VeriWeb: Automatically Testing Dynamic Web Sites

Michael Benedikt   Juliana Freire   Patrice Godefroid

Bell Laboratories, Lucent Technologies

{benedikt,juliana,god}@research.bell-labs.com

Abstract

Web sites are becoming increasingly complex as more and more services and information are made available over the Internet and intranets. At the same time, the correct behavior of sites has become crucial to the success of businesses and organizations and thus should be tested *thoroughly* and *frequently*. Although traditional software testing is already a notoriously hard, time-consuming and expensive process, testing Web sites presents even greater challenges: Web interfaces are very dynamic; the environment of Web applications is more complex than that of typical monolithic or client-server applications; Web applications, most notably e-commerce sites, have a large number of users who have no training on how to use the application and hence are more likely to exercise it in unpredictable ways. Existing testing tools for automating the process of testing dynamic Web sites require the specification of test scenarios, which results in limited test coverage. In this paper, we present an overview of VeriWeb, a tool for automatically discovering and systematically exploring Web-site execution paths that can be followed by a user in a Web application. Unlike traditional crawlers which are limited to the exploration of static links, VeriWeb can navigate automatically through dynamic components of Web sites, including form submissions and execution of client-side script.

Keywords: automated testing, automatic form filling, dynamic content, electronic commerce, model checking, spiders, Web browser, smart bookmarks.

1 Introduction

As more and more services and information are made available over the Internet and intranets, Web sites have become extraordinarily complex, while their correctness is often crucial to the success of businesses and organizations. Although traditional software testing is already a notoriously hard, time-consuming and expensive process, Web-site testing presents even greater challenges. Indeed, unlike traditional GUIs, *Web interfaces are very dynamic*. Web pages are modified frequently: adding links, making user-specific customizations (some pages change every time they are requested), adding new features, or changing the site look-and-feel. Moreover, *the environment of Web applications is more complex* than that of typical monolithic or client-server applications – Web applications interact with many components, such as CGI scripts, browsers, backend databases, proxy servers, etc., which may increase the risk of interoperability issues. Furthermore, many Web applications have a large number of users with no training on how to use the application – they are likely to exercise it in *unpredictable* ways. Therefore, Web sites that are critical to business operations of an organization should be tested *thoroughly* and *frequently*.

Static components of Web sites can be automatically tested by existing spider-like programs (*e.g.,* [22, 11]), which follow recursively all possible static links from a Web page in search of errors such as broken links, misspellings, and HTML-conformance violations. For automatically testing dynamic components, which include execution of client-side scripts and form interactions, the only class of tools currently available are "capture-replay" tools that record specific user-defined testing scenarios, and then generate scripts (sequences of browsers actions) that can be run on browsers (*e.g.,* [13, 7, 14]) in order to automatically replay the recorded scenarios for regression testing. Because Web sites are increasingly complex, manually recording a sample set of testing scenarios with a capture-replay tool can be very time-consuming. Due to the impossibility of recording more than a few possible paths, site coverage using capture-replay tools ends up being typically limited to a small portion of the Web-site functionality. Moreover, although state-of-the-art capture-replay tools do provide some level of abstraction when recording user actions and increased portability of test scenarios (for instance, by recording general browser actions instead of mouse actions on specific screen coordinates), changes in the structure of a Web site may prevent previously recorded test scenarios from being replayed, and hence may require re-generating and re-recording a new set of test scenarios from scratch.

In this paper, we present an overview of VeriWeb, a tool for automatically discovering and systematically exploring Web-site execution paths that can be followed by a user in a Web application. Unlike traditional spiders which are limited to the exploration of static links, VeriWeb can navigate automatically through dynamic components of Web sites, including form submissions and execution of client-side scripts. Whenever examining a new Web page, the system determines *all* possible actions a user might perform – be it via a button with a JavaScript[1] handler or via form submission – and can execute them in a systematic way. When forms are encountered, VeriWeb uses *SmartProfiles* to identify values that should be input to forms. Loosely speaking, user-specified SmartProfiles represent sets of attribute-value pairs that are used to automatically populate forms. An important feature of SmartProfiles is that they are specified independently of the structure of the Web site being tested.

Systematic Web-site exploration is performed under the control of VeriSoft [19], a previously existing tool for systematically exploring the state spaces of concurrent/reactive software systems. (This type of systematic state-space exploration is often referred to as *model checking* in the software testing and verification literature.) Whenever a new Web page is reached and a new set of possible actions is determined, VeriSoft records this set of actions and executes one of them. The process is recursively repeated on the next page until some depth in the state space (*i.e.,* number of successive actions) is reached. At that point, VeriSoft re-initializes the state of the Web-site and starts executing a new scenario from that initial state. By repeating this process, all possible execution paths of a Web application up to the given depth can eventually be exercised and checked. Since the state space of a Web site can be huge in practice, VeriWeb supports various techniques and heuristics to limit the size of the part of the state space being searched. During Web-site ex-

[1] In this paper, we use the terms JavaScript and ECMAscript interchangeably.

ploration, VeriWeb allows checking for many different types of errors, from errors in an isolated Web page (*e.g.,* the presence of a string pattern for an error, conformance to accessibility guidelines), to errors that involve a navigation path (*e.g.,* constraints on length of the deepest paths in the site). An important feature of VeriWeb is that different error-checking components can be easily plugged into the system. Whenever an error is detected, the execution path leading to the error is saved and can be replayed interactively for debugging purposes.

VeriWeb thus combines the flexible navigation capabilities of capture-replay tools with the high level of automation provided by Web crawlers. Note that the functionality provided by VeriWeb is complementary to that of capture-replay tools – whereas capture-replay tools are useful for testing the functionality of a few critical paths in the application, VeriWeb explores a large number of execution paths that may result from unpredictable user behavior.

The purpose of this paper is to introduce the main philosophy and design principles behind VeriWeb. Specifically, we present an overview of VeriWeb that focuses mostly on high-level design issues. The main topics discussed are:

- we describe an extensible platform for performing standard correctness checks and doing functional and regression testing of both static and dynamic elements of a Web application;
- we discuss search algorithms for automatically exploring all the paths a user might follow in a Web application;
- we propose SmartProfiles as a high-level specification of test data to populate forms, and describe various strategies for automatically filling forms during site exploration.

The current prototype implementation of VeriWeb handles simple form and link navigation, checking for errors at the http-level only. Implementation details and experimental results will be presented in a forthcoming paper.

The remainder of this paper is organized as follows. In Section 2 we present the general architecture and different components of VeriWeb. Our approach to automatically filling out forms is discussed in Section 3. Related work is discussed in Section 4 and followed by concluding remarks in Section 5.

2 Automatic Web-Site Exploration

Three main tasks are involved in exploring paths in a Web site: searching (*i.e.,* determining the set of possible actions and systematically go through these), execution (*i.e.,* executing actions), and error handling (*i.e.,* detecting and reporting errors). Figure 1 shows the architecture of VeriWeb and the components responsible for each of these tasks. In what follows, we describe these tasks and how they are performed by the components of VeriWeb.

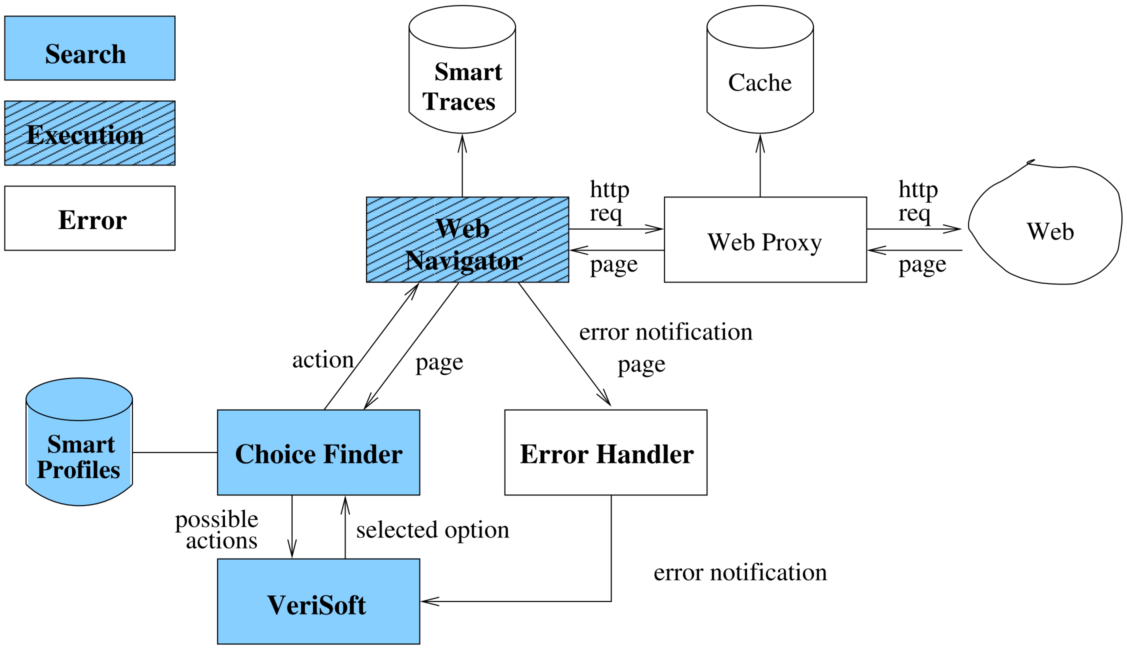


Figure 1: VeriWeb Architecture

2.1 Searching

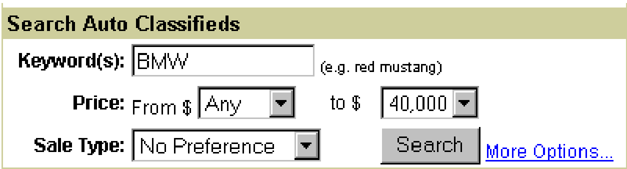
The objective of the search in VeriWeb is to exercise all possible execution paths a user might follow in a Web site. The search process is itself divided into two tasks: action discovery and exploration control. These two tasks are performed by the components *ChoiceFinder* and *VeriSoft*, respectively.

ChoiceFinder. ChoiceFinder traverses the Document Object Model (DOM) [6] of an HTML page and its frames and layers (if present) in search of *active objects* such as links, forms, buttons, items in pull down lists, etc., that have default actions associated (*e.g.,* when a link is clicked, the browser issues an HTTP GET command) or user-defined event handlers (*e.g.,* a link may additionally have an onClick handler that is executed before the browser issues the HTTP GET command). An action in our context corresponds to a browsing action that a user may perform. For instance, clicking on a link is an action that may lead to the execution of any handlers associated with the link, followed by retrieving the HREF specified in the link. [2]

Whereas some objects are associated with a single action, others may be associated with multiple actions. In particular, many actions may be associated with a form, depending on how the form is filled in. For instance, consider the Yahoo Auto Classifieds form shown in Figure 2; there are 23 different possible values for both the `From` and the `to` prices (*i.e.,* `Any`, `1,000`, `2,000`, etc.), 4 values for the `Sale Type` (*i.e.,* `No Preference`, `By Owner`, etc.), and a virtually infinite number of possible choices for `Keyword(s)`; each possible way to fill up this form may result in a different action following our terminology.

The role of ChoiceFinder is to analyze the content of an HTML page in order to find the set of possible

[2] Note that ChoiceFinder is limited to actions that involve HTML objects and ECMAscript. Actions within applets and plugins are outside the scope of VeriWeb.



(a)

```
<FORM action="/display/automobiles" method=get>
   <INPUT type=hidden value=table name=ct_hft>
   <INPUT type=hidden name=intl value="us">
   <B>Keyword(s):</B>
   <INPUT maxLength=50 size=15 name=ck>(e.g. red mustang)
   <input type=hidden name="cr" value="New York City">

   <B>Price:</B>
   From $ <SELECT name=cl_price>
                <OPTION value="" selected>Any
                <OPTION value=1000>1,000
                <OPTION value=2000>2,000
                ...................
                <OPTION value=50000>50,000</OPTION>
          </SELECT>
   to  $  <SELECT name=ch_price>
                <OPTION value="" selected>Any
                <OPTION value=1000>1,000
                ..................
                <OPTION value=50000>50,000</OPTION>
          </SELECT>
  <B>Sale Type:</B>
          <SELECT name="ce_sl">
                <OPTION value="" selected>No Preference
                <OPTION value="By Owner">By Owner
                <OPTION value="Used By Dealer">Used By Dealer
                <OPTION value="New By Dealer">New By Dealer</OPTION>
          </SELECT>
   <INPUT type=submit value=Search> 
   <A href="/display/automobiles?cc=automobiles&ct_hft=advsearch&
nodeid=750000268&cr=New+York+City">More Options...</A>

   <INPUT type=hidden value=automobiles name=cc>
   <INPUT type=hidden value=1 name=cf>
   <INPUT type=hidden value="750000268" name="fullnodeid">
</FORM>
```

(b)

Figure 2: Auto Classified Form

actions from that page. It must address two important and interrelated problems: determining the values to fill in forms and pruning the search space. For form elements such as text fields that do not have an explicit set of possible values defined in the HTML source, values must be provided by the tester. Even when an exhaustive list of possible values is available (*e.g.,* as with selection lists and radio buttons), it may be useful to limit the search space by defining subsets and combinations of values that are thought to be meaningful and representative. The critical problem of how to provide this auxiliary information and use it to guide the search is solved in VeriWeb using *SmartProfiles*, which are defined and discussed in detail in Section 3. ChoiceFinder uses SmartProfile information and analysis of form elements to generate a set of possible form-profile matches. The choice of which action to explore next, among the set of actions identified by ChoiceFinder, is made by VeriSoft.

VeriSoft. The control and management of the search process is performed using an existing tool, VeriSoft [19]. VeriSoft is a tool for systematically exploring the state spaces of software systems that may be composed of several concurrent processes. The state space of a system is defined as a directed graph that represents the combined behavior of all the components of the system being tested. Paths in this graph correspond to sequences of operations (scenarios) that can be observed during executions of the system. VeriSoft systematically explores the state space of a system by controlling and observing the execution of all the components, and by re-initializing their executions. VeriSoft can always guarantee complete coverage of the state space up to some depth. Whenever an error is detected during state-space exploration, a scenario leading to the error state is saved in a file. Scenarios can then be executed and replayed by the user with an interactive graphical simulator.

In the context of VeriWeb, the system being tested is a Web site and it is viewed as a single process by VeriSoft. The state space of the Web site is defined by the set of Web pages (statically or dynamically generated) in the site that can be reached from some initial page, given a SmartProfile. Reachable pages are the states of the Web-site state space, while the set of possible actions from a given page determined by ChoiceFinder defines the set of transitions from the corresponding state.

Systematic State-Space Exploration. The ExploreSite procedure shown in Figure 3 describes a *nondeterministic* algorithm which is executed under the control of VeriSoft for systematically exploring all possible sequences of actions identified by recursive calls to ChoiceFinder.

Starting from a pre-defined URL (startingURL, line 3), the algorithm explores execution paths in the state space reachable from that starting page. Each visited page is checked for errors using the component of VeriWeb called ErrorHandler (line 5); if an error is detected, the error is logged by VeriSoft for later inspection (by invoking VeriSoft.assert in line 7). (Error handling is discussed further in Section 2.3.) If a cycle (see next section) is detected (line 8), the exploration from the current page is aborted by invoking VeriSoft.abort (line 9). Otherwise, the search proceeds by invoking ChoiceFinder (line 11) to compute the set of possible actions from the current page. One of these actions is *nondeterministically* selected by VeriSoft (VeriSoft.toss in line 12) as explained below. The selected action is then executed (line 13) by the component of VeriWeb called Web Navigator and described in Section 2.2. If an error occurs during

```
1   ExploreSite(startingURL,constraints)
2   // Start a new scenario
3   currentPage = Navigator.load(startingURL);
4   while (true) {
5       error = ErrorHandler(currentPage,constraints);
6       if (error.status==true)
7           VeriSoft.assert(currentPage,error);
8       if (this page has been seen before)
9           VeriSoft.abort(currentPage,"cycle");
10      else {
11          choices = ChoiceFinder(currentPage);
12          selectedChoice = VeriSoft.toss(|choices| − 1);
13          currentPage = Navigator.execute(selectedChoice,choices);
14          if (currentPage.error != null)
15              VeriSoft.assert(currentPage,error);
16      }
17  }
```

Figure 3: Algorithm for Site Exploration

action execution, an error is reported (line 14–15). Otherwise, the same procedure is repeated from the page returned by the Navigator.

The execution of the above nondeterministic algorithm is controlled by VeriSoft. The special function VeriSoft.toss, provided by VeriSoft to simulate nondeterminism, takes as argument a positive integer n, and returns an integer in $[0, n]$. The function is nondeterministic: the execution of VeriSoft.toss(n) may yield up to $n+1$ different successor states, corresponding to different values returned by VeriSoft.toss. By controlling the value to be returned for each call to VeriSoft.toss, VeriSoft can drive the state-space exploration along a specific execution path. VeriSoft also forces the termination of the execution of the algorithm of Figure 3 when a certain *depth* is reached. This maximum depth is specified by the tester via one of the several parameters that can be used to control the state-space search performed by VeriSoft, and is measured here by the number of calls to VeriSoft.toss executed so far in the current run of the algorithm.

Every run of ExploreSite traverses a sequence of Web pages, *i.e.,* one path in the state space of the Web site. By controlling the value to be returned for each call to VeriSoft.toss, and by repeatedly executing ExploreSite with different sequences of such values, VeriSoft systematically explores the state space of the site being tested. VeriSoft supports several search strategies (such as depth-first search up to some maximum depth, iterative deepening, randomized search, etc.). We refer the reader to [19] for more details on VeriSoft.

Note that, in the case of Web sites where transactions (*i.e.,* "write" operations) can be performed during execution paths, we assume that the state of the Web site can be reset to some unique initial value using some application-specific procedure that can be invoked between runs of ExploreSite. In order to reset the state of the client side (browser), cookies are removed between runs of ExploreSite.

Pruning strategies. Clearly, the state space of a Web site can be huge, even infinite. Therefore, it is often necessary in practice to limit the size of the state space using various techniques.

One obvious cause of "state explosion" is the astronomical number of ways most forms can be filled in. For example, ignoring `Keywords`, there are 2,116 different ways to fill in the Yahoo Auto Classifieds form of Figure 2. VeriWeb is designed to support a variety of strategies to deal with this issue. The level of pruning can be tuned by defining SmartProfiles and profile policies (search constraints) as described in Section 3.

In addition, simple pruning techniques used in standard crawlers can also be applied such as restricting the search by not following URLs outside of a set of domains, eliminating links that match some pre-defined set of regular expressions (*e.g.,* mail:*, *.ps, *doubleclick*), and setting a limit on the number of links to be followed in each page.

Another source of inefficiency in the search are cyclic paths. For example, in the Travelocity site, all pages have a `Home` link that points to the main page; thus, if all links are explored in all visited pages, the main page will be visited many times. A possible optimization is to record visited URLs and prevent the search from exploring successors from the same URL more than once. However, for dynamic sites, applying this optimization is not always straightforward. For example, sites that generate pages dynamically may generate different links to the same content – different servers can be used for load balancing, or different session ids may be appended to links. For instance, in the Travelocity site, every `Home` link is of the form:

```
<a href="http://dps1.travelocity.com:80/glblwhere.ctl?go_to_ctl=HOME&
go=HOME&SEQ=100567877705164111132001&LANG=EN&
last_pgd_page=logngstexp.pgd">Home</a>
```

Since the session id `SEQ` embedded in the link may change after some period of time, recording the HREF is not enough to avoid revisiting the main page during site exploration.

Even when previously visited pages can be detected, eliminating them may be undesirable. For example, in Travelocity, depending on the exploration context, clicking on the `Flights` link may lead to different pages, namely to the login page if this is the first visit, or to the flight search page if the browser already contains a cookie. Thus, disregarding the `Flights` link the second time around will in effect prevent the exploration of a new, previously unexplored path. Similarly, for personalized sites, clicking on a particular link may lead to different content, depending on the user browsing the site.

Detecting cycles (as done in line 8 of the algorithm of Figure 3) may be more or less important depending on the specific Web site being tested. In our current prototype we do not perform cycle detection, and rely on VeriSoft to limit the depth of the search and guarantee the termination of the search process.

2.2 Execution

Actions selected during site exploration are executed by the Web Navigator. The Web Navigator mimics closely what a user would do while interacting with a browser – the test scenarios are thus realistic simulations of user interactions. In fact, our prototype uses a standard browser to perform the exploration. Note that it is also possible to implement the Navigator using freely available HTTP libraries, DOM interfaces and JavaScript interpretors. We opted to use existing browsers not only because it simplifies the implementation, but also because it allows the detection of errors that may be specific to a particular browser.

The Web Navigator differs from traditional crawlers in the way it treats pages that contain forms and client-side scripts. After ChoiceFinder analyzes a page and an action is selected, it instructs the Navigator to execute the selected action. The execution of an action follows the standards defined for HTML and ECMAscript.

- Given a link, the location of the browser is set to the HREF of the link (effectively loading the HREF); if the link has an onClick event handler, the handler is invoked and the location of the browser is set to the HREF of the link only if the handler returns true.
- Given a form, the values selected by the ChoiceFinder are set, the onSubmit handler (if any) is invoked, and the form is submitted.
- Given an active form element, its associated event handler is invoked.

2.3 Error Handling

Two broad classes of errors are detected by VeriWeb: navigation errors and page errors. Navigation (execution) errors are detected by the Navigator, and include failure in retrieving a Web page (*e.g.,* page not found) and unsuccessful form submission (*e.g.,* onSubmit handler returns false). Page errors are detected by the ErrorHandler component of VeriWeb, which can check various properties on visited Web pages. For example, pages can be analyzed using weblint [21], or checked against accessibility guidelines [3]. They can also be "grepped" for strings that identify application specific errors (*e.g.,* "cannot connect to database", "invalid customer") or for constraints that must hold throughout the Web site (*e.g.,* all pages must contain a navigation bar). More complex graph-theoretic properties involving arbitrary sequences or trees of Web pages (*e.g.,* constraints on frame combinations) could also be specified and checked, along the same lines as the verification of temporal properties of state spaces using model checking [4]. VeriWeb is designed in a modular way so that various existing checking modules can easily be invoked from the ErrorHandler.

Error logging is performed at three different levels: by VeriSoft (error traces), by the Navigator (SmartBookmarks), and by the Web Proxy (Cache of pages retrieved).

Whenever an error is detected during Web-site exploration, a scenario (*i.e.,* a sequence of VeriSoft.toss values defining a state-space path from the starting page) is saved by VeriSoft in a file. These scenarios can be visualized and replayed interactively by the tester for debugging purposes.

In addition to the diagnosis information recorded by VeriSoft, the Navigator itself can optionally save a richer representation of scenarios leading to errors. These scenarios are saved in the SmartTraces repository as SmartBookmarks and can be replayed by the WebVCR [1]. The WebVCR provides a VCR-style interface to transparently record and replay users' browsing actions. This feature makes it possible to replay sequences of actions over Web pages that can change from one testing session to the next. In contrast to capture-replay tools such as e-tester [7] whose scenarios cannot be replayed if minor changes are applied to the Web site (since they refer to links and forms by their DOM address), SmartBookmarks use a more robust representation and can be correctly replayed even in the presence of structural changes in the underlying Web pages [1].

During site exploration, testers can also choose to cache all visited pages, or visited pages in scenarios that lead to errors. Caching these pages allows testers to inspect transient errors (*e.g.,* a database that was temporarily unreachable). Caching pages also allows offline error detection, *i.e.,* apply the ErrorHandler checks over the cached pages; however, since some pages may be created by scripts as they are loaded in the browser, analyzing cached pages may cause errors to be missed.

3 Automatically Filling Forms

Dealing with forms is one of the key issues in automated exploration of dynamic Web sites. For a traditional crawler that only follows links, the set of immediate successor states for any given page is bounded by the size of the page, and the set of state-change actions can be determined statically. In contrast, both of these properties do not hold in the case of forms. Indeed, the collection of immediate successor states could be infinite for all practical purposes – consider a form that queries for the name of the user and then generates a Web page that prints out a personalized greeting. Even if the number of distinct successor states is in fact small, an automated tool may not be able to determine which user inputs are sufficient to cover these states. A standard example is a page containing a form querying for a userid and password: without additional information, no tool can explore successor states of such a page, other than the trivial state corresponding to rejection of an invalid user. Thus, auxiliary information must be provided to enable a tool to crawl through forms. In the sections below we discuss the following questions:

- At what level must the user of the testing tool provide auxiliary information?
- How is auxiliary information used to guide the search?

We propose to address these questions by using a data model structured as a set of profiles, which we call *SmartProfiles*. The guiding principles for designing SmartProfiles are as follows:

- Profiles are described at the level of a *data model*, not that of *forms*. We assume that testers are familiar with the structure of the information to be presented through the interface at the level of an enduser's data model. For instance, a tester of a car-sale site will specify a dataset that describes a set of customer profiles, including userids and passwords if the site demands them, and car models or properties that a user might be interested in. A tester need not describe the information used to populate fields in a form-by-form manner.

  Specifications at the data model level are generally more concise than a form-by-form approach, since sites often give alternative methods of entering the same data, and sometimes require the same information to be entered repeatedly in several forms. The model-driven approach also allows the specification to be maintained as a Web site evolves, while a form-based approach is inherently sensitive to the smallest changes in page format. Most importantly, a form-based approach requires the tester to peruse the forms of a site manually whenever the site changes – exactly the kind of time-consuming activity that we wish to eliminate from the testing cycle.

- The specification should be as *light-weight* as possible – we do not want to burden the tester with a full-fledged data-modeling formalism, and we do not provide the ability to specify how often a particular data value can be used in forms, or to explicitly say what combinations of data values can be entered in a particular path. In addition to the obvious motivation in terms of ease-of-use, the granularity of the specification also has a relationship to test coverage. The more dependencies that are specified on the data to be entered via forms, the greater the possibility that the test runs generated will fail to detect error conditions that result from user behavior violating these dependencies.

From these design principles, we developed SmartProfiles, which are sets of *profiles* formed from *signatures* and *fields*. Signatures and fields have the following structure: [3]

```
type Signature =
      signature [ name [ String ],
                  Field+
                ]
type Field =
      field [ name [ String ],
              @key [ Boolean ],
              synonym [ String ]?
            ]
```

A *signature* consists of a name that identifies the signature and of a list of one or more fields. Each *field* has a name, an attribute which indicates whether the field is a *key*, and optionally a string interpreted as a regular expression and representing *synonyms* of the name of the field. Keys indicate fields that *must* be matched in an HTML form for a profile to be applied. Profiles can then be defined from signatures as follows:

```
type Profile =
  profile [ name [ String ],
            signature [ String ],
            FieldValue+
          ]
type FieldValue =
  fieldvalue [ name [ String ],
               regexp [ String ]
             ]
```

A *profile* consists of a name that identifies the profile, the name of the signature that corresponds to this profile, and a list of *FieldValues*, which provide values for some of the fields in the signature, including keys. Each FieldValue contains the name of a field defined in the signature, and a specification *regexp* for the value(s) of that field. The value of regexp is a string interpreted as a regular expression built up from $|, *, +$. The use of regular expressions both in `regexp` and `synonym` allows the specification of a range of possible values for a field or signature name, respectively. Figure 4 gives a definition of the syntax of signatures and profiles in the form of an XML DTD. Figure 5 shows an example of signature and profiles for a Web site selling cars.

[3]The description is given in alternative notation for XML schemas described in the XML Query Algebra [8].

```
<!DOCTYPE smartprofile [
  <!ELEMENT smartprofile (signature* | profile*)>

  <!ELEMENT signature (name, field+)>
  <!ELEMENT field (name,synonym)>
  <!ATTLIST field key CDATA>

  <!ELEMENT profile (name, signature, fieldvalue+)>
  <!ELEMENT fieldvalue (name,regexp)>

  <!ELEMENT regexp (#PCDATA) >

  <!ELEMENT name (#PCDATA)>
  <!ELEMENT synonym (#PCDATA)>

]>
```

Figure 4: DTD for signature and profile

```
<signature>
   <name> CarType </name>
   <field key=true>
      <name> Make </name>
      <synonym> *carmake* </synonym>
   </field>
   <field key=false>
      <name> Model </name>
   </field>
   <field key=false>
      <name> Year </name>
   </field>
   <field key=false>
      <name> Interior </name>
   </field>
   <field key=false>
      <name> Type </name>
   </field>
</signature>
```

(a)

```
<profile>
   <name> HondaCivicSedan </name>
   <signature> CarType </signature>
   <fieldvalue>
      <name> Make </name>
      <regexp> Honda </regexp>
   </fieldvalue>
   <fieldvalue>
      <name> Model </name>
      <regexp> *Civic*Sedan*  </regexp>
   </fieldvalue>
   <fieldvalue>
      <name> Year </name>
      <regexp> 1999|2000 </regexp>
   </fieldvalue>
   <fieldvalue>
      <name> Interior </name>
      <regexp> Beige </regexp>
   </fieldvalue>
   <fieldvalue>
      <name> Type </name>
      <regexp> Sedan </regexp>
   </fieldvalue>
</profile>

<profile>
   <name> Rolls </name>
   <signature> CarType </signature>
   <fieldvalue>
      <name> Make </name>
      <regexp> Rolls-Royce </regexp>
   </fieldvalue>
   <fieldvalue>
      <name> Model </name>
      <regexp> *Corniche*  </regexp>
   </fieldvalue>
   <fieldvalue>
      <name> Year </name>
      <regexp> 2002 </regexp>
   </fieldvalue>
   <fieldvalue>
      <name> Type </name>
      <regexp> Convertible* </regexp>
   </fieldvalue>
</profile>
```

(b)

Figure 5: Example of signature and profiles

As described in Figure 3, ChoiceFinder (line 11) processes each visited Web page to determine the set of possible actions from that page. If the page is found to include forms, ChoiceFinder uses SmartProfiles to obtain a set of form completion and execution actions for each form in the page using the following process.

1. Form analysis: the form is analyzed to generate a field name for every field. (Methods for identifying labels of text fields are discussed in [12].) The result is a *form schema*, consisting of a set of distinct attribute names optionally paired with a list of possible values (from a selection, checkbox or radio-button list).

2. Form matching: each form schema is matched against SmartProfiles. (This can be viewed as an instance of the *schema matching problem* [9, 10].) The result is a set of partial functions matching profile fields to form fields, that we call *candidate profiles*.

3. Form completion: for each candidate profile, each field in the form is matched with a possible value specified in the candidate profile or with the default value if none is specified in the profile. The result is called the set of all possible *consistent completions* of the form.

Since the set of consistent completions of a form can be quite large (particularly if the number of keys is small), this set can be further reduced with optional *profile policies*. An example of profile policy is to limit to one the number of profiles that can used for a given signature during the same test path – under this policy, once a profile has been selected to match a form during a Web-site execution path, only values specified in that same profile will be used to populate subsequent forms along this specific execution path.

No matter how sophisticated the page analysis, additional information may sometimed be required from the tester in order to perform form matching. For this purpose, VeriWeb includes an interactive profile-refinement *iterative process* where the tester is prompted for specific missing information. For instance, the tester may be asked to define a name for a unknown form field, or to provide a synonym for a signature name that is not matched with any form field. This additional information is then recorded as part of a new SmartProfile. An advantage of this process is that information gathering is driven by the needs of the test engine to carry out Web-site exploration: the tester is not required to navigate the site. Iterations of this process allow the test engine to accumulate large synonym lists for a given profile field for use in subsequent exploration, and these lists may remain valid even when the order of the pages or the order of the fields within forms is modified.

A detailed description of how VeriWeb performs form analysis, matching and completion, as well as of profile policies and the profile-refinement iterative process, is omitted in this paper.

4 Related Work

Software testing for Internet applications has already been discussed extensively in the literature (*e.g.,* see [16]). Many free and commercial tools are already available for testing Web sites (*e.g.,* see [17]). For instance, link and page testers (*e.g.,* [22, 11]) check Web pages for browser compatibility, load time, dead links, spelling errors, and conformance to HTML. In addition to standard link and page checks, commercial testing tools [13, 14, 7, 20] automate functional and regression testing by providing capture-replay capabilities that let testers create, record and replay test scenarios. For scenarios that involve forms, sets of data values can be specified by the tester to be used as sets of inputs to specific forms. VeriWeb goes a step further than these tools: by providing flexible profiles, it allows the automatic discovery of scenarios. An important limitation of capture-replay tools is the relative lack of robustness of the test scenarios. For example, test scenarios recorded by e-tester [7] refer to objects in Web pages by their DOM addresses; thus, if objects are moved around in the page (a new form or link is added), a previously-recorded test scenario cannot be replayed. In contrast, VeriWeb automatically discovers test scenarios dynamically, and uses the techniques described in [1] to save and robustly replay scenarios leading to errors.

Performance-oriented tools (*e.g.,* [18, 15, 13]) automatically perform "stress" tests to check for server speed, responsiveness, stability and reliability. The test scenarios involved in this context are typically limited to traversing static links and their coverage is usually very limited (*i.e.,* many instances of a *same* test scenario are simultaneously executed). Although VeriWeb could be extended to cover performance testing, our main emphasis in this work has been so far on increasing test automation and coverage for functional testing (*i.e.,* many different execution paths are automatically generated, executed and checked for errors).

Extracting content from Web pages hidden behind search forms in large searchable databases is discussed in [12, 2]. The class of Web sites considered in [12] are "read-only" data-driven sites that are cgi-based. In contrast, VeriWeb can navigate through a larger class of dynamic pages that may include client-side scripts, state changes, transactions, etc. Another major difference is our use of high-level SmartProfiles for automatically populating sets of forms, instead of per-form data specifications as done in [12]. In addition, our SmartProfiles are strictly more expressive than the model used in [12], where values associated with different elements are always "independent" of each other. For example, for a login form that requires a username and password, [12] would create two separate tables: one that stores values for username and another that stores values for password. Because no relationship between the two elements can be represented in the model, the system would have to try all the combinations in the Cartesian product of the value tables – clearly leading to many unsuccessful logins.

In [4, 5], de Alfaro et. al describe MCWeb, a tool which is works like a Web crawler and navigates through static links and frames. The main originality of this tool lies in the broad class of graph-theoretic reachability properties the users can specify and check, such as complex properties of frame combinations. However, the problem of navigating through forms or other data-driven interfaces is not addressed in MCWeb.

5 Conclusion

We have presented an overview of VeriWeb, a tool for automatically exploring execution paths of dynamic Web sites. VeriWeb was designed to combine the flexible navigation capabilities of capture-replay tools with the high level of automation provided by Web crawlers. We have discussed the architecture of our current prototype implementation, search algorithms for exploring Web sites, and SmartProfiles and associated techniques for automatically filling forms. Preliminary experimental results are encouraging – a detailed discussion on experiences using the system will be presented in an upcoming paper.

Acknowledgments: Randy Hackbarth was instrumental in initiating the VeriWeb effort. We also thank Avinash Vyas for helpful comments and discussions, and Joanna McCaffrey for collaborating with us on the application of VeriWeb to Lucent Web sites.

References

[1] V. Anupam, J. Freire, B. Kumar, and D. Lieuwen. Automating Web navigation with the WebVCR. In *Proc. of WWW*, pages 503–517, 2000.

[2] M. Bergman. The Deep Web: Surfacing Hidden Value. http://www.brightplanet.com/deepcontent/tutorials/-DeepWeb/index.asp.

[3] Bobby 3.2. http://www.cast.org/bobby.

[4] L. de Alfaro. Model checking the World Wide Web. In *Conference on Computer Aided Verification (CAV)*, pages 337–349, 2001.

[5] L. de Alfaro, T.A. Henzinger, and F.Y.C. Mang. MCWEB: A model-checking tool for Web site debugging. In *World Wide Web Conference (WWW) – Poster Proceedings*, pages 86–87, 2001.

[6] Document Object Model specification. http://www.w3.org/TR/REC-DOM-Level-1.

[7] E-tester. http://www.rswsoftware.com/products/etester_index.shtml.

[8] P. Fankhauser, M. Fernandez, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. The XML query algebra, February 2001. http://www.w3.org/TR/2001/WD-query-algebra-20010215.

[9] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity flooding: A versatile graph matching algorithm. In *Proc. of ICDE*, 2002. To appear.

[10] R. Miller, L. Haas, and M. Hernández. Schema mapping as query discovery. In *Proc. of VLDB*, pages 77–88, 2000.

[11] Netmechanic. http://www.netmechanic.com.

[12] S. Raghavan and H. Garcia-Molina. Crawling the Hidden Web. In *Proc. of VLDB*, pages 129–138, 2001.

[13] Rational robot. http://www.rational.com/products/robot/index.jsp.

[14] Silk test. http://www.segue.com/html/s_solutions/silk/s_family.htm.

[15] Sitetools. http://www.softlight.com/sitea/index.asp.

[16] H. Sneed and S. Göshcl. Testing software for Internet applications. *Software Focus*, 1(1):15–22, September 2000.

[17] Web site test tools and site management tools. http://www.softwareqatest.com/qatweb1.html.

[18] Technovations load testing products. http://www.technovations.com/home.htm.

[19] VeriSoft. http://www.bell-labs.com/projects/verisoft.

[20] Watchfire enterprise solution. http://www.watchfire.com/solutions/wes.asp.

[21] Weblint. http://www.w3.org/Tools/weblint.html.

[22] Web site garage. http://websitegarage.netscape.com.