End-User Programming in the Wild: A Field Study of CoScripter Scripts

Christopher Bogart, *Oregon State University*
Margaret Burnett, *Oregon State University*
Allen Cypher, *IBM Almaden Research*
Christopher Scaffidi, *Carnegie Mellon University*
{bogart, burnett}@eecs.oregonstate.edu, acypher@us.ibm.com, cscaffid@cs.cmu.edu

Abstract

Though a new class of languages has emerged to enable end users to create their own web applications, little is known about how end-user programmers actually use such languages in the real world. In this paper, we report a field study on over 1400 scripts collected from the internet which were created by early adopters of CoScripter, a web macro programming-by-demonstration language. We contrast these internet scripts with those written by users inside IBM, and describe script usage and re-usage patterns, features used, and users' clever workarounds for features not present in the language. The results show how users grapple with such programming notions as repetition, generalization, and reuse, sometimes inventing their own devices for these. Finally, we discuss the many scripts we found with social implications, whose purposes were to circumvent intended rules, regulations, and usage norm assumptions of a number of web sites.

1. Introduction

What kinds of programs do end-user programmers write in the real world? Although there is significant literature on end-user programming in controlled conditions and some literature on real-world end-user programming based upon surveys and interviews (e.g., [9][10][11][12][13][16]), there is little information on real-world programs themselves, especially in the emerging paradigm of web scripting.

Web scripting (sometimes called creating "web macros") is a relatively new way of accomplishing repetitive common tasks in a web browser. For example, consider the task of reserving a shuttle to the airport—going to the shuttle service's web site, navigating to reservations for your city's service, typing your name, contact information, credit card information, and flight time, and clicking the submit button, then repeating the same process for the next trip. This task requires mostly the same typing and navigation for every trip. Worse, people sometimes may not remember all the information needed or how to navigate through a web site to accomplish the task.

Web macro tools address these problems by allowing people to record and replay actions, saving keystrokes and mouse-clicks. Macros remove the need to remember detailed information and tricky navigation sequences. Further, users can help other users with the same needs if macros are publicly available.

Delivering benefits like these are the goals of web scripting languages such as IBM's CoScripter [6]. This web macro recorder incorporates (1) sharing and reuse of macros via a wiki that is tightly integrated into the programming environment and (2) a simple variable substitution scheme to facilitate reuse by others (e.g., automatically substituting each user's own name or phone number where required in a script).

But what tasks do people really automate with scripts? Do they share and extend others' scripts? Very little is known about people's uses of such languages in the real world.

To fill this knowledge gap, we conducted a field study on early adopters of CoScripter, investigating 1445 CoScripter scripts collected from the internet at large and contrasting them with 665 scripts from IBM users. Our research questions were:

(1) *What kinds* of scripts do end-user programmers create? For example, are scripts for work or for play? Oriented toward the author's needs or for other users'? We focus on "what kinds" in Section 4.

(2) *How* were the scripts designed? For example, what kinds of constructs did their creators use? Did they use abstraction? Did they build upon others' scripts? We focus on "how" in Section 5.

(3) How does scripting potentially interact with *assumptions of the web society*? We focus on this issue in Section 6.

2. Background and Related Work

2.1 Background: CoScripter

CoScripter enables end-user programmers to demonstrate actions in the Firefox browser, then saves actions as a "script" on a wiki. Anyone who has installed the CoScripter browser plug-in can run the script to replay the actions. In addition, anyone can add comments to a script's wiki page and rate the script's usefulness. By default, all scripts are public and can be

used and modified by others, but a script's creator can mark it "private" so that it is not visible to others. Scripts are saved in an English-like syntax, with no additional hidden information about the actions (Figure 1). Users can edit these scripts in this syntax, which CoScripter directly parses and executes. For human readability, CoScripter refers to buttons, links, and other web page elements in terms of nearby text (a technique pioneered in Chickenfoot [1]).

It would be inconvenient to share scripts if they always used the creator's personal data (such as name and address), so CoScripter has a Personal Database where each user can supply personal values for variables. For example, the second action in Figure 1 uses a variable, which appears after the keyword "your". At runtime, CoScripter automatically substitutes the user's personal value. If the user's database lacked a personal value for this variable, CoScripter would pause at runtime for the user to enter a value before resuming execution.

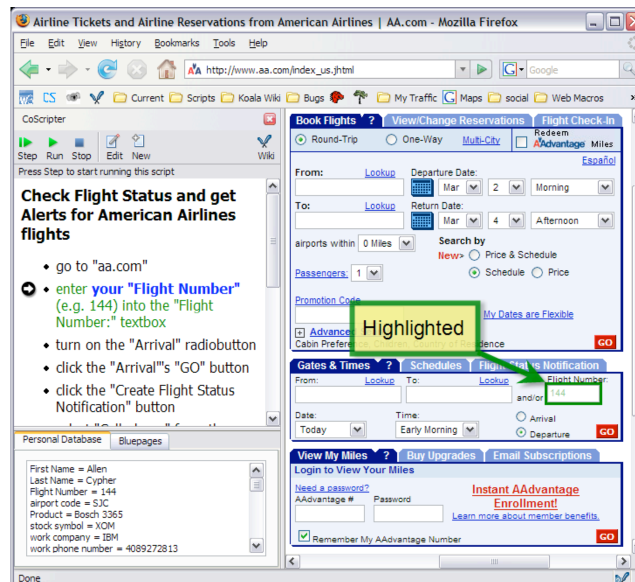

Figure 1. The currently executing step of the "Check Flight Status" script (left) causes CoScripter to highlight the corresponding textbox (right) and then paste the flight number from the user's Personal Database (lower left).

2.2 Related Work

Researchers have studied creation, sharing, and evolution of professional programmers' code (for a survey, see [4]). We aim to broaden this understanding to cover end-user programmers' scripts in the real world.

CoScripter is not the only web scripting tool, but it is the first to feature ready access to numerous publicly accessible end-user scripts. This accessibility is due to integrating a programming-by-demonstration (PBD) interface with a wiki. While other web scripting tools have a PBD interface as well as features not found in CoScripter (such as assertions [3], screen scraping features [2], and email integration [18]), they lack a public script repository. Conversely, Greasemonkey [8] and Chickenfoot [1] have repositories but lack a PBD interface, requiring programmers to write JavaScript. Thus, their repositories mostly contain scripts created by relatively well-trained (often professional) programmers.

There is some end-user programming research into end users' real-world practices, conducted primarily through interviews and surveys. For example, surveys identified web application features that should be possible to implement with web programming tools [11] and the practices of informal web developers [10]. Interviews of scientists revealed that they place little value in creating software, yet they do it anyway out of necessity [15]. Interviews of teacher end-user programmers showed that programming was facilitated when they could reuse code (either via copy-and-paste or by incremental changes to an existing program) and by the presence of many built-in language functions, but programming was inhibited when tools offered many features not relevant to a teacher's task [16]. Interviews of "domestic" end users highlighted two goals for programming household appliances: to make something happen in the future, and to facilitate repetition of a task [12]. A survey of end-user programmers found that abstractions in spreadsheets, web applications, and other programming domains fell into three clusters—PBD macros, imperative functions, and linked data structures—such that people with a propensity to create one abstraction had a propensity to create other abstractions in the same cluster (even across different programming domains) [13].

From both an abstraction and a power perspective, the web scripting context that we consider differs from the contexts of these prior studies. CoScripter supports only two abstractions in the clusters mentioned above: the scripts themselves are PBD macros, and the Personal Database is a minimalist data structure. CoScripter does not yet support conditionals, callable functions, loops, or structured data—all of which are features that have been identified as important for automating common tasks of browser end users [14]. Given these novel design decisions, many open questions arise, such as what useful tasks can still be automated, what abstractions those scripts use, whether and how scripts are successfully reused, and how scripts evolve over time, with or without multiple users' involvement.

The work closest to our own, a series of 26 interviews of CoScripter end-user programmers inside IBM

[5], addressed user motivations and experiences with CoScripter. Although their research used log data on 601 users to summarize usage, it did not analyze content or characteristics of the scripts. Our study builds upon prior findings in three ways. First, it investigates what was actually *in* the scripts that users chose to create. Second, it analyzes scripts created by people on the internet at large (not just IBM employees), thus giving a picture of script creation by a large and varied population of users. Third, it is the first large-scale field study on end-user web scripting, including over 2000 scripts harvested from the real world.

3. Methodology

Our investigation method was the case study, which is the right choice when asking "how" questions about a contemporary set of events over which the investigator has little or no control [17]. Our purpose was to reveal previously unknown details of real-world web scripts, as well as key phenomena that influenced the creation of scripts. Since our goal was to discover and report key phenomena, not to test hypotheses, it would be inappropriate to report inferential statistics, and we do not do so. Instead, we present quantitative summary (non-inferential) statistics and qualitative data.

We gathered 1445 public web scripts and their edit histories (3016 versions) from the public repository on the internet as of Dec. 18, 2007, and the same information for the 665 scripts on the internal IBM intranet site as of Jan. 7, 2008. (Users could also create private scripts that were not available for our analysis.)

We wrote tools to analyze scripts for attributes such as use of variables and comments. In addition, since some script attributes were difficult to detect automatically, such as the purpose of the script, we hand-coded the script attributes shown in Table 1 for 120 scripts.

Hand-coded script attributes
Data-intensive: Has at least one data item hard-coded in the script.
Bending the Rules: Does something that circumvents a website designer's intentions.
Self: Intranet URL, No URL, or hard-coded data. *Everyone*: Not *Self.*
Login Needed: Would an anonymous user have to register somewhere to get through this script? *Browser Fill-in Assumed*: Script logs in by button press without filling in user name. *Login Assumed*: Script assumes a logged in session. *URL Assumed*: Did not start with "go to <URL>" *Intranet Assumed*: Goes to a URL not accessible to most users.
Repetition: Contains the same code multiple times. *Set:* Performs the same task with different parameters each time.

Table 1: The subset of our codes pertinent to this paper.

	Internet	IBM
Script Authors	2510	301
% authors with public scripts	31%	38%
Scripts:		
Public	1445 (26%)	665 (37%)
Private	4028	1117
Total	5474	1782
Runs (Public)	13152	5247

Table 2: The internet repository was larger, newer, and had fewer scripts per author than the IBM repository.

Our hand-coding methodology was as follows. As described in Section 4, the scripts naturally divided into three groups in each repository. After excluding scripts written by authors of this paper and one prolific CoScripter administrator, we randomly chose 20 scripts from each internet group and 20 from each IBM group. One researcher then coded these 120 scripts. To evaluate the code set's robustness and the consistency of its application, a second researcher independently coded a subset (half internet, half IBM). Agreement was 70%-90% for each field, indicating that the code set was reasonably robust and reliably applied.

4. What Kinds of Scripts?

When we collected scripts, the internet site had been available for 6 months, whereas the IBM site had been available for 18 months. Even so, the internet site had more than twice as many scripts and eight times as many authors as the IBM site did (Table 2).

4.1 Internet Scripts and IBM Scripts

Since IBM users had earlier access and perhaps different motivations for using CoScripter, we suspected that their scripts might differ from internet users' scripts. Indeed, internet users who wrote scripts created fewer per person (just over 2/person) than IBM scripters did (about 6/person). In addition, internet users' scripts automated fewer work-focused tasks than those of IBM users.

In the internet repository, some of the most frequently executed scripts involved lotteries and games (Figure 2). Others dealt with consumer web sites like amazon.com; social networking sites like Facebook; classified advertising sites; banking and stock quote sites; bus, train, and airline scheduling and ticketing; sports and entertainment; libraries; job searching; weather and news sites; and generic search engines like Google.

In the IBM repository, scripts encompassed some of the same domains as the internet scripts, but work-related tasks dominated the scripts, many of which automated interactions with IBM's extensive intranet system. Many scripts automated VOIP telephony functions, such as call forwarding and checking messages. Others worked with collaboration tools like wikis and document sharing sites; corporate infrastructure (cafeteria menus, maintenance requests, employee administration); help desk; administrative support for management functions; technical education (accessing online courses); and conference registration.

Leshed et al.'s early study of IBM users conjectured that needs and use patterns would be different outside IBM [5], and our data confirm this conjecture. An implication of these differences for end-user programming researchers is that early data collection within the researchers' own institution may not be externally valid if the ultimate target audience is outside the institution.

Consequently, for the remainder of this paper, we will mainly focus on the internet repository and only mention the IBM repository when there are interesting examples or contrasts.

4.2 Popularity of Usage

In the internet repository, an 80/20 rule applied: 16% of the scripts (211) accounted for 80% of the script runs. Figure 3 plots the average number of runs of a script per user as a function of the number of different users of the script. The values hug the axes, enabling us to identify three groups of scripts for analysis purposes. We classify scripts as "ManyUsers" if they were run by more than three users. Note that these scripts tend to have few runs per user. Of the remaining scripts—which had three or fewer users—we classify as "ManyRuns" those scripts that averaged six or more runs per user. Note that most of these scripts had few users. We classify the remaining scripts as "FewUsers/FewRuns". In both repositories, 9-13% of scripts were ManyUsers, 7% were ManyRuns, and 80-84% were FewUsers/FewRuns (Table 3).

As discussed in Section 3, these three groups formed the structure for sampling the 120 scripts that we hand-coded. The remainder of this paper characterizes most findings in terms of these groups.

- click the "Lager" button
- enter "750000" into the "0,01 ¢" textbox
- enter "0,05" into the first "Einzelpreis" textbox
- click the " versenden " button
- click the "Die Kunden können von größeren Angeboten auch Teilmengen kaufen." button
- click the "Lager" button
- enter "750000" into the "0,01 ¢" textbox

…

Figure 2: The beginning of a repetition-heavy script for a German-language electric utility simulation game. The last five lines repeat 23 more times.

4.3 Me-Oriented or Everyone-Oriented?

We coded our random sample of 120 scripts in terms of potential audience: Self or Everyone. Figure 4 shows the results for the 60 in the internet group. Self scripts were those containing hard-coded data, unspecified URLs, or URLs not reachable by most repository users. Scripts not coded Self were coded Everyone. (Two of the scripts in this random sample happened to be empty files; we left them in the sample but coded them as "blank"). Figure 2 is an example of a Self script that contains hard-coded data.

CoScripter's formative work categorized the needs of surveyed users as "Sharing how-to knowledge" or "Automating frequent tasks" [5]. Although we do not know script authors' intents, Self scripts were at least consistent with the latter category. As Figure 4 shows, about half (27/60) of the scripts were oriented toward the author's own use, and the other half (31/60) may have been more convenient for others to use.

Not surprisingly, the scripts most widely used by people other than the original author were those without the Self-oriented attributes. Still, for scripts with the Self-oriented attributes, many of them stood the

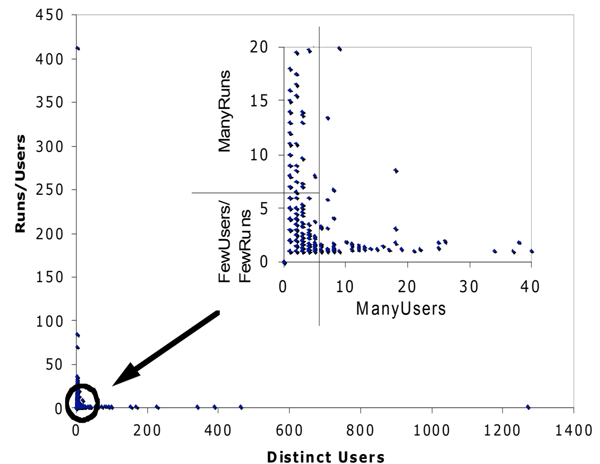

Figure 3: Most scripts hug the axes: run few times by many users, many times by few users, or few times at all.

	Many Users	FewUsers FewRuns	Many Runs	Total
Internet	9% (131)	84% (1208)	7% (106)	100% (1445)
IBM	13% (87)	80% (529)	7% (49)	100% (665)

Table 3: Counts of scripts in each group on each site.

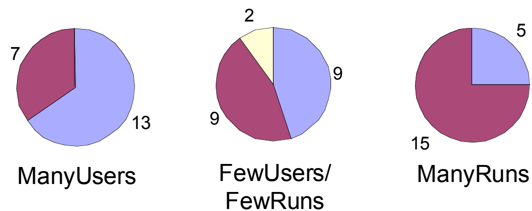

Figure 4: Coded scripts by potential audience, internet repository. Self: dark; Everyone: light. (Blank scripts in FewRuns: white.) Self scripts predominate in ManyRuns, but are less common in ManyUsers.

test of time and were run many times by the script's author (rightmost pie in Figure 4).

Note also that seven of the ManyUsers scripts in our sample were Self scripts. Designers of programming environments have sometimes expressed a vision to see end-user programmers reusing one another's code. These scripts suggest that the ability to easily make scripts available to others, even without explicitly generalizing them, can indeed lead to serendipitous reuse.

5. How the Scripts Were Programmed

5.1 How Users Did Repetition

CoScripter has no repetition constructs. Yet, users found ways to accomplish repetition. One way they did this was via copy-paste, duplicating code the desired number of times. Such sequences were common; about 17% of the 1445 internet scripts had at least one duplicate line, and in our coded sample of 120 scripts, 6 contained repetitive sequences. For example, one script earned a user points in a Facebook game by clicking a button hundreds of times to view a random profile. The script's version history shows that the user first tried to end the script with "repeat" and then "go to start" (both commands unknown to CoScripter), before settling on copy-paste. Figure 2 shows another example.

A different form of repetition was set-based—performing the same operation on different items in a set. For example, one game script shipped identical goods from five different outposts. To create such a script, a user could use copy-paste to perform the same actions five times, and then edit each copy to select a different outpost via the game site's drop-down widget.

Although the scripts described above might have been simpler if the language had "repeat" and "foreach" constructs, another set-based script that we observed would be harder to simplify. This script initially updated the user's Facebook status (e.g., by posting "working" or "watching tv" to the server). Later, other users added code to also update status on two other social networking sites. This is repetition ("foreach site, update status"), but the code to update each site differed considerably, since the different sites have different buttons to click on. In this situation, "simplifying" the code (rolling it into a loop) would require significant forms of abstraction, such as objects with different method implementations (e.g., "foreach ISocialSite s, s.update('watching tv') ").

Finally, one user figured out a way to do recursion, and wrote about in the CoScripter online forum:

> I find a workaround how to force it to automatically start over. Just direct it to your script id, for example
> go to "http://services.alphaworks.ibm.com/ coscripter/browse/script/YOUR_SCRIPT_ID"
> Then click the run link on the website and it will start everything from the scratch.

Although our study period did not include any scripts using this technique, three scripts later appeared, ended with "go to" followed by a specially formatted URL that CoScripter interprets to immediately load and run a script. The scripters may have stumbled on this possibility by hovering over the Run button on their script's wiki page, and trying out the unusual URL that is displayed in the browser's status bar. In all three cases, the construct was used to repeatedly click on buttons in games. Since there are no conditionals in CoScripter, these users would presumably have to terminate execution by hand, such as by clicking Stop, or closing the CoScripter window.

Other researchers have noted that web macros for many tasks would require iteration [14]. The prevalence of repetition in our data offers further evidence of the need for repetition constructs in web macro languages. It also shows evidence of the power of simplicity that allowed end users to find ways to do repetition even without such constructs.

5.2 How Users Did Reuse

CoScripter supports variables. While recording a script, whenever the user types a value that matches data in the Personal Database, CoScripter automatically replaces that value in the script with a variable. The Personal Database is the way variables vary from user to user. Within IBM, the Personal Database is automatically expanded to include the user's "BluePages" information, an internal corporate phone book. Perhaps that helps explain why variables for names, phone numbers, email addresses, office locations and the like abounded in the IBM scripts. But such variables were also fairly common in the internet repository, where each user's Personal Database had to be populated by hand. Of course, a user can add variables that are not really "personal" attributes, and some scripts relied on that. Figure 5 shows such a script. Overall, 20% of scripts referenced the Personal Database, and this greatly promoted reuse: 40% of these scripts were executed by multiple people.

Not everyone used variables for their data. In many cases, a user initially created a script with a hard-coded value and then went back and generalized the script to reference the Personal Database. But sometimes when users encountered a script with a hard-coded value different from the value they needed, they chose to simply edit the hard-coded value. Figure 6 shows that this type of edit was fairly common in the ManyRuns category; overall, it accounted for about 9% of all edits. Interestingly, in the ManyUsers scripts, more than half of these changes were made by users other than the author, showing that they were able to reuse the script despite the hard-coded values.

We saw a preference for editing hard-coded values especially often with the parameters of real estate searches: price range, number of bedrooms, zip code, etc. The program text in these cases is probably as easy to change as the Personal Database, and no variable names need to be invented. The values have clear semantics because of the direct juxtaposition to their use. Figure 7 shows an example; the script would hardly be clearer by introducing variable references.

Another occasion for hard-coding values was when a single user wanted to run the same script with different hard-to-remember values at different times. To handle this, some users created multiple copies of a script and then edited different hard-coded values into each copy. For example, IBM user *U3* (we have anonymized user names in this paper) created a set of scripts, one for each type of printer toner cartridge to be purchased. The scripts differed only in the part numbers and prices entered into the form.

One of the authors (Cypher) handled a similar case personally by having multiple variables with the same name in his Personal Database, and shifting their order before running a script, knowing that the first value encountered would be used. IBM also experimented with the addition of a special feature for importing personal data. It was used by managers of summer interns to run scripts that filled in administrative forms with data about an intern.

In a wiki context, where many users share scripts, edits can cause problems when one user's edits do not suit the needs of other users. We know from Leshed's interview study that some CoScripter users did not even realize that their edits would replace the original script for all users [5]. Our data revealed that site administrators repeatedly had to roll back edits to a certain tutorial script, which searched for "koala" on Google Images. Users' edits included pointing the script to other search engines (such as internationalized versions of Google) and changing the search term to other words such as "bikini".

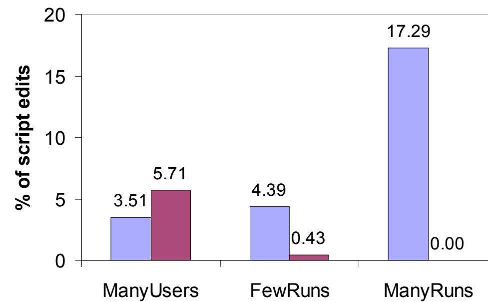

Figure 6: Percent of all script edits that were value edits by the script's author (left bar) or by others (right bar).

5.3 Context: Implicit Preconditions in Scripts

Scripts often reflected assumptions about the browser's state prior to script execution. Some common preconditions we encountered were: the browser being already at a certain URL; the user having access to some non-public URL; the user being already registered to use a site; a cookie being set to indicate that the user had already logged into a site; or the browser having been configured to pre-fill form login and password fields. These assumptions were usually implicit, though a few users did express assumptions in comments inside scripts.

For example, to execute the script in Figure 2, the browser has to be at the right URL before execution, the user must be registered with the site, and the user must have a game in progress.

It was common for a script's first version to include login actions, followed in a few minutes by a revision of the script which assumes that the user is logged in. Apparently, users notice that the script's login actions stop working the very first time they test it, so they de-

```
Make sure you have a "PubMedKey =
   my_pet_biology_subject" entry in your "Personal Da-
   tabase" (bottom left)
 • go to "http://www.ncbi.nlm.nih.gov/sites/entrez?
   db=PubMed&itool=toolbar"
 • enter your "PubMedKey" in the "for" textbox
 • click the "Go" button
```

Figure 5: A script comment (unbulleted line) instructs the user to add a Personal Database variable, which the script then uses in the second command.

```
• go to "http://www.rentometer.com/"
• enter "homestead road" into the "Rental Address"
  textbox
• enter "95014" into the "City & State, or Zip" textbox
• enter "1500" into the "Current Monthly Rent ($)"
  textbox
• select "2" from the "Bedrooms"'s "Bedrooms" listbox
• select "50+" from the "Units in Building" listbox
• click the "Units in Building" button
```

Figure 7: A script with hard-coded values.

lete the script's login actions. The problem with this fix is that the script fails the next day, after the session has expired.

Unlike traditional programs that "start from scratch", CoScripter scripts with preconditions can be characterized as *meta-programs* that manipulate other running programs. This is a powerful capability, but it requires the user to be aware of the exact set-up needed for the script to run properly. If the user's memory or understanding of the preconditions is imperfect, then the script may execute in unanticipated ways. Guarding against failure may call for a mechanism to make preconditions explicit, perhaps by adapting existing research on supporting assertions in web macros [3] to cover the kinds of preconditions that we observed.

5.4 Mixed-Initiative Execution

CoScripter has an affordance that is unusual in end-user programming: mixed-initiative programming. Instructions with the word "you" in them are not parsed further; instead, control is handed to the user, who can perform any desired actions before continuing by clicking the "Run" button.

We saw the "you" keyword serving four different functions: conditional execution, pausing for timing reasons, prompting for data to be provided, and signaling an explicit need for human intelligence.

Conditional execution is needed when a script must run under varying conditions, such as sometimes being logged in and sometimes not. For example, user *U4* inside IBM included the action "you may have to sign in with your intranet id and password and click Submit". This causes CoScripter to pause, so the user can take action and then click the "Run" button to resume.

Timing reasons caused some users to pause scripts. For example, *U6* used "you" lines to stop after each slide in an online presentation. As another example, we saw multiple cases where scriptwriters tried to handle the fact that CoScripter does not always wait until a page is done loading. They tried lines such as "wait 10 seconds" (not recognizable by CoScripter). User *U5*, needing a pause, tried "javascript.sleep(1000)", which CoScripter did not understand, and after some experimentation, ended up with simply "you wait".

Some scriptwriters may have wished for a way to prompt users for input, and used "you" to fill the gap. "You" could be used to let the user fill in a web form directly, when the scripter wanted to avoid hard-coding values or depending on the presence of Personal Database entries.

Regarding explicit need for human intelligence, an internal IBM script avoided ethical problems by inserting "you" before clicking to accept a legal agreement: "You click the first "This update form is electronically signed when you press" button". Similarly, a script to pay traffic fines in London allayed users' potential lack of trust in the script with this final line: "you click the "Pay Now" button (To allow a review)".

The "you" feature eases the learning curve for the end-user programmer, giving the script author a way to write useful scripts even when some portions seem too difficult to write. Mixed-initiative execution also enables incremental development and use of a script before the task is fully automated. Yet the feature was not always used when it would have offered a clean solution, despite the fact that it is prominently featured in the CoScripter site. Perhaps this was due to the feature's novelty to many users, or due to a preconception that programs ought to always run to completion.

6. Changing the Rules

Web sites are designed around a variety of assumptions about how the site will be used. In many cases, these assumptions reflect an implicit social contract or other general rules about the site. For instance, sites that rely on advertising revenue assume that visitors will see and click on ads. Programs such as the Firefox "Adblock Plus" and "Platypus" extensions invalidate this assumption by making it easy for users to remove advertisements. Similarly, the web-scraping software that powers many mashups (e.g., systems from Dapper, Lixto and Kapow) automates the process of clipping data from sites, without having a person ever look at the pages that provide that data.

CoScripter macros can invalidate the assumption that users will manually click on the buttons and links on a page. In our sample of 60 public repository scripts, 18% of them were designed to circumvent this assumption or others underlying web sites.

For example, user *U9* created a script called "Automated Click for Charity", which goes to several sites that donate small amounts of money to different charities whenever pages are visited, as a reward for viewing the advertisements. User *U10* created scripts for playing lottery sites that work on a similar model to the charity sites, but instead of donating to charity, a portion of the advertising revenue goes into a pot that site visitors can win. An even more egregious script logs into a website many times under different usernames to vote for user *U11* in a "Bachelor Search" contest (with a significant monetary prize). At present, this user is winning the contest by a large margin.

As a final example of changing the rules, one IBM script changes a password four times, thereby circumventing an IBM rule that disallows the reuse of any of an employee's last five passwords. Heretofore, changing a password four times has been sufficiently onerous that it is not worth the effort to circumvent this

rule. But CoScripter changes this underlying assumed safeguard because it changes the cost/benefit ratio.

There is another factor at work, too. Unlike previous web scripting tools, CoScripter provides a repository for sharing scripts. In the past, when sophisticated hackers produced "warez" (configurable code for hacking web sites or launching denial-of-service attacks), less sophisticated "script kiddies" who used warez needed at least some minimal programming skills [7]. By unifying an end-user scripting framework with a shared repository, systems like CoScripter may force a change in the assumptions underlying web site design.

7. Conclusions

Our field study of end-user programmers' web macros has revealed what kinds of web scripts exist in the real world and how these programs were designed. We unearthed a variety of phenomena ranging from the staid to the inventive to the mischievous, yielding the following conclusions:

Even if a programming language lacks basic constructs like conditionals and callable functions, it still can be useful. CoScripter does not yet support all requirements needed for every common browser automation task [14], but it provides enough value that many users keep creating and executing scripts. There is a role in the world for non-Turing-complete languages.

End-user programmers can effectively share programs anonymously. Prior research found that end-user programmers often share programs within specific organizational settings [15][16]. Our study generalizes this finding, as the internet CoScripter site's users had no organizational relationships with one another, yet they still had enough needs in common that they could make use of one another's scripts.

The balance of power on the web continues to shift toward site users, and away from site designers. For years, only relatively sophisticated programmers have had the ability to "mashup" information from web sites, reusing data for purposes that are not sponsored by site designers. Our study shows that CoScripter enables even end-user programmers to undermine the assumptions that undergird the web as we know it.

This is an exciting time for end-user programming research. The conclusions above hint at many outstanding research problems—such as how to help macro authors benefit from the web without creating disincentives for site designers to keep creating new site content—and they highlight an unparalleled opportunity to directly affect millions of lives with research.

Acknowledgements

We thank Sam Adams and Rachel Bellamy for helpful discussions about this research. This work was supported by the EUSES Consortium via NSF ITR-0325273, by NSF grants CCF-0438929 and CCF-0613823, and by an IBM International Faculty Award.

References

[1] M. Bolin, P. Rha and R. Miller. Automation and Customization of Rendered Web Pages. *Proc. 18th Annual Symp. on User Interface Software and Technology*, 2005, 163-172.
[2] iOpus corporate website, www.iopus.com
[3] A. Koesnandar. *Building Dependable Web Macros Using Robofox*, Master's Thesis, Computer Science and Engineering Dept., Univ. Nebraska - Lincoln, 2007.
[4] C. Krueger. Software Reuse. *ACM Computing Surveys* (CSUR), Vol. 24, No. 2, 1992, 131-183.
[5] G. Leshed, et al. CoScripter: Automating and Sharing How-To Knowledge in the Enterprise, *Conf. Human Factors in Computing Systems*, 2008, to appear.
[6] G. Little, et al. Koala: Capture, Share, Automate, Personalize Business Processes on the Web, *Conf. Human Factors in Computing Systems*, 2007, 943-946.
[7] J. McDermott and C. Fox. Using Abuse Case Models for Security Requirements Analysis. 15th Annual Computer Security Applications Conf., 1999, 55-64.
[8] N. McFarlane. Fixing Web Sites with Greasemonkey. *Linux Journal*, Vol. 2005, No. 138, 2005.
[9] B. Nardi. *A Small Matter of Programming: Perspectives on End-User Computing*, MIT Press, 1993.
[10] M.B. Rosson, J. Ballin and J. Rode. Who, What and How? A Survey of Informal and Professional Web Developers, *Symp. Visual Lang. and Human-Centric Computing*, 2005, 199-206.
[11] J. Rode and M.B. Rosson, Programming at Runtime: Requirements and Paradigms for Nonprogrammer Web Application Development, *Symp. Human-Centric Computing Languages and Environments*, 2003, 23-30.
[12] J. Rode, E. Toye and A. Blackwell. The Fuzzy Felt Ethnography - Understanding the Programming Patterns of Domestic Appliances. *J. Personal and Ubiquitous Computing*, Vol. 8, No. 2, 2004, 161-176.
[13] C. Scaffidi, et al. Dimensions Characterizing Programming Feature Usage by Information Workers. *Symp. Visual Lang. and Human-Centric Computing*, 2006, 59-62.
[14] C. Scaffidi, et al. Scenario-Based Requirements for Web Macro Tools. *Symp. Visual Lang. and Human-Centric Computing*, 2007, 197-204.
[15] J. Segal. Some Problems of Professional End User Developers, *Symp. Visual Lang. and Human-Centric Computing*, 2007, 111-118.
[16] S. Wiedenbeck, Facilitators and Inhibitors of End-User Development by Teachers in a School Environment, *Symp. Visual Lang. and Human-Centric Computing*, 2005, 215-222.
[17] R. Yin, *Case Study Research: Design and Methods*, Sage Publications, 2003.
[18] J. Zimmerman, et al. VIO: A Mixed-Initiative Approach to Learning and Automating Procedural Update Tasks, *Conf. Human Factors in Computing Systems*, 2007, 1445-1454.