Annals of Software Engineering 14, 93–114, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

Testing Processes of Web Applications

FILIPPO RICCA and PAOLO TONELLA {ricca, tonella}@itc.it
ITC-irst, Istituto per la Ricerca Scientifica e Tecnologica, Via alla cascata, 38050 Povo (Trento), Italy

Abstract. Current practice in Web application development is based on the skills of the individual programmers and often does not apply the principles of software engineering. The increasing economic relevance and internal complexity of the new generation of Web applications require that proper quality standards are reached and that development is kept under control. It is therefore likely that the formalization of the process followed while developing these applications will be one of the major research topics.

In this paper we focus on Web application testing, a crucial phase when quality and reliability are a goal. Testing is considered in the wider context of the whole development process, for which an incremental/iterative model is devised. The processes behind the testing activities are analyzed considering the specificity of Web applications, for which the availability of a reference model is shown to be particularly important. The approach proposed in this paper covers the integration testing phase, which can take advantage of some features of Web applications (e.g., the http protocol employed), thus resulting in a higher level of automation with respect to traditional software.

The testing processes described in this paper are supported by the prototype research tool TestWeb. This tool exploits a reverse engineered UML (Unified Modeling Language) model of the Web application to generate and execute test cases, in order to satisfy the testing criteria selected by the user. The usage of this tool will be presented with reference to a real-world case study.

1. Introduction

Most Web applications have insofar been developed without following a formalized process model [Pressman 2000]. Requirements are not captured and the architecture and detailed design of the system are not considered. Developers quickly move to the implementation phase and deliver the system without testing it. No documentation is usually produced about the internal organization of the application. While this kind of practice was motivated by the characteristics of the first generation of Web sites, now things are changing and there is an increasing demand for better techniques, methodologies and processes.

The delivery of Web based systems developed according to ad hoc methods and with no consideration to the principles of software engineering was justified by the size of these applications, which were typically small, by the estimated lifetime, which was expected to be short, and by the difficulties to capture the user needs, both those to be satisfied by the initial release of the application and those that were considered likely to emerge in the future. Moreover, the first generation of Web sites were little more than fixed advertising material made publicly available: they had no particular relevance for the core business.

Soon companies realized that the Web is not just a way to promote their image, but can be exploited as a means to provide services. At the same time, several technologies have been developed to support the production of increasingly complex Web applications, which can be effectively exploited to support the main company business. Consequently, these applications have begun to be critical for the companies and to incorporate advanced functionalities. The next step in this evolution path will be to absorb some of the lessons learned during the history of software engineering, which started being considered when the practice of software development suffered similar problems as the current one for Web based systems.

While substantial effort was devoted to investigating models and formalisms aimed at supporting the design of Web applications [Bichler and Nusser 1996; Conallen 2000; Isakowitz *et al.* 1997], only few works considered the problems related to Web site maintenance [Antoniol *et al.* 2000; Warren *et al.* 1999] and testing [Chang and Hon 2000; Liu *et al.* 2000; MacIntosh and Strigel 2000; Miller 1998].

The existence of problems, in Web site development, similar to those encountered in software before the advent of software engineering are recognized in [Warren *et al.* 1999], where the evolution of Web sites is characterized by means of metrics.

Statistical testing is proposed in [Chang and Hon 2000] for the automatic selection of the paths to be exercised in a Web application. The number of invalid links (discussed in section 4) encountered along the test paths allows estimating the site reliability, i.e., probability that a user completes the navigation without errors. The CAPBAK/Web tool, explained in [Miller 1998], is a Web testing tool that supports functional testing and regression testing. In [Liu *et al.* 2000] an approach to data flow testing of Web applications is presented. In their approach, the structural test artifacts of a Web application are captured in a Web application Test Model where each component of a Web application is considered as an object. Data flow test cases for Web applications are derived from flow graphs in five levels according to the types of definition use chains for the variables of interest.

In this paper the testing processes of Web applications are considered in the larger context of Web application development, for which the availability of a reference model is central to several activities, such as understanding of the existing system, assessment of the required changes and implementation of the modifications. The testing phase is also expected to benefit from a structural model of the application, which is the starting point for white-box testing. In particular, we will focus on integration testing of Web applications, where the specific features of these applications, compared with traditional software, can be exploited to improve the level of automation and simplify the testing procedures. A support tool will be presented with reference to a real world case study, showing that a practical use of the proposed techniques is affordable.

The paper is organized as follows. Section 2 provides a candidate process model for the development of Web applications, based on their specific characteristics. Section 3 describes the reference model that will be used for testing. An overview on the testing processes of Web applications is given in section 4, while the next section focuses on integration testing and contains the authors' approach to white-box testing. Then, the tool that automates integration testing is described in section 6 and its usage aimed at testing an existing Web application chosen as a case study is presented in section 7. Section 8 contains some concluding remarks.

2. A process model for the development of Web applications

Some features of Web applications and of their development context are very peculiar, so that it is not possible to directly apply the same methods used in software engineering: they rather need some adaptations. Before considering a candidate process model for the development of Web applications, let us consider their main characteristics, which motivate the choice of an incremental/iterative model:

- Web applications have short delivery times.
- Web applications are subject to a tremendous pressure for change.
- Turn over of Web application developers is high.
- Complexity and criticality of Web applications are increasing.
- User needs evolve quickly.

An incremental/iterative process model (see figure 1) assumes that functionalities are delivered to the user incrementally, by traversing several times the different development phases. At each iteration an increment is considered (delta), which is sufficiently small to be completely implemented in a short time interval. After each iteration a complete Web application is delivered, which improves over the previous version because some functionality is added, changed or enhanced.

This process model allows short delivery times and can easily incorporate the change requests coming from the user or from the support technologies. New business opportunities can also push for major changes in the application, which can be achieved by iterating several times through a sequence of finer grain modifications.

While it limits the overhead that may delay the production of a working release, this process model contains all elements necessary to approach the higher complexity

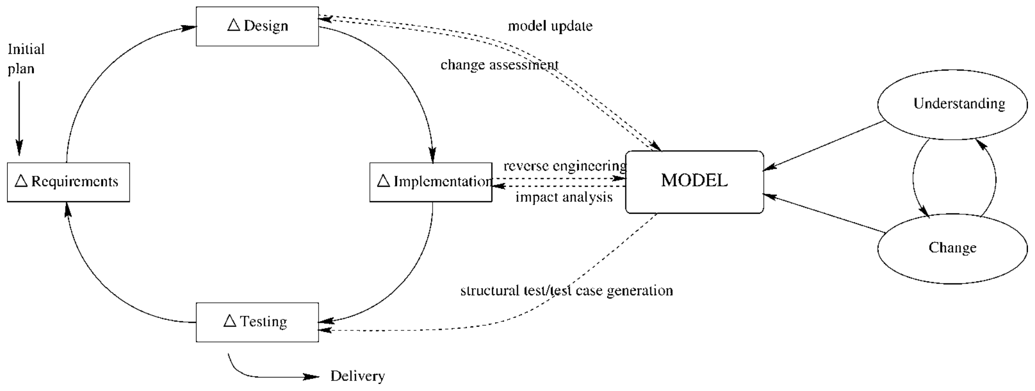


Figure 1. Incremental/iterative process model for the development of Web applications.

of new Web applications. It assumes the existence of requirements and design, so that a new developer is not left only with the code of the application, but can find useful documentation about it. It also attacks the problem of ensuring the needed reliability of these applications by formalizing a testing phase.

The most distinctive feature of the process sketched in figure 1 is the central role of the model of the Web application. Such a model is updated each time a design increment is introduced and is exploited for change assessment each time a requirement increment has to be mapped into the design. In fact, it allows estimating the portion of the application affected by the change, thus supporting the allocation of the change to a proper iteration and, if needed, the split of the change into smaller pieces.

When moving to the implementation, a model of the "as is" system can be retrieved from the code by means of reverse engineering, and can be used to evaluate the impact of the modification in terms of the code portions to be updated. Developers are expected to benefit from the availability of a reference model of the application, since their main activities during the implementation of an increment are the understanding of the existing system and of the portions to be modified. Then, they have to implement the changes and avoid undesirable ripple effects. These activities become particularly difficult in a situation of high turn over, where the persons who developed the system are no longer available and new programmers are charged of understanding and changing it. Reverse engineering and impact analysis can assist programmers in this development phase and a high level of automation can be reached if a proper model of Web applications is adopted.

Finally, the Web application is tested. To reach high levels of reliability, structural testing may be a useful complement to functional testing. Again, the availability of a reference model allows the automation of several activities, related to test case production, execution, to regression testing and to coverage measurement. In the next sections a reference model for Web applications is described, which can be semi-automatically recovered from the code and which can drive testing, becoming the starting point for most testing processes. The same model is a central entity also for all other development phases, as shown in figure 1.

3. A UML model of Web applications

Web applications exploit the navigation and interaction facilities of hypertextual HTML pages to provide and ask information to/from the user. As a consequence, the 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.

Among the design models [Bichler and Nusser 1996; Conallen 2000; Isakowitz *et al.* 1997] which have been proposed in support to Web development, those conceived for the specification of the navigation and presentation structure of the site [Bichler and Nusser 1996; Conallen 2000] are closer to ours. Higher level abstractions (e.g., the entity relationship diagram of the RMM methodology [Isakowitz *et al.* 1997]) are not adequate for our purposes.

The Web site model closer to ours is that proposed by Conallen [2000]. 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 versus 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, while we focused our model on the implementation of the site, which is the starting point for testing.

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. 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. 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 2 shows the meta model used to describe a generic Web application. It is given in the Unified Modeling Language (UML) [Booch *et al.* 1998]. The central entity

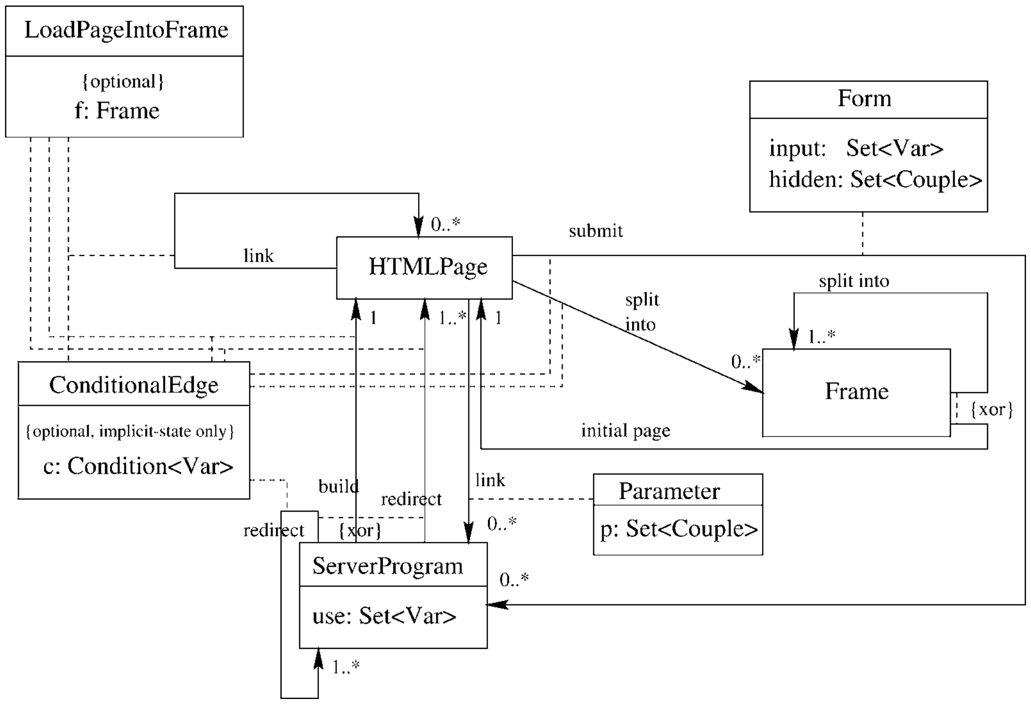


Figure 2. Meta model of a generic Web application. The model of a given site is an instantiation of it.

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). 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 in [Conallen 2000] and [Eichmann 1999]) and may depend on the information provided by the user through input fields. 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 value of a set of input variables, the attribute *use* of class *ServerProgram* contains them. A server side program may 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*⟩. 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.

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 is gathered by exploiting a *Form* and is passed to a server program, which processes it, through a *submit* link (see figure 2). A Web page can include any number of forms; accordingly, the cardinality of this link is arbitrary. Each 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).

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*, a *submit* 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*. Similarly, the action performed by a server program may depend on the input values, and therefore edges outgoing from *ServerProgram* may also be conditional.

The model described above is not adequate to support Web application testing in cases in which the same server program behaves differently according to the interaction state. To clarify this situation it is convenient to classify server programs into two categories:

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

Server programs in the first category exploit always the same mechanism to produce the output (either auto-redirect, redirect or build) and generate a dynamic page whose structure and links are fixed. In other words, there is no *ConditionalEdge* object attached to its links and to the links of its output page. The behavior of these pages 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 this case, *ConditionalEdge* objects are used to distinguish the different behaviors.

In presence of server programs with state-dependent behavior, the paths in the model can still be interpreted as navigation sessions, provided that all *ConditionalEdge* conditions are true. Paths in which conditions are inconsistent or unsatisfiable are infeasible and cannot be used for testing purposes. For this reason we consider a second version of the model in figure 2, which we call the *explicit-state* model, differing from the basic one, called *implicit-state* model, in that it unrolls server programs and dynamic pages with different behaviors into actually different entities, which are given a progressive identification number. In this way the page identity is not associated to a physical entity (page or server program), but is rather differentiated according to the behavior.

With reference to figure 2, additional constraints apply, if we restrict either to the implicit-state or to the explicit-state model. The target of a *redirect* link is multiple in the implicit-state model, where different behaviors are performed according to different interaction states, while exactly one target is allowed in the explicit-state model, in which the exclusive-or constraint is also valid (while it is not in the implicit-state model). The association class *ConditionalEdge* is present only in the implicit-state model, since the explicit-state model is constructed so as to make all input-dependent behaviors explicit and eliminate the related conditions.

In order to define a set of 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*, *Form*, *Parameter* or *ConditionalEdge* relation specifier. In the last three cases, labels are put respectively inside double square brackets, square brackets and normal brackets. 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.

Figure 3 shows an example of Web application for which both implicit-state and explicit-state models are given. The application consists of an initial static page *H* (note that the name is underlined to indicate that it is a UML object and not a class), from which the user can navigate to a server program *S* through a link associated with a parameter, `state`, which is assigned the constant value 1. *S* builds a dynamic HTML page, the content of which depends on the value of variable `state` which is received by *S*. In particular, with `state = 1`, *S* builds a page containing one form which collects the values of variables `x` and `y` and transmits a value of `state` equals to 2 as a hidden variable. This is represented in the implicit-state model (left of figure 3) as a *submit* link guarded by the condition `(state = 1)`. Such a link is generated inside page *D* only when *S* receives a value of `state` equals to 1. Then, the server program *S* is invoked for the second time, now with `state = 2`. The behavior in this situation is different from the previous one, and the output page contains two new

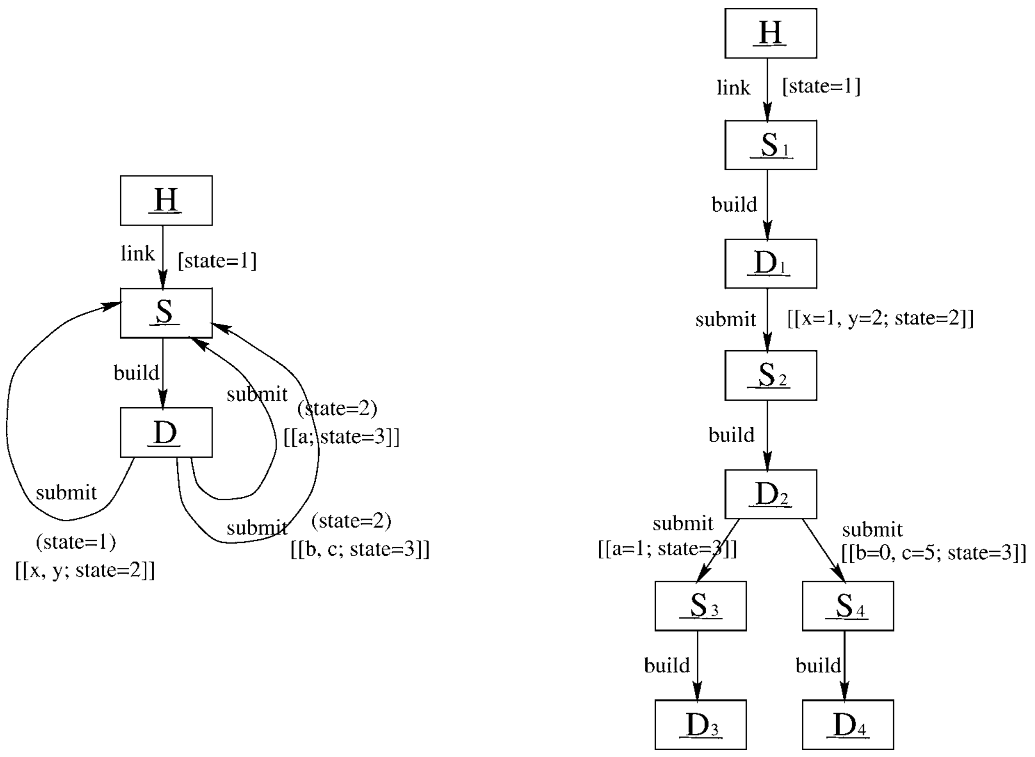


Figure 3. Example of Web application model with implicit (left) and explicit (right) state.

forms, respectively devoted to collecting the values of `a` and of `b`, `c`, while it does not contain the previous form. This is the reason for the two *submit* links guarded by the condition `(state = 2)`. Finally, the server program *S* is executed again, either by the first active form (gathering `a` as input) or from the second one (gathering `b` and `c`). The result of this execution is still different and the dynamic page *D* that is built now does not contain any form (`state` is equal to 3, and therefore, all conditions are false). Its content varies also in the two cases where either `a` or `b`, `c` are filled in by the user.

The explicit-state model of this example of Web application is provided on the right of figure 3. The server program *S* and the dynamic page *D* have been split into 4 pages, associated to the 4 different behaviors that may occur during an interaction, corresponding respectively to `state = 1`, `state = 2`, `state = 3` and `a` gathered, and `state = 3` and `b`, `c` gathered. No condition has to be attached to the edges of this model, since all condition-dependent behaviors have been separated explicitly. The values of input and hidden variables and of link parameters are sufficient to identify a particular navigation path, which is feasible by construction, since specific values (or more generally equivalence classes of values) are assigned to variables and parameters. All paths in the explicit-state model are feasible, while many paths in the implicit-state model are infeasible (e.g., every path going from *H* to *S* and then following any of the submit links with `state = 3`). The problem of determining the feasible paths is thus transformed into the problem of selecting a set of input values representative of all feasible paths. The underlying undecidability still remains, but it can be attacked more effectively, as discussed in section 5.

4. Web application testing

In [Conallen 2000] the three most common architectural styles for the design of Web applications are classified as follows:

Thin Web Client. Only the standard facilities of the client-side browser are exploited (including forms) and all business logic is executed on the server.

Thick Web Client. A portion of the business logic is delegated to the client, where applets, Java script, etc. can be used, but the communication protocol between client and server remains HTTP.

Web Delivery. Executable programs running on the client establish with the server a communication based on an ad hoc protocol different from HTTP.

Actually, a continuum exists between the extreme points of the above classification. At one end, the Web Delivery style becomes a traditional client-server application, defining its own communication protocol, and the Web is just the underlying infrastructure. At the other end, applications adhere completely to the HTTP protocol, the server provides HTML pages without executable parts and all information exchange is mediated by the standard facilities provided by HTML (e.g., forms, links with parameters).

The position that a Web application occupies in this continuum makes relevant differences in the way testing can be conducted. As noted above, the main distinctive feature of Web applications (those closer to the Thin Web Client) is the usage of the HTTP protocol for message exchange. This is also the main feature that makes testing these applications different from testing traditional programs. The HTTP protocol can be exploited to automate test execution, and the organization of the interaction according to the standard facilities of HTML simplifies test case generation. While departing from this style and introducing more elements that are typical of client-server applications, the benefits of HTTP are lost and the testing techniques become those employed for traditional client-server software.

In the next section we will describe an approach to integration testing of Web applications that exploits the HTTP protocol to increase the level of automation. Before considering our approach, integration testing is positioned with respect to the various aspects involved in Web application testing and the different activities conducted during the testing processes.

As with traditional software, testing can be performed to verify either the functional or the nonfunctional requirements. Among the non functional requirements that are typical of Web applications, performance is maybe the most important one. The performance of a Web application can be assessed in terms of the response time (the time necessary to obtain the required page on the browser) and of the number of clients that can be simultaneously served (load testing). Assessment of the minimal computational resources required on the server and on the client side is also an important non functional issue.

The test of the functional requirements can be conducted by considering the Web application as a black-box. In this case it is exercised by partitioning legal inputs into equivalence classes and one or more test cases for each class are defined. Output pages obtained by navigating the application and providing such inputs are compared with the result expected from that interaction according to the system specifications. Each deviation from the expected behavior will be called an *error* in the following. Boundary values are inputs that deserve special attention and therefore specific test cases have to be defined for them. Alternatively, the Web application is considered as a white-box and testing exploits knowledge about the internal functions of the application. In particular, server and client side code, HTML pages and messages exchanged via HTTP are assumed to be known and can drive the definition of the test cases. A widely used example of this kind of test is called *link validation*. It aims at verifying that every link in every page generated by the application is a valid link, i.e., no error is reported when it is traversed. In the next section other white-box testing techniques are described, based on the notion of coverage. As with traditional software, it is possible to define the internal features of the application that must be covered by at least one test case before delivering the application. Novel coverage criteria will be defined, tailored on the characteristics of Web applications and based on the UML model that was introduced for them in the previous section.

Different *levels* of testing can be conducted on a Web application, similarly to a normal software system:

Unit Test. Singular components are tested and stubs/drivers replace missing parts. For example, a server script is invoked by a driver which simulates the browser and receives the HTML page as the resulting output. Another example is a Java script program which validates fields in a client side page and sends a request to a fictitious server simulated by a stub.

Integration Test. Pages are composed and integrated with server programs. The tester can now navigate from page to page and requests can be passed from the browser to the Web server via HTTP. This testing phase is strongly based on the protocol exploited, and, when this is HTTP, it can be supported by ad hoc techniques (see next section).

System Test. The system is validated as a whole in an environment as similar as possible to the real, target environment.

Acceptance Test. The customer installs and runs the application in its own environment.

Regression Test. During evolution, the preservation of previous functionalities is checked by rerunning the test cases defined for them. Exploitation of the HTTP protocol can positively affect also this testing activity.

Among the various testing levels, those for which the difference with respect to traditional software is more relevant are Integration Test and Regression Test, in that these levels heavily depend on the communication protocol used for client-server data exchange. Web applications have their own protocol (HTTP) and the testing methods used can take advantage of it. For the other levels, the methods usually adopted for traditional software systems can still be used in the context of Web applications.

During Integration/Regression Test of a *traditional* application the interaction with the user is simulated by generating the graphical events that trigger the computation from the application interface. One of the main methods used to obtain this result is based on capture/replay tools, which record the interactions that a user has with the graphical interface and repeat them during regression test (figure 4, top). Although this can still be done with Web applications, and is recommended for applications closer to the Web Delivery than to the Thin Web Client, availability of the communication protocol HTTP, which separates user interactions with the browser from server side computations, offers the opportunity to define ad hoc techniques. HTTP messages can replace the graphical events generated by a capture/replay tool during test case execution (figure 4, bottom).

5. An approach to integration testing of Web applications

Differently from traditional software, where white-box testing is typically applied only at the unit level, for Web applications it is possible to exploit the knowledge on the organization of the system into server programs, dynamic pages, forms, links, etc. to define integration testing criteria. The starting point for the proposed approach is the

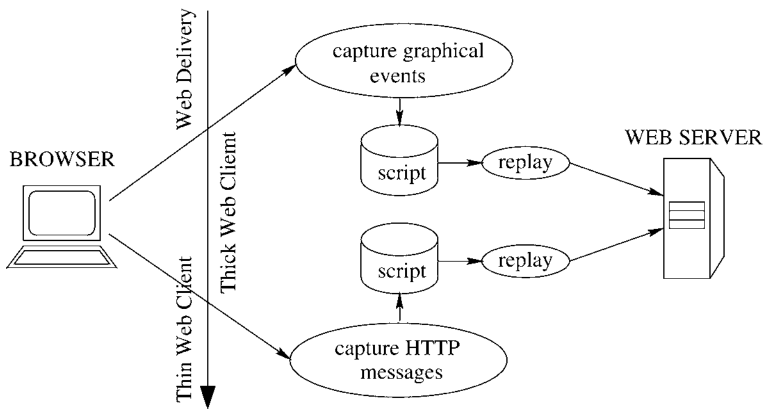


Figure 4. Capture/replay of graphical events versus HTTP messages.

UML model of the Web application. The determination of this model is not a trivial task. First of all, it should be noted that recovering the implicit-state model is a much harder task than recovering the explicit-state one. In fact, in the implicit-state model the behavior of each server program and the content of dynamic pages depend on conditions, associated with the outgoing edges, that have to be specified by the user (it is not possible recovering them) and must be complete and consistent. For large sites the task of their specification is not affordable in practice. Moreover, when the UML model is exploited for coverage assessment and test case generation, it is not easy (in general it is undecidable) determining whether a selected path is feasible (i.e., there exists an input such that all edge conditions in the path evaluate to true). On the contrary, recovering the explicit-state model is more practical and all the paths in this model are feasible by construction.

The operation performed on a Web application to extract its explicit-state model is called *state unrolling*. It consists of replicating each server program and dynamic page that is accessed in a new state, i.e., with inputs belonging to an equivalence class or with hidden state variables or parameters associated to a state never encountered before. This operation cannot be fully automated. The intervention of the user consists of the specification of a sample input for every different behavior of the application. Then, remaining steps of model extraction can be automated. In fact, when a server program is executed and a dynamic page is retrieved, it is possible to automatically check if the related state is new or it was encountered previously, thus deciding whether to unroll it or not.

The model obtained through the state unrolling operation is a conservative explicit-state model, which may include more expansions than strictly required. In fact, it may happen that a given server program P_1 behaves differently in the two states S_1 and S_2, while the server program P_2 has exactly the same behavior in the two states. According to the unrolling procedure described above P_2 is split into two because it is accessed in two distinct states. A second operation, called *state merging*, is applied to unify the server programs and dynamic pages that in different states behave the same. In general, assessing the behavioral equivalence of two software modules is undecidable, but in our case some heuristics can be applied for the simplest cases. For example, when the resulting dynamic pages are exactly the same, it is possible to unify them, as well as the unrolled server programs that generated them, since in the two different states they give exactly the same result. More complex cases of state merging are associated to the generation of dynamic pages with a common structure but different contents and to the generation of dynamic pages in which the structure has some regularity that suggests a common behavior. User intervention in these cases is unavoidable.

The recovered model of the Web application represents its internal structure, which can be 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. Some white box testing criteria, derived from those available for traditional software [Beizer 1990], are:

- *Page testing*: every page in the Web application is visited at least once in some test case.
- *Hyperlink testing*: every hyperlink from every page 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 Web application is traversed in some test case at least once.

Flow analyses can be employed to determine the *data dependences* required by some testing criteria. 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) along a definition-clear path, there is a data dependence between defining node and user node.

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 weaker 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 depends on the user input are good candidates for ending meaningful computations.

5.1. *Test case generation*

Satisfaction of any of the white box testing criteria involves selecting a set of paths in the Web application graph and providing input values. Since in the explicit-state model path selection is independent from input values, it can be automated. Moreover, since in the explicit-state model edges are associated with the actual inputs used to recover the model, given a path of interest it is easy to determine the inputs that allow traversing it. They can be just collected from the path edges. Note that the same operation in the implicit-state model is much harder, in that it involves determining inputs which satisfy the path condition, i.e., the conjunction of the conditions in the path edges.

We propose a test case generation technique based on the computation of the path expression [Beizer 1990] 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 implicit-state and the explicit-state models of the example in figure 3 are:

$$\mathit{link}\ (\mathit{build}\ (\mathit{submit}_1 + \mathit{submit}_2 + \mathit{submit}_3))^*$$
$$\mathit{link}\ \mathit{build}_1\ \mathit{submit}_1\ \mathit{build}_2\ (\mathit{submit}_2\ \mathit{build}_3 + \mathit{submit}_3\ \mathit{build}_4),$$

where edges have been indexed with a numeric identifier when necessary to distinguish them.

Computation of the path expression for a site can be performed by means of the Node-Reduction algorithm described in [Beizer 1990]. 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 reiterate 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, can be met only if restricted to independent or k-limited paths.

Once paths for the test cases are generated from the path expression, the related input values can be obtained from the attributes of the edges in the explicit-state model used for test case generation. 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 reexecuted 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.

6. The tool TestWeb

The UML model of the Web application to be analyzed is generated by the other tool we are developing, *ReWeb* [Ricca and Tonella 2000; Ricca and Tonella 2001] (see figure 5). Among the others, *ReWeb* contains a module called *Spider*, which 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

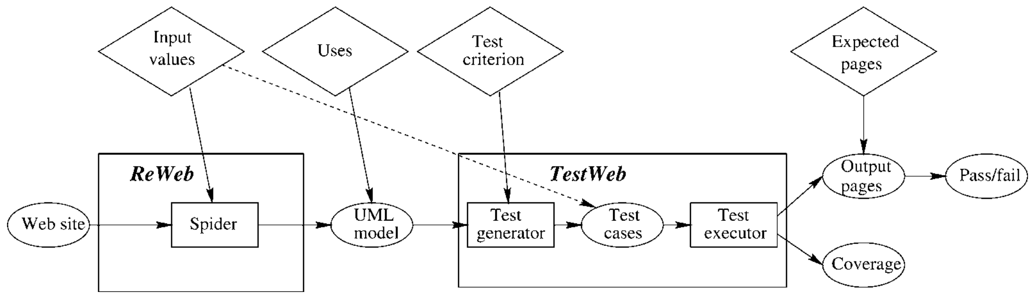


Figure 5. *TestWeb*'s modules and their dependencies on *ReWeb* and the user input.

the web site host are not considered. The pages of a site are obtained by sending the associated requests to the Web server. For dynamic pages, the user has to specify the set of inputs that the Spider will provide to the server programs generating them. To obtain an explicit-state model representing all behaviors of the server programs, more inputs can be provided by the user, associated with the same dynamic page. The resulting UML model is an explicit-state model in which server programs and dynamic pages have been unrolled according to all different conditions specified by the user in terms of different input values. The operation of state merging is currently performed manually, but some strategies for its partial automation can be implemented. They have been sketched in the previous section. If no input is specified for a given dynamic page, the Spider will not expand the model beyond the related server program. This feature of *ReWeb* is useful to analyze the functioning of portions of a Web application. Partitioning the tested functionalities by cutting the model at given nodes is important, especially for large Web applications. The user has also to provide the set of used variables, *use*, for each dynamic page whose content depends on some input value. Such a property is necessary for data-flow testing.

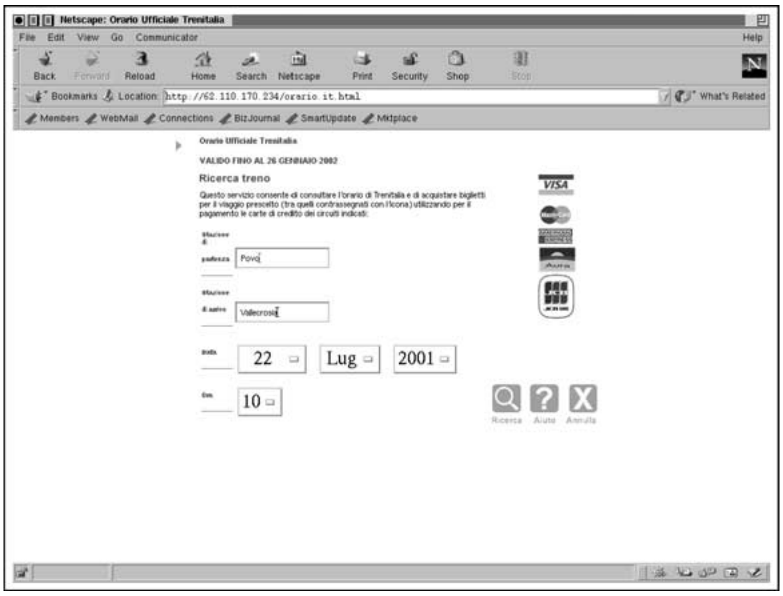
As depicted in figure 5, *TestWeb* contains a test case generation engine (Test generator), able to determine the path expression from the model of a Web application, and to generate test cases from it, provided that a test criterion is specified. Generated test cases are sequences of URLs which, once executed, grant the coverage of the selected criterion. Input values in each URL sequence are those specified for the Spider. Such inputs can be marked with the unrolling index of the server program and dynamic page they allow to download. In this way, during testing it is possible to obtain exactly the dynamic page with the structure required to follow a given path. In fact, a page with the needed structure was obtained by the Spider by providing the same input values. Such a possibility solves one of the major problems in testing traditional software: selecting the inputs to traverse a path of interest. Testing Web applications is simpler because branch selection can be forced, being associated to the user navigation, which is an external input. Moreover, the existence of the hyperlink to be followed is granted if the dynamic page is obtained under the same conditions in which it was downloaded. This can be achieved by exploiting the same inputs that are used by the Spider.

TestWeb's Test executor can now provide the URL request sequence of each test case to the Web server, attaching proper inputs to each form. The output pages produced by the server, 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 an 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.

7. A case study

Several Web applications are periodically downloaded, analyzed and tested by *ReWeb* and *TestWeb*. In [Ricca and Tonella 2000] examples of static analyses of real Web applications can be found, while in [Ricca and Tonella 2001] testing techniques are successfully applied to two complex and well known Web applications: `Wordnet` and `Amazon`. In the following the analysis and testing of a Web application that provides Italian railway timetables will be presented and discussed. The richness of its dynamic structure makes it an interesting case study for testing. Approximate server side information, necessary for analysis and testing, was deduced through interaction sessions and examination of the output page content.

FS-online (`http://orario.fs-on-line.com/orario.it.html` or `http://62.110.170.234/orario.it.html`) is a portion of a very complex Italian web site (`http://www.trenitalia.it/`) providing national rail travel information, devoted to timetables and tickets. The FS-online database can be accessed directly from the initial Web page `orario.it.html` (see figure 6) of the site by means of a `form` collecting user inputs such as: `departure station`, `arrival station`, `date of travel` and `departure time`. After submitting the form, filled-in with correct inputs, the dynamic page `fsbin/fsquery` appears (see figure 7), showing a list of possible solutions to reach the chosen station (two solutions in the example). The user can select one of them, to see detailed information, buy the ticket on-line (only if the related option is active) or return to the initial page. In the case of selection of a solution, the page `fsbin/fsquery` is reloaded, but now it contains details on the chosen solution, such as number of train changes,


Netscape: Orario Ufficiale Trenitalia
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security Shop Stop
Bookmarks Location: http://62.110.170.234/orario.it.html What's Related
Members WebMail Connections BizJournal SmartUpdate Mktplace
Ricerca Aiuto Annulla

Orario Ufficiale Trenitalia

VALIDO FINO AL 26 GENNAIO 2002

Ricerca treno

Questo servizio consente di consultare l'orario di Trenitalia e di acquistare biglietti per il viaggio prescelto (tra quelli contrassegnati con l'icona) utilizzando per il pagamento le carte di credito dei circuiti indicati:

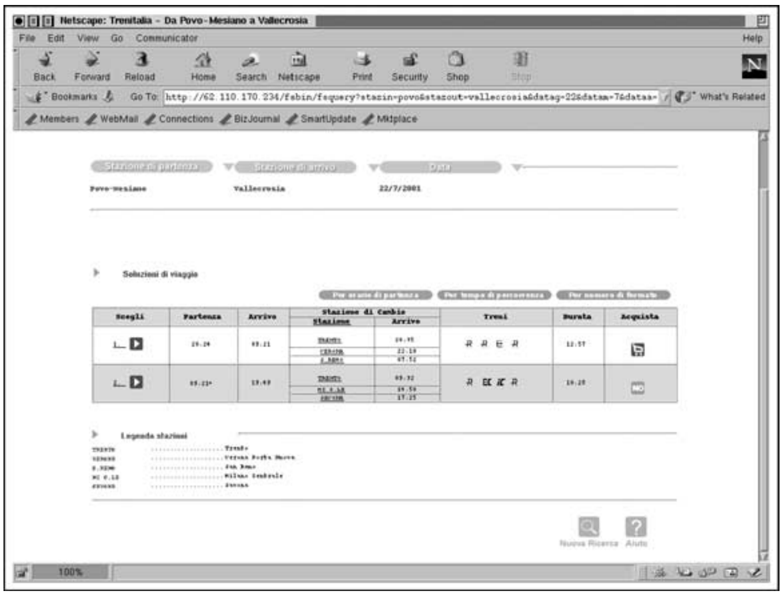
Stazione di partenza: Povo

Stazione di arrivo: Vallecrosia

Data: 22 Lug 2001

Ora: 10

Figure 6. The initial Web page `orario.it.html`


Netscape: Trenitalia – Da Povo-Mesiano a Vallecrosia

Go To: http://62.110.170.234/fsbin/fsquery?stazin=povo&stazout=vallecrosia&datag=22&datam=7&dataa=

Stazione di partenza	Stazione di arrivo	Data
Povo-Mesiano	Vallecrosia	22/7/2001

Soluzioni di viaggio

Scegli	Partenza	Arrivo	Stazione di Cambio		Treni	Durata	Acquista
			Stazione	Arrivo			
1	19.24	08.21	TRENTO VERONA S.REMO	19.55 22.10 07.51	R R E R	12.57	
1	05.21*	19.49	TRENTO MI C.LE SAVONA	05.32 10.50 17.25	R EC IC R	14.28	NO

Legenda stazioni

TRENTO	Trento
VERONA	Verona Porta Nuova
S.REMO	San Remo
MI C.LE	Milano Centrale
SAVONA	Savona

Figure 7. The dynamic page `fsbin/fsquery`.

type of trains and services. This page has also a `form`, containing only hidden variables, whose values are transmitted to the dynamic page `fsbin/prezzi1`. The page `fsbin/prezzi1` collects other inputs (e.g., number of passengers, seat class, etc.) and transmits them, by means of another `form`, to the server program `fsbin/prezzi2`. Finally, `fsbin/prezzi2` computes the price of the chosen solution and displays it.

When it is possible buying the ticket on-line, a clickable red cart icon (see figure 7, last column) is shown in the column "acquista" of the page `fsbin/fsquery`. This link with parameters transmits the parameter values to the server program `fsbin/fsquery` that, this time, redirects them to the server program `fsbin/prezzi1`. The resulting page collects some inputs (e.g., number of passengers, seat class, etc.) and transmits them to the server program `fsbin/preris2` (not to `fsbin/prezzi2` as before) that computes the price of the chosen solution. Other links and forms, starting from `fsbin/preris2` (not considered here), for booking and buying tickets are also available.

An interesting feature of this site is the behavior of the server programs `fsbin/fsquery` and `fsbin/prezzi1`: the content of the dynamic pages they build depends on the *state* of the user interaction. In the case of `fsbin/fsquery` a model with $n + 3$ states is assumed, where state 0 is associated with the page showing the list of solutions (in figure 7, `fsbin/fsquery` is in state 0), states from 1 to n are associated with the pages showing details of the chosen solutions. In the state REDIRECT, `fsbin/fsquery` redirects the collected inputs to the page `fsbin/prezzi1`, while in the state ERR the page displays error messages. FS-online exploits a general mech-

Table 1
Behaviors of the server program `fsbin/fsquery` with respect to variables `det` and `acq`. `OK` is a label indicating a valid sequence of input values, while `NotOK` is used for incorrect inputs.

State	Behavior	Conditions to reach the state
0	displays table of solutions	input = OK
$1, \ldots, n$	displays details of a chosen (i-th, where $1 \leqslant i \leqslant n$) solution	(det = i) $\wedge$ input = OK
REDIRECT	redirects to `fsbin/prezzi1`	(det = i) $\wedge$ acq = 1 $\wedge$ input = OK
ERR	displays an error message	input = NotOK

Table 2
Features of the site FS-online and testing data.

Static pages:	1
Dynamic pages:	4
Server programs:	4
Forms:	4
Merged states:	1
Unrolled states:	2
Unrolled dynamic pages:	8
Unrolled server programs:	8
Unrolled forms:	8

Nodes:	11
Edges:	22
Path expression loops:	30
Path expression alternatives:	33
Independent paths:	13
Total test cases:	4

anism to implement the concepts of state and state transitions. Several hidden variables (`det`, `acq`, `tges`) are used to set the state and pass it to the server, via form submission or link with parameters, when a state transition has to occur. The variable `det` identifies the number of the solution chosen by the user, `acq` becomes equal to 1 only if the red cart icon is selected and `tges` is 1 if, for the given solution, buying the ticket is activated (it is 0 otherwise). Table 1 clarifies the relation between the variables `det` and `acq` and the behavior of the page `fsbin/fsquery`.

The variable `tges` is used by the page `fsbin/prezzi1`. If `tges` is equal to 1 the page `fsbin/prezzi1` collects the inputs and transmits them to the server program `fsbin/preris2`, while if `tges` is equal to 0 `fsbin/prezzi1` transmits the inputs to the dynamic page `fsbin/prezzi2`.

The portion of FS-online devoted to handling timetables and computing ticket prices, as recovered by *ReWeb* with the inputs indicated in figure 6, is shown in figure 8. In this explicit-state model, the server programs and the dynamic pages they build are collapsed into single nodes for space reasons. Only two server programs are represented explicitly in the model (nodes with grey background). They are associated to a redirection (dotted edge from `fsbin/fsquery` to `fsbin/prezzi1`), instead of the construction of a dynamic page.

As indicated in table 2, after unrolling this portion of FS-online includes 1 static page, the initial page `orario.it.html`, 8 forms and 8 dynamic pages. *ReWeb* performed two state unrollings to explicitly represent the different contents displayed under

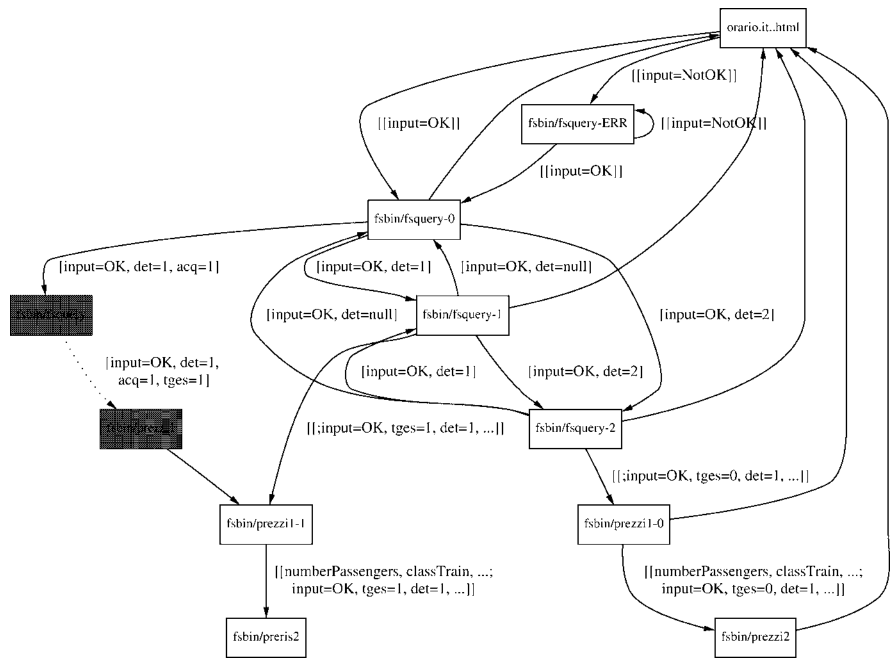


Figure 8. Portion of explicit-state model of the Web application FS-online.

different conditions by the server programs `fsbin/fsquery` and `fsbin/prezzi1` (unrolled dynamic pages are indicated with the name followed by `-id`, where `id` is an identifier). A manual operation of state merging was necessary to unify the dynamic pages `fsbin/prezzi1-1` and `fsbin/prezzi1-2` produced by the server program `fsbin/prezzi1` with different inputs. Since they have the same content, they have been collapsed into `fsbin/prezzi1-1`. An interesting feature of this site, appearing after state unrolling, is that the dynamic pages produced by the server program `fsbin/fsquery` (`fsbin/fsquery-0` ... `fsbin/fsquery-n`) are organized according to a well known navigational pattern: the *guided tour*. After choosing a solution, the user can visualize the previous or next one by simply navigating along the tour (note the use of the variable `det` in figure 8, recording the position in the page sequence).

The first step of the testing activity for the site FS-online was the construction of its path expression. The paths represented in it require 30 loop and 33 alternative operators (see table 2). Four independent feasible (by construction) paths were computed by *TestWeb* from the path expression, associated with 4 test cases (see figure 9 for an example of test case). This set of 4 test cases provides 100% coverage of page and hyperlink testing criteria. Execution of the test cases highlighted the robustness of the site, which could handle appropriately several different interactions, including those for the management of error conditions and incorrect input data.

An anomalous behavior, which may be regarded either as a defect or an improvement area of the server program `fsbin/fsquery` used in the FS-online application,

```
GET fsbin/fsquery0 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery2 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery0 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery1 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery2 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery1 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery0 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery2 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery0 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery0 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery2 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery0 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery1 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery2 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery1 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery0 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery2 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/fsquery0 stazin=Povo,stazout=Vallecrosia,datag=22,datam=7,dataa=2001,time=12
GET fsbin/prezzi1 stazin=Povo,stazout=Vallecrosia,data=15/7/2001,time=12,det=1,tges=1,acq=1, ...
POST fsbin/preris2 npostiordr=1, classe=2, ... ,stazin=Povo,det=1,tges=1,acq=1, ...
```

Figure 9. Test case produced by *TestWeb*.

was evidenced. The solutions produced by the server program `fsbin/fsquery` are sensitive to the departure time, in situations where they should not be. The program searches the FS-online database for all possible travel solutions within 24 hours from the departure time inserted by the user. If, for example, the user inserts the following data: departure station: `Povo`, arrival station: `Vallecrosia`, date of travel: `22/7/2001`, and departure time: `10:00`, the Web application gives two solutions (see figure 7). The first departure time is `20:24` while the second one is `9:21` of the next day (the asterisk, near the departure time, indicates that the solution is for the next day). If the user inserts the same departure station, the same arrival station and the same date, but changes the departure time, for example inserting `7:00`, the travel solutions are still two but the second one is anticipated at `6:46` to respect the constraint that the solution is within 24 hours from the departure time. It is interesting to note that the arrival time is the same in the two cases, i.e. `19:49` (the second and third trains are the same; only the first train changes), with the drawback that in the second case the travel time is approximately 3 hours greater than in the first. These hours are wasted in the station of the first connection, Trento.

8. Conclusion

Web applications require a new perspective on development and testing practices. While many activities are similar to traditional software engineering, others, as for example the testing phase, need adaptation to the specific features of these applications.

In this paper, the testing processes of Web applications are considered in the larger context of Web application development, for which an incremental/iterative, model centered process has been described. The proposed Web application model is exploited when structural testing is performed.

The main advantages offered by the Web testing techniques described in this paper are the possibility to automatically generate test cases (inputs included) from the path expression, and to automatically execute them (by sending HTTP messages to the Web server). Manual validation of the outputs is still required.

The testing processes described in this paper are supported by the *ReWeb* and *TestWeb* tools. The usage of these tools has been presented with reference to a real-world case study, the FS-online Web application. During the testing activity of FS-online, execution of the test cases highlighted an undesired, anomalous behavior, that could have gone unnoticed in a less formalized testing environment.

Manual intervention is necessary for the extraction of the UML model of a Web application. In fact, the inputs necessary to access and unroll dynamic pages have to be specified in a file. Their completeness with respect to the possible behaviors of the Web application is a prerequisite for the construction of an accurate model. Our future work will be devoted to investigating techniques for the (partial) automation of this critical activity in the process presented in this paper.

References

Antoniol, G., G. Canfora, G. Casazza, and A. D. Lucia (2000), "Web Site Reengineering using RMM," In *Proceedings of the International Workshop on Web Site Evolution*, Zürich, Switzerland, pp. 9–16.

Beizer, B. (1990), *Software Testing Techniques*, 2nd ed., International Thomson Computer Press.

Bichler, M. and S. Nusser (1996), "Developing Structured WWW-Sites with W3DT," In *Proceedings of WebNet*, San Francisco, CA.

Booch, G., J. Rumbaugh, and I. Jacobson (1998), *The Unified Modeling Language – User Guide*, Addison-Wesley, Reading, MA.

Chang, W.K. and S.K. Hon (2000), "A Systematic Framework for Ensuring Link Validity under Web Browsing Environments," In *Proceedings of the 13th International Software/Internet Quality Week*, San Francisco, CA.

Conallen, J. (2000), *Building Web Applications with UML*, Addison-Wesley, Reading, MA.

Eichmann, D. (1999), "Evolving an Engineered Web," In *Proceedings of the International Workshop on Web Site Evolution*, Atlanta, GA.

Isakowitz, T., A. Kamis, and M. Koufar (1997), "Extending RMM: Russian Dolls and Hypertext," In *Proceedings of HICSS-30*.

Liu, C.-H., D.C. Kung, P. Hsia, and C.-T. Hsu (2000), "Structural Testing of Web Applications," In *Proceedings of ISSRE 2000, International Symposium on Software Reliability Engineering*, San Jose, CA, pp. 84–96.

MacIntosh, M.A. and M.W. Strigel (2000), "'The Living Creature' – Testing Web Applications," In *Proceedings of QW 2000, 3th International Software/Internet Quality Week*, San Francisco, CA.

Miller, E. (1998), "The Web Site Quality Challenge. Companion Paper: 'WebSite Testing'," In *Proceedings of QW'98, 11th Annual International Software Quality Week*, San Francisco, CA.

Pressman, R.S. (2000), "What a Tangled Web We Weave," *IEEE Software 17*, 1, 18–21.

Ricca, F. and P. Tonella (2000), "Web Site Analysis: Structure and Evolution," In *Proceedings of the International Conference on Software Maintenance*, San Jose, CA, pp. 76–86.

Ricca, F. and P. Tonella (2001), "Analysis and Testing of Web Applications," In *Proceedings of ICSE 2001, International Conference on Software Engineering*, Toronto, ON, Canada, May 12–19, pp. 25–34.

Warren, P., C. Boldyreff, and M. Munro (1999), "The Evolution of Websites," In *Proceedings of the International Workshop on Program Comprehension*, Pittsburgh, PA, pp. 178–185.