WebSite Testing

Edward Miller
Software Research, Inc.
901 Minnesota Street
San Francisco, CA 94107 USA

Email comments to miller@soft.com
See also the companion papers:
The WebSite Quality Challenge.
WebSite Loading and Capacity Analysis.

ABSTRACT

The instant worldwide audience of a WebSite make its quality and reliability crucial factors in its success. Correspondingly, the nature of the WWW and WebSites pose unique software testing challenges. Webmasters, WWW applications developers, and WebSite quality assurance managers need tools and methods that meet their specific needs. Mechanized testing via special purpose WWW testing software offers the potential to meet these challenges. Our technical approach, based on existing WWW browsers, offers a clear solution to most of the technical needs for assuring WebSite quality.

BACKGROUND

WebSites impose some entirely new challenges in the world of software quality! Within minutes of going live, a WWW application can have many thousands more users than a conventional, non-WWW application. The immediacy of the WWW creates immediate expectations of quality and rapid application delivery, but the technical complexities of a WebSite and variances in the browser make testing and quality control that much more difficult, and in some ways, more subtle, than "conventional" client/server or application testing. Automated testing of WebSites is an opportunity and a challenge.

DEFINING WEBSITE QUALITY & RELIABILITY

Like any complex piece of software there is no single, all inclusive quality measure that fully characterizes a WebSite.

Dimensions of Quality. There are many dimensions of quality; each measure will pertain to a particular WebSite in varying degrees. Here are some common measures:

- *Timeliness:* WebSites change often and rapidly. How much has a WebSite changed since the last upgrade? How do you highlight the parts that have changed?
- *Structural Quality:* How well do all of the parts of the WebSite hold together? Are all links inside and outside the WebSite working? Do all of the images work? Are there parts of the WebSite that are not connected?
- *Content:* Does the content of critical pages match what is supposed to be there? Do key phrases exist continually in highly-changeable pages? Do critical pages maintain quality content from version to version? What about dynamically generated HTML (DHTML) pages?
- *Accuracy and Consistency:* Are today's copies of the pages downloaded the same as yesterday's? Close enough? Is the data presented to the user accurate enough? How do you know?
- *Response Time and Latency:* Does the WebSite server respond to a browser request within certain performance parameters? In an E-commerce context, how is the end-to-end response time after a `SUBMIT`? Are there parts of a site that are so slow the user discontinues working?
- *Performance:* Is the Browser->Web->WebSite->Web->Browser connection quick enough? How does the performance vary by time of day, by load and usage? Is performance adequate for E-commerce applications? Taking 10 minutes -- or maybe even only 1 minute -- to respond to an E-commerce purchase may be unacceptable!

Impact of Quality. Quality remains is in the mind of the WebSite user. A poor quality WebSite, one with many broken pages and faulty images, with `Cgi-Bin` error messages, etc., may cost a lot in poor customer relations, lost corporate image, and even in lost sales revenue. Very complex, disorganized WebSites can sometimes overload the user.

The combination of WebSite complexity and low quality is potentially lethal to Company goals. Unhappy users will quickly depart for a different site; and, they probably won't leave with a good impression.

WEBSITE ARCHITECTURAL FACTORS

A WebSite can be quite complex, and that complexity -- which is what provides the power, of course -- can be a real impediment in assuring WebSite Quality. Add in the possibilities of multiple WebSite page authors, very-rapid updates and changes, and the problem compounds.

Here are the major pieces of WebSites as seen from a Quality perspective.

Browser. The browser is the viewer of a WebSite and there are *so* many different browsers and browser options that a well-done WebSite is probably designed to look good on as many browsers as possible. This imposes a kind of *de facto* standard: the WebSite must use only those constructs that work with the *majority* of browsers. But this still leaves room for a lot of creativity, and a range of technical difficulties. And, multiple browsers' renderings and responses to a WebSite have to be checked.

Display Technologies. What you see in your browser is actually composed from many sources:

- *HTML.* There are various versions of HTML supported, and the WebSite ought to be built in a version of HTML that is compatible. This should be checkable.

- *Java, JavaScript, ActiveX.* Obviously JavaScript and Java applets will be part of any serious WebSite, so the quality process must be able to support these. On the Windows side, ActiveX controls have to be handled well.

- *Cgi-Bin Scripts.* This is link from a user action of some kind (typically, from a FORM passage or otherwise directly from the HTML, and possibly also from within a Java applet). All of the different types of Cgi-Bin Scripts (perl, awk, shell-scripts, etc.) need to be handled, and tests need to check "end to end" operation. This kind of a "loop" check is crucial for E-commerce situations.

- *Database Access.* In E-commerce applications you are either building data up or retrieving data from a database. How does that interaction perform in real world use? If you give in "correct" or "specified" input does the result produce what you expect?

  Some access to information from the database may be appropriate, depending on the application, but this is typically found by other means.

  Navigation. Users move to and from pages, click on links, click on images (thumbnails), etc. Navigation in a WebSite is often complex and has to be quick and error free.

  Object Mode. The display you see changes dynamically; the only constants are the "objects" that make up the display. These aren't real objects in the OO sense; but they have to be treated that way. So, the quality test tools have to be able to handle URL links, forms, tables, anchors, buttons of all types in an "object like" manner so that validations are independent of representation.

  Server Response. How fast the WebSite host responds influences whether a user (i.e. someone on the browser) moves on or gives up. Obviously, InterNet loading affects this too, but this factor is often outside the Webmaster's control at least in terms of how the WebSite is written. Instead, it seems to be more an issue of server hardware capacity and throughput. Yet, if a WebSite becomes very popular -- this can happen overnight! -- loading and tuning are real issues that often are imposed -- perhaps not fairly -- on the WebMaster.

  Interaction & Feedback. For passive, content-only sites the only real quality issue is availability. For a WebSite that interacts with the user, the big factor is how fast and how reliable that interaction is.

  Concurrent Users. Do multiple users interact on a WebSite? Can they get in each others' way? While WebSites often resemble client/server structures, with multiple users at multiple locations a WebSite can be much different, and much more complex, than complex applications.

WEBSITE TEST AUTOMATION REQUIREMENTS

Assuring WebSite quality requires conducting sets of tests, automatically and repeatably, that demonstrate required properties and behaviors. Here are some required elements of tools that aim to do this.

Test Sessions. Typical elements of tests involve these characteristics:

- *Browser Independent.* Tests should be realistic, but *not* be dependent on a particular browser, whose biases and characteristics might mask a WebSite's problems.

- *No Buffering, Caching.* Local caching and buffering -- often a way to improve apparent performance -- should be disabled so that timed experiments are a true measure of the Browser-Web-WebSite-Web-Browser response time.

- *Fonts and Preferences.* Most browsers support a wide range of fonts and presentation preferences, and these should not affect how quality on a WebSite is assessed or assured.

- *Object Mode.* Edit fields, push buttons, radio buttons, check boxes, etc. All should be treatable in object mode, i.e. independent of the fonts and preferences.

  Object mode operation is essential to protect an investment in test suites and to assure that test suites continue operating when WebSite pages experience change. In other words, when buttons and form entries change location on the screen -- as they often do -- the tests should still work.

  However, when a button or other object is deleted, that error should be sensed! Adding objects to a page clearly implies re-making the test.

- *Tables and Forms.* Even when the layout of a table or form varies in the browser's view, tests of it should continue independent of these factors.

- *Frames.* Windows with multiple frames ought to be processed simply, i.e. as if they were multiple single-page frames.

Test Context. Tests need to operate from the browser level for two reasons: (1) this is where users see a WebSite, so tests based in browser operation are the most realistic; and (2) tests based in browsers can be run locally or across the Web equally well. Local execution is fine for quality control, but not for performance measurement work, where response time *including* Web-variable delays reflective of real-world usage is essential.

WEBSITE DYNAMIC VALIDATION

Confirming validity of what is tested is the key to assuring WebSite quality -- the most difficult challenge of all. Here are four key areas where test automation will have a significant impact.

Operational Testing. Individual test steps may involve a variety of checks on individual pages in the WebSite:

- *Page Consistency.* Is the entire page identical with a prior version? Are key parts of the text the same or different?

- *Table, Form Consistency.* Are all of the parts of a table or form present? Correctly laid out? Can you confirm that selected texts are in the "right place".

- *Page Relationships.* Are all of the links on a page the same as they were before? Are there new or missing links? Are there any broken links?

- *Performance Consistency, Response Times.* Is the response time for a user action the same as it was (within a range)?

Test Suites. Typically you may have dozens or hundreds (or thousands?) of tests, and you may wish to run tests in a variety of modes:

Unattended Testing. Individual and/or groups of tests should be executable singly or in parallel from one or many workstations.

- *Background Testing.* Tests should be executable from multiple browsers running "in the background" on an appropriately equipped workstation.

- *Distributed Testing.* Independent parts of a test suite should be executable from separate workstations without conflict.

- *Performance Testing.* Timing in performance tests should be resolved to the millisecond; this gives a strong basis for averaging data.

- *Random Testing.* There should be a capability for randomizing certain parts of tests.

- *Error Recovery.* While browser failure due to user inputs is rare, test suites should have the capability of resynchronizing after an error.

Content Validation. Apart from how a WebSite responds dynamically, the content should be checkable either exactly or approximately. Here are some ways that content validation could be accomplished:

- *Structural.* All of the links and anchors should match with prior "baseline" data. Images should be characterizable by byte-count and/or file type or other file properties.

- *Checkpoints, Exact Reproduction.* One or more text elements -- or even all text elements -- in a page should be markable as "required to match".

- *Gross Statistics.* Page statistics (e.g. line, word, byte-count, checksum, etc.).

- *Selected Images/Fragments.* The tester should have the option to rubber band sections of an image and require that the selection image match later during a subsequent rendition of it. This ought to be possible for several images or image fragments.

Load Simulation. Load analysis needs to proceed by having a special purpose browser act like a human user. This assures that the performance checking experiment indicates true performance -- not performance on simulated but unrealistic conditions. There are many "`http` torture machines" that generate large numbers of `http` requests, but that is not necessarily the way real-world users generate requests.

Sessions should be recorded live or edited from live recordings to assure faithful timing. There should be adjustable speed up and slow down ratios and intervals.

Load generation should proceed from:

- *Single Browser Sessions.* One session played on a browser with one or multiple responses. Timing data should be put in a file for separate analysis.

- *Multiple Independent Browser Sessions.* Multiple sessions played on multiple browsers with one or multiple responses. Timing data should be put in a file for separate analysis. Multivariate statistical methods may be needed for a complex but general performance model.

TESTING SYSTEM CHARACTERISTICS

Considering all of these disparate requirements, it seems evident that a single product that supports all of these goals will not be possible. However, there is one common theme and that is that the majority of the work seems to be based on "*...what does it [the WebSite] look like from the point of view of the user?*" That is, from the point of view of someone using a browser to look at the WebSite.

This observation led our group to conclude that it would be worthwhile trying to build certain test features into a "test enabled web browser", which we called eValid.

Browser Based Solution. With this as a starting point we determined that the browser based solution had to meet these additional requirements:

- *Commonly Available Technology Base.* The browser had to be based on a well known base (there appear to be only two or three choices).

- *Some Browser Features Must Be Deletable.* At the same time, certain requirements imposed limitations on what was to be built. For example, if we were going to have accurate timing data we had to be able to *disable* caching because otherwise we are measuring response times within the client machine rather than "across the web."

- *Extensibility Assured.* To permit meaningful experiments, the product had to be extensible enough to permit timings, static analysis, and other information to be extracted.

Taking these requirements into account, and after investigation of W3C's Amaya Browser and the open-architecture Mozilla/Netscape Browser we chose the IE Brower as our initial base for our implementation of eValid.

User Interface. How the user interacts with the product is very important, in part because in some cases the user will be someone very familiar with WebSite browsing and not necessarily a testing expert. The design we implemented takes this reality into account.

- *Pull Down Menus.* In keeping with the way browsers are built, we put all the main controls for eValid on a set of Pull Down menus, as shown in the accompanying screen shot.

Figure 1. eValid Menu Functions.

- "C" Scripting. We use interpreted "C" language as the control language because the syntax is well known, the language is fully expressive of most of the needed logic, and because it interfaces well with other products.

- *Files Interface.* We implemented a set of dialogs to capture critical information and made each of them recordable in a text file. The dialogs are associated with files that are kept in parallel with each browser invocation:

  - *Keysave File.* This is the file that is being created -- the file is shown line by line during script recording as the user moves around the candidate WebSite.

  - *Timing File.* Results of timings are shown and saved in this file.

  - *Messages File.* Any error messages encountered are delivered to this file. For example, if a file can't be downloaded within the user-specified maximum time an error message is issued and the playback continues. (This helps preserve the utility of tests that are partially unsuccessful.)

  - *Event File.* This file contains a complete log of recording and playback activities that is useful primarily to debug a test recording session or to better understand what actually went on during playback.

Operational Features. Based on prior experience, the user interface for eValid had to provide for several kinds of capabilities already known to be critical for a testing system. Many of these are critically important for automated testing because they assure an optimal combination of test script reliability and robustness.

- Capture/Replay. We had to be able both to capture a user's actual behavior online, and be able to create scripts by hand.

- Object Mode. The recording and playback had to support pure-Object Mode operation. This was achieved by using internal information structures in a way that lets the scripts (either recorded or constructed) to refer to objects that are meaningful in the browser context.

  A side benefit of this was that playbacks were reliable independent of the rendering choices made by the user. A script plays back identically the same independent of browser window size, type-font choices, color mappings, etc.

- [Adjustable] True-Time Mode. We assured realistic behavior of the product by providing for recording of user-delays and for efficient handling of delays by incorporating a continuously variable "playback delay multiplier" that can be set by the user.

- Playback Synchronization. For tests to be robust -- that is, to reliably indicate that a feature of a WebSite is working correctly -- there must be a built-in mode that assures synchronization so that Web-dependent delays don't interfere with proper WebSite checking. eValid does this using a proprietary playback synchronization method that waits for download completion (except if a specified maximum wait time is exceeded).

- Timer Capability. To make accurate on-line performance checks we built in a 1 millisecond resolution timer that could be read and reset from the playback script.

- Validate Selected Text Capability. A key need for WebSite content checking, as described above, is the ability to capture an element of text from an image so that it can be compared with a baseline value. This feature was implemented by digging into the browser data structures in a novel way (see below for an illustration). The user highlights a selected passage of the web page and clicks on the "Validate Selected Text" menu item.

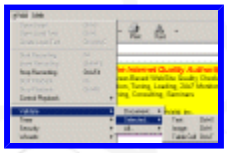

Figure 2. Illustration of eValid Validate Selected Text Feature.

What results is a recorded line that includes the ASCII text of what was selected, plus some other information that locates the text fragment in the page. During playback if the same text is **not** found at the same location an error message is generated.

- Multiple-playback. We confirmed that multiple playback was possible by running separate copies of the browser in parallel. This solved the problem of how to multiply a single test session into a number of test sessions to simulate multiple users each acting realistically.

Test Wizards. In most cases manual scripting is too laborious to use and making a recording to achieve a certain result is equally unacceptable. We built in several *test wizards* that mechanize some of the most common script-writing chores.

- *Link Wizard.* This wizard creates a script based on the current Web page that visits every link in the page. Scripts created this way are the basis for "link checking" test suites that confirm the presence (but not necessarily the content) of URLs.

Here is a sample of the output of this wizard, applied to our standard sample test page `example1.html`:

```
void name()
{
/* Produced by eValid Ver. 3.0 Link Wizard */
/* (c) Copyright 2001 by Software Research, Inc. */

InitLink "http://www.soft.c../../Products/Web/CAPBAK/example1/"
GotoLink "http://www.soft.c../../Products/Web/CAPBAK/example1/#target"
GotoLink "http://www.soft.c../../Products/Web/CAPBAK/example1/#notdefined"
GotoLink "http://www.soft.c../../Products/Web/CAPBAK/example1/example1.out.html"
GotoLink "http://www.soft.c../../Products/Web/CAPBAK/example1/example1.notoutside.html"
GotoLink "http://www.soft.c../../Products/Web/CAPBAK/example1/#topofpage"
}
```

Figure 3. Sample of Output of Link Test Wizard.

- *FORM Wizard.* For E-Commerce testing which involves FORMS we included in the system a FORM Wizard that generates a script that:
  - Initializes the form.
  - Presses each pushbutton by name.
  - Presses each radio button by name.
  - Types a pre-set script fragment into each text field.
  - Presses `SUBMIT`.

Here is a sample of the output of this wizard, applied to our standard test page: `example1.html`:

```
void name()
{
/* Produced by eValid Ver. 3.0 Form Wizard */
/* (c) Copyright 2001 by Software Research, Inc. */

InitLink("http://www.testworks.c../../Products/Web/CAPBAK/example1/");
SubmitForm(FORM:0:12, "RESET FORM");
SelectOneRadio(FORM:0:0, "now", "TRUE");
SelectOneRadio(FORM:0:1, "next", "TRUE");
SelectOneRadio(FORM:0:2, "look", "TRUE");
SelectOneRadio(FORM:0:3, "no", "TRUE");
SelectCheckBox(FORM:0:4, "concerned", "TRUE");
SelectCheckBox(FORM:0:5, "info", "TRUE");
SelectCheckBox(FORM:0:6, "evaluate", "TRUE");
SelectCheckBox(FORM:0:7, "send", "TRUE");
FormTextInput(FORM:0:8, "TestWorks");
FormTextInput(FORM:0:9, "TestWorks");
```

```
FormTextInput(FORM:0:10, "TestWorks");
FormTextInput(FORM:0:11, "TestWorks");
SubmitForm(FORM:0:13, "SUBMIT FORM");
}
```

Figure 4. Sample of Output of FORM Test Wizard.

The idea is that this script can be processed automatically to produce the result of varying combinations of pushing buttons. As is clear, the wizard will have pushed all buttons, but only the last-applied one in a set of radio buttons will be left in the `TRUE` state.

- *Text Wizard.* For detailed content validation this wizard yields up a script that includes in confirmation of the entire text of the candidate page. This script is used to confirm that the content of a page has not changed (in effect, the entire text content of the subject is recorded in the script).

EXAMPLE USES

Early application of the eValid system have been very effective in producing experiments and collecting data that is very useful for WebSite checking. While we expect eValid to be the main engine for a range of WebSite quality control and testing activities, we've chosen two of the most typical -- and most important -- applications to illustrate how eValid can be used.

Performance Testing Illustration. To illustrate how eValid measures timing we have built a set of *Public Portal Performance Profile* TestSuites that have these features:

- *Top 20 Web Portals.* We selected 20 commonly available WebSites on which to measure response times. These are called the "P4" suites.
- *User Recording.* We recorded one user's excursion through these suites and saved that keysave file (playback script).
- *User Recording.* We played back the scripts on a 56 kbps modem so that we had a realistic comparison of how long it would take to make this very-full visit to our selected 20 portals.
- *P4 Timings.* We measured the elapsed time it took for this script to execute at various times during the day. The results from one typical day's executions showed a playback time range of from 457 secs. to 758 secs (i.e. from -19% of the average to +36% of the average playback time).
- *Second Layer Added.* We added to the base script a set of links to each page referenced on the same set of 20 WebSites. This yielded the P4+ suite that visist some 1573 separate pages, or around 78 per WebSite. The testsuite takes around 20,764 secs (~5 Hrs 45 mins) to execute, or an average of 1038 secs per WebSite. per WebSite).
- *Lessons Learned.* It is relatively easy to configure a sophisticated test script that visits many links in a realistic way, and provides realistic user-perceived timing data.

E-Commerce Illustration. This example shows a typical E-Commerce product ordering situation. The script automatically places an order and uses the Validate Selected Text sequence to confirm that the order was processed correctly. In a real-world example this is the equivalent of (i) selecting an item for the shopping basket, (ii) ordering it, and (iii) examining the confirmation page's order code to assure that the transaction was successful. (The final validation step of confirming that the ordered item was actually delivered to a specific address is also part of what eValid can do -- see below.)

- *Example Form.* We base this script on a sample page shown below. This page is intended to have a form that shows an ordering process. On the page the "Serial Number" is intended as a model of a credit card number.

Figure 5. Sample Input Form For E-Commerce Example.

- *Type-In with Code Number.* Starting with the FORM Wizard generated script, we modify it to include only the parts we want, and include the code number `8889999`.

- *Response File.* Once the playback presses the `SUBMIT` button the WebServer response page shows up, as shown below.

Figure 6. Response Page for E-Commerce Example.

- *Error Message Generated.* If the Cgi-Bin scripts make a mistake this will be caught during playback because the expected text `8889999` will no be present.

- *Completed TestScript.* Here is the complete testscript for eValid that illustrates this sequence of activities.

```
void name()
{
/*  Recording by eValid Ver. 3.0
      (c) Copyright 1999 by Software Research, Inc. */

InitLink("http://www.soft.c../../Products/Web/CAPBAK/example1/example1broken.html");
SelectOneRadio(FORM:1:0, "buying-now", "TRUE");
SelectOneRadio(FORM:1:1, "next-month", "FALSE");
SelectOneRadio(FORM:1:2, "just-looking", "FALSE");
SelectOneRadio(FORM:1:3, "no-interest", "FALSE");
SelectOneRadio(FORM:1:4, "Yes", "TRUE");
SelectOneRadio(FORM:1:5, "Yes", "TRUE");
SelectOneRadio(FORM:1:6, "Yes", "TRUE");
SelectOneRadio(FORM:1:7, "Yes", "TRUE");
FormTextInput(FORM:1:8, "Mr. Software");
FormTextInput(FORM:1:9, "415-550-3020");
FormTextInput(FORM:1:10, "info@soft.com");
FormTextInput(FORM:1:11, "8889999");
SubmitForm(FORM:1:13, "SUBMIT FORM");
Wait(3425);
ValidateText(12, 143, "88899999");
}
```

Figure 7. Script for E-Commerce Test Loop.

- *Lessons Learned.* This examples illustrates how it is possible to automatically validate a website using eValid by detecting when an artificial order is mis-processed.

FUTURE EXPANSION AND EXTENSIONS

We are presently using eValid in support of various customers' WebSite quality control activities. As rich as we believe our implementation of a test enabled web browser is with eValid there are many areas where there is need for expansion.

Obviously we need to expand the capability to the Mozilla class of browsers, and possibly others as well. And, certain of the user-control functions have to be refined to get the best use out of the product set.

SUMMARY

All of these needs and requirements impose constraints on the test automation tools used to confirm the quality and reliability of a WebSite. At the same time they present a real opportunity to amplify human tester/analyst capabilities. Better, more reliable WebSites should be the result.

REFERENCES AND RELATED MATERIAL

- This paper is based on many sources, but it relies heavily on a prior White Paper *The WebSite Quality Challenge*.
- You can learn more about the test system being described in a Tour of eValid

- There is detailed information about the P4 Examples and the P4+ Examples.
- You can study the E-commerce example described above by going to these URLs:

  `example1.html` Input Page :
  `example1.html` Response Page:

- Readers may also be interested in seeing one way that the eValid product is applied by considering subscribing to one of the family of eValid Test Services.