Creating Dynamic World Wide Web Pages By Demonstration

Robert C. Miller, Brad A. Myers

May 1997
CMU-CS-97-131
CMU-HCII- 97-101

Human Computer Interaction Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891

{rcm,bam}@cs.cmu.edu
http://www.cs.cmu.edu/~rcm

Abstract

Turquoise is an intelligent browser and editor for the World Wide Web (WWW) that allows users to create dynamic pages by demonstration rather than by writing program code. With Turquoise, users without programming experience can create scripts that combine data from several Web pages, automate repetitive browsing or editing tasks, convert other data formats into Hypertext Markup Language (HTML), and process submitted forms. Scripts are demonstrated by familiar browsing and editing actions, which Turquoise records and generalizes into a program. In order to generalize the locations of the user's actions on a page, Turquoise includes a novel pattern matcher that finds locations within an HTML document. Turquoise infers patterns automatically by picking from a knowledge base of pattern templates, heuristically chosen to be robust and comprehensible to the user. With a good pattern knowledge base, Turquoise can often infer the correct script after only a single demonstration.

This research was partially sponsored by NSF under grant number IRI-9319969. Robert Miller is partially supported by a National Defense Science and Engineering Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Keywords: World Wide Web (WWW), end-user programming, programming-by-demonstration, Hypertext Markup Language (HTML), Client Gateway Interface (CGI), scripting languages.

1. INTRODUCTION

The World Wide Web's enormous popularity shows that almost anyone can publish documents on the Web. Would-be publishers do not even need to learn HTML, thanks to what-you-see-is-what-you-get (WYSIWYG) editors like Netscape Navigator Gold, Adobe PageMill, and AOLpress. These editors make publishing static documents on the Web as easy as ordinary word processing.

Unfortunately, end-users get far less help with creating *dynamic pages.* Dynamic pages are documents generated on-the-fly, typically by running a script through the Common Gateway Interface (CGI) protocol [13]. Dynamic pages enable the Web to be used as a medium for database access, commercial transactions, gateways to other information systems, and even chat forums.

Automating repetitive tasks is another area where end-users get little help from today's Web tools. Web document maintainers often face tedious, repetitive chores. Existing tools can help with the most typical chores, like finding broken links in a Web site, but specialized tasks are beyond their reach. End-users often need some kind of *personal assistant* to help them browse and edit the Web.

Creating dynamic pages and personal assistants is a major challenge for most end-users, requiring intimate knowledge of both HTML and a programming language, such as Perl, Tcl, Java, or C. Consider a typical Web user with no programming experience. How can such a user do any of the following:

- create a customized newspaper, showing the weather forecast, news headlines, and a favorite cartoon, all on a single Web page for easy reading?
- maintain a history of a stock portfolio by automatically retrieving new stock quotes from a Web site each day and appending them to a table?
- provide colleagues with a Web-based form that collects orders for lunch, combining them into a single list for easy ordering and delivery?

Typical users are unable to automate any of these tasks, without an appropriate existing script or a programmer to help them. For these users, the World Wide Web must be manually operated, with no way to retrieve, process, or generate data automatically.

There is an alternative to traditional scripting languages: creating the script *by demonstration*. In programming-by-demonstration [11], the user describes a program by operating on example data, while the system watches and attempts to infer the intent of the user's actions. The end result of the demonstration is an abstract, executable program.

Turquoise is a WWW browsing and editing system that supports the creation of dynamic pages and personal assistants by demonstration. With Turquoise, a user demonstrates a script like the customized newspaper by browsing to the desired news sources and using cut-and-paste to construct an example of the newspaper. From this single example, Turquoise can infer a script that regenerates the newspaper automatically, reflecting the latest information from its sources.

The Web documents used by a script are liable to change from time to time – in fact, the customized newspaper's sources change every day. In order to handle changes gracefully, Turquoise includes a *pattern matcher* designed to find portions of an HTML document. A Turquoise pattern is an abstract description of a region of interest on a page, in terms of HTML markup elements and text. For instance, the pattern "Bulleted List **after** Heading **containing** 'Headlines'" might describe how news headlines are presented by a particular Web site. Like Halbert's *data descriptions* [7], patterns are used to describe the parameters of every command in a script. For the copy-and-paste commands used to build the customized newspaper, both the copied region and the paste location are described by patterns. Patterns are the essential abstractions that allow Turquoise scripts to be generalized beyond the specific examples used in a demonstration.

Although Turquoise patterns could be written by the user (after some exposure to the grammar and vocabulary), the challenge for Turquoise is to infer patterns automatically from the user's demonstration. A huge number of patterns might describe a selected region, but few of them will be useful. To reduce the search space, we use a heuristic knowledge base of *pattern templates*, which are patterns containing placeholders, like "<HTML element> **after** Heading **containing** <text>". When Turquoise needs a pattern to describe a region, it tests the pattern templates against the selected region and instantiates the placeholders as needed to make a matching pattern. If several matching patterns are found in the template knowledge base, then Turquoise presents the possibilities to the user for a final decision. If none of the guesses are appropriate, then the user can edit the pattern directly to fix it.

The remainder of this paper is organized as follows. First we describe the system architecture, which motivates a discussion of the kinds of scripts that Web users write, in order to delimit the subset that can be usefully demonstrated. To illustrate, we give two examples of demonstrating scripts with Turquoise. Next, we present the pattern language used to describe regions of an HTML page, followed by the inference mechanism that

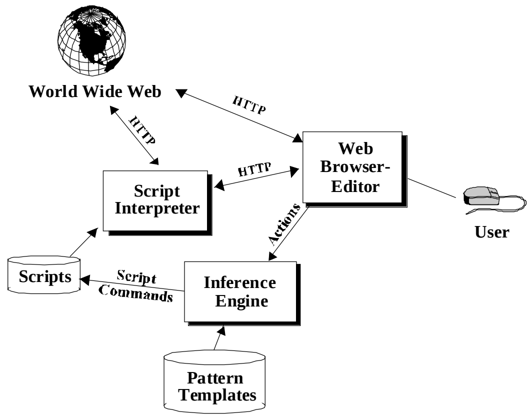

Figure 1. Turquoise system architecture.

Turquoise uses to guess patterns automatically. Finally, we survey related work and present some conclusions.

2. SYSTEM ARCHITECTURE

The Turquoise system is shown inFigure 1. It consists of the following components:

- a Web *browser/editor*;
- an *inference engine*, which watches the user's actions and infers script commands;
- a *pattern knowledge base*, which provides the pattern templates that are used for guessing what the user meant by a selection;
- a *script interpreter*, which replays scripts on demand, either for the local user or for a remote Web user; and
- the *scripts* that the user has developed.

The Web browser/editor in the prototype system is AOLpress[1], which is freely available from America Online[1]. AOLpress is a WYSIWYG editor, enabling users to create and edit pages without learning HTML, including pages containing forms and tables. Our version of AOLpress has been specially instrumented to report the user's browsing and editing actions to an external program, in this case the inference engine.

[1] AOLpress was originally known as NaviPress, and then GNNpress. It was one of the first WYSIWYG editors for HTML.

The inference engine receives the stream of user actions, watching for actions which affect a browser window containing a Turquoise script. These actions are assumed to be part of a demonstration. For each demonstrated action, the inference engine generalizes the action's parameters by searching for them in the pattern knowledge base. For instance, when the user pastes text into a script window, the inference engine attempts to find patterns describing the origin of the pasted text and the location where it is inserted. After generalizing the action, the inference engine adds it as a command to the script.

The script interpreter runs the scripts that have been demonstrated and saved. Like other Web resources, Turquoise scripts are identified by a uniform resource locator (URL), so the local user can run a script simply by opening its URL in the Web browser. The script interpreter also includes a small Hypertext Transfer Protocol (HTTP) server that can run scripts for remote users, as long as the local user has identified those scripts as publicly accessible. Scripts might also be invoked automatically, either at regular intervals (such as hourly, daily, or weekly) or by events within the Web browser/editor. For example, a script that updates a page's last-modified date might be invoked whenever the user saves an HTML document to disk.

The Turquoise prototype is written primarily in Java (with some C++ code at the interface with AOLpress). The prototype currently runs on Windows 95 and NT, but it could be ported readily to Macintosh or Unix, since its off-the-shelf components are also portable to those platforms.

3. APPLICATION DOMAIN

The Turquoise architecture is well-suited for scripts that access, create, or modify HTML or flat text documents, since such scripts can be readily demonstrated in a Web browser/editor. In an informal survey of 50 existing CGI scripts whose source code was publicly available, we found that the scripts available on the Web can be broken down into seven categories (with significant overlap): composite pages, assistants, filters, form processors, active pages, gateways, and applets. These categories are described next, and most of them are suitable to be demonstrated to Turquoise.

Composite page scripts extract information from one or more dynamic sources to produce a page of HTML. A typical example might be a customized newspaper which retrieves news headlines from Yahoo/Reuters, sports scores from ESPN, and a weather report from the National Oceanic and Atmospheric Administration, then reformats all the retrieved information into a single page. Another example might be a meta-search which submits a search phrase to several Web search engines simultaneously.

Assistants perform chores that the user encounters while browsing the Web or editing HTML pages. Examples of typical chores include updating your stock portfolio history, updating the last-modified date on an edited page, validating links, and downloading or prefetching a set of links.

Filter scripts translate data between other textual formats and HTML, or from one HTML layout into another. Examples of filters include converting Unix *man* pages to HTML, converting Emacs *info* files to HTML, and generating a table-of-contents for a page from its section headings.

Form processors accept a form as input and use the supplied information to access other resources, invoke external processes, record some information, or otherwise change the state of the Web. Examples include merchandise order forms, comment solicitation forms, guest books, and Web-based bulletin boards.

Active pages contain scripting code embedded in the HTML that is executed by the browser. The popular languages for active pages are JavaScript and VB Script. Active pages are typically used for form validation, simple calculations, and eye-catching visual effects.

Gateways translate non-HTML information sources or services into HTML. For instance, there are gateways that allow Web users to browse other hypertext systems, like HyperG, or retrieve information via the *finger* or *whois* services. Most Web browsers include built-in gateways for FTP, Gopher, the local filesystem, and Usenet news.

Finally, *applets* contain an embedded user interface which is downloaded and run by the Web browser. The latest hot tools for developing applets are Java and ActiveX.

The Turquoise architecture is capable of creating composite pages, assistants, filters, form processors, and active pages (insofar as its Web browser/editor supports a client-side scripting language like JavaScript or VB Script). These types of scripts are alike in that they operate on HTML or plain text.

Applets and gateways, however, are beyond the scope of the architecture. Creating an applet is tantamount to creating a graphical user interface, which cannot be done in a Web browser/editor because it lacks the tools for drawing it. Even automating an interaction with an applet is difficult for Turquoise, since it involves interpreting and controlling the state of an arbitrary, unconstrained user interface. Likewise, most gateways are beyond the ability of Turquoise to create, because the protocol on the other side of the gateway is unconstrained, and usually involves access mechanisms that are not accessible to the Web browser.

On the other hand, if the gateway already exists (either out on the Web or built into the browser), then a Turquoise script can interact with it as it would any other WWW-based information source. In order to extend the range of Turquoise, we plan to augment the prototype system with several built-in gateways not normally found in Web browsers: a calculator for numerical computation, a command line for executing external programs, and an OLE interface for exchanging information with other Microsoft Windows applications. These gateways will enable Turquoise scripts to do a wider variety of useful processing.

4. DEMONSTRATING SCRIPTS

We present two examples to show how Turquoise might be used. The first creates a composite page script for a customized newspaper. The second example is a form processor for lunch orders submitted over the Web.

4.1 Creating a Customized Newspaper

Suppose Lucy wants to demonstrate to Turquoise how to assemble a customized newspaper containing a weather report and news headlines (Figure 2). The demonstration begins with Lucy opening a new script window within the Web browser. Initially the script window is empty, indicating that the script has no output yet.

Lucy types “Weather” in the new script window and changes it to section heading style. Since these actions were performed directly on the script window, Turquoise records them as constant text in the script. Then, in a separate browser window, Lucy navigates to a weather page, highlights the region of the page containing her local three-day weather forecast, and copies it into the script window after “Weather” (Figure 2a).

Next, Lucy types in a section heading for “News” in the script window. She navigates to a news page and copies the region containing the day’s headlines to the script window just after “News” (Figure 2b).

Finally, Lucy saves the script, calling it “newspaper,” and runs it to check that it works as expected. Running the script is simply a matter of reloading it into the browser. When the script runs, it retrieves the weather page and the news page, extracts the regions of the page that match the patterns, and inserts them after “Weather” and “News”, respectively. The result is a page that looks the same as the demonstration. When Lucy comes back and runs her newspaper script the next day, however, the script retrieves the latest weather and news and displays an updated page (Figure 2c).

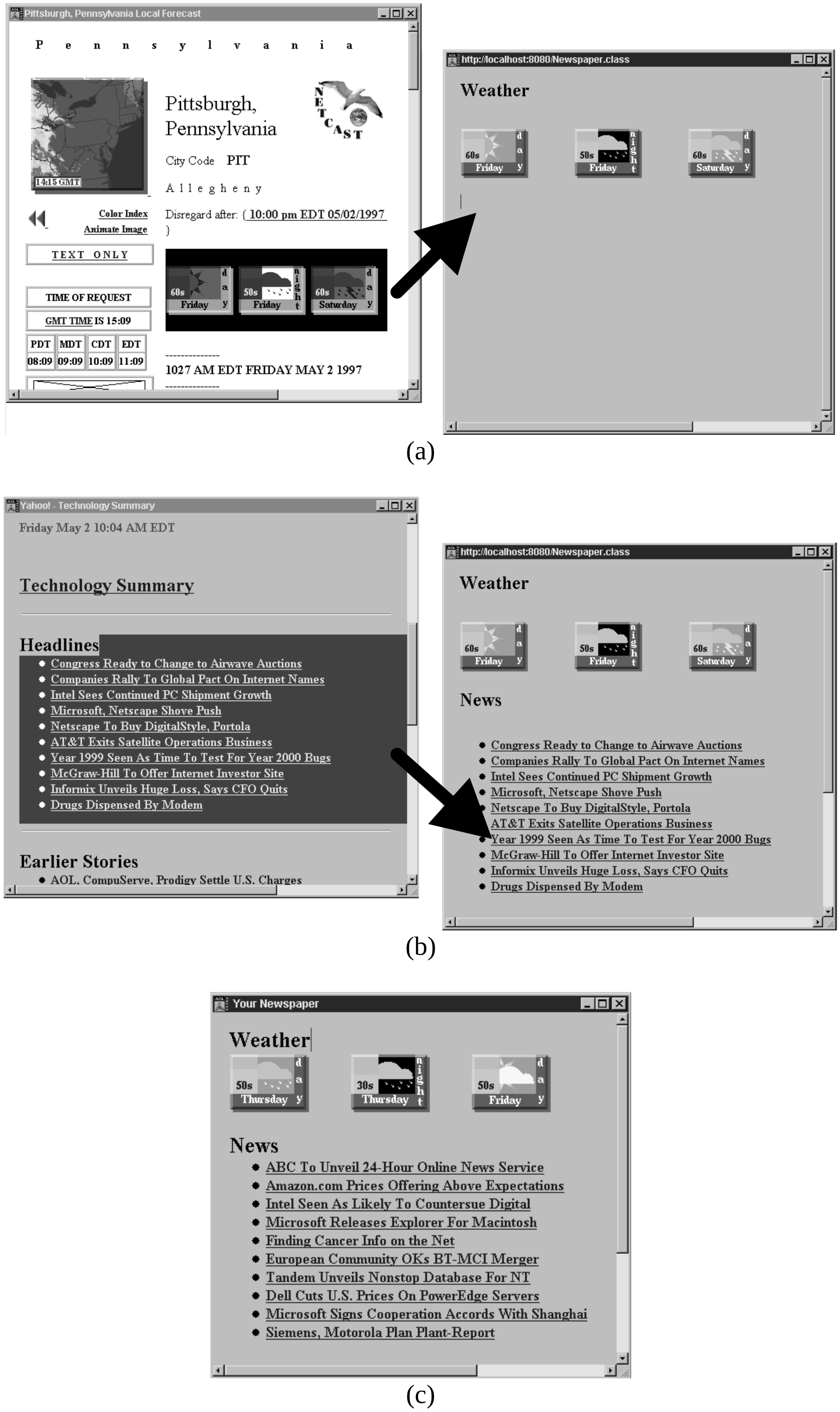

Figure 2. Demonstrating a customized newspaper script. In (a), the user copies the forecast from a weather page, on the left, and pastes it into a new script window, on the right. A similar operation copies the list of headlines from a news page (b). From these actions, Turquoise infers a script that can produce a new version of the newspaper on another day (c).

4.2 Collecting Lunch Orders

Harry's workgroup likes to order lunch from a local sandwich shop, and they find it convenient to send one person out every day with the entire group's order. To automate the process of collecting the sandwich orders, Harry wants to demonstrate a script that accepts order forms over the Web and collates them into a single report.[2]

Harry starts by designing a form using the Web editor (Figure 3a). The form contains fields for the person's name, sandwich choice, and any special requests. Creating the form requires a uniform resource locator (URL) to which it should be submitted, so Harry creates a new Turquoise script "lunch" and uses its URL. Harry also designs a template for the final report (Figure 3b), which will list the sandwich orders on one page for easy ordering and delivery.

To begin the demonstration, Harry fills out the form with an example order, and attempts to submit it. Since the "lunch" script is currently empty, the result is just an empty window. In this window, Harry proceeds with the demonstration by loading the report file. Since the load action occurs in the script window, Turquoise adds it as a command to the script. Then, using copy-and-paste operations, Harry copies the order form field-by-field to the end of the report, and saves the modified report back to disk.

Next, Harry wants to show how to construct a result page, which should be sent to the form submitter to confirm that the sandwich order has been recorded (Figure 3c). He uses the browser's Back button to return to the script output page, which is still empty, and designs a page that repeats the information in the form, again using cut-and-paste operations.

By default, Turquoise scripts can be run only by the local user. In order to make the lunch-ordering script accessible to his colleagues, Harry must mark the script as public, which specifies that Turquoise should run the script in response to remote requests for it. Every remote request runs the script on Harry's machine, which is appropriate for a small workgroup application like ordering lunch. Scripts which are frequently run or resource-hungry should be translated to a conventional language, like Perl, C, or Java, and made available on a high-speed Web server. We plan to extend Turquoise to perform these translations automatically wherever possible.

---

[2] This example does not yet work in the prototype, because AOLpress is not yet instrumented to report user actions that occur in form controls.

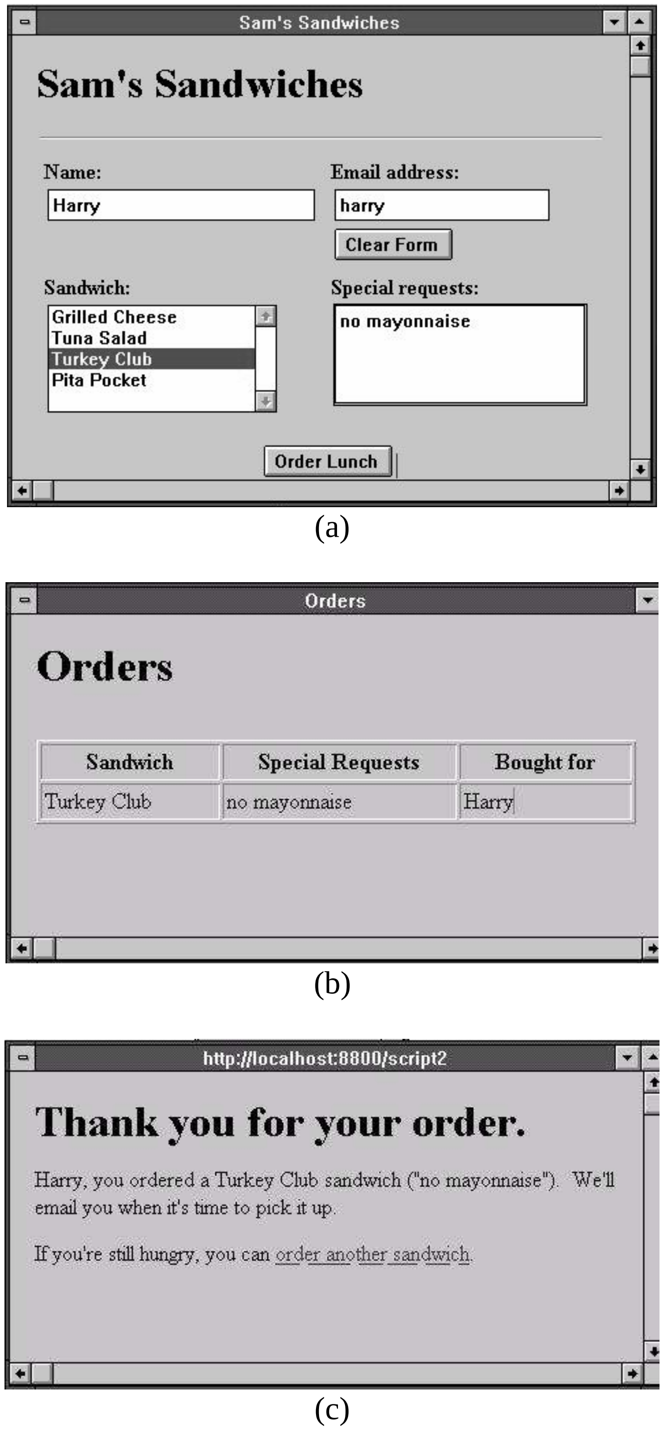

Figure 3. Demonstrating a lunch ordering script. After designing a form for the lunch order (a), the user demonstrates how to process an example order – loading a report file (b), copying fields from the form, and saving the file back to disk. The user also demonstrates the script output (c), which is returned to the form submitter to confirm that the order has been received.

5. PATTERN MATCHING

As the examples show, Turquoise scripts operate on HTML documents – copying, inserting, and deleting text and markup elements. These operations require a way to identify locations within an HTML document. Although HTML provides a few mechanisms for marking locations in a document (the most standard being the anchor tag, <A>), this is not sufficient, because the documents we want to manipulate may not have the markers we need.

Turquoise solves this problem by providing a pattern language for HTML (Figure 4).

5.1 Pattern Language

A Turquoise pattern matches a *region*, which is a contiguous, possibly empty range of characters and HTML markup. The primitive patterns are HTML elements, regular expressions, and literal text. For the benefit of users who do not know HTML, patterns

pattern →
HTML-element (e.g. Paragraph, Bulleted List, Image)
"*literal-text*"
/regular-expression/
pattern identifier (e.g. Date, Number)
region identifier (e.g., Current Page, Selection)
URL
point (a cursor position)
everything
the only *pattern*
the first *pattern*
the last *pattern*
the *n***th** *pattern*
$pattern_1$ **containing** $pattern_2$
$pattern_1$ [**just**] **after** $pattern_2$
$pattern_1$ [**just**] **before** $pattern_2$
$pattern_1$ **in** $pattern_2$
$pattern_1$ **at start of** $pattern_2$
$pattern_1$ **at end of** $pattern_2$
from $pattern_1$ **to** $pattern_2$

Examples

the first Image **in** "http://www.cs.cmu.edu/~rcm/"
(a picture of the first author)
the only Date **in** Address **at end of** Current Page
(the last-modified date on a page)
the 3**rd** List Item **in** Bulleted List **in** "http://www.yahoo.com/"
(a category in the Yahoo! Index of the Web)

Figure 4. The Turquoise pattern language.

refer to HTML elements by human-readable names like “Link” and “Paragraph”, rather than their HTML tags.

A central feature of the Turquoise pattern language is its ability to represent relationships between regions, using the operators **in**, **containing**, **before**, and **after**. For instance, the pattern “Link **in** Address **in** http://…” would find only links (HTML element <A>) appearing in the address (HTML element <ADDRESS>).

If a pattern might match more than once, the language can express assertions about the particular occurrence desired. For instance, “**the first** Paragraph **after the only** Horizontal Rule **in** http://…” asserts that only one horizontal rule (<HR>) will appear on the page, and the pattern should match the first paragraph (<P>) appearing after it.

Patterns may be named and reused in other patterns. Turquoise will include a library of predefined patterns, including commonly-used regular expressions, like Character, Word, Sentence, Number, Date, and URL.

A pattern may also refer to a named region. Web pages are the simplest form of named region, where the name is just the page’s URL. Turquoise also defines Current Page and Selection to represent regions that the user has selected interactively, and Output to represent the page returned by a script.

Finally, Turquoise patterns may contain placeholders of certain types: <literal text>, <HTML element>, and <named region>. Patterns containing placeholders can be instantiated by substituting a pattern of the same type. Thus, “<HTML element> **in** <named region>” can be instantiated to “Table **in** http://…”. A pattern with placeholders is called a *pattern template*. Pattern templates are used to infer patterns automatically from examples, which will be explained later.

5.2 Implementation

To search for a pattern in an HTML document, Turquoise parses the document into an explicit parse tree and walks over the tree trying to match the pattern. The prototype pattern matcher uses a straightforward recursive implementation, which backtracks when a subpattern fails. In the worst case, this implementation may take exponential time. In practice its performance is acceptable for searching a typical Web page, which is less than 10 kilobytes long [3]. For searching larger documents, we are considering adapting one of the algorithms devised by Kilpeläinen [9] for searching structured text databases.

command →
insert $pattern_1$ **at** $pattern_2$
replace $pattern_1$ **with** $pattern_2$
delete *pattern*
go to *pattern*
save *pattern* [**to** *pattern*]

Figure 5. Turquoise script commands.

6. INFERENCE

When the user demonstrates some actions, such as clicking on a link or pasting some text, Turquoise must infer a command that would perform the same actions in the script.

Part of the Turquoise command language is shown in Figure 5. User actions are translated into these commands by a fixed mapping. For example, the user might select an picture on a Web page, copy it, and paste it into the script window. This sequence of actions is always translated into the same script command, **insert** $pattern_1$ **at** $pattern_2$. The challenge for Turquoise is to infer $pattern_1$ and $pattern_2$, which describe the copied picture and the location to which it was copied, respectively.

An inferred pattern should satisfy four criteria. It should be:

- *Correct*: the pattern should correctly describe the region the user specified in the demonstration, otherwise it is not a proper generalization of the user's action.
- *Eager*: the inference engine should strive to guess the right pattern from only one example.
- *Robust*: the pattern should continue to work if the world changes in irrelevant ways. For instance, a pattern intended to describe the last-modified date from the bottom of a page should not be affected by modifications to the rest of the page.
- *Comprehensible*: the pattern should be understandable by the user, so that an incorrect inference can be detected and corrected. A comprehensible pattern should be easily verified, so patterns involving large numbers or invisible HTML elements, like "**the** 87$^{\mathbf{th}}$ Word **after** Comment," are not desirable.

The inference mechanism in Turquoise attempts to meet all these goals by using a knowledge base of pattern templates. These templates are heuristically chosen to be robust and comprehensible.

In order to infer a pattern for a specific region, Turquoise searches the pattern knowledge base for patterns that match the region, instantiating placeholders as needed. All the patterns that match are presented as possibilities to the user, but Turquoise chooses one pattern as its best guess. In the prototype system, the knowledge base is statically sorted, with the most robust and most frequently-used patterns first, so the prototype inference engine just uses the first match as its best guess. In the future, we will explore more sophisticated evaluation functions for determining the best pattern.

This inference mechanism guarantees correctness because only patterns that match the example region are considered. It also attempts to be eager, by making its best guess about the user's intention and using it by default.

Advanced users can improve the inference mechanism by adding new pattern templates to the knowledge base using a text editor. We may also experiment with providing different knowledge bases for demonstrating different kinds of scripts. The user may explicitly indicate which kind of script is being demonstrated, or Turquoise may use some heuristics to guess which knowledge base is appropriate.

7. RELATED WORK

Turquoise is similar to other programming-by-demonstration systems. SmallStar [7], a system for demonstrating macros in Xerox Star, introduced the notion of *data descriptions*, which are like Turquoise patterns. TELS [16] infers programs for text editing tasks, such as reformatting a bibliography. Like Turquoise, it abstracts the user's actions into abstract commands like *insert* and *delete*, but its data descriptions are limited to flat text divided into words, lines, and paragraphs. Eager [5] infers repetitive tasks in Hypercard. Eager includes a pattern language for data descriptions which is similar to the Turquoise pattern language, but Eager never shows its patterns to the user. As a result, Eager typically requires at least three examples to infer a program. By displaying the possible patterns and allowing the user to verify or fix them, Turquoise can often infer from only one example.

Other systems give end-users the ability to develop CGI scripts, though not by demonstration. Several systems, like Zelig [15], represent a dynamic page as an HTML template with variable fields, which are computed at runtime by database queries or script code. Unlike Turquoise, these systems generally require knowledge of HTML. WebWriter [4] allows users to create the templates without knowing HTML, in a WYSIWYG editor. Since WebWriter is a CGI application itself, however, its editing interface is limited to form fill-out, which can be tedious.

The Turquoise pattern language, which identifies locations in HTML, is similar to other languages for representing transformations or doing searches in structured text. Bonhomme and Roison [2] described a language for specifying HTML editing transformations which allows an HTML editor to be extended with new transformations, but this language can only match a local area of the HTML tree – a few nodes and their immediate neighbors and children. Pattern matching of this kind has been called the classic "tree matching problem" [8]. The Turquoise pattern language is more general, however, permitting arbitrary intervening neighbors and descendants in its **in**, **before**, and **after** patterns. Kilpeläinen [9] studied this general tree matching problem in the context of searching large structured-text databases, and found algorithms that solve it in polynomial time. Turquoise also draws ideas from *p-strings* [6], another language for searching structured-text databases.

8. STATUS AND FUTURE WORK

The prototype system uses copy-and-paste events to infer composite page scripts. It needs further development before it can be used to demonstrate a broader class of scripts. One important area of future development is inferring conditional branches and iteration, which are needed by many scripts. Much of the ease of using Turquoise stems from its ability to infer from a single example, but conditions and iterations generally require multiple examples to demonstrate. The prototype could be augmented to generalize from multiple examples, possibly using hints from the user, as in Gamut [10].

More work is needed to develop good pattern knowledge bases for the kinds of scripts we identified as demonstrable. We also want to explore different evaluation functions that Turquoise can use to pick out a good pattern from all the matches in the knowledge base.

To extend the scope of Turquoise scripts, we plan to add a number of built-in gateways to the prototype, including a calculator, a command shell, and an OLE interface. In addition, a future version of the prototype will be capable of translating Turquoise scripts into a traditional Web scripting language, such as Perl, Tcl, or Java.

We also plan to user-test the pattern language to determine whether users can read and write patterns, and whether the system's inferred patterns are comprehensible.

9. CONCLUSION

The Turquoise architecture is suitable for demonstrating a broad range of Web scripts, including composite pages, assistants, filters, form processors, and active pages. With

Turquoise, Web users only need their familiar browsing and editing skills to create dynamic pages and automate repetitive tasks on the Web.

The Turquoise pattern language satisfies the need for identifying locations in HTML documents. Unlike HTML anchors, which must be explicitly inserted in a document by its author, Turquoise patterns allow an arbitrary user to identify regions of a page for extraction, reformatting, or other processing, without needing to change the original document. By inferring patterns from example, Turquoise puts these capabilities in the hands of users who do not know HTML.

10. ACKNOWLEDGMENTS

For help with this paper, we would like to thank Ellen Borison, Laura Cassenti, Edwin Chance, Richard McDaniel, and Eric Tilton. We also gratefully acknowledge Dave Long and America Online for providing a special version of AOLpress for the Turquoise prototype.

11. REFERENCES

1. America Online. AOLpress. http://www.aolpress.com/
2. Bonhomme, S., and Roisin, C. Interactively restructuring HTML documents. In Proc. 5th Intl World-Wide Web Conference, WWW'96 (Paris, France, May 1996). http://www5conf.inria.fr/fich_html/papers/P16/Overview.html
3. Bray, T. Measuring the Web. In Proc. 5th Intl World-Wide Web Conference, WWW'96 (Paris, France, May 1996). http://www5conf.inria.fr/fich_html/papers/P9/Overview.html
4. Crespo, A., and Bier, E.A. WebWriter: a browser-based editor for constructing Web applications. In Proc. 5th Intl World-Wide Web Conference, WWW'96 (Paris, France, May 1996). http://www5conf.inria.fr/fich_html/papers/P35/Overview.html.
5. Cypher, A. Eager: programming repetitive tasks by example. in Proc. of CHI '91 (New Orleans, May 1991), ACM Press, 33-39.
6. Gonnet, G.H., and Tompa, F.W. Mind your grammar: a new approach to modelling text. In Proc. 13th Very Large Databases Conference (Brighton, 1987), 339-345.
7. Halbert, D. SmallStar: programming by demonstration in the desktop metaphor. in Watch What I Do: Programming By Demonstration, Allen Cypher, ed. MIT Press, Cambridge MA, 1993, pp. 104-123.
8. Hoffmann, C.M., and O'Donnell, M.J. Pattern matching in trees. JACM 29, 1 (January 1982), 68-95.

9. Kilpeläinen, P. Tree Matching Problems With Applications to Structured Text Databases. Technical Report A-1992-6, Department of Computer Science, University of Helsinki, Finland. November 1992.

10. McDaniel, R. Improving communication in programming-by-demonstration. In CHI'96 Conference Companion (Vancouver BC, Canada, April 1996), 55-56.

11. Myers, B. Demonstrational interfaces: a step beyond direct manipulation. IEEE Computer 25, 8 (August 1992), 61-73.

12. Myers, B., et al. The Amulet 2.0 Reference Manual. Carnegie Mellon University Computer Science Dept Technical Report CMU-CS-95-166-R1. April 1996.

13. National Center for Supercomputing Applications (NCSA). The Common Gateway Interface. http://hoohoo.ncsa.uiuc.edu/cgi/

14. Nielsen, H.F., and Håkon, W.L. Towards a uniform library of common code. in Proc. 2nd Intl World-Wide Web Conference, WWW'94, (Chicago, IL, December 1994. http://www.w3.org/pub/WWW/Library/User/Paper/LibraryPaper.html

15. Varela, C.A., and Hayes, C.C. Providing data on the Web: from examples to programs. in Proc. 2nd Intl World-Wide Web Conference, WWW'94 (Chicago IL, December 1994). http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/varela/paper.html.

16. Witten, I.H., and Mo, D. TELS: learning text editing tasks from examples. in Watch What I Do: Programming By Demonstration, Allen Cypher, ed. MIT Press, Cambridge MA, 1993, pp. 183-203.