End User Programming for Web Users

Robert C. Miller
MIT Lab for Computer Science
200 Technology Square, Cambridge, MA 02139 USA
rcm@lcs.mit.edu
http://graphics.lcs.mit.edu/~rcm

Introduction

The World Wide Web is increasingly a focus of business and entertainment. Applications which formerly would have been designed for the desktop — calendars, travel reservation systems, purchasing systems, library card catalogs, map viewers, even games like crossword puzzles and Tetris — have made the transition to the Web, largely successfully.

Applications have moved to the Web for a number of reasons. First, and probably most important, web applications need no installation. Just click on a link, and you can use the application immediately. Bugs can be fixed and new features can be rolled out without requiring users to install upgrades or apply patches. Multiple platforms are also easier to support. Platform-independent standards, such as XHTML, DOM, ECMAScript (aka JavaScript), and CSS, make it possible to target a web application to any standards-compliant web browser, regardless of operating system or windowing environment.

The migration of applications to the Web opens up a new vista of opportunity for end user programming. Applications that would have been closed and uncustomizable on the desktop suddenly sprout numerous hooks for customization when implemented in a web browser. Structured displays are represented by machine-readable HTML. Commands are invoked by generic HTTP requests. Graphical layouts can be tailored by stylesheet rules.

Unfortunately, although web browsers have a long history of built-in scripting languages, these languages are not designed for the *end user* of a web application. Instead, languages like JavaScript and Curl [8] are aimed at *designers* of web applications. Granted, many web designers lack a traditional programming background, so they may be considered end users in that respect. But the needs of a designer, building an application from whole cloth, differ greatly from the needs of a user looking to tailor or script an existing application. Current web scripting languages do not serve those needs.

In this paper, I consider the problem of end-user automation of web applications. After covering background and related work, I will present several motivating examples, and distill from those examples some essential requirements on a programming system for web users. Preliminary steps toward such a system have been taken, and the resulting research prototype (LAPIS) will be briefly described. Finally, I will mention some of the hard problems that arise.

Background

Closed, uncustomizable applications have long been a bugaboo for end-user scripting on the desktop. Despite the long existence of scripting frameworks like AppleScript, OLE Automation, and Visual Basic for Applications, and exhortations by platform vendors to support them, many desktop applications still do not provide the hooks required for scripting. In a software development environment that demands tight development cycles and short times to market, scripting and customization get short shrift compared to more pressing concerns like feature set, performance, reliability, and usability.

When a desktop application fails to provide an application programming interface (API), the end user must resort to automating the user interface — a technique often called, somewhat derogatively, *screen scraping*. Cross-application macro recorders support this technique by recording the user's mouse movements and keystrokes, then playing them back by inserting simulated mouse and keyboard events in the system queue. Macro recorders have a serious flaw in that they can only simulate input; they have no way to read an application's display to extract information or condition their behavior on the application's state. Triggers [14] and VisMap [15] address this problem by interpreting the screen contents at a pixel level, but this approach is challenging to program and has so far been applied only to simple tasks.

Interpreting desktop application output is hard. Web applications, however, display their output in structured, machine-readable HTML, making screen scraping much easier. As a

result, web screen scraping abounds. Comparison-shopping sites, such as Priceline.com, use screen scraping behind the scenes to extract price information from online retailers. A Boston Red Sox fan used screen scraping to try to stuff the ballot box for baseball's 1999 All-Star Game ballot [2].

The component of a web screen scraper that interprets a web page and extracts information from it is called a *wrapper*. Many wrappers are written by hand in scripting languages like Perl or Python, but work has also been done on inducing wrappers from examples [7].

Most web screen scrapers are written in a scripting language that dwells *outside* the web browser, like Perl, Python, or WebL [4]. For an end-user, the distinction is significant. Cookies, authentication, session identifiers, plugins, user agents, client-side scripting, and proxies can all conspire to make the Web look different outside the web browser than inside. But perhaps the most telling difference, and the most intimidating one for an end user, is the simple fact that outside a web browser, a web page is just raw HTML. Even the most familiar web portal looks frighteningly complicated when viewed as HTML source.

Unfortunately, only a handful of systems have looked at putting end-user web automation into the browser, where it belongs. LiveAgent [5] is a macro recorder that can record and play back a sequence of browsing actions, using a local HTTP proxy to snoop on the user's actions. SPHINX [10] is a user-configurable web crawler that runs as a Java applet in the user's web browser, so that it would see the same pages seen by the user. TrIAS [1] constructs wrappers from examples given in a web browser.

Although JavaScript is primarily intended for site designers, end users can access it with *bookmarklets* [3]. A bookmarklet is a short piece of JavaScript code encoded as a URL and stored in a bookmark. When the user clicks on the bookmark, the JavaScript code runs on the current page. For example, here is a simple bookmarklet that changes the current page's background to white:

```
javascript:void(document.bgColor='white')
```

Bookmarklets can extract data from pages, change display properties, adjust window properties, and visit other sites. However, a bookmarklet must fit into a URL, strongly constraining its length and making it hard to read and modify.

Mozilla has brought some promising developments in browser-centric web automation [13]. All the "chrome" in Mozilla — the toolbars, panels, and dialog boxes that surround the browser itself — are specified in XUL, an XML-based user interface description language. A combination of XUL, JavaScript, and CSS is used to implement web screen scrapers directly in the browser. For example, when a Google search results page is displayed in the browser, Mozilla automatically parses it to present the results as a list of hyperlinks in the sidebar. Mozilla promises to be a powerful testbed for future research into end-user web automation.

Scenarios

For further motivation, let us consider some scenarios in which end-users of web applications would want scripting and customization. These scenarios offer concrete examples that guide the requirements to be discussed in the next section.

Scenario 1: Reviewing. Many conferences — including CHI — now use a web application to receive papers, distribute them to peer reviewers, and collect the reviews. A reviewer assigned 10 papers to read and review faces a lot of repetitive web browsing to download each paper, print it, and (later) upload a review for it. Some reviewing applications require the review to be submitted in a web form, so a review prepared off-line must be copied and pasted into the appropriate fields of the form. Tedious repetition is a strong argument for automation. Unfortunately, since the reviewing application is protected by authentication, a simple Perl script won't do the job.

Scenario 2: House hunting. Prospective home buyers in the US can use the Multiple Listings Service (MLS) to search for homes matching various criteria. A number of real estate companies now offer web interfaces that search the MLS (e.g., www.realtor.com). Interestingly, different MLS search interfaces provide different subsets of the available information, forcing a home buyer to search several sites to get a more complete picture. Furthermore, many location preferences that may be personally important to a home buyer cannot be specified in the search. If I buy this house, how far will I have to commute to my work? How far is the nearest grocery store, subway stop, or public park? How far is it from my mother's house? These questions can be answered by plugging the house address into an online map site (e.g., MapQuest).

Scenario 3: Book shopping. A voracious reader may frequently visit an online bookstore (e.g., Amazon.com) with a list of books to buy. A voracious reader on a budget, however, may want to check first whether any of the books are available in a local public or university library by searching its online catalog. This is a feature that Amazon is unlikely ever to offer.

Requirements

The scenarios above suggest a number of desirable criteria for an end-user web automation system.

Browser centricity. In these scenarios, the web browser is the center of the user's activity. Tasks interleave manual operations, such as logging in to a site, with automatable operations, such as downloading papers or searching for books. If the automation takes the user out of the browser, or digs below the familiar rendered world of the Web into raw HTML, the user's work flow is interrupted.

Data-parallel operations. Much of the repetitive activity in these scenarios revolves around sets of data items: papers to print, reviews to upload, houses to search, addresses to map, books to look up. The ability to apply an operation to multiple items at once would be extremely valuable in web browsing, just as it is in file managers, word processors, and drawing editors.

Cross-site scripting. The scenarios often require interacting with multiple web applications in the same task: e.g., multiple real estate sites, or a real estate site and a mapping site, or a bookstore and a library catalog. Instead of being confined to the environment of a single page, as JavaScript typically is, end-user automation must smoothly interact with multiple pages, extracting data from one page and using it in another.

Both manual and automatic invocation. Suppose the user creates a *distance-to-work* script that takes a house address as input and uses an online mapping site to compute how far the user would have to commute to work from that address. This script might be invoked in several ways. With manual invocation, the user selects a house (or list of houses) and triggers the script from a menu or toolbar. Bookmarklets support only manual invocation. With automatic invocation, on the other hand, the browser automatically runs the script on any page recognized as a list of houses. The resulting distances might be inserted in the house's description, or they might be used to filter the list of houses, hiding any that are farther than a given threshold. Mozilla's search sidebar uses automatic invocation; whenever it detects a Google search results page, it automatically parses the page and displays the results in the sidebar. Automatic invocation allows custom behavior to be injected into a web application in ways that were impossible with desktop applications.

Approach

The LAPIS research project at MIT is working toward this vision of end-user automation in the web browser. Our current prototype, LAPIS, is written entirely in Java. The LAPIS browser can display simple HTML, visit hyperlinks, and submit web forms, but it fails to support all the standards (such as cookies, JavaScript, CSS, and SSL) required by modern web applications. Work is underway to port some of the novel features of LAPIS into Mozilla, giving a much richer, standards-compliant testbed for web automation.

LAPIS is described in detail elsewhere [9]. Features that are most relevant to end user automation are highlighted below:

Pattern library. LAPIS includes an extensible library of patterns and parsers that can be referred to by simple names, such as Link, Paragraph, Button, or Table. An HTML parser is included in the library, naturally, but so are patterns for other common kinds of text structure, including dates, times, phone numbers, email addresses, URLs, etc. Wrappers for web sites, such as Google or Amazon, would naturally fall into the pattern library. A pattern library raises the abstraction level of data descriptions, so that when users think about identifying elements and extracting data from a web page, they can think in terms of *books* or *addresses* rather than low-level features of HTML. The LAPIS library is designed to be extended, and is language-independent in the sense that a library pattern can be implemented by an arbitrary kind of scanner — regular expression, context-free grammar, parser generator, neural network, or even a Turing-complete program.

Pattern language. Library patterns can be glued together with a pattern language called *text constraints*, which uses relational operators such as *before*, *after*, *in*, and *contains* to describe a set of regions in a page. The matches to a pattern are displayed as multiple selections, and editing commands can affect all selections at once [12]. LAPIS was designed with data-parallel operations in mind.

Command language. LAPIS has an embedded scripting language aimed at the end user, not the page designer. (Tcl was chosen as the scripting language, partly because of its syntactic simplicity and partly because a good pure-Java implementation was available. Tcl is also well-suited for interactive command execution.) Commands take patterns as arguments to indicate how to manipulate a web page. For example, the *keep* command extracts a set of regions matching a pattern; *delete* deletes the regions; *sort* sorts the regions in-place; and *replace* replaces each region with some replacement text, which may be a function of the original region. Other commands interact with the web page as a user would: *click* simulates a click on a hyperlink or form control matching a pattern, and *enter* places text in a form field. JavaScript can also access form controls, of course, but an important difference is that LAPIS patterns can be written without looking at the underlying HTML source, e.g.:

```
click {Link containing "Download this paper"}
click {Checkbox just after "Garage"}
```

Writing equivalent commands in JavaScript requires digging into the HTML source to find the names of the fields.

Browser shell. Instead of presenting the Tcl interpreter in a separate window, LAPIS integrates the interpreter directly into the browser window. Tcl commands may be typed into the Location box. The typed command is applied to the

current page, and the command's output is displayed in the browser as a new page that is added to the browsing history. A command may also invoke an external program, passing the current page as standard input and displaying the program's standard output and error streams as a new page. This "browser shell" interface [11] allows legacy programs and scripts written in other languages to be integrated seamlessly into the browser environment.

Challenges

The primary challenge for end-user automation in the web browser can be simply stated: the user should never have to view the HTML source of a web site to customize or automate it. Web sites are becoming increasingly complicated. Even when a web interaction could be scripted outside the browser (with no trouble from cookies, authentication, or dynamically-generated content), the need to examine and understand the HTML source is a roadblock that discourages spur-of-the-moment innovation. Web automation must be done at the level of rendered pages.

This problem is far from trivial. What the user sees as "blue text" in a rendered page may be blue for many reasons: because it is a hyperlink; because it is contained in a FONT tag; because it has a CSS style attribute; because it matched by a CSS stylesheet rule; or because its color attribute was set by some JavaScript code. Worst of all, the "blue text" may be only a picture of text, embedded in a GIF or JPG image! The text pattern matching approaches used for web screen scraping outside the browser no longer work in general.

An automation system must deal smoothly with the proliferation of Web standards and syntaxes — XHTML, XML, CSS, MathML, SVG — while hiding the distinctions between them from the user. It must be integrated with a fully standards-compliant web browser, so that the user's web applications are functional and usable. Where previous approaches used a web proxy to extend the browser (e.g., LiveAgent), embedding automation into the browser is more likely to achieve the desired results.

Another challenge facing end-user web automation, like all web screen scrapers, is dealing with changes in web applications. One of the benefits of web applications (for their designers) is that changes can be rolled out without notice to users, but this turns out to be detrimental to end-user automation. Some steps toward solving this problem include regression tests that can detect when a wrapper is going wrong [6] and intelligent agents that relearn failed wrappers with the user's help [1]. Web services with well-specified XML APIs will also help, although considering how few desktop applications have scriptable APIs, it is hard to be optimistic about web applications.

Acknowledgements

This research was sponsored in part by the National Science Foundation, the Army Research Office, the Defense Advanced Research Project Agency, and the USENIX Association. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any sponsoring party or the U.S. Government.

References

[1] Mathias Bauer, Dietmar Dengler, Gabriele Paul, and Markus Meyer. Programming by demonstration for information agents. *Communications of the ACM*, 43(3):98–103, March 2000.

[2] Gordon Edes. This hack tried but couldn't connect. *Boston Globe*, July 1999.

[3] Steve Kangas. Bookmarklets. http://www.bookmarklets.com/, 1998.

[4] Thomas Kistler and Hannes Marais. WebL – a programming language for the Web. In *Proceedings of the 7th International World Wide Web Conference (WWW7)*, 1998.

[5] Bruce Krulwich. Automating the internet: Agents as user surrogates. *IEEE Internet Computing*, 1(4):34–38, 1997.

[6] Nicholas Kushmerick. Wrapper verification. *World Wide Web*, 3(2):79–94, 2000.

[7] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper induction for information extraction. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.

[8] Friedger Müffke. The Curl programming environment. *Dr. Dobb's Journal*, September 2001.

[9] Robert C. Miller. *Lightweight Structure in Text*. PhD thesis, Carnegie Mellon University, May 2002.

[10] Robert C. Miller and Krishna Bharat. SPHINX: a framework for creating personal, site-specific web crawlers. *Computer Networks and ISDN Systems*, 30(1–7):119–130, 1998.

[11] Robert C. Miller and Brad A. Myers. Integrating a command shell into a web browser. In *USENIX 2000 Annual Technical Conference*, pages 171–182, June 2000.

[12] Robert C. Miller and Brad A. Myers. Multiple selections in smart text editing. In *Proceedings of the Sixth International Conference on Intelligent User Interfaces (IUI 2002)*, pages 103–110, 2002.

[13] Ian Oeschger, Eric Murphy, Brian King, Pete Collins, and David Boswell. *Creating Applications with Mozilla*. O'Reilly, 2002.

[14] Richard Potter. Triggers: Guiding automation with pixels to achieve data access. In Allen Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 360–380. MIT Press, 1993.

[15] Luke Zettlemoyer and Robert St. Amant. A visual medium for programmatic control of interactive applications. In *Proceedings of ACM Conference on Human Factors in Computer Systems (CHI '99)*, pages 199–206, 1999.