Automation and Customization of Rendered Web Pages

Michael Bolin, Matthew Webber, Philip Rha, Tom Wilson, and Robert C. Miller
MIT CSAIL
32 Vassar St
Cambridge, MA 02139 USA
{mbolin,wmatthew,prha,tom_w,rcm}@mit.edu

ABSTRACT
On the desktop, an application can expect to control its user interface down to the last pixel, but on the World Wide Web, a content provider has no control over how the client will view the page, once delivered to the browser. This creates an opportunity for end-users who want to automate and customize their web experiences, but the growing complexity of web pages and standards prevents most users from realizing this opportunity. We describe Chickenfoot, a programming system embedded in the Firefox web browser, which enables end-users to automate, customize, and integrate web applications without examining their source code. One way Chickenfoot addresses this goal is a novel technique for identifying page components by keyword pattern matching. We motivate this technique by studying how users name web page components, and present a heuristic keyword matching algorithm that identifies the desired component from the user's name.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces; D.3.3 [Programming Languages]: Language Constructs and Features; D.2.6 [Programming Environments]: Interactive environments; H.5.2 [User Interfaces]: User-centered design.

General terms: Design, Algorithms, Experimentation, Human Factors, Languages.

Keywords: Web automation, web browsers.

INTRODUCTION
The World Wide Web has become a preferred platform for many kinds of application development. Over the past decade, applications which formerly would have been designed for the desktop — calendars, travel reservation systems, purchasing systems, library card catalogs, map viewers, crossword puzzles, and even Tetris — have made the transition to the Web, largely successfully.

The migration of applications to the Web opens up new opportunities for user interface customization. Applications that would have been uncustomizable on the desktop sprout numerous hooks for customization when implemented in a web browser, without any effort on the application developer's part. Displays are represented primarily by machine-readable HTML or XML, navigation and commands are invoked by generic HTTP requests, and page designs can be tailored by stylesheets. Here are some of the customization possibilities that arise when an application is moved to the Web:

Automating repetitive operations. Web automation may include navigating pages, filling in forms, and clicking on links. For example, many conferences now use a web site to receive papers, distribute them to reviewers, and collect the reviews. A reviewer assigned 10 papers to read and review repetitive web browsing to download each paper, print it, and (later) upload a review for it. Tedious repetition is a good argument for automation. Other examples include submitting multiple queries and comparing the results, and collecting multiple pages of search results into a single page for easy printing or additional sorting and filtering.

Integrating multiple web sites. The simplest kind of integration is just adding links from one site to another [9], but much richer integration is possible. For example, many retailers' web sites incorporate maps and directions provided by a mapping service directly into their web pages, to display their store locations and provide driving directions. But end users have no control over this kind of integration. For example, before buying a book from an online bookstore, a user may want to know whether it is available in the local library, a question that can be answered by submitting a query to the library's online catalog interface. Yet the online bookstore is unlikely to provide this kind of integration, not only because it may lose sales, but because the choice of library is inherently local and personalized to the user.

Transforming a web site's appearance. Examples of this kind of customization include changing defaults for form fields, filtering or rearranging web page content, and changing fonts, colors, or element sizes. Web sites that use Cascading Style Sheets (CSS) have the potential to give the end user substantial control over how the site is displayed, since the user can override the presentation with personal stylesheet rules. With the exception of font preferences, however, current web browsers do not expose this capability in any usable way.

UIST'05, October 23–27, 2005, Seattle, Washington, USA.

These examples involve not only *automating* web user interfaces (clicking links, filling in forms, and extracting data) but also *customizing* them (changing appearance, rearranging components, and inserting or removing user interface widgets or data). The openness and flexibility of the web platform enables customizations that would not have been possible on the desktop.

Existing approaches to web automation use a scripting language that dwells outside the web browser, such as Perl, Python, or WebL [1]. 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 significantly different to an agent running outside the web browser. Recent highly-interactive web applications like Google Mail and Google Maps — sometimes called AJAX applications [13] because they use asynchronous JavaScript and XML — have made this situation worse.

But perhaps the most telling difference, and the most intimidating one to a 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.

Chickenfoot is a new programming system we are developing that provides a platform for automating and customizing web applications through a familiar interface – as web pages rendered in a web browser. The challenge for Chickenfoot is simply stated: a user should never have to view the HTML source of a web page in order to customize or automate it.

Chickenfoot addresses this challenge in three ways. First, it runs inside the web browser, so that the rendered view of a web page is always visible alongside the Chickenfoot development environment. Second, its language primitives are concerned with the web page's user interface, rather than its internal details. For example, Chickenfoot uses commands like `click`, `enter`, and `pick` to interact with forms. Third, it uses novel pattern-matching techniques to allow users to describe components of a web page (targets for interaction, extraction, insertion, or customization) in terms that make sense for the rendered view. For example, `click` identifies the button to be clicked using keywords from its text label, rather than the name it was given by the web page designer.

Chickenfoot is implemented as an extension for the Mozilla Firefox web browser, written in Java, JavaScript, and XUL. It consists of a development environment, which appears as a sidebar of Firefox, and a library built on top of JavaScript. Chickenfoot customizations are essentially JavaScript programs, so Chickenfoot currently does not support nonprogramming users. We assume that a Chickenfoot developer has *some* knowledge of JavaScript and HTML — not an unreasonable assumption, since many power users showed the ability and willingness to learn these during the explosive growth of the Web. The problem Chickenfoot is addressing is not learning JavaScript and HTML syntax, but rather reading and understanding the complex HTML used by today's web applications.

Naturally, many users would benefit from a web automation system that avoids the need to learn programming language syntax. We regard Chickenfoot as a step towards this goal, but a crucial one, since it provides a level of expressiveness and completeness unavailable in special-purpose web automation systems.

One system similar to Chickenfoot in implementation is Greasemonkey[1], a Firefox extension that can run user-written JavaScript on web pages just after they are loaded in the browser. Though Greasemonkey enables mutation and automation of web pages in the browser, it does not address the need to inspect the HTML of the page. Platypus[2] is another Firefox extension, designed to work with Greasemonkey, that allows some customization of rendered web pages, but not automation or integration of multiple web sites.

The rest of this paper is organized as follows. First we give an overview of the Chickenfoot language and development environment, and describe a range of applications we have built using Chickenfoot. Then we delve deeper into a novel aspect of Chickenfoot: the pattern matching used to identify web page elements for automation or customization. We describe a survey of web users that motivated the design of the pattern matching, and present the algorithm we developed as a result. Finally we review related work and make some conclusions.

CHICKENFOOT

Chickenfoot is an extension to the Mozilla Firefox web browser, consisting of a library that extends the browser's built-in JavaScript language with new commands for web automation, and a development environment that allows Chickenfoot programs to be entered and tested inside the web browser. This section describes the essential Chickenfoot commands, including pattern matching, form manipulation, page navigation, and page modification. The section concludes by describing the development environment (Figure 1).

Language

Chickenfoot programs are written in JavaScript, using the JavaScript 1.5 interpreter built into Mozilla Firefox. As a result, Chickenfoot users have access to the full expressiveness of a high-level scripting language, with a prototype-instance object system, lexically scoped procedures, dynamic typing, and a rich class library.

Because Chickenfoot uses JavaScript, web developers can easily transfer their knowledge of JavaScript from web page development over to Chickenfoot. Chickenfoot predefines the same variables available to JavaScript in web pages – e.g., `window`, `document`, `location`, `frames`, `history` –

[1] http://greasemonkey.mozdev.org

[2] http://platypus.mozdev.org

so that JavaScript code written for inclusion *inside* a web page can also be used as a Chickenfoot script *outside* the web page. JavaScript has its own ways of visiting new pages (`location`), manipulating form controls (`document.forms`), and modifying page content (using the Document Object Model, or DOM [12]). These mechanisms can be used by Chickenfoot scripts in addition to the Chickenfoot commands described in the next few sections. We have found that using the native JavaScript mechanisms generally require reading and understanding a web page's HTML source. But by providing access to them, Chickenfoot provides a smooth escape mechanism for script developers that need to do something more low-level.

All current web browsers, including Firefox, implement a security model for JavaScript to protect web users from malicious downloaded scripts. A major part of this security model is the *same-origin policy*, which prevents JavaScript code downloaded from one web server from manipulating a web page downloaded from a different server. This restriction is clearly too severe for Chickenfoot, since its primary purpose is integrating and customizing multiple web sites. As a result, Chickenfoot scripts run at a privileged level, where they have access to the entire web browser, all pages it visits, and the user's filesystem and network. Users must trust Chickenfoot code as much as they trust any other desktop application. As a result, Chickenfoot scripts cannot be embedded in downloadable web pages like other JavaScript. But Chickenfoot code *can* inject new behavior into downloaded pages, which is explained in more detail later.

Pattern Matching

Pattern matching is a fundamental operation in Chickenfoot. To operate on a web page component, most commands take a pattern describing that page component.

Chickenfoot supports two kinds of patterns: *keyword patterns* and *text constraint patterns*. A keyword pattern consists of a string of keywords that are searched in the page to locate a page component, followed by the type of the component to be found. For example, `"Search form"` matches a form containing the keyword Search, and `"Go button"` matches a button with the word "Go" in its label. The component type is one of a small set of primitive names, including `link`, `button`, `textbox`, `checkbox`, `radiobutton`, and `listbox`. When a keyword pattern is used by a form manipulation command, the type of page component is implicit and can be omitted. For example, `click("Go")` searches for a hyperlink or button with the keyword "Go" in its label. Case is not significant, so `click("go")` has the same effect.

A text constraint pattern combines a library of primitive patterns (such as `link`, `textbox`, or `paragraph`), literal strings (such as `"Go"`), and relational operators (e.g., `in`, `contains`, `just before`, `just after`, `starts`, `ends`). Text constraint patterns have been described in more detail previously [10]. Text constraint patterns are generally used

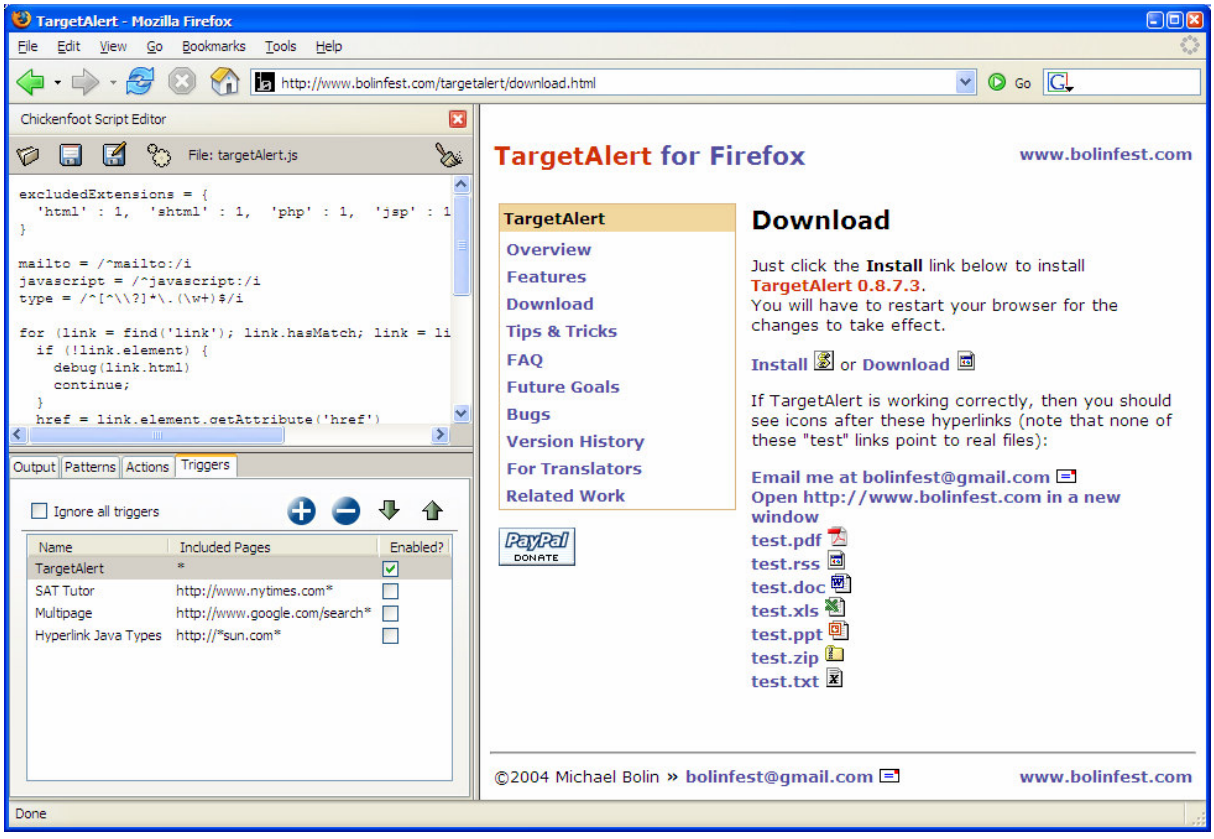


Figure 1. Chickenfoot development environment running inside the Firefox web browser.

to identify parts of a page for modification and extraction, although they can also be used for form manipulation.

The `find` command takes a pattern of either kind and searches for it in the current page, e.g.:

```
find("Search form")
find("link in bold")
```

`Find` returns a `Match` object which represents the first match to the pattern and provides access to the rest of the matches. Here are some common idioms using `find`:

```
// test whether a pattern matches
if (find(pattern).hasMatch) { ... }

// count number of matches
find(pattern).count

// iterate through all matches
for (m=find(pattern);
     m.hasMatch;
     m = m.next) { ... }
```

A `Match` object represents a contiguous region of a web page, so it also provides properties for extracting that region. If `m` is a `Match` object, then `m.text` returns the plain text it represents, i.e., the text that would be obtained by copying that region of the rendered web page and pasting it to a text editor that ignored formatting. Similarly, `m.html` returns the source HTML of the region, which is useful for extracting the region with formatting intact. Finally, `m.element` returns the DOM `Element` object represented by the region, if the region contains a single outermost element. This `element` can be used to get and set element attributes, e.g.:

```
find("link").element.href
```

The `find` command is not only a global procedure, but also a method of `Match` and `Document`. Invoking `find` on one of these objects constrains it to return only matches within the page or part of a page represented by the object. Here are some common idioms:

```
// nested finds
for (t=find("table");
     t.hasMatch;
     t = t.next) {
  r = t.find("row");
  ...
}

// find in a page not currently showing
otherDocument.find(pattern)
```

Clicking and Form Manipulation

The next few commands allow Chickenfoot scripts to click on hyperlinks and fill in and submit forms.

The `click` command takes a pattern describing a hyperlink or button on the current page and causes the same effect as if the user had clicked on it. For example, these commands click on various parts of the Google home page:

```
click("Advanced Search") // a hyperlink
click("I'm Feeling Lucky") // a button
```

Keyword patterns do not need to exactly match the label of the button or hyperlink, but they do need to be unambiguous. Thus, `click("Lucky")` would suffice to match the I'm Feeling Lucky button, but in this case, `click("Search")` would be ambiguous between the Google Search button and the Advanced Search link, and hence would throw an exception. (Exact matches take precedence over partial matches, however, so if there were a single button labeled "Search", then the `click` command would succeed.) Buttons and links labeled by an image can be matched by keywords mentioned in their ALT text, if any. The keyword matching algorithm is described in more detail later in this paper.

The `click` command can take a `Match` object instead of a pattern, if the button or hyperlink to be clicked has already been found. For example, to identify a button using a text constraint pattern, the user might write this:

```
click(find("button just after textbox"))
```

The `enter` command enters a value into a textbox. Like `click`, it takes a keyword pattern to identify the textbox, but in this case, the keywords are taken from the textbox's caption or other visible labels near the textbox. For example, to interact with the Amazon login page, a script might say:

```
enter("e-mail address", "rcm@mit.edu")
enter("password", password)
```

When the page contains only one textbox, which is often true for search forms, the keyword pattern can be omitted. For example, this sequence does a search on Google:

```
enter("uist 2005")
click("Google Search")
```

Checkboxes and radio buttons are controlled by the `check` and `uncheck` commands, which take a keyword pattern that describes the checkbox:

```
check("Yes, I have a password")
uncheck("Remember Me")
```

Finally, the `pick` command makes a selection from a listbox or drop-down box (which are both instantiations of the HTML `<select>` element). The simplest form of `pick` merely identifies the choice by a keyword pattern:

```
pick("California")
```

If only one choice in any listbox or drop-down on the page matches the keywords (the common case), then that choice is made. If the choice is not unique, `pick` can take two keyword patterns, the first identifying a listbox or drop-down by keywords from its caption, and the second identifying the choice within the listbox:

```
pick("State", "California")
```

Like `find`, all the clicking and form manipulation commands are also methods of `Match` and `Document`, so that the keyword search can be constrained to a particular part of a page:

```
f = find("Directions form")
f.enter("address", "32 Vassar St")
f.enter("zip", "02139")
f.click("Get Directions")
```

The form manipulation commands described so far permit setting the value of a form widget, which is the most common case for web automation. To read the current value of a widget, a script can use `find` to locate it, and then access the value of its Element object in conventional JavaScript fashion, e.g.:

```
find("address textbox").element.value
```

Chickenfoot also provides a `reset` command which resets a form to its default values, though `reset` is rarely needed.

Navigation and Page Loading

In addition to loading pages by clicking on links and submitting forms, Chickenfoot scripts can jump directly to a URL using the `go` command:

```
go("http://www.google.com")
```

If the string is not a valid URL, `go` automatically prefixes it with "http://".

Conventional browser navigation operations are also available as Chickenfoot commands: `back`, `forward`, and `reload`.

To retrieve a page without displaying it, the `fetch` command can be used. It returns a `Document` object representing the invisible page:

```
google = fetch("www.google.com")
```

The JavaScript `with` statement is convenient for performing a sequence of operations on an invisible page, by implicitly setting the context for Chickenfoot pattern matching and form manipulation:

```
with (fetch("www.google.com")) {
  enter("syzygy")
  click("Google Search")
  n = find("number just after
about").text
}
```

Pages retrieved by `fetch`, `go`, or `click` are loaded asynchronously by the browser, while the Chickenfoot script continues to run. Thus, a script can fire off several `fetch` requests in parallel, without forcing each request to complete before the next one starts. When a subsequent Chickenfoot command needs to access the content of a page, such as `find`, the command automatically blocks until the page is fully loaded. The `wait` and `ready` commands make this blocking available to programmatic control. Both commands take a `Document` object or an array of `Documents` as an argument. With no arguments, the default is the current page. `Wait` blocks until at least one of the specified pages is fully loaded, and returns that page. `Ready` returns a loaded page only if it has already completed, otherwise it immediately returns `null`.

Page Modification

Chickenfoot offers three primitive commands for changing the content of web pages: `insert`, `remove`, and `replace`.

The `insert` command takes two arguments: a location on a page and a fragment of web page content that should be inserted at that location. In its simplest form, the location is a text constraint pattern, and the web page content is simply a string of HTML:

```
insert("just before textbox",
       "<b>Search: </b>")
```

The location can also be derived from a `Match` object, but it must represent a single point in the page, not a range of content. The `before` and `after` commands can be used to reduce one of these objects to a point:

```
t = find("textbox")
insert(after(t), "<b>Search: </b>")
```

The page content to be inserted can also be a `Match` object, allowing content to be extracted from another page and inserted in this one:

```
map = mapquest.find("image")
insert("just after Directions", map)
```

The `remove` command removes page content identified by its argument, which can be a text constraint pattern or `Match` object. For example:

```
remove("Sponsored Links cell")
```

The `replace` command replaces one chunk of page content with another. It is often used to wrap page content around an existing element:

```
keyword = find("syzygy")
replace(keyword, "<b>"+keyword+"</b>")
```

Widgets

When a Chickenfoot script needs to present a user interface, it can create links and buttons and insert them directly into a web page. Hyperlinks are created by the `Link` constructor, which takes a chunk of HTML to display inside the hyperlink and an event handler to run when the link is clicked:

```
new Link("<b>Show All</b>", showAll)
```

The event handler can be either a string of Chickenfoot code (like the `onclick` attribute in HTML) or a JavaScript Function object. Buttons are created similarly by the `Button` constructor.

Other widgets can be created by inserting HTML, e.g.:

```
insert(..., "<input type=checkbox>")
```

If an `onclick` attribute is included in this HTML element, however, the code it contains will execute like conventional, untrusted JavaScript code downloaded with the page. Commands defined by Chickenfoot would be unavailable to it. To add a Chickenfoot handler to a button created from HTML, one can use the `onClick` command:

```
onClick(button, showAll)
```

Development Environment

Figure 1 shows a screenshot of the development environment presented by the current Chickenfoot prototype, which appears as a sidebar in Firefox. At the top of the sidebar is a text editor used to compose Chickenfoot code, which may be merely a single expression or command to execute, or a larger program with function and class definitions. This simple interface goes a long way toward making the JavaScript interpreter embedded in every web browser ac-

cessible to the user. Previously, there were only two ways to run JavaScript in a web browser: by embedding it in a web page (generally impossible if the page is fetched from a remote web site, since the user can't edit it), or by using a *javascript:* URL, which requires the entire program to be written on a single line. The Chickenfoot sidebar makes it much easier for an end-user to write and run scripts.

The bottom of the sidebar has four tabbed panels. First, the Output panel is an output window that displays error messages, output printed by the Chickenfoot `output` command, and the result of evaluating the Chickenfoot script (i.e., the value of the last expression).

The Actions panel (not yet implemented) will display a history of browsing actions performed by the user, constantly updated as the user browses the web. This panel will enable self-disclosure [3], displaying the Chickenfoot commands that would reproduce the user's actions. A manual browsing sequence can be used as the basis for a script by selecting it in the Actions pane and copying it into the script editor.

The Patterns panel displays an interface for developing text constraint patterns, which allows the user to type in a pattern and see what it matches in the current page. This pane also displays the primitive patterns available in the text constraints pattern library.

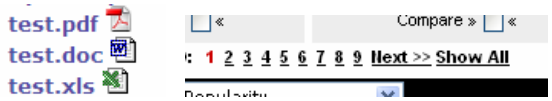
Finally, the Triggers panel allows a Chickenfoot script to be installed into the browser for regular use. For manual invocation, a script can be installed as a bookmark, which appears like any other bookmark in the Bookmarks menu and toolbar. For automatic invocation, a script can be associated with a *trigger*, which is a URL pattern such as `http://www.amazon.com/*`. Whenever a page is loaded, if its URL matches a trigger, then the associated script executes automatically. If a page matches multiple triggers, the associated scripts execute in the fixed order given by the Triggers pane. The Triggers pane provides an interface for adding and removing triggered scripts, temporarily disabling triggers, and viewing errors in automatically triggered scripts.

APPLICATIONS

This section describes a few of the example scripts we have created using Chickenfoot.

Adding File Type Icons to Links

When a hyperlink points at a resource other than a web page (such as a PDF document, a ZIP archive, or a Word document), it is often helpful for the link to be visually distinguished – first, because the user may be actively scanning the page for one of these resources, and second, because they may want to avoid them while casually surfing. Only a few web sites provide a visual cue to the file type of a hyperlink. TargetAlert is a Firefox browser extension, developed by the first author, that adds file type icons to hyperlinks on any web site.



(a) (b)

Figure 2. Chickenfoot examples: (a) file type icons added to links, and (b) "Show All" link for page concatenation.

TargetAlert was originally written in 217 lines of JavaScript and XUL. We rewrote it in 29 lines of Chickenfoot. The essence of the script is the following loop:

```
for (l=find('link'); l.hasMatch; l=l.next)
{
  href = l.element.getAttribute('href')
  if (m = href.match(/\.(\w+)$/)) {
    extension = m[1]
    src = 'moz-icon://.' + extension
          + '?size=16';
    insert(after(l),
           ' <img src="' + src + '"> ')
  }
}
```

To simplify retrieving file type icons, the script exploits a feature of Firefox that works only on Windows: URLs of the form moz-icon://*.ext*?size=16 return the icon associated with file extension *.ext* in the Windows registry. Firefox normally uses these URLs to display local directories in the browser. The result of running this script on part of a web page is shown in Figure 2a.

Sorting Tables

Another feature that some web sites have, but many lack, is the ability to sort a table of data by clicking one of its column headers. A Chickenfoot script can add this functionality automatically to most tables, by replacing every table header cell it finds with a link that sorts the table.

Most of the script is concerned with managing the sort, but here is the part that replaces headers with links:

```
column=0
for (h=table.find('text in cell in first
row'); h.hasMatch; h=h.next) {
  var f = makeRowSorter(table,column++)
  replace(h, new Link(h, f))
}
```

The makeRowSorter function returns a function that sorts the specified table by the specified column number.

Concatenating a Sequence of Pages

Search results and long articles are often split into multiple web pages, mainly for faster downloading. This can inhibit fluid browsing, however, because the entire content isn't accessible to scrolling or to the browser's internal Find command. Some articles offer a link to the complete content, intended for printing, but this page may lack other useful navigation.

We have written a Chickenfoot script that detects a multi-page sequence by searching for its table of contents (generally a set of numbered page links, with Next and Previous).

When a table of contents is found, the script automatically adds a "Show All" link to it (Figure 2b). Clicking this link causes the script to start retrieving additional pages from the sequence, appending them to the current page. In order to avoid repeating common elements from subsequent pages (such as banners, sidebars, and other decoration), the script uses a conservative heuristic to localize the content, by searching for the smallest HTML element that contains the list of page links (since the content is nearly always adjacent to this list) and spans at least half the height of the rendered page (since the content nearly always occupies the majority of the page). The content element from each subsequent page is inserted after the content element of the current page.

Adjusting Contrast

An unfortunate number of web pages make poor color choices, making the text hard to read even for users with good eyesight – and all but impossible for users with reduced vision. A simple Chickenfoot script can reset all the text colors to black and white for easier reading:

```
for (e=find('element contains text');
     e.hasMatch;
     e=e.next) {
  e.element.style.color = 'black';
  e.element.style.background = 'white';
  e.element.style.backgroundImage='none';
}
```

Coloring Java Syntax and Linking to Documentation

The text constraint patterns used by Chickenfoot can draw on the rich library of patterns and parsers implemented in LAPIS [10]. In the library is a Java parser, which is capable of finding and parsing Java syntax even if it is embedded in a web page. One of our Chickenfoot scripts uses this parser for coloring embedded Java syntax, e.g.:

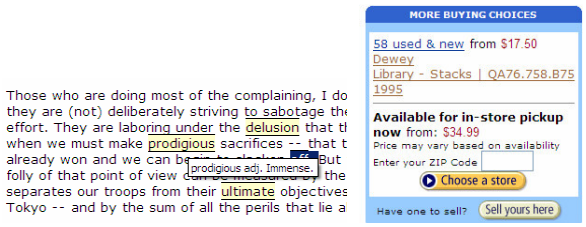
```
for (c = find('Java.Comment');
     c.hasMatch;
     c = c.next) {
  replace(c, "<span style='color:green'>"
             + c + "</span>")
}
```

The script also links each occurrence of a class name to its Javadoc documentation:

```
for (c = find('Java.Type');
     c.hasMatch;
     c = c.next) {
  if (c.text in classURL) {
    replace(c, "<a href='"
             + classURL[c.text] + "'>"
             + c.text + "</a>")
  }
}
```

The classURL array maps a class name (e.g. "String") to the URL of its documentation, which another part of the script extracts from Sun's web site:

```
go("java.sun.com/j2se/1.5.0/docs/api")
click("No Frames")
click("All Classes")
for (l=find('link');l.hasMatch;l=l.next){
  classURL[l.text] = l.element.href
}
```



(a) (b)

Figure 3. Chickenfoot examples: (a) highlighting vocabulary words, and (b) integrating a link to a library into a bookstore page.

This simple script ignores the problem of ambiguous class names (java.util.List and java.awt.List) and only uses the standard Java class library, but it would be straightforward to extend it to deal with these problems.

Highlighting Vocabulary Words

Students studying for college placement exams, such as the SAT, often work hard to expand their vocabulary. One way to make this learning deeper is to highlight vocabulary words while the student is reading, so that the context of use reinforces the word's meaning. One of our Chickenfoot scripts takes a list of vocabulary words and definitions (posted on the web) and automatically highlights matching words in any page that the user browses. The script uses a title attribute to pop up the word's definition as a tooltip if the mouse hovers over it:

```
for (w=find('word');w.hasMatch;w=w.next){
  if (w.text in vocab) {
    html = '<span style="background-
color: yellow" title="'
              + vocab[w.text] + '">'
          + w + '</span>'
    replace(word, html)
  }
}
```

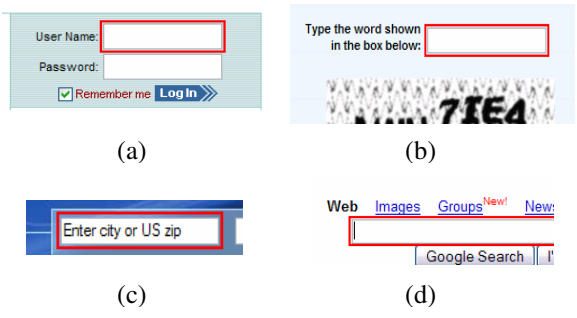
Integrating a Bookstore and a Library

The last example is a short script that augments book pages found in Amazon with a link that points to the book's location in the MIT library:

```
isbn = find('number just after isbn')
with (fetch('libraries.mit.edu')) {
  pick('Keywords');
  enter(isbn)
  click('Search')
  link=find('link just after Location')
}
// back to Amazon
if (link.hasMatch) {
  insert(before('first rule after "Buying
Choices"'), link.html)
}
```

USER STUDY OF KEYWORD MATCHING

One of the novel aspects of Chickenfoot is the use of keyword patterns to identify page elements, such as "Search button" and "address textbox". To explore the usability of this technique, we conducted a small study to learn what kinds of keyword patterns users would generate for one kind of page component (textboxes), and whether



(a) (b) (c) (d)

Figure 4. Some of the textboxes used in the web survey.

users could comprehend a keyword pattern by locating the textbox it was meant to identify.

Method

The study was administered over the Web. It consisted of three parts, always in the same sequence. Part 1 explored freeform generation of names: given no constraints, what names would users generate? Each task in Part 1 showed a screenshot of a web page with one textbox highlighted in red, and asked the user to supply a name that "uniquely identified" the highlighted textbox. Users were explicitly told that spaces in names were acceptable. Part 2 tested comprehension of names that we generated from visible labels. Each task in Part 2 presented a name and a screenshot of a web page, and asked the user to click on the textbox identified by the given name. Part 3 repeated Part 1 (using fresh web pages), but also required the name to be composed only of "words you see in the picture" or "numbers" (so that ambiguous names could be made unique by counting, e.g. "2nd Month").

The whole study used 20 web pages: 6 pages in Part 1, 8 in Part 2, and 6 in Part 3. The web pages were taken from popular sites, such as the Wall Street Journal, the Weather Channel, Google, AOL, MapQuest, and Amazon. Pages were selected to reflect the diversity of textbox labeling seen across the Web, including simple captions (Figure 4a), wordy captions (b), captions displayed as default values for the textbox (c), and missing captions (d). Several of the pages also posed ambiguity problems, such as multiple textboxes with similar or identical captions.

Subjects were unpaid volunteers recruited from the university campus by mailing lists. Forty subjects participated (20 females, 20 males), including both programmers and non-programmers (24 reported their programming experience as "some" or "lots", 15 as "little" or "none", meaning at most one programming class). All but one subject were experienced web users, reporting that they used the Web at least several times a week.

Results

We analyzed Part 1 by classifying each name generated by a user into one of four categories: (1) *visible* if the name used only words that were visible somewhere on the web page (e.g., "User name" for Figure 4a); (2) *semantic* if at least one word in the name was not found on the page, but was semantically relevant to the domain (e.g., "login name"); (3) *layout* if the name referred to the textbox's position on the page rather than its semantics (e.g., "top box right hand side"); and (4) *example* if the user used an example of a possible value for the textbox (e.g. "johnsmith056"). About a third of the names included words describing the type of the page object, such as "field", "box", "entry", and "selection"; we ignored these when classifying a name.

Two users consistently used *example* names throughout Part 1; no other users did. (It is possible these users misunderstood the directions, but since the study was conducted anonymously over the Web, it was hard to ask them.) Similarly, one user used *layout* names consistently in Part 1, and no others did. The remaining 37 users generated either *visible* or *semantic* names. When the textbox had an explicit, concise caption, *visible* names dominated strongly (e.g., 31 out of 40 names for Figure 4a were visible). When the textbox had a wordy caption, users tended to seek a more concise name (so only 6 out of 40 names for Figure 4b were visible). Even when a caption was missing, however, the words on the page exerted some effect on users' naming (so 12 out of 40 names for Figure 4d were visible).

Part 2 found that users could flawlessly find the textbox associated with a visible name, as long as the name was unambiguous. When a name was potentially ambiguous, users tended to resolve the ambiguity by choosing the first likely match found in a visual scan of the page. When the ambiguity was caused by both visible matching and semantic matching, however, users tended to prefer the visible match: given "City" as the target name for Go.com, 36 out of 40 users chose one of the two textboxes explicitly labeled "City"; the remaining 4 users chose the "Zip code" textbox, a semantic match that appears higher on the page. The user's visual scan also did not always proceed from top to bottom; given "First Search" as the target name for eBay.com, most users picked the search box in the middle of the page, rather than the search box tucked away in the upper right corner.

Part 3's names were almost all visible (235 names out of 240), since the directions requested only words from the page. Even in visible naming, however, users rarely reproduced a caption exactly; they would change capitalization, transpose words (writing "web search" when the caption read "Search the Web"), and mistype words. Some Part 3 answers also included the type of the page object ("box", "entry", "field"). When asked to name a textbox which had an ambiguous caption (e.g. "Search" on a page with more than one search form), most users noticed the ambiguity and tried to resolve it with one of two approaches: either counting occurrences ("search 2") or referring to other nearby captions, such as section headings ("search products").

KEYWORD PATTERN MATCHING ALGORITHM

We now describe the heuristic algorithm that resolves a keyword pattern to a web page component, such as a textbox. Given a name and a web page, the output of the algorithm is one of the following: (1) a textbox on the page that best matches that name; (2) *ambiguous match* if two or more textboxes are considered equally good matches; or (3) *no match* if no suitable match can be found.

The first step is to identify the text labels in the page that approximately match the provided name, where a *label* is a visible string of content delimited by block-level tags (e.g. <P>,
, <TD>). Button labels and ALT attributes on images are also treated as visible labels. Before comparison, both the name and the visible labels are normalized by eliminating capitalization, punctuation, and white space. Then each label is searched for an approximate occurrence of the name, using a conventional edit distance algorithm to tolerate typos and omitted words. Matching labels are ranked by edit distance, so that closer matches are ranked higher.

For each matching label, we search the web page for textboxes that it might identify. Any textbox that is roughly aligned with the label (so that extending the textbox area horizontally or vertically would intersect the label's bounding box) is paired with the label to produce a candidate *(label,textbox)* pair.

These pairs are further scored by several heuristics that measure the degree of association between the label and the textbox. The first heuristic is pixel distance: if the label is too far from the textbox, the pair is eliminated from consideration. Currently, we use a vertical threshold of 1.5 times the height of the textbox, but no horizontal threshold, since tabular form layouts often create large horizontal gaps between captions and their textboxes. The second heuristic is relative position: if the label appears below or to the right of the textbox, the rank of the pair is decreased, since these are unusual places for a caption. (We don't completely rule them out, however, because users sometimes use the label of a nearby button, such as "Search", to describe a textbox, and the button may be below or to the right of the textbox.) The final heuristic is distance in the document tree: each (*label,textbox*) pair is scored by the length of the shortest path through the DOM tree from the *label* node to the *textbox* node. Thus labels and textboxes that are siblings in the tree have the highest degree of association.

The result is a ranked list of (*label, textbox*) pairs. The algorithm returns the textbox of the highest-ranked pair, unless the top two pairs have the same score, in which case it returns *ambiguous match*. If the list of pairs is empty, it returns *no match*.

We evaluated this algorithm on the 240 names (40 for each of 6 pages) generated by Part 3 of the study. Its performance is shown in Figure 5. For each name, Chickenfoot either found the right textbox (*Match*), reported an ambiguous match (*Ambiguous*), or returned the wrong textbox (*Mismatch*). Precision is high for 5 of the 6 pages. Performance is poor on the MIT page because it involved an ambiguous caption, and our heuristic algorithm does not yet recognize the disambiguation strategies used for this caption (counting and section headings).

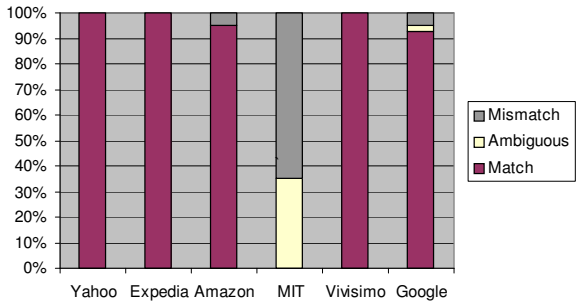


Figure 5. Precision of keyword patterns.

This evaluation is only preliminary; a proper evaluation should use a larger selection of web sites. Nevertheless, it suggests that keyword patterns can be automatically resolved with high precision.

RELATED WORK

Several systems have addressed specific tasks in web automation and customization, including adding links [9], building custom portals [14], crawling web sites [15], and making multiple alternative queries [2]. Chickenfoot is a more general toolkit for web automation and customization, which can address these tasks and others as well.

One form of general web automation can be found in scripting language libraries such as Perl's WWW::Mechanize or Python's ClientForm. These kinds of scripts run outside the web browser, where they cannot easily access pages that require session identifiers, secure logins, cookie state, or client-side JavaScript to run.

In an attempt to access these "hard-to-reach pages" [4], some systems give the user the ability to record macros in the web browser, where the user records the actions taken to require access to a particular page, such as filling out forms and clicking on links. Later, the user can play the macro back to automate access to the same page. LiveAgent [7] takes this approach, recording macros with a proxy that sits between the user's browser and the Web. The proxy augments pages with hidden frames and event handlers to capture the user's input, and uses this information to play back the recording later. Unfortunately, the proxy approach is also limited – for example, pages viewed over a secure connection cannot be seen, or automated, by the proxy. WebVCR [4] is another macro recorder for web navigation, which skirts the proxy problem by using a signed Java applet to detect page loads and LiveConnect [8] to instrument the page with event-capturing JavaScript after the page loads. Because part of WebVCR runs as an applet inside the browser, it can record all types of navigation. But neither LiveAgent nor WebVCR enable the user to modify the pages being viewed.

Toolkits such as WBI [5] and Greasemonkey focus on giving the user the ability to modify pages before, or just after, they are loaded in the user's web browser. WBI uses a proxy to intercept page requests, letting user-authored Java code mutate either the request or the resulting page. Giving users the ability to automate pages with Java and all its libraries is a powerful tool; however, WBI is still hampered by the limitations of a proxy.

Though both WBI and Greasemonkey enable the user to mutate pages, neither of them eliminates the need to inspect the HTML of the page to mutate it. For example, the sample scripts on the Greasemonkey site are full of XPath patterns [11] that identify locations in web pages. These scripts are difficult to create because they require the author to plumb through potentially messy HTML to find the XPath, and they are difficult to maintain because they are not resilient to changes in the web site. Chickenfoot avoids this problem by giving users a high-level pattern language based on keyword matching that enables the user to identify pages without knowledge of the page's HTML structure, facilitating development and increasing readability and robustness. In Chickenfoot, users can fall back on JavaScript to manipulate a page's DOM or XPath expressions to select elements; however, they are not restricted to these tools.

WebL [1], a programming language for the Web, also focused on giving users a higher-level language to describe Web page elements. In WebL, the user provides names of HTML elements to create *piece-sets*, where a *piece-set* is a set of *piece* objects, and a *piece* is a contiguous text region in a document. WebL provides various methods to combine *piece-sets* called *operators*, including set operators such as `union` and `intersection`, positional operators such as `before` and `after`, and hierarchical operators such as `in` and `contain`. Although these operators help produce more readable scripts, the language does not eliminate the need to inspect a web page for the names of its HTML elements, as the user must provide those to construct the basic pieces on which the operators work.

Another drawback of WebL, and of most of the aforementioned tools (with the exception of the macro recorders), is that they do not allow scripts to be developed inside the web browser. We consider the ability to experiment with a web site from the script development environment one of the greatest advantages of Chickenfoot – the user does not have to wait to see how it will affect the appearance of the web page, because Chickenfoot gives immediate feedback on the rendered page. LAPIS [10], a predecessor of Chickenfoot, took a similar approach, giving the user an interactive environment in which to experiment with pattern matching and web automation. Unfortunately, the LAPIS web browser does not support web standards like JavaScript, cookies, and secure connections, so it fails to provide the user with a complete web experience.

CONCLUSION, STATUS, AND FUTURE WORK

Chickenfoot is a programming system for web automation, integrated into the Firefox web browser. Chickenfoot enables the user to customize and automate web pages without inspecting their source, using keyword pattern matching to name page components. We showed that keyword patterns correspond closely to the names users actually generate for page components, and we presented a heuristic algorithm that implements keyword matching. Future work includes more demonstrational programming (such as the Actions pane) and techniques for detecting changes to web applications that break Chickenfoot scripts and helping the user fix them. We are also working to unify the two kinds of pattern matching (keywords and text constraints), to make the language simpler and more uniform.

Chickenfoot is under active development. The latest version is available at `www.bolinfest.com/chickenfoot`.

ACKNOWLEDGMENTS

We thank all the users who participated in our web study, as well as Maya Dobuzhskaya, Vineet Sinha, ChongMeng Chow, David Huynh, Marcos Ojeda, and other members of the UID group who provided valuable feedback on the ideas in this paper. This work was supported in part by the National Science Foundation under award number IIS-0447800. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] Kistler, T. and Marais, H. "WebL – a programming language for the Web." *Proc. WWW7*, 1998, pp. 259–270.

[2] Fujima, J., Lunzer, A., Hornbæk, K., Tanaka, Y. "Clip, connect, clone: combining application elements to build custom interfaces for information access." *Proc. UIST 2004*, pp. 175–184.

[3] DiGiano, C. and Eisenberg, M. "Self-disclosing design tools: a gentle introduction to end-user programming." *Proc. DIS '95*, pp. 189–197.

[4] Anupam, V., Freire, J., Kumar, B., and Lieuwen, D. "Automating web navigation with the WebVCR." *Proc. WWW9*, 2000, pp 503–517.

[5] Barrett, R., Maglio, P., and Kellem, D.. "How to Personalize the Web." *Proc. CHI*, 1997, pp. 75–82.

[6] Ekiwi, LLC. "Screen-scraper: solutions for web data extraction. " www.screen-scraper.com/

[7] Krulwich, B. "Automating the Internet: Agents as User Surrogates." *IEEE Internet Computing*, v1 n4 (July/August 1997), pp. 34–38.

[8] Flanagan, D. *JavaScript: The Definitive Guide.* O'Reilly, 2001.

[9] Microsoft. "Smart Tags and Smart Documents." msdn.microsoft.com/office/understanding/smarttags/default.aspx

[10] Miller, R.C. and Myers, B.A. "Integrating a Command Shell into a Web Browser." *Proc. USENIX*, 2000, pp. 171–182.

[11] W3C. "XML Path language (XPath) Version 1.0," 1999.

[12] W3C. "Document Object Model (DOM)." ww.w3.org/DOM/.

[13] Garrett, J. "Ajax: A New Approach to Web Applications." www.adaptivepath.com/publications/essays/archives/000385.php

[14] Sugiura, A. and Koseki, Y. "Internet Scrapbook: Automating web browsing tasks by demonstration." *Proc. UIST '98*, pp. 9–18.

[15] Miller, R.C. and Bharat, K. "SPHINX: a Framework for Creating Personal, Site-Specific Web Crawlers." *Proc. WWW7*, 1998, pp. 119–130.