Building a Tool for the Analysis and Testing of Web Applications: Problems and Solutions

Filippo Ricca and Paolo Tonella

ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{ricca, tonella}@itc.it
tel. +39.0461.314592, fax +39.0461.314591

Abstract. Web applications are becoming increasingly complex and important for companies. Their design, development, analysis and testing need therefore to be approached by means of support tools and methodologies. In this paper we consider the problems related to building tools for the analysis and testing of Web applications and we try to provide some indications on possible solutions, based upon our experience in the development of the tools ReWeb and TestWeb.
The definition of a proper reference model will be discussed, as well as the impact of dynamic pages during Web site downloading and subsequent model construction. Visualization techniques addressing the large amount of extracted data will be presented, while infeasibility problems will be considered with reference to the testing phase.

1 Introduction

In the last years, Web applications have become important assets for several companies, being a convenient and inexpensive way to provide product information, e-commerce and services on-line. Since a software bug in a Web application could interrupt an entire business and cost millions of dollars, there is a strong demand for methodologies, tools and models that can improve the web site quality and reliability [7,8]. For example, tools can support developers understanding the abstract structure of a Web application by means of views and analyses, ensuring that the requirement specifications are satisfied by the application, and they can help in the testing phase. Developing a tool that extracts a model of a Web application, implements some static analyses and supports the developers in the testing phase is not easy. Main problems are related to: modeling the abstract structure of Web applications, adapting known analysis and testing techniques to the characteristics of Web based systems, and visualizing large graphs [6,10]. Only few works have insofar considered the problems related to Web site static analysis, maintenance, testing and to building the associated tools. One of the first systematic studies on Web maintenance is [12], where the authors recognize the similarity between software systems and web based systems and the importance of the maintenance phase. They have built a tool called SiteSeer

T. Margaria and W. Yi (Eds.): TACAS 2001, LNCS 2031, pp. 373–388, 2001.
© Springer-Verlag Berlin Heidelberg 2001

that downloads web sites and computes some metrics on them. The paper [9] describes SPHINX, a Java toolkit and interactive development environment for Web spiders. SPHINX consists of two parts: the Spider workbench, a customizable spider that supports a graphical user interface and visualizes the web site recovered as a graph, and the WebSPHINX class library, that provides support for writing Web spiders in Java. The CAPBAK/Web tool, explained in [8], is a web testing tool that supports functional testing and regression testing. In [7] an approach to data flow testing of Web applications is presented.

In this paper we consider the problems related to building tools for the analysis and testing of Web applications and we try to provide some indications on possible solutions, based upon our experience in the development of the tools **ReWeb** and **TestWeb**. **ReWeb** downloads and analyzes the pages of a Web application with a twofold purpose: building a model of the application and supplying some views and analyses to the developer. **TestWeb**, a structural testing tool, generates and executes a set of test cases for a Web application whose model was computed by **ReWeb**.

The remainder of this paper is organized as follows: the next section describes a generic Web application infrastructure, Section 3 introduces the general architecture of our tools and presents the adopted analysis model for Web applications, Sections 4 and 5 explain problems encountered and solutions adopted in the development of the tools **ReWeb** and **TestWeb**. Finally, Section 6 concludes the paper.

2 Web Applications

A typical generic Web Application infrastructure is shown in Figure 1 (a similar schema is proposed in [13]).

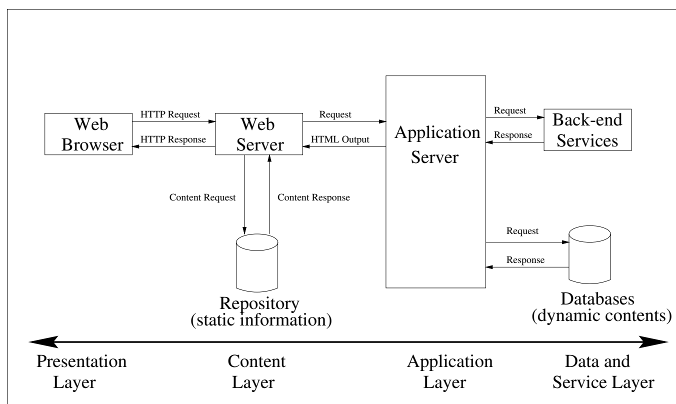

Fig. 1. *Web application infrastructure.*

The browser sends the requests via HTTP to the server for an interactive view of Web pages. Web pages can be static or dynamic. While the content of a static page is fixed and stored in a repository, the content of a dynamic page is computed at run-time by the application server and may depend on the information provided by the user through input fields (a similar distinction is proposed in [4] and [5]). The programs that generate dynamic pages at run-time, as for example CGI scripts and servlets, run on the application server and can use information stored in databases and other resources. The Web server and the application server can be located on the same machine or on different machines.

Similarly to [5], we classify Web applications[1] according to a taxonomy, ordered by growing complexity, which is characterized by dynamism, page decomposition and data flow.

- Level 0: static pages without frames.
- Level 1: static pages with frames.
- Level 2: dynamic pages without data transfer from client.
- Level 3: dynamic pages with data transfer from client.

The difficulties and problems in the construction of a tool, that supports developers in the phases of analysis and testing of Web applications, grow with increasing levels in the taxonomy. Applications at levels 2 and 3 typically exploit information stored inside a database to build the content of the dynamic pages. All four levels are in the scope of the proposed techniques.

3 Tool Architecture

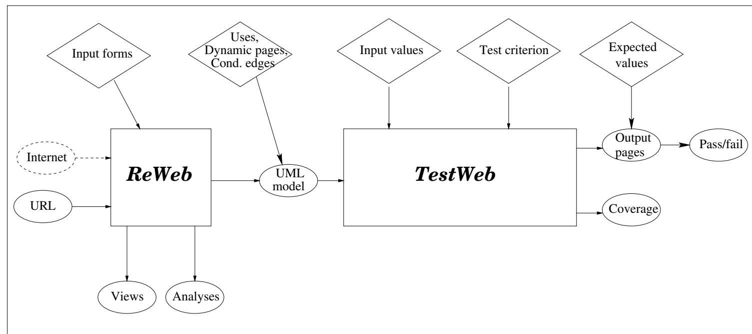

Fig. 2. *Roles of Reweb and TestWeb.*

The two tools **ReWeb** and **TestWeb** have been developed to support analysis and testing of Web applications. Their relative roles are schematized in

[1] Although some authors distinguish between Web sites and applications, using the latter term only in presence of dynamic pages (our levels 2 and 3), we will use them interchangeably in the following if the distinction is not important.

Figure 2. **ReWeb** downloads and analyzes the pages of a Web application with the purpose of building a model of it and producing some analyses and views. **TestWeb** generates and executes a set of test cases for a Web application whose model was computed by **ReWeb**. The whole process is semi-automatic, and the interventions of the user are indicated within diamonds in Figure 2. Explanations on the manual interventions will be given in the following sections.

Both tools perform their operations on an abstraction of the Web applications, indicated in Figure 2 as UML model. UML, the Unified Modeling Language [3], was exploited to express such a model. Let us consider the key requirements on the model. We are interested in a model that can be directly abstracted from the implementation. Some important characteristics that it should have can be summarized as follows:

- The focus should be on the navigational features of the site;
- It should be complete i.e. the most important entities as, for example links, frames, forms and dynamic pages must be explicitly represented in the model;
- It should be possible to provide (partial) automatic support for its extraction;
- It should be possible to apply to it some static analyses and testing techniques derived from those used with traditional software systems.
- It should be possible to derive some views from it that represent the Web site in an intuitive mode;

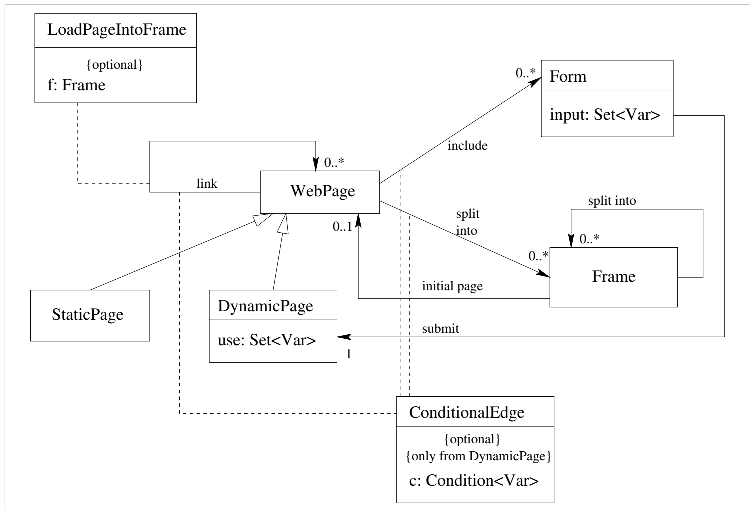

Fig. 3. *Meta model of a generic Web application structure.*

Figure 3 shows our meta model used to describe the elements in the model of a Web application. It satisfies all key requirements given above. The central entity in a Web site is the *WebPage*. A Web 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).

The two subclasses of *WebPage* model the static and dynamic pages. When the content of a dynamic page depends on the value of a set of input variables, the attribute *use* of class *DynamicPage* contains them.

A *frame* is a rectangular area in the current 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 can be gathered by exploiting *forms*. A Web page can include any number of forms (association *include*). Each form is characterized by the input variables that are provided by the user through it (data member *input*). Values collected by forms are submitted to the Web server via the special link *submit*, whose target is always a dynamic page. Since links, frames and forms are part of the content of a Web page, and for dynamic pages the content may depend on the input variables, even the organization of a page is, in general, not fixed and depends on the input. This is the reason for the association class *ConditionalEdge*, which optionally adds a boolean condition, function of the input variables, representing the existence condition of the association (which can in turn be a *link*, an *include* or a *split into*). The target, page, form or frame, is referenced by the source dynamic page only when the input values satisfy the condition in the *ConditionalEdge*.

4 ReWeb

The **ReWeb** tool consists of three modules: a Spider, an Analyzer and a Viewer. The Spider downloads all pages of a target web site, starting from a given URL and providing the input required by dynamic pages, and it builds a model of the downloaded site. Each page found within the site host is downloaded and marked with the date of downloading. The HTML documents outside the web site host are not considered. The user has to specify the set of inputs for each page that contains Forms. The Analyzer uses the UML model of the web site and the downloaded pages to perform several analyses, presented in the following, some of which are exploited during static verification. The Viewer provides a Graphical User Interface (GUI) to display the Web application model as well as the output of the static analyses. The graphical interface supports a rich set of

navigation and query facilities. Web Spider and Analyzer are written in Java, while the Viewer is based on Dotty[2].

4.1 Spider

```
SPIDER(target_url)
1  UML_Model ← ∅
2  Error_urls ← ∅
3  Pages_already_visited ← ∅
4  Urls_found ← ∅
5  S ← {target_url}
6  while (S ≠ ∅)
7      choosen_url ← chooseElement(S)
8      S ← S \ {choosen_url}
9      if not (choosen_url ∈ Pages_already_visited) then
10         if (choosen_url is OK) then
11             Pages_already_visited ← Pages_already_visited ∪ {choosen_url}
12             if (choosen_url is a HTML page) then
13                 Download(choosen_url)
14                 Urls_found ← scanPage(choosen_url)
15                 S ← S ∪ Urls_found
16                 AddElementsToModel(UML_Model, choosen_url, Urls_found)
17             endif
18         else
19             Error_urls ← Error_urls ∪ {choosen_url}
20         endif
21     endif
22 endwhile
```

Fig. 4. *Pseudo-code of the Spider.*

Web pages are not actually written using a single language. They can be rather regarded as multilanguage documents, where code fragments in languages different from HTML can be loaded (e.g. Applets) or interpreted (e.g. Javascript). Libraries for the construction of Spider programs are available for programming languages such as Perl, C/C++, or Java. An example is the WebSPHINX class library [9]. We decided to implement our Web Spider just exploiting the Java language and its standard library, starting from scratch, in order to have total control on the multilingual aspects of the downloaded pages. We developed a parser which recognizes both HTML and Javascript code fragments, and extracts the needed information (links, forms, frames, etc.) from them.

Figure 4 shows the pseudo-code of our Spider. The procedure *SPIDER* takes a given URL in input and builds the associated UML model. The body of the command *while* (contained in lines 6-22) is executed until there are elements in the set S. The function *chooseElement* (line 7) chooses an element in the set S while the condition *chosen_url is OK* (line 10) is true if chosen_url is well-formed and the corresponding page exists in the Web site. The condition *chosen_url is*

[2] Dotty is a customizable graph Editor developed at AT&T Bell Laboratories by Eleftherios Koutsofios and Stephen C. North.

an HTML page (line 12) is true if the content type of the document connected to chosen_url is HTML. In line 13 the procedure *Download* is called to store the retrieved page in the file-system. The function *scanPage* (line 14) scans the page and returns the set of URLs found within it and contained in the site host. The procedure *AddElementsToModel* (line 16) adds nodes and edges to the model in accordance with our meta model. If the set S is implemented as a stack, the algorithm visits the Web site in depth-first way, while using a queue produces a breadth-first visit.

Problems encountered in the construction of the Spider are due to irregularities and ambiguities present in HTML code, also noted in [12], and to the current state of the Web technology, offering a large spectrum of alternatives to implement a web site. Our solution was to improve the robustness of the parser, so that it could accept a superset of HTML including the main irregularities commonly recognized and properly interpreted by available browsers.

Dynamic pages pose additional problem to the activity of the Spider. Since the content of these pages is decided at run time, it may in general depend on the input previously provided by the user. In particular, the structure of a dynamic page may change when it is encountered in a different interaction. Since the model of a Web site encompasses all possibilities, the Spider has to recover all variants of a dynamic page, and has to merge them into a single representative object. This can be achieved by specifying the input values to be provided before downloading the dynamic page of interest. Moreover, the same dynamic page has to be downloaded several times, with different inputs, when the different conditions generate a different page structure. All sequences of input values to be provided before each page download are specified in a file which is read by the Spider. All dynamic pages specified in the file are downloaded after providing the Web server with the given inputs. Finally, all versions of the same page are merged. Although the number of inputs to be provided may explode combinatorially, out experience suggests that in practice few alternatives are sufficient to cover all variants.

An additional input that may affect the content displayed in a dynamic page is the *cookie* that the browser provides to the Web server. A *cookie* is a user identifier that is stored by the browser in the local file system and is provided to the Web server to allow user identification each time a new connection with a given server is established. After recognizing the user, the Web server can provide a customized version of the dynamic pages in the site. Since their structure may depend on the cookie, the Spider needs the ability to send a cookie to the Web server, in order to obtain also the pages that are generated when the user is identified.

4.2 Analyzer

The UML model of a Web site can be interpreted as a graph by associating objects with nodes and associations with edges. Some simple analyses may determine the presence of **unreachable pages**, i.e., pages that are available at the server site but cannot be reached along any path starting from the initial

```
FLOW_ANALYSIS(graph G = (N, E))
1   for each (n ∈ N)
2       initialize IN_n and OUT_n
3   endfor
4   change ← true
5   while (change)
6       change ← false
7       for each (n ∈ N)
8           IN_n ← ⊕_{p∈pred(n)} OUT_p
9           OLDOUT_n ← OUT_n
10          OUT_n ← GEN_n ∪ (IN_n − KILL_n)
11          if OUT_n ≠ OLDOUT_n then
12              change ← true
13          endif
14      endfor
15  endwhile
```

Fig. 5. *Pseudo-code of the flow analysis algorithm.*

page. They are obtained as the difference between the pages available in the Web server file system and those downloaded by the Spider. **Ghost pages** are associated with pending links, which reference a non existing page.

More advanced analyses [11] can be derived from the general framework of flow analysis [1], described by the algorithm in Figure 5. The algorithm propagates flow information inside a graph until the fix-point is reached. The kind of flow information to be propagated depends on the purpose of the analysis being performed. Some examples are given below. Moreover, the confluence operator exploited at line 8 to collect outgoing information from predecessor nodes is also dependent on the analysis, and is typically either the intersection or the union. After initializing the input and output sets of each node (IN_n and OUT_n, lines 1-3) with the initial flow information, propagation is achieved inside a fix-point loop (line 5) by subtracting the destroyed information ($KILL_n$ set) and adding the generated information (GEN_n set) to the incoming information (line 10) for each node n in the graph.

An example of analysis which specializes the algorithm in Figure 5 is the computation of the **reaching frames**, which determines the set of frames in which each page can appear. When a page is loaded into a frame as its initial page or is reachable through an edge decorated with a *LoadPageIntoFrame* association class instance, it *generates* the name of the frame as flow information. By propagating such information along the site graph until the fix-point is reached, the reaching frames of each page are determined. The outcome of the reaching frames analysis is useful to understand the assignment of pages to frames. The presence of undesirable reaching frames is thus made clear. Examples are the possibility to load a page at the top level, while it was designed to always be loaded into a given frame, or the possibility to load a page into a frame where it should not be.

Flow analyses can be employed in a more traditional fashion to determine the **data dependences**. Nodes of kind *Form* generate a definition of each variable in the *input* set. Such definitions are propagated along the edges of the Web site

graph. If a definition of a variable reaches a node where the same variable is used (*use* attribute of a dynamic page), there is a data dependence between defining node and user node. Data dependences are useful to represent the information flows in the application. They may reveal the presence of undesirable possibilities, such as using a variable not yet defined or using an incorrect definition of a variable. Data dependences are also extremely important for dynamic validation, when data flow testing techniques are adopted.

When the pages of a site are traversed, it is impossible to reach a document without traversing a set of other pages, called its **dominators**. Sites in which traversing a given page is considered mandatory, e.g., because it contains important information, will have it in the dominator set of every node. Dominator analysis, also derived from the algorithm in Figure 5, automates the check.

The evolution of web sites [10,12] is another interesting object of investigation. Such an analysis requires the ability to compare successive versions of its pages and to graphically display the differences. Given two versions of a web site, downloaded at different dates, their comparison aims at determining which pages were added, modified, deleted or left unchanged. It can be combined with the static analyses described above, since their re-computation over time allows controlling the evolution of the application quality.

4.3 Viewer

The *graph view* of a Web application is 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* or *ConditionalEdge* relation specifier. In the *graph view*, to intuitively suggest decomposition into frames, we adopt the convention of joining horizontally the nodes of type frame contained in the same page, and collapsing the edges of type "split into" into a single edge. An example of decomposition into frames is shown in Figure 6. Page `madmaxpub/index.html` (the main page) is divided into two frames with identifiers `a` and `b`, and frame `a` is used as a menu to force the loading of pages into the other frame.

The *graph view* of a web application can be enriched with information about its history [10], by coloring the nodes and associating different colors to different time points (see Figure 6). In particular, a scale of colors ranging from the blue, going through the green and reaching the red can be employed to represent nodes added/modified in the far past, in the medium past or more recently.

The Viewer is based on Dotty, and uses the algorithm explained in [6] for drawing directed graphs. The aesthetic principles followed by the algorithm are: to expose hierarchical structure (if any) in the graph, to avoid edge crossings (if possible) and sharp bends, to keep edges short, to favor symmetry and balance. The layout algorithm of the *graph view* of a web site is very important to understand its structure, especially when the site is very complex.

Another problem connected with the visualization of a web site is the fact that also small sites (e.g. with 100 pages) can have an entangled structure difficult to understand. A way to improve the Viewer display is to use techniques to

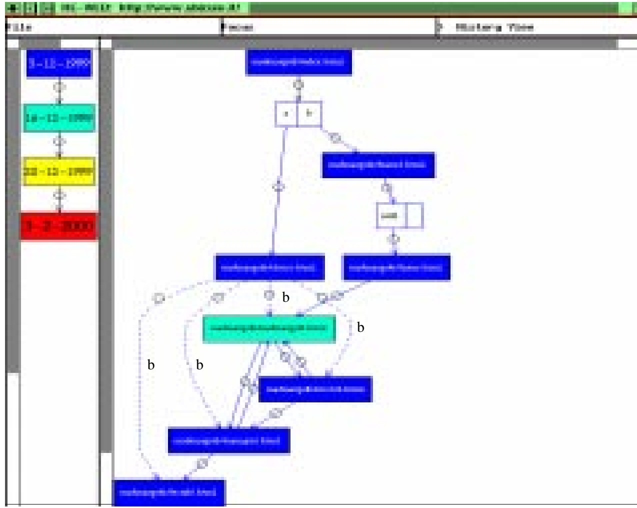

Fig. 6. *Colored graph view of the site www.ubicum.it/madmaxpub at date 3-2-2000.*

abstract, to simplify, to extract a portion of, or to see only a part of the *graph view*. Another possibility can be to add other views of a web site as for example the birdeye and overview diagrams or to display only the depth-first tree without return edges (solution adopted in [9]). The views and facilities we propose follow. The *system view* represents the organization of pages into directories; the *data flow view* displays the read/write accesses of pages to variables, respectively through incoming/outgoing edges linking pages to variables; the *history representation with percentage bars* describes, in compact way, the percentages of nodes with the same color. Among the provided facilities, the viewer supports zoom, search, deletion of incoming or outgoing edges, and focus. The facilities for focusing on and searching a node are useful when the visualized graphs are very large. By exploiting the focusing facility it is possible to display only a limited neighborhood of a selected node. Another possibility to access the *graph view*, not yet implemented, is the identification and extraction of a portion of a web site by means of pattern matching techniques. Recurrent patterns are expected to be used in the design of web sites (for example *tree*, *hierarchy*, *full connectivity*, *indexed-sequence*).

5 TestWeb

Web sites can involve a complex interaction among Web browser, operating systems, plug-in applications, communicating protocols, Web servers, databases, server programs (for example CGI programs) and firewalls. Such complexity makes the test of Web Sites a great challenge [8]. Ideally all components and functionality of a Web site on both client and server sides should be completely tested. However, this is rarely possible in modern Web site projects because of

the extreme time pressure under which Web systems are developed. Available testing techniques [8] differ on the features of the Web site we want to test. For example it is possible to execute link testing, HTML validation, performance testing, and security testing. We are interested in *dynamic validation* using our UML model as a base for this type of testing. In general, dynamic validation methods aim at exercising the system by supplying a vector of input data (*test case*) and comparing the expected outputs with the actual ones after execution. In particular, we considered *white box testing* of Web Applications: the internal structure of a Web application is accessed to measure the coverage that a given *test suite* (collection of test cases) reaches, with respect to a given *test criterion* (stating the features to be tested). Some white box testing criteria, derived from those available for traditional software [2], are: Page testing, Hyperlink testing, Definition-use testing, All-uses testing, All-paths testing. A *test case* for a Web application is a triple: URL, input (a sequence of variable-value assignments separated by the character '&'), type of parameter passing (GET or POST). Execution consists of requesting the Web server for the URL in the triple with the associated input and storing the output pages. Satisfaction of any of the white box testing criteria involves selecting a set of paths in the Web site graph and providing input values. Since path selection is independent (conditional edges excluded) from input values, it can be automated.

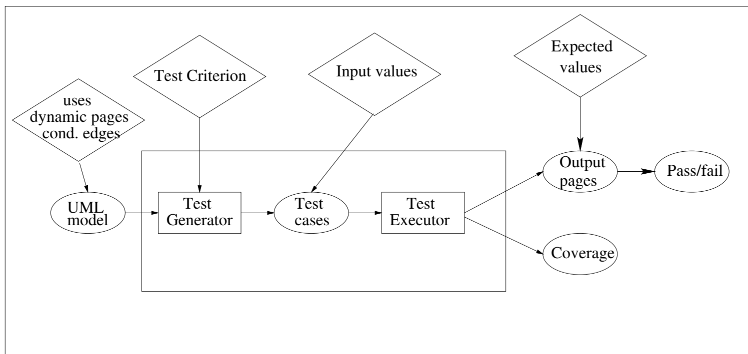

Fig. 7. *Architecture of the tool TestWeb.*

TestWeb (see Figure 7) contains a test case generation engine (Test generator), able to generate test cases from the UML model of a Web application. The user has to add some information to the model produced by **ReWeb** to complete it for testing purposes and furthermore the user has to choose a test criterion. The user specifies the page type when the distinction between static and dynamic pages cannot be obtained automatically (e.g., dynamic pages with no input). The user also provides the set of used variables, *use*, for each dynamic page whose content depends on some input value. Finally, the user has to attach conditions to the edges whose existence depends on the input values.

Additional manual interventions, related to state unrolling, will be described in the following on an example. Generated test cases are sequences of URLs which, once executed, grant the coverage of the selected criterion. Input values in each URL sequence are left empty by the Test generator, and the user has to fill in them, possibly exploiting the techniques traditionally used in black box testing (boundary values, etc.). **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. A second, numeric output of test case execution is the level of coverage reached by the current test suite.

Regression testing highly benefits from the automation in test case execution, since each test case can be re-executed unattended on a new version of the Web application, and its output pages can be automatically compared with those obtained from a run of the previous version.

5.1 Test Generator

Given the graph representation of a Web application, a *reduced graph* can be computed for the purposes of white box testing: each static page without forms is removed from the graph by a Cross-Term step described in [2] (see Figure 8).

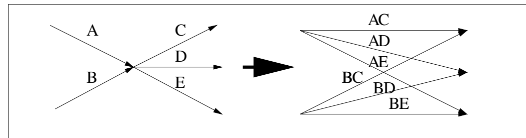

Fig. 8. *Step of the Cross-Term algorithm at a node selected for removal.*

In the resulting graph, a fictitious *entry* node is added, connected with all nodes with no predecessor, and a fictitious *exit* node is directly reachable from all *output* nodes, i.e., dynamic nodes with non empty *use* attribute. In fact, the end of a computation is reached, in a Web application, when some result is displayed to the user, but no intrinsic notion of termination for a navigation session exists.

Differently from the flow-graph of a structured program, the *graph view* of a Web application can contain horrible-loops [2], i.e., there may be nodes jumping into or out of a loop and/or there may be more than one iterating node for the same loop. In presence of horrible-loops the usual strategies used to cover nested-loops and concatenated-loops do not work. We have chosen a general solution: a test case generation technique based on the computation of the path expression [2] of the reduced Web site graph. A *path expression* is an algebraic representation of all paths in a graph. Variables in a path expression are edge labels. They can be combined through operators $+$ and $*$, associated respectively with selection and loop. Brackets can be used to group subexpressions.

```
REDUCTION(graph)
1  Combine all serial links by multiplying their expressions
2  Combine all parallel links by adding their path expressions
3  Remove all self-loops by replacing them this a link of the form X*
4  while (number of nodes in the graph > 2)
5      n ← choose a node of the graph different from initial or final node
6      Apply Cross-Term elimination to n
7      Combine any remaining serial links as in step 1
8      Combine all parallel links as in step 2
9      Remove all self-loops as in step 3
10 endwhile
```

Fig. 9. *Reduction algorithm.*

Computation of the path expression for a site can be performed by means of the Reduction algorithm described in [2] and depicted in Figure 9.

The lines 1-3 of the Reduction algorithm initialize the process and put the graph in normal form. The body of the command *while* is executed until the number of nodes in the graph is greater than 2. Line 5 assigns a node of the graph different from the initial or final node to variable n, while line 6 executes a Cross-Term step on node n. This step eliminates the node n and transforms the graph according to the diagram shown in Figure 8. Lines 7-10 combine all serial and parallel links and remove self-loops. At the end of the execution, the path-expression of the input graph is obtained.

```
PATH_GENERATION(path_expression)
1  while criterion not satisfied
2      for each alternative from inner to outer nesting
3          choose one never considered before, if any
4          or randomly choose one
5      endfor
6      if computed path increases coverage then
7          add it to the resulting paths
8      endif
9  endwhile
```

Fig. 10. *The heuristic technique adopted to obtain the paths satisfying a criterion.*

Since the path expression directly represents all paths in the graph, it can be employed to generate sequences of nodes (test cases) which satisfy any of the coverage criteria. Determining the minimum number of paths, from a path expression, satisfying a given criterion is in general a hard task. However, heuristics can be defined to compute an approximation of the minimum. The heuristic technique adopted for this work is based on the scheme of Figure 10 (the alternative at line 2 for a loop is whether to re-iterate or not). Definition-use and all-uses testing can be achieved by considering, for each data dependence, the definition as *entry* node and the use as *exit* of the subgraph to be tested. Criteria

such as definition-use and all-paths testing, for which the coverage of possibly infinite paths should be achieved, could require that only independent paths be considered or that loops be k-limited.

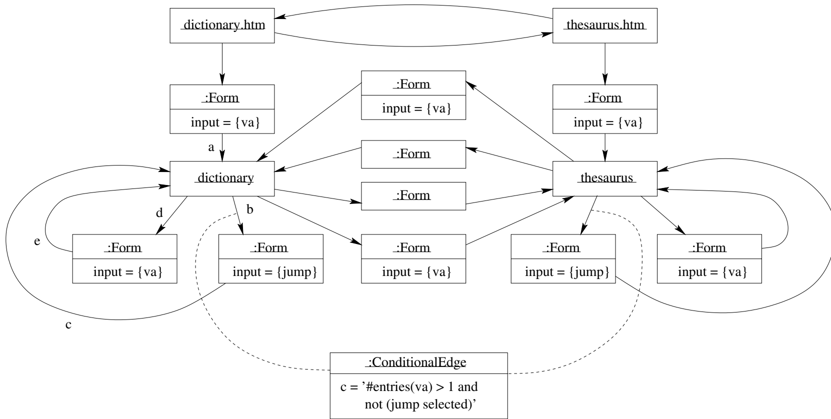

Fig. 11. *Model of a portion of* `www.m-w.com`*, including two conditional edges.*

Figure 11 shows the portion of the Web site `www.m-w.com` which provides on line access to the Merriam-Webster English dictionary and thesaurus. A word can be entered in the initial page (either `dictionary.htm` or `thesaurus.htm`). A dynamic page `dictionary` is then composed in response to the input word, stored in variable `va`. The content of the resulting page depends on the number of entries found in the dictionary. If there are more than one entry, a selection list is displayed to the user, together with an explanation of the main entry. The user can choose among the alternatives – the selection is stored in variable `jump` – and move to a page, still named `dictionary`, with an explanation of such an entry. The model of the site represents the conditional existence of the list of alternatives with a ConditionalEdge object associated with the edge labelled b. The site offers the possibility to enter a new word from both the dynamic pages `dictionary` and `thesaurus`, and allows switching from dictionary to thesaurus and vice versa from the initial and result pages.

In order to determine a set of paths to be exercised during white box testing, the path expression is computed. Let us consider the portion of the site devoted to the extraction of entries from the dictionary (symmetric considerations can be made for the thesaurus). The path expression associated with the labelled edges of Figure 11 is $a(bc + de)^*$. Some of the paths generated from it can be traversed only if proper inputs are provided, while some other paths are infeasible for every input. For example, the path abc can be traversed only if the input word has more than one entry in the dictionary. This condition is quite easy to achieve and requires only a careful selection of input data. A path whose infeasibility does not depend on the input is $abcbc$. In fact, if one of the entries is

selected from the list displayed to the user, the next dynamic page `dictionary` that is obtained will not include the list of alternative entries any longer. This condition is represented as `'not (jump selected)'` in Figure 11: if a selection was performed by the user, edge b does not exist. As a consequence the path expression cannot be easily exploited for path generation.

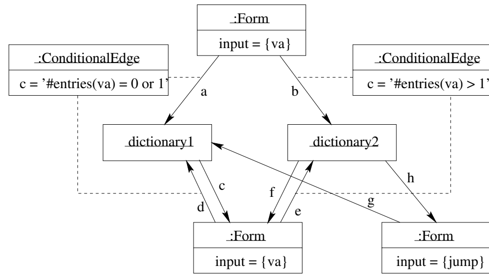

Fig. 12. *Page* `dictionary` *was unrolled into* `dictionary1` *and* `dictionary2`.

The problem highlighted above derives from the possibility to use a same dynamic page for different purposes. Actually, some Web sites consist of just one dynamic page which displays different information according to an internally recorded state of the interaction. In other words, while for static sites the Web page is coincident with the state of the interaction, this is not necessarily true with dynamic sites. A possible solution to this problem is to perform an operation of state unrolling on the dynamic pages that are used to display different contents under different conditions. In the example of Figure 11, page `dictionary` is used for two purposes: to propose a list of alternative dictionary entries and to provide the final result of the search, once a single entry is identified. Such two purposes may be represented explicitly by the two pages `dictionary1` and `dictionary2` into which the initial page is unrolled (see Figure 12). The conditions in the *ConditionalEdge* objects are now simplified and verify only the number of dictionary entries. The path expression of such site portion becomes $(ac + bf + bhgc)(dc + ef + ehgc)^*$ and all the paths that can be generated from it are feasible, provided that an input word with the appropriate number of dictionary entries is selected.

6 Conclusions and Future Work

We proposed some analysis and testing techniques working on Web applications. The starting point for their definition is a model of Web sites, designed to include all characteristics that are relevant from an architectural point of view. Page downloading and model construction were achieved by providing input values for the forms in the site. Moreover, the site model was enriched with information about conditional edges and variable uses, exploited during testing.

Facilities for the display of the resulting model are provided by the analysis tool **ReWeb**, while test generation and execution is automated by **TestWeb**. Our experience with **ReWeb** and **TestWeb** suggests that the choice of a good model is fundamental for both analysis and testing. We showed some views and facilities of **ReWeb** on a real example (`www.ubicum.it`). Path testing in presence of an internal representation of the interaction state can be simplified by means of a state unrolling operation, which was also described with reference to a real world example (`www.m-w.com`).

Our future work will be devoted to extending the set of analyses available (to include, for example, pattern matching), adding abstraction techniques to support a high level view of the site, (partially) automating input selection during testing in presence of conditional edges and providing better support to state unrolling.

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1985.
2. B. Beizer. *Software Testing Techniques, 2nd edition*. International Thomson Computer Press, 1990.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language – User Guide*. Addison-Wesley Publishing Company, Reading, MA, 1998.
4. J. Conallen. *Building Web Applications with UML*. Addison-Wesley Publishing Company, Reading, MA, 2000.
5. D. Eichmann. Evolving an engineered web. In *Proc. of the International Workshop on Web Site Evolution*, Atlanta, GA, USA, October 1999.
6. E.R. Gasner, E. Koutsofios, S. North, and Kiem-Phong Vo. A technique for drawing directed graphs. In *IEEE-TSE 1993*, March 1993.
7. Chien-Hung Liu, David C.Kung, Pei Hsia, and Chih-Tung Hsu. Structural testing of web applications. In *Proc. of ISSRE 2000, International symposium on software reliability engineering, San Jose, California*, pages 84–96, October 2000.
8. Edward Miller. The web site quality challenge. - companion paper: "website testing". In *Proc. of QW1998 conference*, 901 Minesota street San Francisco, CA 94107 USA, 1998.
9. Robert C. Miller and Krishna Bharat. Sphinx: A framework for creating personal, site-specific web-crawlers. In *Proc. of WWW7, Brisbane Australia*, April 1998.
10. F. Ricca and P. Tonella. Visualization of web site history. In *Proc. of the International Workshop on Web Site Evolution*, pages 30–33, Zurich, Switzerland, 2000.
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. P. Warren, C. Boldyreff, and M. Munro. The evolution of websites. In *Proc. of the International Workshop on Program Comprehension*, pages 178–185, Pittsburgh, PA, USA, May 1999.
13. Y. Zou and K. Kontogiannis. Enabling technologies for web-based legacy system integration. In *Proc. of the International Workshop on Web Site Evolution*, Atlanta, GA, USA, October 1999.