Proceedings of 2000 USENIX Annual Technical Conference

San Diego, California, USA, June 18–23, 2000

INTEGRATING A COMMAND SHELL INTO A WEB BROWSER

Robert C. Miller and Brad A. Myers

Integrating a Command Shell Into a Web Browser

Robert C. Miller and Brad A. Myers
Carnegie Mellon University
{rcm,bam}@cs.cmu.edu

Abstract

The transition from command-line interfaces to graphical interfaces has resulted in programs that are easier to learn and use, but harder to automate and reuse. Another transition is now underway, to HTML interfaces hosted by a web browser. To help users automate HTML interfaces, we propose the *browser-shell*, a web browser that integrates a command interpreter into the browser's Location box. The browser-shell's command language is designed for extracting and manipulating HTML and text, and commands can also invoke local programs. Command input is drawn from the current browser page, and command output is displayed as a new page. The browser-shell brings to web browsing many advantages of the Unix shell, including scripting web services and creating pipelines of web services and local programs. A browser-shell also allows legacy command-line programs to be wrapped with an HTML/CGI interface that is graphical but still scriptable, and offers a new shell interaction model, different from the conventional typescript model, which may improve usability in some respects.

1 Introduction

The transition from command-line interfaces to graphical interfaces carries with it a significant cost. In the Unix shell, for example, programs accept plain text as input and generate plain text as output. This makes it easy to write scripts that automate user interaction. An expert Unix user can create sophisticated programs on the spur of the moment, by hooking together simpler programs with pipelines and command substitution. For example:

```
kill ‘ps ax | grep xterm | awk '{print $1;}'‘
```

This command uses `ps` to list information about running processes, `grep` to find just the `xterm` processes, `awk` to select just the process identifiers, and finally `kill` to kill those processes.

These capabilities are lost in the transition to a graphical user interface (GUI). GUI programs accept mouse clicks and keystrokes as input and generate raster graphics as output. Automating graphical interfaces is hard, unfortunately, because mouse clicks and pixels are too low-level for effective automation and interprocess communication. Attempts to introduce Unix shell features like pipelining into graphical user interfaces [3, 6, 7, 8, 15, 16] have been unsuccessful, largely because they were not integrated well with existing applications, required extra work from application developers to expose hooks and programming interfaces, or were too hard to use.

With the advent of the World Wide Web, another transition is underway, this time to distributed web applications that run on a web server and interact with the user through a web browser. Most web services accept input from HTML forms and generate output as HTML pages. Since HTML is textual and capable of being parsed and manipulated, we have the opportunity to recover some of the interactive automation capabilities that were available in the Unix shell, but missing in graphical interfaces. Consider the following web-browsing tasks that could be partially or totally automated:

- Download and print a group of links on a page;
- Compare airfares and schedules for several choices of departure and arrival dates;
- Look up a colleague in the online university phonebook, obtain a home address, locate the address on a map, and get driving directions;
- Given a list of books to read, search for each book in the local library catalog, and if the book is not on the shelves, buy it from an online bookstore;
- Make a smart alarm clock that announces the current temperature from an online weather report, and the time until the next bus departs from an online schedule, while you dress in the morning.

As a step towards automating these tasks and others, we have extended a web browser in several ways:

1. *Embedding a pattern language for matching text and HTML, and a suite of text-processing tools for*

extracting and manipulating web page data. High-level pattern-matching and text manipulation are essential to web automation, acting as a *glue language* for connecting unrelated web services and programs.

2. *Embedding a scripting language and integrating a command interpreter into the Location box.* In addition to accepting a typed URL, the browser window's Location box can also accept a typed command with arguments. A command may be a built-in command, a user-defined script, or an external program. The built-in scripting language includes commands for automatic web browsing, such as clicking on hyperlinks, filling out forms, and extracting data from web pages.

3. *Using the browser window to display command output and construct pipelines of commands.* When a command is invoked, it takes its input from the current page in the browser window, and sends its output back to the browser window as a new page.

4. *Including executed commands in the browsing history.* Forward and Back navigate through command output pages as well as web pages. Part of the history can be extracted and saved as a script for later execution.

We have implemented these extensions in a prototype web browser named LAPIS (Lightweight Architecture for Processing Information Structure). The first extension, consisting of a pattern language and text-processing tools, was described in a previous paper [14], which is summarized below. This paper focuses on the other three features, which integrate a command shell into the web browser to create a *browser-shell*.

The browser-shell addresses the problem of interactive web automation by allowing the user to apply patterns, script commands, and external programs directly to the browser page. For one-shot tasks, commands can be interleaved with manual browsing to perform the task as quickly and directly as possible. For repeated tasks, the user can interactively define a script by invoking a sequence of commands on example data, using the Back button to correct mistakes, and then copying the command sequence out of the browsing history and saving it as a script.

The browser-shell concept has implications beyond web automation, two of which are considered in this paper:

1. *HTML interfaces for local programs.* Currently, programs with HTML interfaces must be installed in a web server in order to handle form submissions. LAPIS can submit forms to local programs by the Common Gateway Interface (CGI) [17], an existing standard used by web servers. This opens the possibility of running HTML applications entirely locally. HTML offers benefits of both a graphical user interface (GUI) and a command-line interface (CLI). An HTML interface can be as easy to learn and use as a GUI, yet still open to automation like a CLI. As a demonstration, we have wrapped an HTML interface around the Unix *find* program.

2. *Using the browser as a command shell, in place of the Unix shell or MS-DOS command prompt.* The browser-shell can be used to invoke local programs, but it behaves differently from a conventional typescript shell. Whereas a typescript shell interleaves commands with program output in the same window, a browser-shell displays commands and program output in separate parts of the browser window, and automatically redirects a program's input from the current page. These differences make some tasks easier, such as viewing program output and constructing pipelines, but others harder, such as running legacy programs that use standard input to interact with the user. The tradeoffs are discussed in more detail in section 5.

The remainder of this paper is organized as follows. Section 2 covers related work. Section 3 describes important features of the LAPIS browser-shell, including the pattern language, the scripting language, and invocation of external programs. Section 4 describes our prototype implementation of LAPIS and contrasts some implementation alternatives. Section 5 discusses some of the implications of integrating a command shell into a web browser, in particular creating local programs with HTML interfaces and using the browser as an alternative interface to the system command prompt. Section 6 reports on the status of the LAPIS prototype, and Section 7 concludes.

2 Related Work

Several systems have addressed the problem of web automation. One approach is *macro recording*, typified by LiveAgent [11]. LiveAgent automates a task by recording a sequence of browsing actions in Netscape through a local HTTP proxy. Macro recording requires little learning on the part of the user, but recorded macros suffer from limited expressiveness, often lacking variables, conditionals, and iteration.

Another approach is *scripting*, writing a program in a scripting language such as Perl, Tcl, or Python. These scripting languages are fully expressive, Turing-complete programming languages, but programs written in these languages must be developed, tested, and invoked outside the web browser, making them difficult to incorporate into a web user's work flow. The overhead of switching to an external scripting language tends to discourage the kind of spur-of-the-moment automation required by the tasks described above, in which interactive operations might be mixed with automation in order to finish a task more quickly.

A particularly relevant scripting language is WebL [9], which provides high-level *service combinators* for invoking web services and a *markup algebra* similar to the LAPIS pattern language for extracting results. Like other scripting languages, WebL lacks tight integration with a web browser, forcing a user to study the HTML source of a web service to develop markup patterns and reverse-engineer form interfaces. In LAPIS, web automation can be done while viewing rendered web pages in the browser, and simple tasks can be automated entirely by demonstrating the steps on examples.

Other systems have tackled more restricted forms of web automation by demonstration. Turquoise [13] and Internet Scrapbook [22] construct a *personalized newspaper*, a dynamic collage of pieces clipped from other web pages, by generalizing from a cut-and-paste demonstration. SPHINX [12] creates a web crawler by demonstration, learning which URLs to follow from positive and negative examples.

Wrapping GUI frontends around CLI programs is a common way to support both ease-of-use and scriptability. Many integrated development environments follow this pattern, in which the graphical user interface invokes the compiler, linker, and other tools using command-line interfaces. Particularly relevant is the Commando dialog box system in the Macintosh Programmer's Workshop [1], which allows a developer to specify a dialog box interface for an arbitrary Macintosh command-line program. A Commando dialog box resource is an abstract description specifying the dialog box controls and how the controls are mapped to command-line options. In that sense, it resembles an HTML interface, but is more platform-dependent than HTML.

Others have investigated wrapping HTML interfaces around command-line programs on a web *server*, but not on the client. For example, Phanouriou and Abrams [19] described an HTML interface that presented status information about a web server (network, filesystem, memory, kernel, etc.) obtained from Unix commands.

The browser-shell is not the first alternative to the standard typescript Unix shell. Another is Sam [21], a graphical text editor which integrates external program execution in three ways: "**<** *command*" replaces the current selection with the output of a command, "**>** *command*" runs the command with the current selection as input, and "**|** *command*" redirects both input and output. The Emacs *shell-command-on-region* command provides similar capabilities. In a later editor, Acme [20], each external command's output appears in a new window, with a *tag line* similar to a browser's Location box that can be used to invoke another external program. Unlike Sam, Acme had no provision for supplying a command's input from a window, and both systems lacked the output history provided by a browser-shell.

3 User Interface

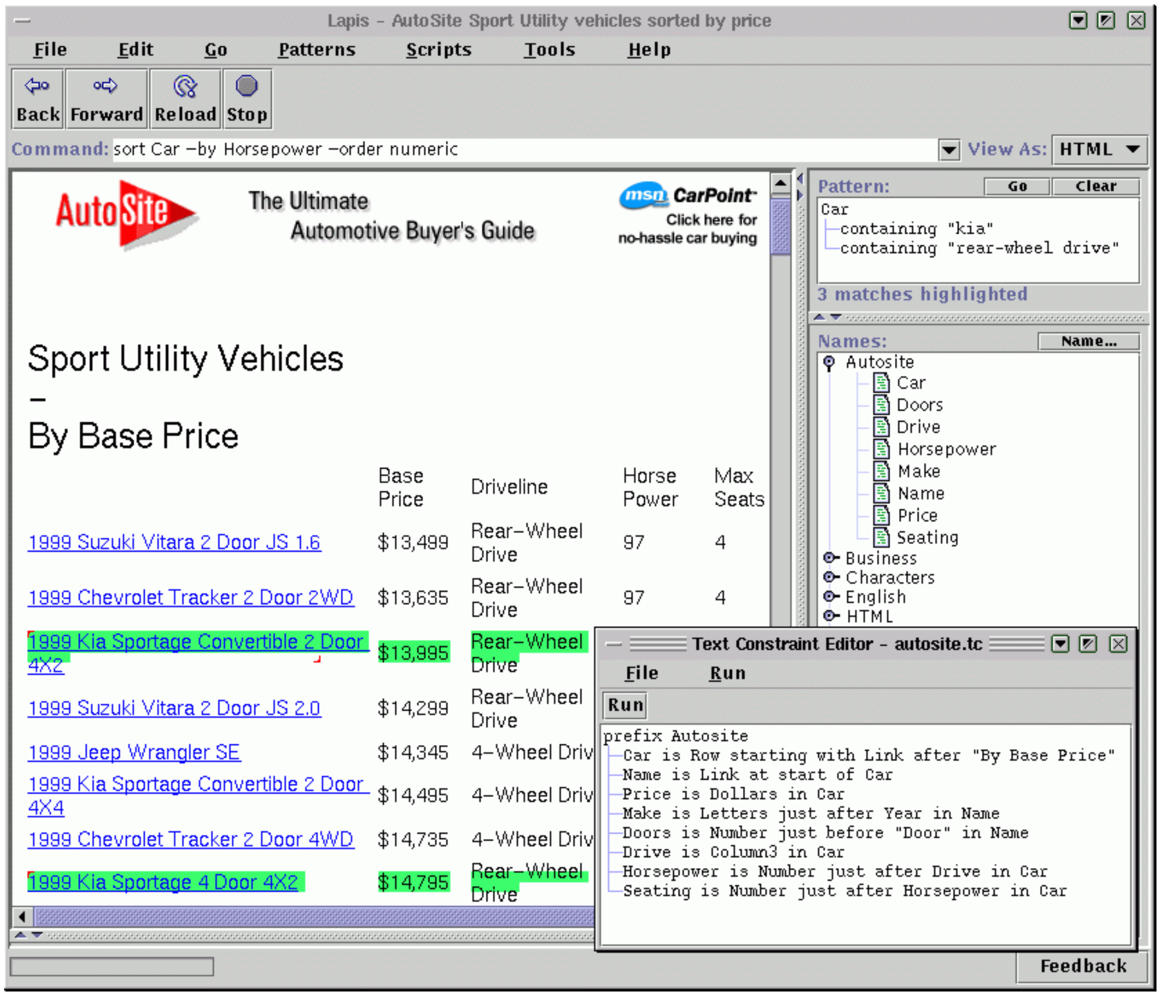
We now describe some important features of the browser-shell user interface. The first section is a summary of some previous work on which we are building. Subsequent sections describe new work: the command interpreter, web automation, creating web scripts by example, and invoking external programs and CGI programs.

3.1 LAPIS

The web browser we used to prototype the browser-shell is called LAPIS (Figure 1), part of a system of generic tools for structured text that we call *lightweight structured text processing* [14]. Lightweight structured text processing enables users to define text structure interactively and incrementally, so that generic tools can operate on the text in structured fashion. Our lightweight structured text processing system has four components:

- a *pattern language* for describing text structure;
- *parsers* for standard structure, such as HTML and programming language syntax;
- *tools* for manipulating text using structure, including sorting, searching, extracting, reformatting, editing, computing statistics, graphing, etc;
- a *document viewer* (in this case, a web browser) for viewing documents, developing and testing patterns, and invoking tools.

LAPIS includes a new pattern language called *text constraints*. Text constraints describe a set of regions in a page in terms of relational operators,



```
Command: sort Car -by Horsepower -order numeric
```

```
Pattern:
Car
  containing "kia"
  containing "rear-wheel drive"
3 matches highlighted
```

```
Names:
Autosite
    Car
    Doors
    Drive
    Horsepower
    Make
    Name
    Price
    Seating
Business
Characters
English
HTML
```

Sport Utility Vehicles
–
By Base Price

	Base Price	Driveline	Horse Power	Max Seats
1999 Suzuki Vitara 2 Door JS 1.6	$13,499	Rear-Wheel Drive	97	4
1999 Chevrolet Tracker 2 Door 2WD	$13,635	Rear-Wheel Drive	97	4
1999 Kia Sportage Convertible 2 Door 4X2	$13,995	Rear-Wheel Drive		
1999 Suzuki Vitara 2 Door JS 2.0	$14,299	Rear-Wheel Drive		
1999 Jeep Wrangler SE	$14,345	4-Wheel Driv		
1999 Kia Sportage Convertible 2 Door 4X4	$14,495	4-Wheel Driv		
1999 Chevrolet Tracker 2 Door 4WD	$14,735	4-Wheel Driv		
1999 Kia Sportage 4 Door 4X2	$14,795	Rear-Wheel Drive		

Text Constraint Editor - autosite.tc

File Run

Run

```
prefix Autosite
  Car is Row starting with Link after "By Base Price"
  Name is Link at start of Car
  Price is Dollars in Car
  Make is Letters just after Year in Name
  Doors is Number just before "Door" in Name
  Drive is Column3 in Car
  Horsepower is Number just after Drive in Car
  Seating is Number just after Horsepower in Car
```

Figure 1: The LAPIS web browser, displaying a web page that lists new cars. The page structure is described by patterns shown in the inset window (Text Constraint Editor). Some of the terms used in these patterns (Row, Link, etc.) are defined by other patterns not shown, and others are defined by the built-in HTML parser. The user has entered a pattern in the Find box to highlight certain cars (rear-wheel drive Kias), and is now about to run a command in the Command box to sort all cars by horsepower.

such as *before*, *after*, *in*, and *contains*. Text constraints can refer to structure defined by arbitrary parsers, such as the built-in HTML parser that identifies HTML elements and assigns them names, such as Link, Paragraph, and Heading. A single text constraint pattern can refer to multiple parsers — for example, `Line at start of Function` refers to both `Line`, a name defined by a line-scanning parser, and `Function`, a name defined by a programming-language parser. In general, text constraints are designed to be more readable and comprehensible for users than context-free grammars or regular expressions, because a structure description can be reduced to a list of simple, intuitive constraints which can be read and understood individually. More details about the text constraints language can be found in a previous paper [14].

The LAPIS browser includes several tools for transforming web pages. For example, *keep* extracts a set of regions matching a text constraint pattern, *delete* deletes a set of regions, *sort* sorts a set of regions in-place, and *replace* replaces a set of regions with some replacement text. In the LAPIS browser described in a previous paper [14], a tool could only be invoked from a menu, and its output was directed to a new page in the browser. The browser-shell extensions described in this paper make it possible to invoke these tools from the Location box and from user-defined scripts.

3.2 The Browser-Shell

In order to create scripts of commands, we embedded Tcl [18] into LAPIS. Tcl was chosen partly because of its syntactic simplicity, and partly because a good Java implementation was available [5]. Tcl is also well-suited to interactive command execution.

Instead of presenting a 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 its output is displayed in the browser as a new page that is added to the browsing history.

Using the Location box as a command line has several advantages. The page generated by a command can be browsed like a page generated by a URL. The browsing interface — Back, Forward, Stop, and Reload — also applies to command outputs. The Back button returns the browser to the previous page, Stop aborts a long-running command, and Reload runs the command again.

Since either a URL or a command can be typed into the Location box, LAPIS must be able to distinguish between them. The problem is trivial if the typed entry begins with a protocol prefix, such as `http:` or `file:`, and LAPIS also recognizes the protocol `cmd:` for invoking a command unambiguously. If the typed entry does not begin with a prefix, LAPIS tries every possible interpretation: first as a command to execute, then as a filename to display, then as a domain name for a web server. This is an extension of the heuristics already used by the Location box of most web browsers.

For security reasons, LAPIS only executes a `cmd:` URL if it originates locally — e.g., if it is typed into the Location box or found in a page loaded from the local filesystem. A link in a remote web page cannot invoke a Tcl command.

3.3 Web Automation

Web browsing has two basic actions: clicking on hyperlinks and submitting forms. Automating web browsing requires equivalent script commands for these actions.

Clicking on a link has the same result as typing in its URL in the Location box. Thus the script command for clicking on a link is simply the link's URL, such as:

```
http://weather.yahoo.com/
```

For some links, however, the URL varies depending on when the page is viewed. Variable links are often found in online newspapers, for example, where links to top stories change from day to day. The `click` command can be used to click on a variable link by describing its location in the web page with a LAPIS text constraint pattern. For example:

```
http://www.salon.com/ # Start at Salon
click {Link after Image in Column3}
         # Click on top story
         # (curly braces are Tcl quoting)
```

For entering data into forms, the `enter` command is provided, with two arguments. The first argument is a pattern describing the form field to affect. Since HTML form fields are named, this pattern may simply be the field name. Alternatively, the pattern may describe the field in terms of its context (e.g., `TextBox just after "Email Address:"`), which has the advantage of being comprehensible without looking at the HTML source. The second argument to `enter` specifies the value to enter in the field. For text fields, this value is entered in the field directly. For menus or lists, the value is selected in the list. For radio buttons or checkboxes, the value should be "on" or "off" (or yes/no, true/false, or 0/1).

Forms are submitted either by a `click` command describing the form's submit button. For example, here is a complete script that searches Google for the USENIX 2000 conference home page:

```
http://www.google.com/
enter {Textbox just after \
       "Search the web using Google"} \
      {USENIX 2000}
click SubmitButton
```

LAPIS also provides script commands for other web browsing actions, including Home, Back, Forward, Stop, Reload, and Save.

The examples presented so far have been web-site-specific, but some browsing tasks are sufficiently uniform across web sites to be handled by a generic script. For example, the following script can log into many web sites, assuming the user's login name and password have been stored in the Tcl variables `id` and `password`:

```
enter {Textbox \
        just after Text containing \
        ("login"|"email"|"id"|"user")} \
      $id
enter {Textbox just after Text containing \
         "password"} \
      $password
click SubmitButton
```

3.4 Automation by Demonstration

To create a browsing script quickly, the user can *demonstrate* it by recording a browsing sequence. The demonstration begins with an arbitrary example page, the *input page*, showing in the browser. Invoking the Demonstrate command pops up a new browser window, in which the browsing demonstration will take place. A new window is created so that the browsing sequence can refer to the input page for parameters. Like any LAPIS browser window, the Demonstrate window records a browsing history: URLs visited and commands typed. Unlike a normal browser window, however, the Demonstrate window's history also records user events in form controls. For example, if the user types into a form field, the history will record an equivalent `enter` command.

To fill in a form with text from the input page, the user can make a selection in the input page, then drag-and-drop (or copy-and-paste) to a form field in the Demonstrate window. If the copied text was selected by searching for a pattern, then this action records the command `enter` *`field-name pattern`* in the history. If the copied data was selected manually, then the command `enter` *`field-name`* `{Selection}` is recorded in the history. When the script is run at a later time, `Selection` will return the user's selection at that time. More complex dependencies can be expressed by typing a Tcl command instead of pointing-and-clicking. For example, if a radio button should be selected only if the input page has certain features, then the user might type the command `if {[find` *`pattern`*`]} {click` *`field-name`*`}`.

Using Back and Forward, the user can revise the demonstration as necessary until the desired results are achieved. The browsing history, which is essentially a Tcl script, can also be opened in an editing window, where the user can insert conditionals, iteration, and comments, if desired. When the user is satisfied with the demonstration, the Demonstrate window is closed, the history is saved as a script, and the script becomes available as a named command.

LAPIS demonstrations have two advantages over the macro recorders in previous systems, such as LiveAgent [11]. First, the recorded transcript is represented by the browsing history, which is visible, easy to navigate, and very familiar. A crucial part of making this work is that LAPIS inserts commands as well as URLs in the browsing history. Second, an experienced user can generalize the demonstration on-the-fly by typing commands at crucial points instead of pointing-and-clicking. Since a full scripting language is supported, the resulting scripts can be significantly more expressive than recorded macros, without taking much more time to develop.

3.5 Script Optimization

A script created by demonstration may include unnecessary steps, which may be expensive if they fetch web pages. To address this problem, LAPIS includes an optimizer that tries to compact the browsing script. For example, a sequence of simple link-clicking may result in a list of URLs:

```
# Start at Yahoo
http://www.yahoo.com/

# Click on Weather
http://weather.yahoo.com/

# Click on US
http://weather.yahoo.com/regional/US.html
```

Since the URLs are constant, depending neither on the input page nor on previous pages in the demonstration, the optimizer can delete all but the last, saving several page fetches.

The optimizer can also streamline form submissions. Submitting a form normally requires two page fetches, one to retrieve the form and another to submit the form. The optimizer can eliminate the first fetch by hard-coding the form submission URL, the form field names, and their values.

These optimizations are not always safe, however. For example, some forms have a variable submission URL or variable default values, often referring to unique session identifiers or persistent state. Thus the optimizer does not run by default. Instead, the user selects some or all of the script and invokes the optimizer on it manually. In the future, the optimizer may be able to gather information from repeated runs of a script to determine which optimizations would be safe to make automatically.

An optimized form submission may stop working correctly if the form changes, which happens from time to time when web sites are redesigned or moved. Gross changes can be detected by various techniques, such as the modification time or checksum of the form page, but the cost of detecting changes in just the *form* (as opposed to page content around the form, which might change often) would overwhelm the savings of optimization. This is a special case of a general challenge for web automation: recognizing and dealing with change on the Web. LAPIS helps with the problem by providing a rich pattern language, enabling browsing scripts to be insulated from many kinds of changes, but otherwise leaves detecting and debugging broken scripts to the user.

3.6 External Programs

In addition to built-in Tcl commands, LAPIS can also run an external command-line program from the Location box. If the command name is not found as a built-in Tcl command or user-defined script, then LAPIS searches for an external program by that name. If an external program happens to share the same name as a Tcl command, the user can force the external program to run with the `exec:` prefix.

Like a Tcl command, an external program is applied to the current page and displays its output as a new page added to the browser history. For example, if the user types (on BSD-style Unix) `ps aux`, then the browser displays a list of running processes. If the next command is `grep xclock`, then the process listing is filtered to display only those lines containing “xclock.”

To make this work with legacy programs such as *ps* and *grep*, the external program is invoked in a subprocess with its input and output redirected. Standard input is read from the current page of the browser, passing the HTML source if the current page is a web page. Standard output is sent to a new page of the browser, which is displayed incrementally as the program writes output. Standard error is sent to a subframe of the page, to separate it from standard output.

A program’s output may be parsed and manipulated like any other page in LAPIS. For example, `ps aux` displays information about running processes:

```
USER    PID %CPU %MEM SIZE RSS TTY...
bin     160  0.0  0.4  752 320  ? ...
daemon  194  0.0  0.6  784 404  ? ...
rcm     294  0.0  1.0 1196 660  ? ...
```

The output of *ps* can be parsed by simple LAPIS text constraint patterns:

```
Process = Line,
          but not starting with "USER";
User = Alphanumeric at start of Process;
PID = Number just after User;
```

These identifiers can be used with LAPIS commands that search and manipulate the output of *ps*:

```
# sort processes by PID
sort Process -by PID -order numeric

# display only xterm processes
keep {Process containing "xterm"}

# kill all xterm processes
kill [extract {PID in Process \
               containing "xterm"}]
```

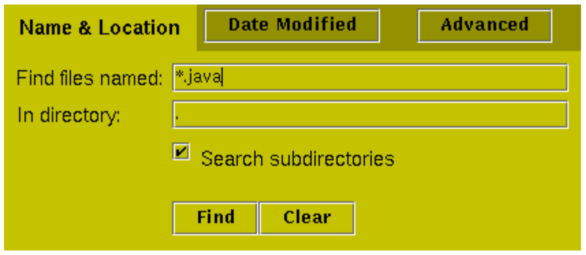
By default, patterns and commands are applied only to standard output, but standard error may also be processed by referring to the Tcl variable `$error`, as in `find {"Warning:"} $error.`

3.7 CGI Programs

If an external program outputs HTML instead of plain text, the browser-shell detects it and renders it as a web page. HTML output is detected by several simple heuristics, such as an initial `<html>` or `<doctype>` tag.

The HTML output may contain embedded forms. To submit a filled-out form back to the external program, LAPIS passes form parameters using the Common Gateway Interface (CGI) [17]. CGI passes form fields and other request information by setting environment variables, such as `QUERY_STRING`. Although CGI is commonly used by web servers to invoke external programs, no major web browser can invoke a CGI program locally. (The closest we’ve found is the Help Viewer in KDE 1.1, which displays HTML help documents and uses CGI to invoke a local search engine.) One beneficial side-effect of using CGI to communicate with external programs is that existing CGI scripts can be run directly by the browser-shell, without installing them in a web server. This feature may be useful for developing and testing CGI applications outside a web server.

Whether a form is being submitted or not, LAPIS always sets the CGI environment variables when it invokes an



*.java

.

[✓] Search subdirectories

Figure 2: HTML interface for Unix *find*.

```
#!/usr/bin/perl -w

# Check if invoked outside of browser-shell
# or passed arguments.
if (!defined $ENV{"GATEWAY_INTERFACE"}
    || @ARGV > 0) {
    # Pass arguments directly to find
    exec ("/usr/bin/find", @ARGV);
}

# Otherwise act as CGI script.
use CGI qw/:standard/;

if (!param()) {
  # No form submitted.
  # Display the HTML interface.
  exec "cat /usr/doc/find/find-form.html";
} else {
  # Handle form submission.
  exec ("/usr/bin/find",
        param("directory"),
        param("search_subdirectories")
         ? () : ("-maxdepth", "1"),
        "-name", param("name"),
        "-print");
}
```

Figure 3: Perl wrapper for *find* that displays the HTML form interface shown in Figure 2 when invoked inside the browser-shell. Form submissions are handled by the Perl CGI.pm module.

external program. A program can use the presence or absence of these variables to determine whether it was invoked from the browser-shell, in which case it can present an HTML interface and act like a CGI program, or from the ordinary typescript shell, in which case it should present a text-only or command-line interface.

One use for this facility is wrapping a friendlier HTML interface around an existing command-line program. For example, some users have trouble remembering the syntax for the Unix *find* command, which searches for files matching certain constraints. *Find* supports a variety of predicates on filename, date, user ownership, etc., and Boolean operators for combining predicates. We wrote a Perl CGI script wrapper around *find* which displays a simple HTML form (Figure 2). The first part of the wrapper script (Figure 3) tests whether the script is running under LAPIS. If not, or if the user passed command line arguments, then the wrapper simply invokes the original *find*. Otherwise, the script prints an HTML page containing the form. When the form is filled out and submitted back to the wrapper script, the script invokes *find* appropriately.

The HTML wrapper makes it possible to use *find* without learning or remembering its command-line syntax. A GUI frontend for *find* would offer the same benefits, but at greater cost: a GUI frontend has no ready hooks for automation, but the HTML form interface can be scripted in LAPIS exactly as if it were a web service. For example, a Java programmer may want a script that searches all subdirectories for files ending with `.class` and stores them in a ZIP file. The user pops up a Demonstrate window, invokes *find* to display its HTML form, fills in the form to search for files named `*.class`, and applies *zip* to the resulting list of files. This sequence of actions is then saved as a script. Thus the user can include *find* in a script without learning its more complex command-line interface.

4 Implementation

The browser-shell prototype described in this paper was implemented by modifying an existing web browser, LAPIS, originally designed to test new user interface ideas. LAPIS is written in Java 1.1 using the HTML layout component `JEditorPane` from the Java Foundation Classes. Before modification, LAPIS consisted of about 18,000 lines of code. The browser-shell features added about 2,000 lines of code. The LAPIS browser-shell has been tested on Linux, Solaris, and Windows NT.

Modifying a browser is not the only way to implement browser-shell capabilities. Two other general strategies exist for adding features to web browsers. One scheme uses an internal browser extension mechanism, such as a Netscape plugin, an Internet Explorer ActiveX component, or a Java applet. The other scheme is an HTTP proxy, external to the browser but running on the same machine, that filters the browser's HTTP requests.

Both schemes have the advantage of working with existing browsers, but lack of tight integration with the browser makes some browser-shell features difficult or impossible to implement. For example, neither scheme would allow commands to be typed directly into the browser's Location box. Highlighting the results of pattern matches would be much harder, as would monitoring the user's entries in form fields to generate scripts by demonstration. The lack of control over the browser's user interface makes these browser-extension schemes too constraining for use as a research testbed. For a developed product, however, one of these schemes may be the best bet, even if it can only deliver a subset of the capabilities described in this paper.

We suggest that there are several levels of browser-shell complexity. Higher levels are harder to design and implement, but deliver correspondingly greater benefits. In increasing order of complexity, the levels are:

1. *Local program invocation.* Implementing this level requires spawning a subprocess and redirecting its input and output to the browser. This level is sufficient for using the browser as a command shell.

2. *Local CGI invocation*. Implementing this level requires encoding a form submission into environment variables and invoking a local program. This level is sufficient to support local HTML interfaces with form submission.

3. *Embedded scripting language*. Many web browsers already embed Javascript, but do not support automatic browsing (i.e., a sequence of script commands invoked on successive web pages). With automatic browsing, this level is sufficient to support web automation.

4. *Embedded pattern language*. A pattern language like text constraints enables the user to describe, manipulate, and extract parts of web pages and program outputs. This level acts as a glue language for connecting unrelated information sources or programs, so an ideal pattern language should be capable of describing not only HTML, but also text and XML.

5. *Web automation by demonstration*. Implementing this level requires recording user events and generalizing them into script commands. This level helps novice users learn the scripting language and helps expert users streamline the construction of scripts.

5 Discussion

We now discuss some general implications of integrating a command shell into a web browser, in particular the new applications, architectures, and interaction styles that such a hybrid enables.

5.1 HTML Interfaces

Much interest in recent years has focused on creating and deploying HTML-based applications that run in web servers. The advantages of deploying an application as a web service are well understood: it can be accessed by millions of users at the click of a button, it can be upgraded easily, and it can even be given away for free, paid for by advertising. The most popular sites on the Web are HTML interfaces in this sense.

The browser-shell opens up a new possibility: deploying HTML interfaces on the client. There are still many reasons to deploy applications on the client, including performance, security, and ability to run disconnected from the network. Current browsers cannot submit HTML forms to client-side programs, however, forcing a client-side HTML application to handle its user input in a more complicated way (e.g., with Javascript, Java, or ActiveX). The browser-shell's ability to submit forms to local programs allows client-side programs to have pure HTML user interfaces, displayed entirely in the browser.

HTML interfaces have several advantages. First, an HTML interface is easy to implement portably, since it needs only the standard I/O library rather than larger, less portable GUI libraries. Second, a wide variety of HTML editors and CGI libraries already exist, making the job easier. Third, compared to a command-line interface, an HTML interface is easy to use, not only because it is visual, but also because users are familiar with similar interfaces on the Web. Finally, compared to a GUI, an HTML interface is easier to script because it is declarative and textual, allowing systems like LAPIS to parse the interface and control it automatically.

Some applications are well-suited to HTML; others are not. User input is limited to forms with standard controls such as buttons, menus, and text fields, so applications that demand richer interaction would be poorly suited. On the other hand, applications with high information content, such as detailed help or reference materials, would be well-suited, since HTML makes it easy to intersperse forms with formatted text, pictures, and hyperlinks.

Any program that already has a command-line interface is a prime candidate for an HTML interface. As our *find* example showed, wrapping an HTML interface around a legacy program is simple if the program takes all user input as command-line arguments. Programs that conduct an interactive dialog with the user are trickier to wrap, however, because the CGI protocol does not support persistent connections. The wrapper must be reinvoked for every form submission. This problem could be solved by a more complex wrapper that maintains its own persistent connection to the legacy program, or by an alternative form submission protocol with a persistent connection to the wrapper.

HTML interfaces allow command-line programs to be self-describing. Instead of the terse "usage" message printed by command-line interfaces, a program running in a browser-shell would display its HTML documentation, and embedded in the documentation itself would be the program's user interface. Thus, the usage message of an HTML interface not only explains what the program does, but also presents an interface for actually invoking it.

5.2 New Shell Interaction Model

The web browser is becoming a central part of the desktop interface. Modern browsers, such as Microsoft Internet Explorer and KDE's *kfm*[10], already include file management among the web browser's responsibilities. Integrating the system command prompt is another step along the same path, which makes sense because file management and command execution are often intertwined.

The browser-shell interface behaves differently from a traditional typescript shell, however. Whereas a typescript shell interleaves commands with program output in the same window, a browser-shell separates the command prompt from program output. The browser-shell also automatically redirects program input from the current browser page, and automatically sends program output to a new browser page.

One effect of these differences is on scrolling. In a typescript interface, long output may scroll out of the window. To view the start of the output, the user must either scroll back, or else rerun the command with output redirected to *more* or *head*. The browser-shell, by contrast, initially displays the *first* windowful of output, rather than the *last*, reducing the need for scrolling. When output is less than a windowful, a typescript can become cluttered by outputs of several commands, forcing the user to scan for the start of the latest output. The browser-shell displays each program output on a new, blank page. The overall effect of the browser-shell is like automatically redirecting output to *more*.

Unlike *more*, however, the browser-shell's display is not ephemeral. The displayed output can be passed as input to another command, which allows pipelines to be assembled more fluidly than in the typescript interface. Developing a complicated pipeline, such as `ps ax | grep xclock | cut -d ' ' -f 1`, is often an incremental process. In typescript interfaces, where input redirection must be specified explicitly, this process typically takes one of two forms:

- Repeated execution: run `A` and view the output; then run `A|B` and view the output; then (`B` turned out wrong) run `A|B`$'$ and view the output; etc. This strategy fails if any of the commands run slowly or have side-effects.
- Temporary files: run `A > t1` and examine `t1`; then run `B < t1 > t2` and examine `t2`; then (`B` was wrong) run `B`$'$ `< t1 > t2`, etc.

The browser-shell offers a third alternative: run `A` and view the output; then run `B` (which automatically receives its input from `A`) and view the output; then press Back (because `B` was wrong) and run `B`$'$ instead. The browser-shell displays each intermediate result of the pipeline while serving as automatic temporary storage.

Automatic input redirection makes constructing a pipeline very fluid, but it is inappropriate for programs that use standard input for interacting with the user, such as *passwd*. Such programs cannot be run in a browser-shell without modification, such as wrapping an HTML interface around the program, or running the program in a terminal emulator, possibly embedded in the browser-shell window.

One problem with the browser-shell model is the linear nature of the browsing history. If the user runs A, backs up, and then runs B, the output of A disappears from the browsing history. To solve this problem, the LAPIS prototype lets the user duplicate the browser window, including its history, so that one window preserves the original history while the other is used to backtrack. (Netscape's New Window command worked similarly before version 4.0.) A more complex solution might extend the linear browsing history to a branching tree [2].

6 Status and Future Work

The LAPIS web browser described in this paper, including Java source code, is available from

```
http://www.cs.cmu.edu/~rcm/lapis/
```

LAPIS is only a prototype, but it demonstrates the basic ideas described in this paper. Unfortunately, the LAPIS prototype is not robust enough for everyday use, largely because `JEditorPane` renders many web pages poorly. An important avenue of future work will be to convert a production-quality web browser into a browser-shell and experiment with using it on a daily basis.

Several features are needed to make the browser-shell more useful and more efficient as a command shell, including:

- Background processes. Web browsers generally stop loading a page when a new URL is typed in the Location box. Similarly, LAPIS automatically stops the currently executing command when a new command is typed. As a result, only one command can be running in each LAPIS browser window. An improvement would be support for background-process syntax. If a command ends with &, it could continue running in the background, storing its output in case the user ever backs up through the history.

- Handling large outputs. A command may generate too much output for the browser to display efficiently. The same problem often happens in typescript shells, usually forcing the user to abort the program and run it again redirected to a file. To handle this problem, the browser could automatically truncate the display if the output exceeds a certain user-configurable length. The remaining output would still be spooled to the browser cache, so that the entire output can viewed in full if desired, or passed as input to another program.

- Streaming I/O. A pipeline may process too much data for the browser's limited cache to store efficiently. Although the browser-shell's automatic I/O redirection could still be used to assemble the pipeline (presumably on a subset of the data), the pipeline would run better on the real data if its constituent commands were invoked in parallel with minimal buffering of intermediate results. The browser-shell might do this automatically when invoking a script.

- Shell syntax. Expert users would be more comfortable in the browser-shell if it also supported conventional operators for pipelining and I/O redirection, such as |, <, >, and >>. The most direct way to accommodate expert users might be to embed an existing shell, such as *bash* or *tcsh*, as an alternative to Tcl.

7 Conclusions

We have integrated a command shell into a web browser, and shown how this arrangement delivers benefits in three areas: (1) web automation; (2) HTML user interfaces for command-line applications; and (3) using a web browser as a new way to interact with the system command prompt.

We would hope that the next generation of web browsers will include at least some of these features, enabling future web users to put the power of automation to work in browsing and manipulating the Web.

Acknowledgements

The authors are grateful to David Garlan, John Pane, and the anonymous referees for their helpful suggestions on improving this paper.

This research was funded in part by a USENIX Student Research Grant.

References

[1] Apple Computer, Inc. *Macintosh Programmer's Workshop*. http://devworld.apple.com/tools/mpw-tools/

[2] E.Z. Ayers and J.T. Stasko. "Using Graphic History in Browsing the World Wide Web." *Proc. 4th International World Wide Web Conference WWW*4, December 1995, pp 259–270.

[3] K. Borg. "IShell: A Visual UNIX Shell." *Proc. Conference on Human Factors in Computing Systems (CHI '90*), 1990, pp 201–207.

[4] M. A. Cusumano and D. B. Yoffie. "What Netscape Learned From Cross-Platform Software Development."*Comm. ACM*, v42 n10, October 1999, pp 72–78.

[5] M. DeJong, et al. *Jacl and Tcl Blend*. http://www.scriptics.com/software/java

[6] P. E. Haeberli. "ConMan: A Visual Programming Language for Interactive Graphics." *Proc. ACM SIGGRAPH 98*, 1988, pp 103–111.

[7] T. R. Henry and S. E. Hudson. "Squish: A Graphical Shell for Unix." *Graphics Interface*, 1988, pp 43–49.

[8] B. Jovanovic and J. D. Foley. "A Simple Graphics Interface to UNIX." Technical Report GWU-IIST-86-23, George Washington University Institute for Information Science and Technology, 1986.

[9] T. Kistler and H. Marais. "WebL - A Programming Language For the Web." In *Computer Networks and ISDN Systems* (*Proc. 7th International World*

Wide Web Conference WWW7), v30, April 1998, pp 259–270. Also appeared as DEC SRC Technical Note 1997-029.

[10] K Desktop Environment. *KFM*. http://www.kde.org/

[11] B. Krulwich. "Automating the Internet: Agents as User Surrogates." *IEEE Internet Computing*, v1 n4, July/August 1997. http://computer.org/internet/v1n4/ krul9707.htm

[12] R. C. Miller and K. Bharat. "SPHINX: A Framework for Creating Personal, Site-Specific Web Crawlers." In *Computer Networks and ISDN Systems* (*Proc. 7th International World Wide Web Conference WWW7*), v30, April 1998.

[13] R. C. Miller and Brad A. Myers. "Creating Dynamic World Wide Web Pages By Demonstration." Carnegie Mellon University School of Computer Science Tech Report CMU-CS-97-131 (and CMU-HCII-97-101), May 1997.

[14] R. C. Miller and B. A. Myers. "Lightweight Structured Text Processing." *Proc. USENIX 1999 Annual Technical Conference*, Monterey, CA, June 1999, pp 131–144.

[15] F. Modugno and B. A. Myers. "Typed Output and Programming in the Interface." Carnegie Mellon University School of Computer Science Technical Report, no. CMU-CS-93-134. March 1993.

[16] F. Modugno and B. A. Myers. "Pursuit: Visual Programming in a Visual Domain." Carnegie Mellon University School of Computer Science Technical Report, no. CMU-CS-94-109. January 1994.

[17] NCSA. C*ommon Gateway Interface.* http://hoohoo. ncsa.uiuc.edu/cgi/

[18] J. Ousterhout. "Tcl: An Embeddable Command Language." *Proc. USENIX 1990 Winter Technical Conference*, pp 133–146.

[19] C. Phanouriou and M. Abrams. "Transforming Command-Line Driven Systems to Web Applications." *Proc. 6th International World Wide Web Conference (WWW6*), 1997, Santa Clara CA, pp 599–606.

[20] R. Pike. "Acme: A User Interface for Programmers." *Proc. USENIX 1994 Winter Technical Conference*.

[21] R. Pike. "The Text Editor sam." *Software Practice & Experience*, v17 n11, November 1987, pp 813–845.

[22] A. Sugiura and Y. Koseki. "Internet Scrapbook: Automating Web Browsing Tasks by Demonstration." *Proc. ACM Symposium on User Interface Software and Technology (UIST 98)*, 1998, pp 9–18.