Smart Bookmarks:
Automatic Retroactive Macro Recording on the Web

Darris Hupp and Robert C. Miller
MIT CSAIL
32 Vassar St
Cambridge, MA 02139 USA
{dhupp,rcm}@mit.edu

ABSTRACT
We present a new web automation system that allows users to create a smart bookmark, consisting of a starting URL plus a script of commands that returns to a particular web page or state of a web application. A smart bookmark can be requested for any page, and the necessary commands are automatically extracted from the user’s interaction history. Unlike other web macro recorders, which require the user to start recording before navigating to the desired page, smart bookmarks are generated retroactively, after the user has already reached a page, and the starting point of the macro is found automatically. Smart bookmarks have a rich graphical visualization that combines textual commands, web page screenshots, and animations to explain what the bookmark does. A bookmark’s script consists of keyword commands, interpreted without strict reliance on syntax, allowing bookmarks to be easily edited and shared.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces; H.5.2 [User Interfaces]: User-centered design.


Keywords: Web automation, browsers, macro recording.

INTRODUCTION
As the Web has grown and matured, new technologies such as scripting languages, cookies, and XML have made the Web much more dynamic and personalizable. At the same time, however, web interaction has become more complicated. Today, many web pages are dynamically generated by client-side Javascript, and it is not always possible to use a conventional URL bookmark to return to them. This is unfortunate, because these kinds of pages may require significant effort from users to repeatedly access, and would therefore benefit from a bookmarking facility that allows a page to be recovered with a single click.

Our solution to this problem is a smart bookmark, consisting of a starting URL plus a script of commands that returns to a particular web page or restores a particular state of a web application. Smart bookmarks are useful not only for recalling hard-to-reach, dynamic web pages, but also for automating common tasks on the Web. Smart bookmarks can be parameterized, so that by entering just one or two parameters each time (e.g. the departure and return dates for a flight search for instance), users can reach the state or information they need more quickly. Smart bookmarks would also allow users to easily share dynamically-generated content. For example, a user could customize a computer he or she wishes to purchase online, save the completed configuration page as a bookmark, and send the bookmark to a friend to look over. Smart bookmarks might also be used as step-by-step tutorials, showing how to perform a task on a website.

Our prototype Smart Bookmarks system is implemented as a sidebar inside the Firefox web browser. From the Smart Bookmarks sidebar, users can bookmark the current web page, replay an existing bookmark, or view and edit a graphical representation of a bookmark. Smart Bookmarks continuously monitors and records the actions the user makes while browsing the Web and stores them in a history in the background. When a web page is bookmarked, the system automatically determines the sequence of commands needed to return to the page, and saves the sequence as a bookmark.

Once a bookmark has been created, it can be run with a single click. Smart Bookmarks replays the bookmark’s

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UIST’07, October 7-10, 2007, Newport, Rhode Island, USA.
Copyright ACM 1-59593-023-1/07/0010...$5.00.

Figure 1. Smart Bookmarks sidebar showing a bookmark that configures a PC on Dell’s web site.
command sequence in the web browser to return to the page that was originally bookmarked. Users can also indicate that they should be prompted for the inputs to certain commands in the bookmark before each run.

In order to clearly show users what a particular bookmark will do when run, Smart Bookmarks displays a graphical representation of the bookmark. The graphical view includes a screenshot of each command in the bookmark showing the part of the web page that command acted on when originally created. Smart Bookmarks also allows users to easily edit a bookmark once it has been created, for instance by changing the inputs to a command, copying commands, or removing a command from the bookmark entirely. Users can dynamically edit or even create bookmarks in real time by explicitly performing the actions they want to capture in their browser.

One issue with automatically capturing browsing actions and making it easy for users to replay those commands later is that it is possible that one or more of those commands may have some undesirable side effect that the user does not want to be repeated. For instance, if the user purchases a book online or requests to join a mailing list, he or she probably does not want to repeat those actions when running a bookmark. While the graphical bookmark representations should make it easier for users to catch when cases like this may occur, a more reliable and less error-prone solution is desired. To that end, we are currently working on a two-part system that detects potential undesirable side effects when a bookmark is run by using various heuristics in combination with a shared database that will store votes provided by users of Smart Bookmarks about a particular command’s possible side effects.

A key contribution of this paper relative to previous efforts at macro recording on the web [1],[10],[13] is the notion of automatic, retroactive macro recording. Other macro recorders have a VCR-like interface that requires the user to be proactive, explicitly starting the recorder before performing actions for the macro, and explicitly stopping it when the macro is done. Smart Bookmarks, in contrast, generates a macro with a single retroactive command, invoked after the actions have been performed, and automatically scans backwards through the history to find the macro’s starting point. Automatic retroactive macro recording is better suited to just-in-time automation tasks, and neatly fits the bookmarking model that is already familiar to many web users.

A second contribution is a rich graphical visualization of what a recorded bookmark does, including text explanations, screenshots, and animations. This rich feedback is intended to make bookmarks easier to understand, inspire greater confidence in the automation, and allow users to exchange bookmarks with each other more easily and understandably. Furthermore, the textual representation of a bookmark consists of keyword commands [9], which allow the user to edit the bookmark directly without needing to learn the syntax of a scripting language.

The rest of this paper is organized as follows. First we survey related work. Then we describe several scenarios in which Smart Bookmarks solve problems. Subsequent sections describe the user interface of the bookmarking system in more detail, and explain how it is implemented. We then present an evaluation of smart bookmarks on 25 web sites, and discuss some of the limitations and implications of this bookmarking approach, particularly with respect to security, side-effects, and robustness to change. Finally, we conclude and discuss future work.

RELATED WORK

The Smart Bookmarks system is based on Chickenfoot [2], a scripting language that allows users to automate and customize web sites within the web browser. Chickenfoot scripts are written at the abstraction level of the user interface, with commands like click(“search button”), rather than at the level of raw HTML or HTTP targeted by other web scripting approaches. Chickenfoot also introduced the idea of keyword commands [9] as a syntax-free alternative to traditional scripting languages.

Koala [10] combines keyword-command scripting with macro recording, so it is very closely related to Smart Bookmarks. With Koala, the user can record a sequence of actions and generate a script of keyword commands that can be replayed later. Recorded scripts are stored automatically on a wiki, which might be shared by a workgroup, allowing easy exchange and improvement of scripts. Koala uses proactive macro recording, where the user explicitly starts and stops the recording; it does not support retroactive recording. Recorded Koala scripts are also pure text, unlike the rich visual representation captured by Smart Bookmarks. Smart Bookmarks also continuously records the user’s actions, which it can display as a graphical history. Finally, Smart Bookmarks can record various representations of each action (not just keyword commands) for greater robustness.

WebVCR [1] and Web Macros [13] are earlier systems that used explicit proactive recording. WebVCR is a Java applet designed for Netscape that could track the user’s actions while recording was underway. The WebVCR paper also coined the term smart bookmark for a web macro that returns to a particular page. The other system, Web Macros, is a proxy-based system that rewrites the HTML of web pages while recording, allowing the user to parameterize a recording while the recording is in progress. Web Macros can also detect when a macro playback might be going off the rails, by comparing fingerprints of the pages seen during recording with pages seen during playback. Web Macros also proposed a “Get Me Here” feature similar to retroactive macro recording, but didn’t describe how to implement it. Both WebVCR and Web Macros use a low-level, internal representation for the recorded macro, which is not editable by the user or displayed in the interface.

Other research has looked at applying programming-by-demonstration to the Web for purposes other than macros, including constructing dynamic pages and portals [15],...
combining web sites into a dataflow pipeline [5], and extracting data from web pages [4][6].

One kind of retroactive macro recording can be found in Chimera [8], a drawing editor that offers a graphical history of the user’s recent actions. The user can create a macro retroactively by selecting and copying a part of the history. Smart Bookmarks carries this idea farther, by selecting the necessary part of the history automatically when the user requests a bookmark. The rich graphical feedback in Smart Bookmarks has its roots in the comic strip metaphor used by Chimera and also by Pursuit [11]. The Smart Bookmarks history is also similar to work by Kaasten and Greenberg [7], which integrated history and bookmarking into a single visual page history. Smart Bookmarks displays finer-grained actions, however, not whole pages.

Some Javascript-heavy (AJAX) web sites that are hard to bookmark offer a feature that generates a bookmarkable page by packaging up the current state of the site into a URL. A notable example is the “Link to this page” feature on Google Maps. Most AJAX applications do not offer such a feature, however. Out of the thousands of Google Maps mashups, very few offer bookmarkable links.

SCENARIOS

In this section, we present three scenarios to illustrate how Smart Bookmarks can be used in ordinary web browsing.

Scenario 1: Sharing Bookmarks with Other People

Tanya needs to buy a new office PC. She uses Dell’s web site to select a base model, and then goes through a number of forms in which she customizes the hardware components and preloaded software to meet her needs. Once she’s done configuring the new PC, however, she realizes that she can’t continue the order process, because she doesn’t know how to pay for it. She needs to send it to Luis, a colleague who actually handles the payments. So Tanya presses Create Bookmark to save her place, generating a smart bookmark that is able to return to the customized PC she has created (Figure 1). She saves the bookmark to a file and emails it to Luis. Luis imports the bookmark into his web browser, scans its visual representation briefly to see what it does, and runs it in order to recreate the same configuration and complete the ordering process.

If Luis did not have Smart Bookmarks installed in his browser, then Tanya has another option as well: she can copy the bookmark to the clipboard and paste it into her email client (Figure 2), where it appears as a list of keyword commands. Luis can follow these instructions by hand to recreate the same configuration. He can also paste them into a new Smart Bookmark, or even run them as a keyword command script in Chickenfoot or Koala.

Scenario 2: Parameterized Bookmarks

Henry lives in Boston, but his girlfriend lives in New York. He often flies down to visit for the weekend. He can create a smart bookmark that searches for flights from Boston to New York by bookmarking the result page of a flight search (Figure 3). Since Henry does this search frequently, but using different days each time he does the search, he edits the bookmark to turn the Leave and Return dates into parameters that the browser requests every time he runs the bookmark (Figure 4).

Scenario 3: Bookmarking Dynamic Pages

Phyllis would like to have a bookmark that goes directly to the statement of her net worth, but this page is not normally bookmarkable – she has to log in and navigate to it. Both the login information and navigation can be captured as part of a smart bookmark, however. Her password is automatically detected and replaced with ***** when the bookmark is displayed.

Figure 2. A smart bookmark pasted as text into an email message, so that a user without Smart Bookmarks installed can follow the steps manually.

Figure 3. Smart bookmark for a flight search.

Figure 4. The flight search bookmark is parameterized, so that when the bookmark is replayed, commands that need parameters are displayed in the sidebar, with textboxes for the user to enter the parameter values.
The Smart Bookmarks interface appears in the web browser as a sidebar that displays a list of the smart bookmarks that the user has created (Figure 6a). Clicking on a bookmark displays the steps of the bookmark (Figure 6b).

Creating a Bookmark
The user makes a bookmark for the current page by pressing the Create Bookmark button ( ). The system responds by generating a sequence of commands that would return to the current page in the browser, if those commands were replayed. We refer to this page, the destination of a smart bookmark, as the target page. The generated command sequence always begins with a starting URL, so that the bookmark can be replayed regardless of the current state of the browser. In general, however, a user can edit a smart bookmark to start with any command, so a smart bookmark can be used as a general macro, not just as a pointer to a particular target page.

If the target page is normally bookmarkable (i.e., its URL is a valid way to return directly to it), then the generated smart bookmark contains only one command, the URL. If the target page is not bookmarkable, however, the system searches backwards through the browser’s page history for a page that is bookmarkable. The most recent bookmarkable page, which we refer to as the starting page, becomes the initial URL of the smart bookmark. The details on how we determine whether or not a page is bookmarkable are found in the Implementation section, later in this paper.

Having found a starting URL for the bookmark, the system then appends a sequence of commands corresponding to the actions the user took to get from the starting page to the target page. The commands used in smart bookmarks are the same set of commands used in Chickenfoot (go to URL, click button or hyperlink, type text into textbox, check/uncheck checkbox or radio button, choose list-item). The

Replaying a Bookmark
To replay a bookmark, the user first opens it in the Smart Bookmarks sidebar, and then presses the Go button ( ). A smart bookmark can also be replayed directly from the bookmark list, without opening it, by pressing the play button next to its name or right-clicking and pressing Play. While a bookmark is playing, it can be stopped by pressing the Stop button in the sidebar ( ) or the Stop button in the standard Firefox toolbar.
Since one use for a smart bookmark is to save a session at a web site (such as a point in an ordering process) with the intention of returning to it later, the system also has a reminder feature. Whenever the user visits a smart bookmark’s starting page by any means (e.g., with an ordinary bookmark, a search engine, or typing a URL in the address bar), a drop-down message is displayed in the browser window, indicating that a smart bookmark is ready to run from this page, and offering a Play button to replay it immediately (Figure 9).

Like ordinary bookmarks, which may break when URLs change and web resources move, smart bookmarks can also break when a web site changes, and fail to play back correctly. When a command in a bookmark cannot be played because the page component it uses can no longer be found, the playback stops at that command, and the broken command is highlighted in red in the sidebar. The user can then continue the process by interacting with the web site manually, or edit the broken bookmark to correct it (as described in the next section).

For debugging or partial playback, the sidebar has a secondary toolbar that offers buttons for single-stepping ( ), playing up to a selected command ( ), and playing from a selected command until the end of the bookmark ( ).

**Editing a Bookmark**

A bookmark can also be edited manually by the user. This might be done for several reasons, including: (1) fixing a bookmark that has become broken by a web site change; (2) adapting an existing bookmark for a new purpose without re-recording it; (3) streamlining an automatically-generated bookmark by removing unnecessary steps from it; and (4) creating a bookmark macro by hand.

One simple (and common) kind of edit merely changes the text entered by a *type* command. For example, in the command *type “alice” into username textbox*, the user name can be changed to “bob” simply by clicking on “alice” and editing it. Since this kind of edit doesn’t change the textbox involved, the visual representation of the command (the screenshot) is still preserved, even though the value shown in the textbox may be stale.

The textual representation of the whole command can also be edited by right-clicking on it and selecting Edit Command. The user can use this feature to change *uncheck* into *check* or vice versa, or to change a *click* command so that it clicks a completely different button. It is not essential that edited commands follow a precise syntax, however, because the commands in a Smart Bookmark are actually *keyword commands*. When a keyword command is executed, the system searches for a page component (such as a button) and an action on that component (such as clicking) that best match the words found in the command. As a result, even if the user changes the command to something like *username = alice*, it is still very likely to succeed.

When the full textual representation of a command is changed by the user, the command’s visual representation (the screenshot) is discarded, because the system has no way to know whether the command still refers to the page component shown in the screenshot. Whenever the bookmark is played back, however, new screenshots are taken automatically, so the visual representation will be filled in as soon as the user tests the edited bookmark.

Commands can be deleted from a bookmark by selecting them and deleting them through a context menu, which is useful for removing unnecessary steps from a long bookmark. Commands can also be copied and pasted from one bookmark to another using the clipboard. Whole bookmarks can also be copied and pasted, in order to make variants of bookmarks without re-recording them. Commands and bookmarks that are copied and pasted carry along both textual and visual representations.

In addition to manual editing, the user can change a bookmark by recording new commands at any point in the bookmark. New commands are recorded by toggling on a recording mode called Live Editing ( ), selecting a point in the bookmark where the new commands should be inserted, and then doing actions in the web browser (e.g., clicking buttons and hyperlinks or operating form controls). When Live Editing is on, the recorded commands are immediately inserted in the bookmark. Live Editing can also be used to fix a broken command by selecting it and doing the action that the broken command should have done instead.

Live Editing can also be used to support the proactive recording model used by other systems [1],[10],[13]. A user can record a bookmark manually by creating a new empty bookmark using another toolbar button ( ) and toggling the Live Editing mode to start and stop recording actions into the bookmark.

**Graphical History**

In order to support automatic bookmark generation, Smart Bookmarks is in fact constantly recording user actions. A graphical history of recorded actions can be viewed by pressing View Browsing History ( ). The history is displayed in the same form as a bookmark, using both textual and visual representations of each user action.

The history can be used as a source of commands for bookmarks as well, so that the user can reach into their recorded history and copy and paste commands into bookmarks. Thus the system also supports manual retroactive macro creation, in the manner of Chimera [8], in addition to automatic retroactive creation.
of their original commands, for two reasons. First, it makes
bar (bookmark are collected and displayed together in the sid-
the user in-
Alternatively, a parameter value can be entered
the web page
s when the user fills it in and presses
the smart bookmark by default. After the user fills in the requested parameters and presses the button again, the bookmark finally runs.

IMPLEMENTATION
Our Smart Bookmarks prototype is implemented as an ex-
tension for the Firefox web browser, using a combination
of XUL, Javascript, and Java. It also uses Chickenfoot, a
Firefox extension for web automation and customization,
which it relies on for interpreting keyword commands in a
bookmark.

This section describes key parts of the prototype imple-
mentation. First we explain how the system records the user’s
web browsing actions in a history, in anticipation that the
user will request a smart bookmark. Then we describe how
to generate a smart bookmark from the history, and how a
smart bookmark is replayed.

Recording the Action History
While the user is using the Web, the Smart Bookmarks
extension is constantly recording the user’s actions in the
background. Unlike the conventional web browser history,
however, this recording includes not just the URLs that the
user visits, but also low-level clicking and form-filling ac-
tions. Low-level actions are observed by installing
Javascript event listeners for click, change, and load
events at the root of the Firefox web browser (to be precise,
on the tabbrowser XUL element that contains Firefox’s
browsing tabs). Click events occur when the user clicks
the mouse on a button or hyperlink; change events occur when
a textbox, radio button, or checkbox is changed; and load
events occur when the browser visits a new URL.

When one of these events occurs, the system records it in
the action history, along with a description of the page
component that received the event (e.g., a button or
textbox). The description has several parts that will be use-
ful for generating a bookmark later: (1) a text label for the
page component; (2) an XPath pattern identifying the com-
ponent’s location in the Document Object Model (DOM) of
the web page; and (3) a screenshot of the page component
with some of its context.

The label of a page component is extracted from the web
page. For components that contain their own label, such as
buttons or hyperlinks, the label is constructed from the text
nodes contained by the component, text found in its value
attribute, or text found in the alt or title attributes of an
image inside the component. For checkboxes and radio
buttons, the label consists of the text following the element.
For textboxes and listboxes, the label is found by an ap-
proach similar to Koala’s [10]: first checking the element’s
previous siblings for visible nonempty text, and if that fails,
then searching the previous siblings of the element’s ances-
tors until text is found. This algorithm generally finds a

Parameterization
A bookmark can also be parameterized, so that instead of
having hard-coded values for textbox entries, the user can
fill in the values when the bookmark is replayed.

The system supports two kinds of parameterization, which
can be chosen by right-clicking on a command and select-
ing either “Choose value while running bookmark” or
“Choose value before running bookmark.” The former
selection means that when bookmark playback reaches the
command, the bookmark pauses, highlights the textbox on
the web page that the user needs to fill in, and waits until
the user fills it in and presses on the bookmark again.

Alternatively, a parameter value can be entered before the
bookmark starts running. When the user initially presses
the button on a bookmark, all such parameters in the
bookmark are collected and displayed together in the side-
bar (Figure 4). The parameters are displayed in the context
of their original commands, for two reasons. First, it makes
the parameter display consistent with the bookmark dis-
play. Second, the user sees a richer description of the par-
parameter, not only a textual name for it (e.g. Leave textbox)
but also a screenshot of the textbox where it will actually
be entered. Each parameter is ready for editing, set to the
value found in the original recorded bookmark by default.

Figure 10. Graphical history of the user’s actions.
Page changes are marked by a gradient boundary.

The browsing history includes a go to command for every
new page loaded, even if the page load was actually trig-
gered by a preceding click command. These page loads are
further emphasized with a gradient boundary, so that the
user actions affecting a single page are visually grouped
together (Figure 10). In a bookmark, these extra go to
commands would be at best redundant, since the previous
click command should have the same effect, and at worst
completely wrong, since the go to URL may actually be
unbookmarkable. The extra go to commands are included
in the history so that a starting point is available when the
user copies a macro manually out of the history. In the
current prototype, the remaining go to commands must be
manually deleted from the bookmark, but we plan to fix
this so that unnecessary go to commands are automatically
removed.

Parameterization
A bookmark can also be parameterized, so that instead of
having hard-coded values for textbox entries, the user can
fill in the values when the bookmark is replayed.

The system supports two kinds of parameterization, which
can be chosen by right-clicking on a command and select-
ing either “Choose value while running bookmark” or
“Choose value before running bookmark.” The former
selection means that when bookmark playback reaches the
command, the bookmark pauses, highlights the textbox on
the web page that the user needs to fill in, and waits until
the user fills it in and presses on the bookmark again.

Alternatively, a parameter value can be entered before the
bookmark starts running. When the user initially presses
the button on a bookmark, all such parameters in the
bookmark are collected and displayed together in the side-
bar (Figure 4). The parameters are displayed in the context
of their original commands, for two reasons. First, it makes
caption correctly when it appears on the left or above the component.

As a backup representation, the system also generates an XPath pattern for the page component, which is a sequence of tag names and child indices that traces out the path to the component from the root of the DOM, e.g. 
\[HTML/BODY/TABLE[2]/TBODY/TR[3]/DIV/INPUT\]. If no label could be found, then the XPath pattern is used during playback instead.

Finally, the system takes a screenshot representing the action, by creating an offscreen canvas element, rendering the current web page to it with the Canvas API, and obtaining the resulting bitmap in PNG format. For load events, a screenshot of the entire page is stored, after the page has finished loading. For other actions, the screenshot contains only the affected page component, plus 50 pixels of context on each side.

The system records a separate action history for every browsing tab currently open in Firefox. When a tab is closed, its browsing history is discarded. Actions are not stored persistently unless the user actually generates a smart bookmark.

**Generating a Bookmark from the Action History**

When the user requests a smart bookmark for the current web page, the system must extract a sequence of commands from the history that return to the same page. In principle, the entire history could be used as this sequence, because it is guaranteed to start with a bookmarkable page (the URL that started the browser window) and reach the current web page. Since the full history might be long, however, we want to find the shortest suffix of the history that successfully returns to the page. This is done by searching for the most recent bookmarkable URL in the history, and using that as the starting point for the bookmark.

We search for the most recent bookmarkable URL by attempting to refetch each URL in the history, starting with the current page and proceeding backwards. While refetching pages, we temporarily discard session cookies (cookies which expire when the browser is closed), in order to simulate trying to revisit the URL in a later browser session.

A URL is bookmarkable if refetching it produces the same page (or nearly the same page) as when it was originally fetched in the user’s history. Even bookmarkable pages may differ when refetched a short time later (e.g. showing different ads or a different time of day), so the comparison must tolerate some change, but we require that change to be small relative to the whole page. We compare the refetched page with the original page by computing the edit distance between the two DOM trees [3], which finds a set of insertion, deletion, and update operations that transform the original page into the refetched page. The cost of inserting or deleting a subtree \( T \) is measured by the total weight found in the subtree, denoted \( w_T \), where an element node has weight 1 and a text node containing \( n \) characters has weight \( \log n \).

The overall amount of change between the original page \( P \) and the refetched page \( R \) is thus the sum of the weights of the differences between \( P \) and \( R \), divided by \( w_P + w_R \). If \( P \) and \( R \) are identical, this measure is 0, indicating no change. If \( P \) and \( R \) are completely different, such that the smallest set of edits to transform \( P \) to \( R \) requires deleting all of \( P \) and substituting all of \( R \), then the measure is 1.

Our current prototype uses a threshold of 0.1 to identify bookmarkable URLs. Intuitively, if the refetched page \( R \) is less than 10% different from the original page \( P \), then we conclude that \( R \) and \( P \) are effectively the same page, so the URL is bookmarkable. In our experience, the results are not particularly sensitive to this threshold. A bookmarkable URL tends to produce very similar (though frequently not identical) pages, while a nonbookmarkable URL tends to produce radically different pages (e.g., producing a form rather than the form’s results, or a redirect to a higher part of the web site).

We also briefly experimented with comparing screenshots of the original page and the refetched page, because it might be closer to how a user would judge whether pages are the same. In practice, however, small changes to a page could cause significant differences in its screenshot, even to the point of moving layout or reflowing text. Aligning two screenshots is significantly harder than aligning DOMs.

**Playback**

A bookmark is replayed by running each command in sequence. Go to commands are run simply by loading the URL in the browser. Other commands, such as click and enter, are run using Chickenfoot’s algorithm [2] to match the label in the command against the components found on the web page. If no label was found for a component when the command was recorded, then the recorded XPath pattern is used to find the component instead. Finally, if the user edited the bookmark and typed in a new command (rather than recording it), then Chickenfoot’s keyword commands algorithm [9] is used to interpret the command. Commands can fail during playback. For example, the command click Search button will fail if no button labeled “Search” can be found on the page. Usually this would happen because the web site has changed and broken the bookmark, but sometimes it happens because of a race condition between bookmark playback and the loading page. Although the bookmark player does not attempt to run the command until the page has been fully downloaded from the web server, some Javascript-heavy web pages may still not be complete, which is hard to detect. As a result, when a command fails, the system waits three seconds and then retries it. If the command still fails after three tries, then playback stops and the failed command is highlighted in red in the sidebar.

**EVALUATION**

To demonstrate the range of web sites and tasks that Smart Bookmarks can support, we created bookmarks for 25 popular websites, using tasks that fall into one of the three usage scenarios described earlier in this paper: accessing
dynamic web pages, automating repetitive tasks, and saving or sharing customized dynamically-generated content.

Each of the tasks was performed using normal browsing in Firefox 2.0 with the Smart Bookmarks extension installed. When the task was complete, or the desired web page was reached, we used Smart Bookmarks to automatically create a bookmark that could repeat the task. Table 1 lists the 25 tasks, including the amount of time Smart Bookmarks required to generate the bookmark, and the number of actions that were included in the bookmark. Most bookmarks took only a few seconds to generate, but some needed up to 15 seconds, largely because the web site was slow to respond during the search for the starting point of the bookmark. The average creation time over all the tasks was 4.2 seconds, and the average number of actions was 6.

Table 1 only shows successful bookmarks, to demonstrate the range of the system, but we have also found a few web sites that our prototype does not handle. For example, pages inside Gmail cannot be bookmarked because of a buggy interaction between Gmail’s redirections and the Smart Bookmarks history mechanism. For other sites, the label-finding algorithm sometimes generates the wrong label for a textbox, which makes the script harder to read but doesn’t affect correct playback, because the playback uses the same algorithm to find the textbox again. Sometimes label finding fails entirely — for example, Amazon makes heavy use of image buttons with no alt attribute — but the backup XPath patterns are sufficient to play back these commands. We have not yet found a site on which bookmark generation fails to identify a suitable starting page automatically.

In order to gauge the robustness of the bookmarks to web page changes over time, we attempted to replay the bookmarks one month and four months after they were originally created. Of the 25 bookmarks, 18 could still be replayed successfully after four months. The seven failing bookmarks were broken for various reasons.

- Two failures (the Dell and Gateway laptop customization bookmarks) were caused by changes in product offerings: Dell no longer offered the particular printer chosen by the bookmark, and Gateway no longer sold the base computer model.
- Three failures (Facebook, GoogleMaps, and LiveMaps) occurred because of changes in a web page encountered by the bookmark. For example, the Google Maps bookmark included clicks on the Zoom In button, which were recorded as an XPath pattern because the button originally lacked a tooltip label. A restructuring of the Google Maps page broke this XPath.
- MySpace failed because an interstitial advertisement page is sometimes but not always presented, disrupting the bookmark. A similar problem can arise with a login bookmark when the user is already logged in, since the login page is then skipped. It may be useful to automatically detect common kinds of skipped pages (such as login pages and ads) so that the replay can adapt automatically.
- Finally, the Apple bookmark failed because its starting URL contained a temporary session identifier, which eventually expired.

In all cases, the bookmark replay correctly stopped at the first broken command.

**DISCUSSION**

A system like Smart Bookmarks faces a number of challenges on today’s Web. A few are particular to our approach, but others are general problems of web automation, shared by other systems as well. This section discusses some of these challenges, specifically: (1) recording and replaying low-level mouse and keyboard events; (2) preserving the user’s security and privacy when automating sensitive web sites; (3) protecting the user from undesirable side-effects triggered by automation; and (4) creating automation that is robust to changes on the Web.

<table>
<thead>
<tr>
<th>Website &amp; Task</th>
<th>Time</th>
<th>N</th>
<th>Replay after 1mo</th>
<th>Replay after 4mo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hotmail: login and view inbox</td>
<td>4.6 s</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MITWebmail: login, view sent mail</td>
<td>1.8 s</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>YahooMail: login, view inbox, mark messages read</td>
<td>2.0 s</td>
<td>9</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Facebook: login, view user profile</td>
<td>1.7 s</td>
<td>5</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>MySpace: login, view messages</td>
<td>2.5 s</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>BankofAmerica: login, view portfolio</td>
<td>1.7 s</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Chase: login, view account summary</td>
<td>3.5 s</td>
<td>4</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Vanguard: login ,view performance</td>
<td>1.7 s</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Expedia: search for a flight</td>
<td>11.1 s</td>
<td>8</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Orbitz: search for a flight</td>
<td>14.2 s</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>CheapTickets: search for a flight</td>
<td>12.4 s</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Travelocity: search for a hotel</td>
<td>7.5 s</td>
<td>9</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Greyhound: find, choose bus schedule</td>
<td>1.5 s</td>
<td>7</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Amtrak: find, choose train schedule</td>
<td>2.4 s</td>
<td>3</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dell: get updated drivers for a model</td>
<td>4.1 s</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Zip Realty: search for real estate</td>
<td>1.3 s</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MyNewPlace: search for apartments</td>
<td>1.8 s</td>
<td>8</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Amazon: login, view package tracking</td>
<td>1.9 s</td>
<td>6</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Dell: customize laptop to buy it</td>
<td>2.3 s</td>
<td>5</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Gateway: customize laptop to buy it</td>
<td>13.6 s</td>
<td>8</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Lenovo: customize laptop to buy it</td>
<td>2.9 s</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Apple: customize a laptop to buy it</td>
<td>2.3 s</td>
<td>3</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>HousingMaps: view apartments</td>
<td>1.2 s</td>
<td>5</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GoogleMaps: locate a region on map</td>
<td>1.4 s</td>
<td>7</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>LiveMaps: locate a region on map</td>
<td>3.4 s</td>
<td>7</td>
<td>✓</td>
<td>×</td>
</tr>
</tbody>
</table>

Table 1. Smart bookmarks for 25 tasks created in Feb-Mar 2007. *Time* is the time required to generate the bookmark; *N* is the number of commands in the bookmark; *Replay* shows which bookmarks could be successfully replayed 1 month and 4 months later.
Low-Level Mouse and Keyboard Events
As web sites present richer AJAX interfaces that more closely resemble desktop applications, the simple widget-change recording techniques used by existing web macro recorders (including WebVCR, WebMacros, Koala, and Smart Bookmarks) can no longer completely capture the user’s interaction. For example, Smart Bookmarks does not currently record mouse dragging events on sites like Google Maps or Live Maps, so an interaction that involves dragging the map will not be completely replayed by the bookmark. (The bookmarks in Table 1 did not drag the map.) Low-level mouse movements are hard to generalize and replay correctly in the face of changes in screen resolution, page layout, and window size. Smart Bookmarks does not currently record low-level keystrokes, either, so keyboard shortcuts (e.g. on Gmail) are not replayed.

Security
When a generated bookmark includes a command that enters a password into a web site, then our system detects and obscures the password in the usual way, e.g. as type="password" into password textbox. This behavior is triggered automatically for password entry textboxes, marked by the HTML attribute type="password".

Bookmarks that contain passwords are therefore easy to detect and handle more sensitively than other bookmarks, if desired. For example, replay of password-containing bookmarks might be controlled by a master password. When the user exports or copies a bookmark containing a password, with the intention of sharing the bookmark with another user, the system could warn the user that the bookmark might expose the user’s password, and offer to erase the login information and parameterize it instead. Our current prototype does not yet implement these additional features, but they would be straightforward to add.

More challenging are bookmarks that contain undetected sensitive information, such as credit card numbers, personal identifiers like Social Security numbers, or other sensitive personal information. Additional heuristic rules may help detect some of these – e.g., credit card numbers are particularly easy to detect reliably. In general, however, Smart Bookmarks may pose a greater risk than other web automation systems of inadvertently including sensitive data in a bookmark, because Smart Bookmarks are generated retroactively and automatically. Our design seeks to mitigate that risk by providing rich feedback about what the bookmark does (with text, screenshots, and animations), and displaying that feedback immediately after the bookmark is created.

Smart Bookmarks continuously records the user’s actions, which at first glance also seems like a potential risk to privacy and security. But modern web browsers already record a rich history, storing URLs, caching web pages, and recording text entered into form fields, so this privacy risk already exists. Smart Bookmarks adds screenshots and commands to the set of recorded data, and provides a graphical history that makes it easy to see what the user recently typed into a form, even if they subsequently undid it before submitting the form. Smart Bookmarks also records screenshots of all pages, including pages marked not-for-caching by the web server. The Smart Bookmarks history is not persistent, however; it is discarded as soon as the browser window or tab is closed.

Side-Effects
Side-effects are another source of possible risks. An automatically generated bookmark may include commands with side-effects, which may change the state of a web site (e.g. deleting mail from a webmail account), affect other users of the network (e.g. sending mail), or even affect the real world (e.g. ordering a book and charging it to a credit card). For smart bookmarks used like macros, the side-effects may be intentional, but for bookmarks that are used simply to recall a hard-to-reach page, side-effects are undesirable.

One solution that has been proposed [14] requires web site authors to annotate actions that may have side-effects. For example, the final button of a purchasing process might have an attribute indicating that it triggers a monetary transfer. If widely adopted, this approach is likely to be the safest, since the web site knows best which actions have side-effects and which do not. The conventional use of GET and POST methods is an example of this kind of approach, in fact. Since ordinary browser bookmarks only use GET, most web sites use POST for actions with side-effects. Unfortunately, POST is also widely-used for forms with no side effects at all, so POST is generally necessary but not sufficient to identify a side-effect.

Since widespread adoption of side-effect annotations may take a long time, our prototype system tries to bridge the gap by detecting undesirable side-effects automatically. Since virtually all side-effects on the Web are triggered by a button click, the essential problem is to classify buttons into two classes: side-effecting, or non-side-effecting. (In fact, we have several classes of side-effects similar to those proposed by Safonov et al. [14], including monetary transfers, sending messages, and registering accounts.) When the system is about to replay a click command, it passes the command to the side-effect classifier. If the command is
likely to have an undesirable side-effect, the system pauses and asks the user whether to proceed (Figure 11).

To classify a button automatically, we use a combination of machine learning and collaborative feedback. For buttons that have never been classified before, we use a naïve Bayes classifier learned from examples of other buttons. The features used for classification are tokens in the button’s label, in the attributes of its HTML element, and in the attributes of its containing <form> element. Typical features used by the classifier include checkout, purchase, and buy.

The naïve Bayes classifier is supplemented by a memory-based learner that aggregates feedback from a community of Smart Bookmarks users. Users can classify a button as side-effecting or not-side-effecting, either using the warning (Figure 11) or by right-clicking on a button and labeling it by hand. Each label provided by a user is stored as a vote in a shared database. Once enough votes are accumulated for a particular web site, then buttons from that site are no longer classified by the naïve Bayes classifier, but instead by adding up votes from the database, weighted by their similarity to the unknown button. We are still experimenting with this approach, and have yet to evaluate it.

Robustness
An open question for web macro systems is how well they can survive changes in the web sites that they automate. Change comes in many forms. Some changes are minor, such as adding new content to a page without affecting navigation between pages. Other changes may be drastic, such as changing the order of pages in a form submission process, or requesting new information from the user.

The results reported in the Evaluation section suggest that Smart Bookmarks are robust over short intervals, on the order of weeks to a few months. Testing robustness over longer periods is future work.

Ordinary bookmarks can also be fragile, of course, since a web site that changes its domain name or its URL structure will break existing bookmarks, resulting in the dreaded Page Not Found error. Web site authors are generally sensitive to this, however, and have tools like URL redirection at their disposal to help make ordinary bookmarks more robust. Keyword signatures [12] can also be used as more robust locators for web pages that are indexed by search engines. Automatically correcting web macros, however, is still future work.

CONCLUSION, STATUS, AND FUTURE WORK
We have described Smart Bookmarks, a new approach for automatic, retroactive macro recording in a web browser. The system continuously records the user’s actions in a graphical history, so that when the user requests a bookmark, it can automatically find a short suffix of the history to use as the bookmark. Bookmarks are displayed with a rich visual representation, including text, screenshots, and animations, and the user can easily edit, parameterize, play, single-step, and share bookmarks. Since a bookmark is represented by a script of keyword commands, users can edit the commands without attention to syntax, and a bookmark sent to another user can be read and followed manually if the other user doesn’t have the Smart Bookmarks system installed.

The current prototype system runs in Firefox 2.0 using the Chickenfoot web automation system. It is available for download at http://uid.csail.mit.edu/bookmarker.

Future work includes tighter integration with Firefox’s own bookmarking system, so that smart bookmarks can be created and managed with the user’s ordinary bookmarks; more elaboration of the side-effect detection mechanism briefly outlined in this paper, and an evaluation of its effectiveness; and evaluation of the robustness of smart bookmarks over longer periods of time.

ACKNOWLEDGMENTS
We thank Greg Little, Jones Yu, members of the UID group, and the anonymous reviewers, 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, and by Quanta Computer under the T-Party project. Any opinions, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the sponsors.

REFERENCES