Analysis and Testing of Web Applications

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

ABSTRACT
The economic relevance of Web applications increases the importance of controlling and improving their quality. Moreover, the new available technologies for their development allow the insertion of sophisticated functions, but often leave the developers responsible for their organization and evolution. As a consequence, a high demand is emerging for methodologies and tools for quality assurance of Web based systems.

In this paper, a UML model of Web applications is proposed for their high level representation. Such a model is the starting point for several analyses, which can help in the assessment of the static site structure. Moreover, it drives Web application testing, in that it can be exploited to define white box testing criteria and to semi-automatically generate the associated test cases.

The proposed techniques were applied to several real world Web applications. Results suggest that an automatic support to the verification and validation activities can be extremely beneficial. In fact, it guarantees that all paths in the site which satisfy a selected criterion are properly exercised before delivery. The high level of automation that is achieved in test case generation and execution increases the number of tests that are conducted and simplifies the regression checks.

Keywords
Web applications, testing, UML modeling, code analysis, reverse engineering.

1 INTRODUCTION

The development of Web applications is following an evolution similar to that observed for software systems: production is moving from an artistic phase based on highly skilled craftsmen to an industrial phase in which quality is controlled by introducing a structured workflow. Such a change is still in progress for Web applications, and there is a demand for methodologies, tools and models to be employed within the new development process.

Web based systems tend to evolve rapidly and undergo frequent modifications, due to new technological and commercial opportunities, as well as feedback from the users. For such reasons iterative development process models, based on the notion of rapid prototyping and continuous change, seem to fit the conditions in which Web sites are produced and maintained [7].

In the context of an iterative process, the role of analysis is crucial, since a working system comes to life very soon. Analysis results can be exploited to support understanding and modification, since they provide a high level view of the existing system and they can answer queries about its organization and functions.

At the end of each iteration the application is tested. When high quality standards have to be achieved, it may be appropriate to complement the usual functional, black-box testing with the structural, white-box one. White-box testing exploits the internal structure of a Web application to define the coverage criteria. Therefore, analysis is a prerequisite, in that it allows determining the testing points (e.g., branches, data-flows, etc.). It is also the starting point when test cases are automatically or semi-automatically generated.

In this paper an analysis model is defined and adopted for the high level representation of Web applications. Several static analyses are based on it, derived from those used with software systems. Testing techniques are also adapted to the peculiarities of Web based systems. The internal structure and data flows of the Web application under test drive the definition of the testing criteria and the generation of the test cases. The usefulness of the proposed methods, implemented by the tools **ReWeb** and **TestWeb**, will be discussed with reference to two case studies.

The paper is organized as follows: Section 2 presents the adopted analysis model. In Section 3, some verification and validation techniques are proposed which can be used with Web applications. The tools developed to support them are described in Section 4, while their application to real world examples is discussed in Section 5. A comparison with related works is conducted in Section 6. Finally, conclusions are drawn in Section 7.

0-7695-1050-7/01 $10.00 © 2001 IEEE

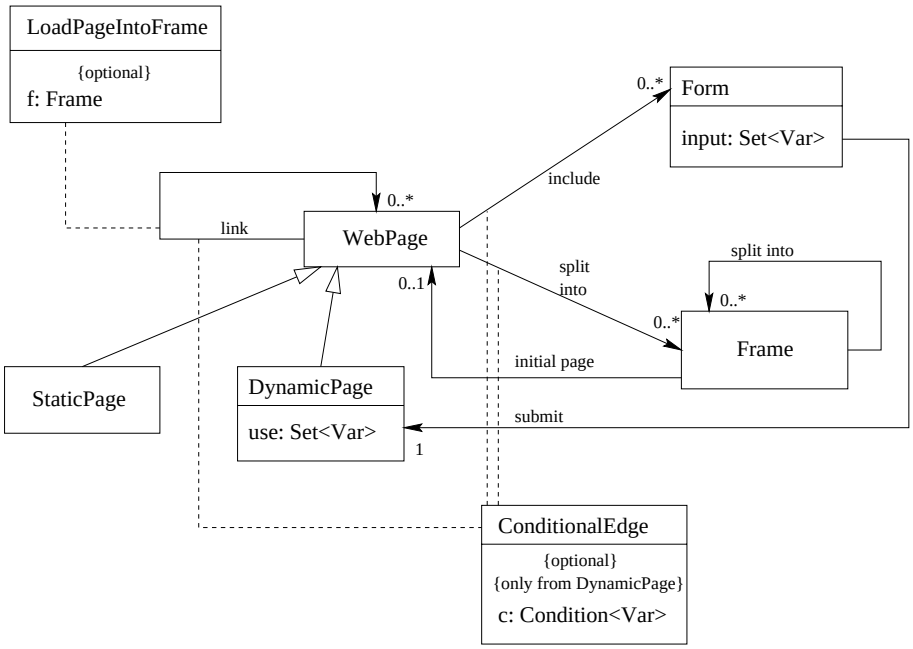

Figure 1: *Meta model of a generic Web application structure. The model of a given site is an instantiation of it.*

2 ANALYSIS MODEL

Web applications[1] exploit the navigation and interaction facilities of hypertextual HTML pages to provide and ask information to/from the user. As a consequence, the analysis model proposed here emphasizes the navigation and interaction patterns over other architectural perspectives. Alternative, more standard, architectural views could be given for the other aspects (e.g., entity-relationship diagrams for data modeling).

In the following, a Web application is identified with all the information that can be accessed from a given Web server. Documents accessed from different servers are considered external to the given application. The analysis of a Web application is supposed to be conducted in the development environment. Consequently, some information that cannot be accessed by external users browsing the site is considered available. In some cases it can be extracted automatically from artifacts used by the Web server (e.g., CGI scripts), while in other cases it is assumed to be manually provided by the developers.

Figure 1 shows the meta model used to describe a generic Web application. It is given in the Unified Modeling Language (UML) [5]. The central entity in a Web site is the *Web Page*. 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). Navigation from page to page is modelled by the auto-association of class *WebPage* 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 in [7] and [8]) and may depend on the information provided by the user through input fields. The two subclasses of *WebPage* model such alternatives. 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

[1] Although some authors distinguish between Web sites and applications, using the latter term only in presence of dynamic pages, we will use them interchangeably when the distinction is not important.

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*.

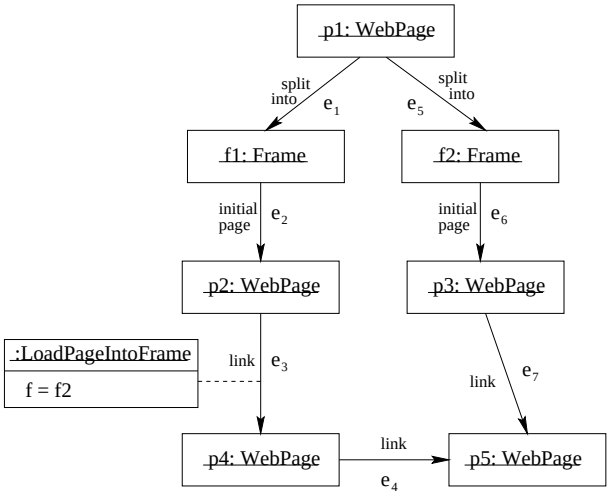

Figure 2: *Example of model with frames.*

Figure 2 shows an example of model for a site with frames. The model of a given site is an instance of the meta model in Figure 1, and therefore its entities are objects (while entities in the meta model are classes).

The links between `p3` and `p5`, and between `p4` and `p5` are normal navigation connections between HTML pages. Page `p1` is decomposed into the two frames `f1` and `f2`. The links between `f1` and `p2` and between `f2` and `p3` indicate that the pages initially loaded into `f1` and `f2` are respectively `p2` and `p3`. Finally, the association connecting `p2` to `p4` has an attribute specifying that the link in `p2` does not result in the navigation within `f1` toward a different page, but rather produces the loading of page `p4` into frame `f2`, with no regard to the page currently loaded into `f2`.

In the example in Figure 3, the static page `p1` includes a form to gather data from the user. The `input` attribute of the related unnamed *Form* object contains the two input variables of the site, `x1` and `x2`. Page `p2` is dynamic and its content depends on the value of the input variable `x1`. Pages `p3` and `p4` are static, but the dynamic page `p2` contains links toward them only when the input variable `x1` is respectively equal to the string `"books"` or `"movies"`, as represented in the *ConditionalEdge* instances. In this simple example of

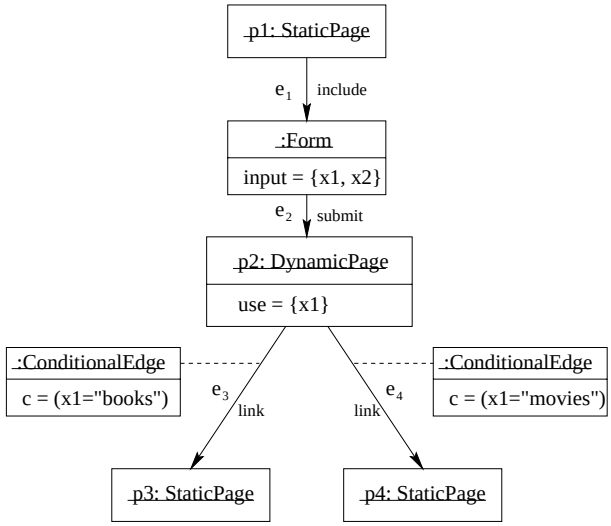

Figure 3: *Example of model with a form.*

Web application, page `p1` provides a search facility and `p2` displays the result of the search. When the search category is `"books"` or `"movies"`, the site can provide additional information through links respectively to `p3` or `p4`.

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. Labelled edges are used for the links having a *LoadPageIntoFrame* or *ConditionalEdge* relation specifier. Since the model adopted for Web sites is aimed at explicitly representing navigation, 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. In the following, model and graph terms (e.g., page vs. node, link vs. edge, etc.) are used interchangeably.

3 TESTING

As with traditional software, verification and validation processes have the purpose of respectively checking the quality of the implementation and revealing non compliances with the user requirements. Static techniques differ from dynamic ones in that they do not require execution. They rather work on artifacts such as design documents and HTML source. 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.

Static verification

In addition to inspections, which are a valuable verification strategy also for Web applications, static analyzers can be employed to scan the HTML pages in a Web site and detect possible faults and anomalies.

Static checks can be focused on the navigation paths provided to the user or on the data flows of the information gathered from the user. More details on the static analyses defined for Web applications, which are only summarized in the following, can be found in [10]. An experience of Web site re-structuring based on these analyses is described in [11].

The model of a site can be analyzed to 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 page. **Ghost pages** are associated with pending links, which reference a non existing page (for this analysis external pages are also of interest).

The analysis of the **reaching frames** is a specialization of the flow analysis framework [1], which computes 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 labelled with the frame (*LoadPageIntoFrame* association class instance), it *generates* the name of the frame as flow information. By propagating such information along the site graph until the fixpoint 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 web site are traversed, it is impossible to reach a given 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 automates the check.

A useful information about a web site is the minimum number of pages that must be visited before reaching the target document. Information about the **shortest path** to each page in the site is an indicator of potential troubles for the user searching a given document, when such path is long.

Re-computation of the static analyses described above over time allows controlling the evolution of the application quality.

Dynamic validation

This section is focused on *white box testing*: 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). A *test case* for a Web application is a sequence of pages to be visited plus the input values to be provided to pages containing forms. Therefore it can be represented as a sequence of URLs specifying the pages to ask and, if needed, the values to assign to the input variables. Execution consists of requesting the Web server for the URLs in the sequence and storing the output pages. Differently from traditional software, branch selection can be forced by choosing the associated hyperlink, without having to track conditions back to input values (except for conditional edges). Some white box testing criteria, derived from those available for traditional software [3], are:

- **Page testing**: every page in the site is visited at least once in some test case.
- **Hyperlink testing**: every hyperlink from every page in the site is traversed at least once.
- **Definition-use testing**: all navigation paths from every definition of a variable to every use of it, forming a data dependence, is exercised.
- **All-uses testing**: at least one navigation path from every definition of a variable to every use of it, forming a data dependence, is exercised.
- **All-paths testing**: every path in the site is traversed in some test case at least once.

The definition-use and all-paths criteria are often impractical, since there are typically infinite paths in a site, if loops are present. They can be satisfied if additional constraints are imposed on the paths to be considered. Examples are loop k-limiting and path independence. In the first case, loops are traversed at least k times, while in the second case only *independent paths* are considered. A path is *independent* from a given set of paths if its vector representation is linearly independent from that of any other path in the set.

When designing and executing test cases for a Web application, not all pages are equally of interest. Static pages not containing forms can be disregarded, since they do not collect user input, process it or display results. They contain fixed information that needs not be examined during dynamic validation. The site can thus be reduced, while retaining only the relevant entities. 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 and all its predecessors (if any) are linked to all its successors. 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. Therefore, dynamic pages whose content depend on the user input are good candidates for ending meaningful computations.

Test case generation
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 of input values, apart from conditional edges, it can be automated.

We propose a test case generation technique based on the computation of the path expression [3] of the reduced Web site graph. A *path expression* is an algebraic representation of the 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. The path expressions for the two examples in Figure 2 and 3 are: $e_1e_2e_3e_4 + e_5e_6e_7$ and $e_1e_2(e_3 + e_4)$. In the former, the two alternative paths go from `p1` to `p5` respectively through `f1` or `f2`. In the latter, a selection is encountered at `p2`, with e_3 = `(p2, p3)` and e_4 = `(p2, p4)`. Another example of path expression is: $(e_1e_3 + e_2e_4)^*$. It corresponds to a site where the initial page, say `p1`, is followed either by `p2` or by `p3` (edges e_1 and e_2). In turn, pages `p2` and `p3` are connected to `p1` via edges e_3 and e_4 respectively, thus generating a loop.

Computation of the path expression for a site can be performed by means of the Node-Reduction algorithm described in [3]. 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 following scheme:

while criterion not satisfied
  for each alternative from inner to outer nesting
    choose one never considered before, if any
    or randomly choose one
  if computed path increases coverage
    add it to the resulting paths

where the alternative 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, requiring the coverage of all paths, could require that only independent paths be determined or that loops be k-limited.

Once test cases are generated from the path expression of the site, the test engineer has to insert input values for the variables collected through forms. Then, their execution can be automated and, after downloading, the output pages can be inspected to assess whether the test case was passed or not.

Regression testing highly benefits from the automation described above, 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.

4 REWEB AND TESTWEB TOOLS

The two tools **ReWeb** and **TestWeb** have been developed to support analysis and testing of Web applications. Their relative roles are schematized in Figure 4. **ReWeb** downloads and analyzes the pages of a Web application with the purpose of building a UML model of it, in accordance with the meta model described in Section 2. **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 4.

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. 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 pages of a site are obtained by sending the associated requests to the Web server. The result of such requests is always an HTML page, so that it is not possible to discriminate between dynamic and static pages. The only exception is represented by pages that are the target of form submissions. Since they have to handle input values, they are necessarily dynamic. As a default assumption, **ReWeb** marks all pages as static apart from those immediately reachable from a form. The user can modify the resulting model by switching the page type from static to dynamic whenever appropriate. The user has also to provide the set of used variables, *use*, for each dynamic page whose content depends on some input value. Finally, the user is required to attach conditions to the edges whose existence depends on the input values. Input values making such conditions true have also to be provided to the Spider for a complete download of the site (dashed arrow in Figure 4). Additional manual interventions, related to state unrolling and merging, will be described in Section 5.

The Analyzer uses the UML model of the web site to perform

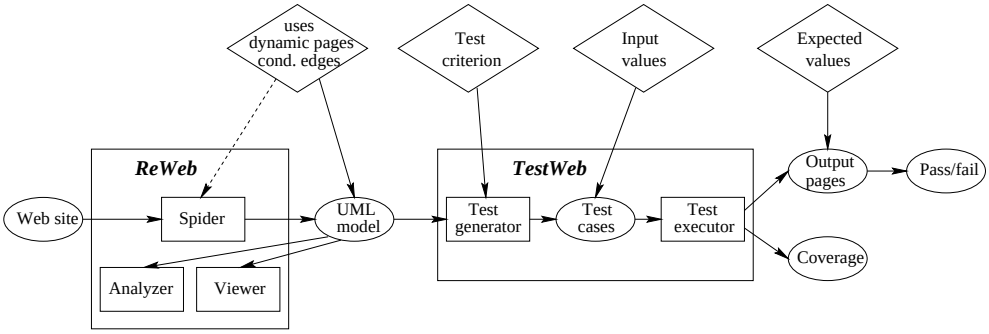

Figure 4: *Roles of* **ReWeb** *and* **TestWeb** *in the analysis and testing process.*

several analyses [10], some of which are exploited during static verification (see Section 2).

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 including zoom, search, focus and HTML code display. Among the available views, the *history view* shows the structure of the site over time, the *system view* represents the organization of pages into directories, and the *data flow view* displays the read/write accesses of pages to variables, respectively through incoming/outgoing edges linking pages to variables.

TestWeb contains a test case generation engine (Test generator), able to determine the path expression from the model of a Web application, and to generate test cases from it, provided that a test criterion is specified. 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, marked in the UML model with a non empty *use* attribute, are stored for further examination. After execution, the test engineer intervenes to assess the pass/fail result of each test case. For such evaluation, she/he opens the output pages on a browser and checks whether the output is correct for each given input. 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.

5 EXPERIMENTAL RESULTS

Over 15 Web sites were periodically downloaded and analyzed by **ReWeb**. Examples of outcomes of their static verification can be found in [10, 11]. In the following the analysis and dynamic validation of the two sites Wordnet and Amazon will be presented and discussed. The richness of their dynamic structure makes them an interesting case study for analysis and testing. Approximate server side information, necessary for their analysis, was deduced through interaction sessions and examination of the output pages content.

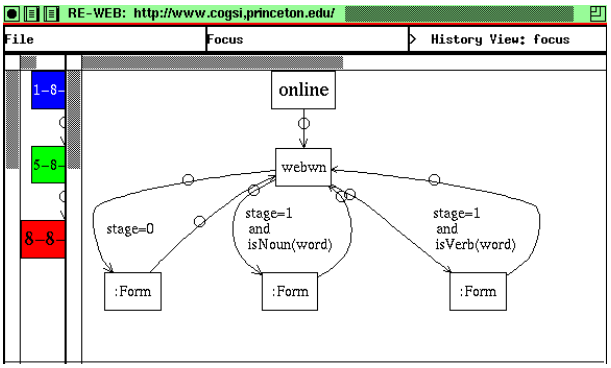

Figure 5: *Portion of UML model of Wordnet.*

Wordnet (`http://www.cogsci.princeton.edu/~wn/`) is a lexical reference system developed at the Cognitive Science Laboratory of the Princeton University. English nouns, verbs, adjectives and adverbs are organized into synonym sets, each representing one underlying lexical concept. Different relations link the synonym sets. The Wordnet database can be either downloaded or accessed through a Web interface. The portion of the Wordnet site providing Web access to Wordnet functionalities was selected for further investigation. A part of its model, as recovered by **ReWeb**, is shown in Figure 5. Information attached to the edges was manually added, and represents the condition of each *ConditionalEdge* association class instance.

Page `online` is linked to the dynamic page `webwn`. When loaded for the first time from `online`, page `webwn` displays a form where the user can digit a word. After submitting the form, page `webwn` is re-loaded, but now it contains a list of forms, the first associated with the senses of the input word considered as a noun, the second considers it a verb, and so

on. After filling and submitting one of the available forms, page `webwn` is re-loaded for the third time, now providing the final result of the query.

An interesting feature of this site is the behavior of page `webwn`, whose content depends on the *state* of the user interaction. Actually, a simple three state model is assumed, where state 0 is associated with the initial input page, state 1 with the selection among alternative senses and lexical functions of the word and state 2 with the display of the final output. Wordnet exploits a general mechanism to implement the state and the state transitions. It inserts a hidden variable, `stage`, in each instance of page `webwn`, and it passes an incremented value of it to the server, via form submission, when a state transition has to occur. A variant of this pattern is the use of cookies: when a user connects to the Web server to request a new page, a user identifier (*cookie*) is also transmitted, which allows storing and updating the state of the interaction of each connected user. As a consequence of such implicit state representation, the edge from `webwn` to the leftmost form in Figure 5 exists only when `stage` is equal to zero, while the other forms are available from `webwn` only when the state of the interaction is the successive one (`stage = 1`) and the input word has a lexical type of verb, noun, etc. Last loading of `webwn` receives a hidden input value of `stage` equal to 2, giving origin to no outgoing conditional edge.

Another possible organization of Wordnet, aimed at avoiding the internal representation of the interaction state, would replicate page `webwn` for each successive state. The resulting model is depicted in Figure 6, where `webwn0`, `webwn1` and `webwn2` are the pages previously provided by the server in states 0, 1 and 2 respectively. Note that the condition about the value of `stage` was removed from the edges. With this choice the state of the interaction is coincident with the page currently loaded into the browser.

When testing a Web application such as Wordnet, with a partially implicit state, it may be necessary to preliminarly *unroll* it. In fact, computing the set of feasible paths, i.e., those with conditional edges evaluating to true, on the model in Figure 5 is a hard task. On the contrary, feasible paths can be easily computed from the path expression $e_1e_2e_3(e_4e_5 + e_6e_7)$ (see Figure 6) of the unrolled site. The all paths coverage, as well as any other weaker coverage, is thus achieved: for the remaining edge conditions (`isNoun(word)`, etc.), it is simple to select inputs making them true. Finally, test cases were executed, after restoring the original page names (`webwn` instead of `webwn0`, `webwn1` and `webwn2`). No defect was revealed by the white-box testing of Wordnet, thus indicating that the application behaves properly along all paths of the site, for the selected inputs.

Amazon (`http://www.amazon.com/`) is a well-known e-commerce site for the purchase of books, music and other goods. It allows searching an item by keyword (and optionally category). The selected item can be inserted into a shopping cart, which is associated to each customer. Customer identification at the beginning of the interaction is achieved by means of cookies, while accounting information (email and password) is required for more advanced operations, like shipping and payment of the selected goods. Other features of this site include the possibility to buy and send gift certificates, by which the receiving person can obtain goods from Amazon for a given amount. When shipping a gift to a person, items can be wrapped and an accompanying message can be specified. The highly dynamic structure of this site makes it an interesting case study for our tools.

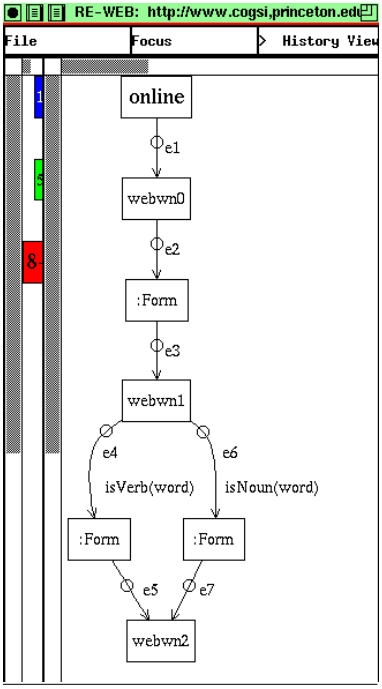

Figure 6: *Portion of UML model of the Web application Wordnet, with state unrolled.*

The portion of Amazon devoted to handling the transactions for buying goods (and gift certificates) was analyzed by means of **ReWeb**. It is shown in Figure 7 (forms have grey background). As indicated in Table 1, this portion of Amazon includes 39 nodes (20 pages and 19 forms) and 73 edges. Similarly to Wordnet, 3 pages were unrolled to explicitly represent the different contents displayed under different conditions. In addition, this site required 10 operations of page merging. *Page merging* is necessary whenever the Web server generates links to different dynamic pages that provide the same content to the user. This typically occurs when the name (path included) of the dynamic page is exploited as an indicator of the state of the interaction. Consequently the same page, generated under different interaction conditions, is given different names. In Amazon, pages are named with a path indicator followed by a name which is always equal to the user cookie. Therefore only

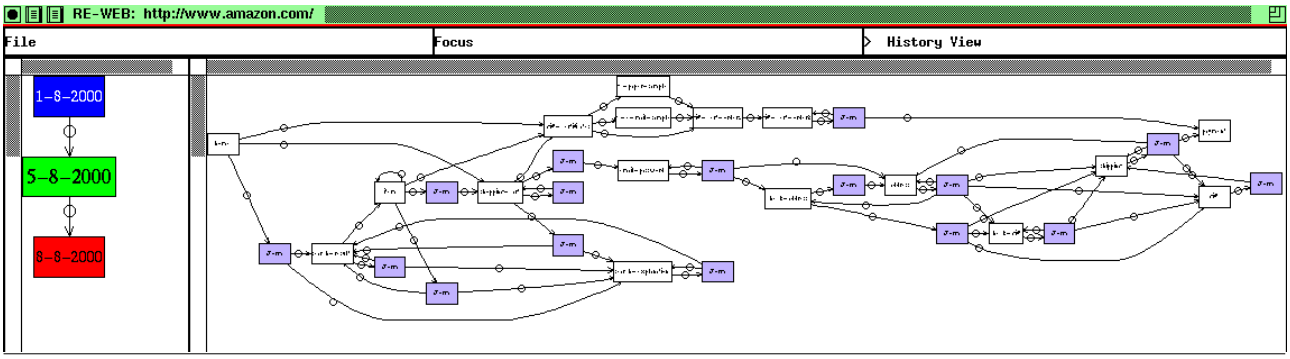

Figure 7: *Portion of Amazon handling purchase transactions. History view of* **ReWeb**.

the directories in the path contain useful naming information (for this reason pages were renamed before displaying them with **ReWeb**). An example of merging is page `address`, whose content is obtained from Amazon both when accessing the directory `checkout-shipping-select` and `checkout-address-edit`, and the distinction is just a means to record the different state of the interaction.

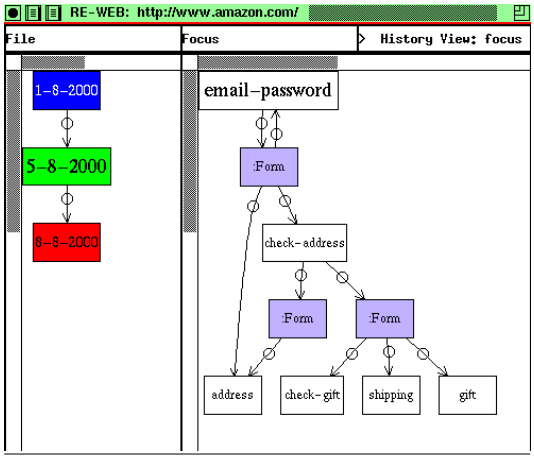

Figure 8: **ReWeb**'*s history view focused on page* `email-password`.

Figure 8 shows an example of focus in the history view provided by **ReWeb**. Page `email-password` was selected as focus and only a limited neighborhood of this page is displayed. Such a page contains a form, through which new customers can provide their email address, and later obtain a password, while returning customers can login by providing their password. New customers are asked for detailed address information, at the page `address`, while returning customers are presented with a list of addresses associated to them, among which the proper one can be selected (`check-address`). If none is appropriate, a new one can be entered (form leading to `address`). Once the address is defined, the `shipping` page can be reached or, if the item is a gift, a wrapping and a message can be specified for it. By navigating inside the history view and exploiting the focus facility of **ReWeb** other similar details about the organization of this site were obtained.

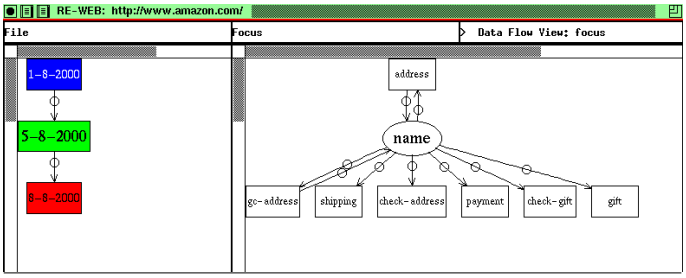

Figure 9: **ReWeb**'*s data flow view focused on* `name`.

Another view which was very helpful to understanding the site functioning is the data flow view. Figure 9 shows such a view focused on variable `name`. Only two pages, namely `address` and `gc-address`, can set the value of this variables. The former is traversed during check-out, after selecting the goods to buy, and the provided variable value is the full name of the customer. The latter is traversed when sending a gift certificate to a person via mail (a claim code will be attached for redeeming) and the inserted information identifies the person to whom the certificate has to be sent. The value of this variable is subsequently used by the other pages in Figure 9, among which `shipping` and `payment` for the related operations, and the two pages `address` and `gc-address` themselves, which redisplay such datum when looping under error conditions.

The first step in the testing activity for the site Amazon was the construction of its path expression. The paths represented in it required 398 loop and 192 alternative operators (see Table 1). 16 independent paths were computed from the path expression, associated with 16 test cases. Six of such paths are infeasible, because they contain mutually exclusive conditions. An example is the following sequence of 3 pages: `gc-sample1, gc-preorder, gc-order2`. Page `gc-sample1` provides a sample of email gift certificate, while `gc-order2` is the order page for paper gift certificates. The intermediate page `gc-preorder` always

Dynamic pages:	20
Forms:	19
Merged pages:	10
Unrolled pages:	3
Nodes:	39
Edges:	73
Path expression loops:	398
Path expression alternatives:	192
Independent paths:	16
Infeasible paths:	6
Data dependences:	328
Infeasible data dependences:	2
Data dep. to be covered for all uses:	5
New test cases for all uses:	2
Indep. paths between data dep.:	195
Indep. paths to be covered for def-use:	141
Total test cases:	18

Table 1: *Features of the site Amazon and testing data.*

generates a link to `gc-order1` in case of email gift certificate sample, making the path infeasible. For symmetry reasons, the sequence `gc-sample2, gc-preorder, gc-order1` cannot be traversed as well. The problem with these two paths can be solved by simply exchanging `gc-sample1` and `gc-sample2`. The resulting paths are feasible and do not alter the coverage offered by the whole set of paths. The other 4 cases of infeasible paths are similar. The final set of 16 adjusted paths contains only feasible paths, which are still independent and provide 100% of page and hyperlink testing. The all-paths testing criterion is achieved, restricted to independent paths.

After completing the 16 test cases with appropriate input values, they were executed by **TestWeb**. Their execution highlighted the robustness of the site, which could handle appropriately several different interactions, including those for the management of error conditions and incorrect data input. An anomalous behavior, which may be regarded either as a defect or an improvement area of this application, was however revealed by one of the test cases. The sequence of pages traversed is the following: `shipping shipping gift shipping shipping payment`. The associated interaction involves the modification of the quantities of items purchased (loop `shipping shipping`). Then a gift wrapping and a message are added, and finally the payment form is filled in, after a second modification of the quantities. If, for example, quantities are augmented, from 1 to 2, the 2 items are wrapped separately, and two messages are generated, in case of quantity modification *before* gift data specification. If quantities are modified *after* leaving the page `gift`, one wrapping of both items is assumed. Therefore, the user cannot rely on the commutativity of the two operations (wrapping a gift and modifying quantities) and has to learn non trivial paths leading to the desired effect. If, for example, the user wishes to edit the message of a wrapped package containing 2 books, without splitting it into two packages, the following (counter-intuitive) sequence of operations has to be followed: (1) edit the gift packaging and change the message; (2) change the quantities of the 2 resulting packages (with 1 item each) respectively to 0 and 2, and update the form.

A possible solution to this problem could be the introduction of an explicit representation of packages, so that quantities and wrapping can be updated in an intuitive way.

Path testing was then complemented with data flow testing. The **ReWeb**'s analyzer was run to obtain the data dependences, and its outcome consists of 328 definition-use pairs (see Table 1). Two of them are associated with infeasible paths, similar to those discussed above, and therefore were excluded from the set of dependences to be covered. The 16 independent paths resulting from the path expression provide the coverage of all but 5 data dependences, according to the all-uses criterion. To obtain a complete coverage, two new test cases had to be defined, thus leading the test suite to its final size (18 elements). Finally, the definition-use criterion was considered, with the restriction that only independent paths between definitions and uses be considered. The total number of such paths is 195, and only 54 of them are traversed when executing the 18 test cases. For the remaining 141 paths proper test cases should be defined, but the coverage achieved for the other criteria was considered satisfactory and suggested to stop test case production. Execution of the full test suite and manual verification of the outcomes required about 1 working day. Its re-execution for regression check can be obtained at no additional cost.

6 RELATED WORK

While substantial effort was devoted to investigating models and formalisms aimed at supporting the design of Web applications [4, 7, 9], only few works considered the problems related to web site evolution and maintenance [2, 12] and to testing [6]. The contribution of this paper to the available literature on Web applications is in the definition of a UML model which can be exploited for analysis and testing, and in the study of the analysis and testing techniques that can be effectively applied to Web applications.

Among the design models [4, 7, 9] which have been proposed in support to Web development, those conceived for the specification of the navigation and presentation structure of the site [4, 7] are closer to ours. Higher level abstractions (e.g., the entity relationship diagram of the RMM methodology [9]) are out of the scope of our current investigation.

The Web site model closer to ours is that proposed by Conallen [7]. 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 other side, we focused our model on the implementation of the site, which is the starting point for analysis, and on the navigational features of the site. An example of this difference is the representation of frames. In Conallen's model a frameset contains two aggregations: an aggregation of targets (frame names) and an aggregation of the pages initially loaded into the different frames. On the contrary, in our model only the former aggregation, between a page and the enclosed frames, is retained, and in turn each frame is linked to the page loaded initially. In such a representation, each link sequence is a navigation path (with possibly some links traversed automatically), while in Conallen's one some links are used to express properties (namely, the list of available targets).

Statistical testing is proposed in [6] for the automatic selection of the paths to be exercised in a Web application. The number of invalid links (ghost pages, discussed in Section 3) encountered along the test paths allows estimating the site reliability, i.e., probability that a user completes the navigation without errors. The proposed path selection method could be a useful complement to the white-box techniques described here, in that it accounts for the typical interactions with the site, rather than its structure and data flows.

We share with [2] and [12] a common view of Web site development and evolution. The existence of problems, in web site development, similar to those encountered in software before the advent of software engineering and the importance of the maintenance phase were recognized in [12], where the evolution of web sites is characterized by means of metrics. An experience of web site re-engineering is described in [2], where the target representation of the reverse engineering phase is based on RMM. The recovering process and the following re-design activity were conducted almost completely manually. On the contrary, since our analysis model is closer to the implementation than theirs, we can provide strong automatic support for its extraction.

7 CONCLUSION

The analysis and testing techniques proposed in this paper were successfully applied to several real world Web applications, among which Wordnet and Amazon.

ReWeb's views were useful to understand the site organization, both in terms of navigation paths (history view) and of variable usage (data flow view).

TestWeb's generator and executor of test cases were exploited to exercise the two sites up to a satisfactory level of coverage. An anomalous behavior of Amazon was revealed during the testing activity. It was highlighted by the sequence of operations to be performed during the execution of one of the automatically generated test cases.

The experimented analysis and testing techniques were extremely useful in the assessment of the site quality. They allowed a deep insight in the internal functioning of the Web applications, highlighting strengths and weaknesses. Future work will be devoted to the reduction of the manual activities still required (e.g., for state unrolling and merging).

REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers. Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, MA, 1985.

[2] G. Antoniol, G. Canfora, G. Casazza, and A. D. Lucia. Web site reengineering using rmm. In *Proc. of the International Workshop on Web Site Evolution*, pages 9–16, Zurich, Switzerland, March 2000.

[3] B. Beizer. *Software Testing Techniques, 2nd edition*. International Thomson Computer Press, 1990.

[4] M. Bichler and S. Nusser. Developing structured www-sites with w3dt. In *Proc. of WebNet*, San Francisco, California, USA, 1996.

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

[6] 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.

[7] J. Conallen. *Building Web Applications with UML*. Addison-Wesley Publishing Company, Reading, MA, 2000.

[8] D. Eichmann. Evolving an engineered web. In *Proc. of the International Workshop on Web Site Evolution*, Atlanta, GA, USA, October 1999.

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

[10] 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.

[11] F. Ricca and P. Tonella. Web site understanding and re-structuring with the reweb tool. *IEEE MultiMedia*, page (to appear), April-June 2001.

[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.