A Conversational Interface to Web Automation

Tessa Lau[†], *Julian Cerruti*[⋆], *Guillermo Manzato*[⋆], *Mateo Bengualid*[⋆],
Jeffrey P. Bigham[⋆⋆], *Jeffrey Nichols*[†]

[†]IBM Research – Almaden
650 Harry Road
San Jose, CA 95120 USA

[⋆]IBM Argentina
Ing. Butty 275 - C1001AFA
Buenos Aires, Argentina
tessalau@us.ibm.com

[⋆⋆]University of Rochester
Dept of Computer Science
Rochester, NY, 14627 USA

ABSTRACT

This paper presents CoCo, a system that automates web tasks on a user's behalf through an interactive conversational interface. Given a short command such as "get road conditions for highway 88," CoCo synthesizes a plan to accomplish the task, executes it on the web, extracts an informative response, and returns the result to the user as a snippet of text. A novel aspect of our approach is that we leverage a repository of previously recorded web scripts and the user's personal web browsing history to determine how to complete each requested task. This paper describes the design and implementation of our system, along with the results of a brief user study that evaluates how likely users are to understand what CoCo does for them.

Keywords: automation, natural language interfaces, intelligent assistants

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. - Natural language.

General terms: Algorithms, Design, Human Factors

INTRODUCTION

For decades, science fiction writers and technology pundits have described a future in which computers respond to high-level natural language commands, carrying out relatively complex tasks with limited interaction, allowing humans to focus on other more interesting work. These intelligent computerized assistants would learn as we interacted with them, becoming more accurate and useful over time. Unfortunately, building an intelligent agent with these capabilities for the general case requires advances in natural language understanding, dialog management, planning, sensing and actuating, user modeling, and machine learning – all of which continue to be challenging problems individually.

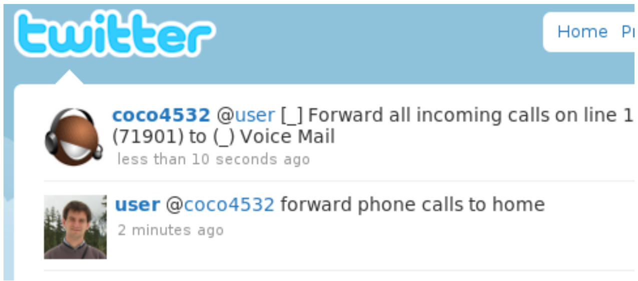
If we limit the scope of the problem however, perhaps this vision of an intelligent assistant can become a reality. In this paper, we explore the idea of creating an intelligent agent with two key limitations. First, we limit the domain of tasks to those that can be carried out entirely on the web. As opposed to domains involving physical space, the web is entirely machine readable and manipulable, which substantially reduces the problem of sensing and actuating. Moreover, we focus on bounded tasks with known outcomes, rather than open-ended sense-making tasks that require human judgement to complete.

In this paper, we present CoCo, an intelligent assistant that can perform routine web tasks on a user's behalf and report back with the results. CoCo (short for CoScripter Concierge) listens for commands on low-bandwidth textual channels such as Twitter or SMS (Figure 1). It synthesizes a plan to complete a task using a sequence of actions on the web, engages the user in conversation to confirm its plan and solicit necessary task parameters, executes its plan in a server-side web browser, and informs the user of the result. Plans are synthesized by mining two repositories of web activity: a public collection of web automation scripts [6], and logs of users' web browsing history [7].

We believe that one enabling factor in our design is our choice of input channels. First, these channels are primarily textual, which avoids the challenges of speech recognition that have stymied systems in the past. Second, these channels restrict the length of users' messages to a relatively small number of characters (˜140). This restriction limits the complexity of the task that users can specify and also constrains them to be precise, generally describing their task using just a few meaningful terms.

In summary, this paper makes the following contributions:

- a novel architecture for building a conversational intelli-



coco4532 @user [_] Forward all incoming calls on line 1 (71901) to (_) Voice Mail
less than 10 seconds ago

user @coco4532 forward phone calls to home
2 minutes ago

Figure 1: Example CoCo interaction, over Twitter

UIST'10, October 3–6, 2010, New York, New York, USA.

gent assistant that builds on existing web automation components;

- an auto-clipping algorithm to automatically extract relevant regions from web pages;
- an implemented system that validates our architecture;
- and a *user study* that addresses concerns about trust and whether users will understand the results of automation.

The remainder of this paper is structured as follows. We begin by describing several usage scenarios that exercise some of CoCo's interaction capabilities. Next, we describe CoCo's architecture, and the concrete implementation we have developed. We then present the results of a user study designed to evaluate the level of understanding users have of textual automation interfaces such as CoCo's. Next, we discuss our practical experience with the system, along with some of the security and privacy considerations of our system, and ideas for future work. Finally, we conclude with related work.

USAGE SCENARIOS

In this section, we describe several illustrative scenarios showing how CoCo could be used in practice.

Automating scripted interaction

Bob's workplace uses a VOIP-based phone system that allows him to forward his work calls to his home phone when he is out of the office. This forwarding is controlled via a web application. Bob performs this task so frequently that he has created a CoScripter [6] script to automate the task, and routinely runs this script before leaving the office.

However, while boarding the bus to head home from work, Bob remembers that he is expecting a call from an important client this evening, but he has forgotten to forward his phone. Using Twitter, Bob sends a directed message to CoCo:

@bob → @coco: forward phone calls to home

CoCo searches through Bob's recently-used CoScripter scripts, using his command as a query, and finds the phone forwarding script. CoCo runs Bob's script and tells him that the script completed successfully (Figure 1).

Mining past web activity

Contractor Alice is required to punch in and out of work using an online timecard application each day. Alice works a regular 8-hour day so her actions for doing this are very repetitive. These actions are recorded in her web history using CoScripter Reusable History.

One day Alice leaves work without remembering to punch out. Using her cellphone, she sends a direct message to CoCo via Twitter:

@alice → @coco: punch out 17 30

Alice does not use CoScripter, so CoCo does not have any pre-recorded scripts for her. Instead, CoCo searches through her CRH logs and identifies a sequence of steps on the timecard website that contain the words "punch" and "17 30". CoCo sends the text of these steps to Alice, asking whether this is what she wants it to do. Alice reads the steps, verifies that they look correct, and responds with yes. CoCo automatically runs these timecard steps on her behalf and tells Alice that it was successful.

Parameterizing interaction

Carol is driving to the mountains to spend a week skiing with her family. However, a snowstorm has resulted in road closures and tire chain restrictions on some roads. Carol pulls out her phone to get the most up-to-date information from CoCo. Using the CoCo Android app, Carol can speak her query and see the results displayed on her phone.

@carol → @coco: get highway conditions

CoCo finds a script for getting highway conditions, but the script uses a parameter to specify which highway. CoCo returns a message to Carol explaining that it cannot complete the script without that parameter:

@coco → @carol: I don't know which "highway" to use

Carol then speaks her command again, supplying the missing parameter:

@carol → @coco: get highway conditions for highway 88

Because Carol's command uses the correct syntax to specify parameters, CoCo can extract the parameter value (highway=88) from her command, and supply this value as a parameter to the script. CoCo then executes the parameterized script and displays the highway conditions on Carol's phone.

Learning from interaction

On her way back from the mountains, Carol again wants to check the road conditions before leaving. Since she has worked with CoCo before on this task, CoCo remembers their previous interaction and streamlines the next interaction. Carol gives CoCo the following command:

@carol → @coco: highway conditions

CoCo remembers that she had previously asked for highway conditions, and that she was interested in highway 88. CoCo retrieves this information and displays it on her phone.

Lightweight interactions over email

Developer Dion uses an enterprise-class bug tracking application that requires filling out a web form with multiple fields to report each defect. Most of these fields have poorly chosen defaults, requiring Dion to override them each time he submits a new bug. He has mostly automated this task using a CoScripter script, except for entering the description of the defect. However, context-switching to the bug tracker interrupts his flow and causes him to lose concentration.

Instead, Dion fires off a quick email to CoCo to report a bug, with the command in the subject line: "log a bug against myproduct, with Crash On Reply as the bug". CoCo uses his CoScripter script to log a new bug report, using the description "Crash On Reply", and sends Dion a confirmation email with the id of the newly logged defect in the bug tracking system.

Because CoCo is accessible via email, Dion also discovers that he can use CoCo to log bugs even when he is onsite with a customer, debugging problems in their installation, with no access to his company's intranet (and therefore no access to the web-based bug tracker).

SYSTEM ARCHITECTURE

The architecture of our system is shown in Figure 2. At a high level, the system works as follows:

- Commands enter the system via a number of *transports* (input channels that permit the user to address CoCo with a short textual command).
- These commands are first interpreted by a *parser*, which extracts and removes any parameters, leaving the "core command."
- Next, a *planning* component uses the core command to identify sequences of web actions that can satisfy the core command.
- Scripts are *parameterized* to plug in missing variable values.
- These scripts are routed to a browser automation server, which runs each script in a headless web browser.
- Next, a clipping component automatically clips relevant portions of the web pages visited during script execution in order to construct a meaningful response for the user.
- Finally, this response is returned to the user either via the originating transport, or via an alternate transport if requested.

The following sections describe each of the steps in more detail, starting with the simplest scenario and incrementally adding more complexity.

TURNING COMMANDS INTO SCRIPTS

CoCo's primary functionality is to enable a user to direct CoCo to perform a task on the web. While this could have been done by presenting a list of task options in a menu, and letting the user select one, we decided to explore a natural language interface for directing CoCo. While menus work well when the number of choices is small, we envision CoCo being used as an interface to a large number of tasks. Considering that CoCo also provides the ability to locate and perform previously-completed tasks from a user's web browsing history, the number of potential tasks CoCo can perform grows quickly. Therefore we focused on a text-based, search-style interface.

Given a command, such as "forward phone calls to home", CoCo tries to map from this command to a sequence of actions that accomplish this command on the web — a planning problem. Our approach to planning relies on the use of plan libraries, in the form of user-centric repositories of scripts that describe how to accomplish tasks on the web. In CoCo, we leverage two sources of scripts: 1) the set of user-created scripts in the CoScripter [8, 6] repository, and 2) the log of a user's web browsing history stored by CoScripter Reusable History (formerly known as ActionShot [7]).

To find CoScripter scripts, CoCo does a search of the CoScripter repository using the command as a query. Assuming an authenticated user, we search the scripts that user has created, as well as the set of scripts that she has executed recently. Searching recent scripts allows CoCo to broaden the search to include scripts that were created by other users but that may be relevant to this command. Search is implemented using a vector-space model (treating script titles and script text as bags of words) and TF/IDF is used to rank script relevance relative to the query. The top N scripts are returned for the query.

In the simplest case, CoCo then picks the top-ranked script, automatically executes it on the web using Highlight [10], and returns "script successful" to the user if the script was successfully executed.

MINING SCRIPTS FROM WEB HISTORY

CoCo can also mine scripts from web history logs such as those captured by CoScripter Reusable History [7] (CRH). Searching web history logs is more challenging because these logs capture an undifferentiated stream of user actions that are not segmented by task or web site. While CRH provides a search feature, it only returns individual steps in response to a query (e.g., "click the Search button" or "go to mylibrary.com"). However, most CoCo tasks span multiple steps and a query will likely reference multiple steps in sequence. For example, the task of searching for a book at the library might involve first clicking on a "library catalog" link, followed by selecting "books" from a drop-down menu, then clicking a "search" button. No single step contains all the words in the query; instead, we would like to retrieve a group of related steps in response to the query.

Our initial algorithm performs simple segmentation of the history stream in order to group steps into units that could be used as plans in CoCo. While preliminary, this algorithm is enough to enable users to experiment with CoCo by trying different commands, much as users try multiple web search queries today. We leave it to future work to design better history search mechanisms in order to pull more useful plans out of users' web histories.

Our algorithm starts by segmenting CRH's single stream of steps into groups that are each likely associated with a task. Logically, a segment is defined as a group of steps $S_1, S_2, ...S_n$ such that $t(S_{i+1}) - t(S_i) < \delta$, where $t(S_i)$ denotes the timestamp of step S_i and δ is an arbitrary threshold. In our experiments, we found that 5 minutes was a reasonable tradeoff between making segments too inclusive, and splitting segments into too many pieces.

In practice however, users often move from one task to the next without waiting several minutes between tasks. This task switching is typically accompanied by going to a different website in order to start the new task. Therefore, in addition to segmenting based on time, we also segment based on location changes. If the user enters a new URL into the location bar, or clicks a bookmark, or otherwise triggers going to a different location, then we insert a segment boundary right before that step.

Once a user's CRH logs have been segmented, we use the same vector-space model and TF/IDF ranking scheme to rank these segments relative to the user's original query. The resulting hits are merged with the results of the CoScripter script search to form a single ranked list of possible script results.

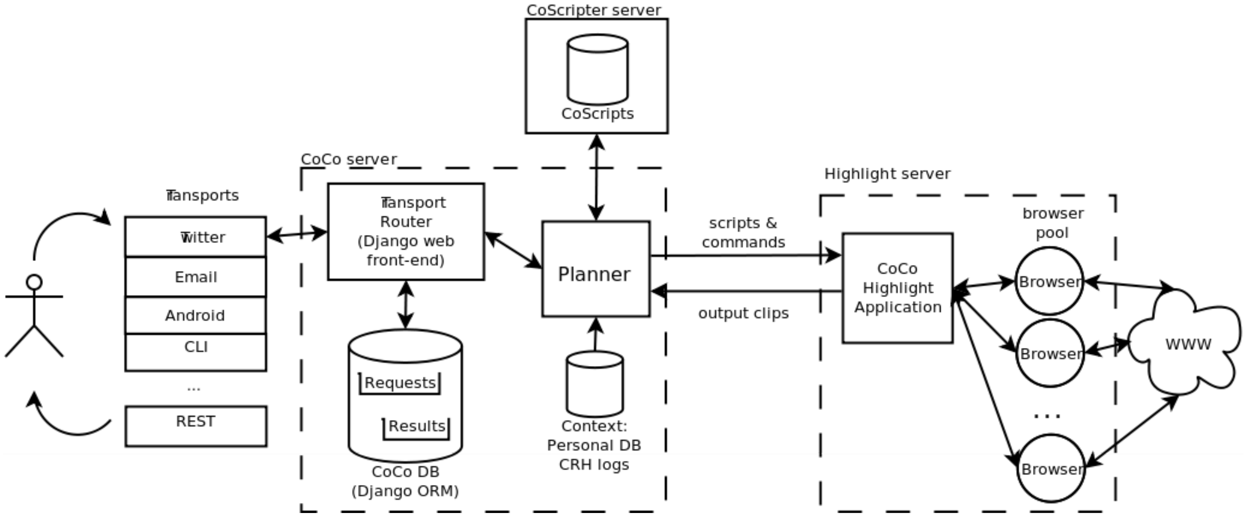


Figure 2: CoCo archicture diagram

CoCo then returns the top script from the ranked list as the result of the planning algorithm. Since our algorithm computes a similarity score for each result in the ranked list, we imagine someday using smarter algorithms to pick which script to run. For example, if the top script has a similarity much greater than the second-ranked script, then the top script is probably a good choice. However if the results included many scripts with similar similarity scores, then perhaps CoCo ought to ask the user which script to use, or examine the scripts to see if they are similar, and if so, combine them into a single "best practice" script.

PARSING COMPLEX COMMANDS

Many tasks require custom data values to be provided at runtime (for example, retrieving highway conditions for a specific highway, or the name of a product to search for, or the title of a bug to be reported). To enable users to supply parameters in their command, CoCo uses a *parser* to extract certain bits of information from a command before using it to search for a script.

We considered several approaches to parsing. At one extreme, we could require the user to obey a strict syntax that exactly specified what CoCo should do, using which parameters, and so on. The downside to this approach is that users are required to memorize and use correct syntax, which may be too difficult for casual or novice users. At the other extreme, we could treat a user's input as completely unstructured text, with no syntax whatsoever. While this would sidestep the pitfalls of enforced syntax, it is also less expressive: users could not explicitly call out certain words as parameter values. Alternatively, we could apply natural language processing to the user's input to semantically interpret their words. We leave it to future work to investigate whether NLP can be used to improve parsing.

In this paper, we have taken a hybrid approach between strict syntax and unstructured text: scanning for specific keyword-delimited statements, and treating the remainder as unstructured text. We believe this approach preserves some of the ease-of-use of the unstructured approach, while providing the ability for users to express certain types of structured information in a natural way. In our approach, CoCo assumes that a command may contain up to three types of information:

- *what* to do
- which *parameters* to use
- where to send the *output*

The specification of what to do is probably the most complex part of the command, and generally consists of freeform text with little structure. We call this piece the "core command." It is unlikely that the format of a core command will be similar for different requests. However, parameter and output specification are simple properties that could be expressed using a straightforward syntax. For example, these commands show how parameters might be naturally specified using keywords such as `for` and `using`, while output modalities might use the word `via`:

- check the library for book "neuromancer"
- get highway conditions for highway 88 via email
- log a bug, using Parser crash on line 45 as the title

Using our hybrid approach, CoCo parses these commands as follows:

- core command(check the library), parameters (book = neuromancer)
- core command(get highway conditions), parameters (highway = 88), output(email)
- core command(log a bug), parameters (title = Parser crash on line 45)

The benefit of the hybrid approach is that if the user fails to specify parameters using the correct syntax, then their entire utterance will be treated as a core command, which allows the system to fail gracefully rather than report a syntax error.

However, even with our hybrid approach, users may not always clearly differentiate between the name and value of a parameter. To reduce the burden on the user of specifying which words are the parameter name and which are the value, we developed an algorithm for non-deterministic parameter recognition which tries all possible combinations of parameter names and values. For example, if the command is “get the phone number for full name marc jones”, then CoCo generates the following potential name/value pairs:

- full = name marc jones
- full name = marc jones
- full name marc = jones

Even though only one of these interpretations is correct, all are passed to the next stage of processing. In practice, this works because only the correctly named parameter will be required for script execution. Incorrect parameters will be discarded because their names are nonsensical and not referenced in a script.

PARAMETERIZING SCRIPTS

The original CoScripter system provided a way to parameterize scripts with variables, such as the following:

- enter your “highway” into the “Road conditions:” textbox

At runtime, CoScripter would retrieve a variable named “highway” from this user’s ”CoScripter personal database”: a short list of name/value pairs that users can create to customize script execution at runtime.

CoCo uses a similar mechanism to allow users to customize the execution of their scripts at runtime. When a script retrieved from the CoScripter repository contains variable references[1], CoCo uses the following three sources to supply the paremeter value, in order:

- parameters specified in the command
- recently-used parameters from a previous CoCo interaction
- parameters from the user’s personal database

Parameters provided in the command override all other sources of parameters. If the value is not supplied in the command, CoCo retrieves previous values used in prior interactions. For example, if a user wanted to retrieve the same highway information a second time, CoCo remembers the “highway” parameter from her last interaction.

A user can optionally grant CoCo access to the personal database she created in CoScripter. If the parameter cannot be found in either the command or in recent history, CoCo can retrieve the value from her personal database.

If the variable cannot be found in any of these sources, CoCo returns a message to the user asking them to supply the missing value. A user may then repeat her command, including the missing information, as described in Alice’s highway conditions scenario above.

CONVERSING WITH THE USER

Since CoCo is capable of performing arbitrary actions on the web, users may be concerned about automation going awry and taking actions that they did not intend. To address this problem, CoCo conducts a dialogue with the user to build trust in its actions and ensure that it is doing what the user intended.

The first time a user gives CoCo a command to run a script that CoCo has not executed before for that user, CoCo will explain what it is about to do, and ask the user to confirm that the script it found accomplishes her desired task. For example:

@user → @coco: forward my phone
@coco → @user: Run script Forward phone calls to home? go to callmgr.corp.com, click the “Forward all calls” link, ...
@user → @coco: yes

CoCo would then respond with the result of the script.

The next time this user asks CoCo to forward her phone, CoCo remembers that it has previously executed this script for the user, so the confirmation step can be skipped. This memory is based on the steps of the script to be executed, not on the command the user supplied. For example, if the user next asked CoCo to “update phone forwarding”, and it retrieved the same script using different query words, CoCo would still remember that this is a script the user had previously approved, and run it without confirmation.

Through conversation, CoCo can learn from interacting with the user, and remember what it has done for the user in the past, while enhancing user trust in the system. This allows tasks with CoCo to be done quickly without the overhead of constant confirmation, while new tasks (which may have unexpected side effects) first go through this confirmation step.

GENERATING A RESPONSE

Many tasks involve fetching information from the web, which means that CoCo ought to return this information to the user in its responses. Therefore, we have provided two mechanisms by which more information can be returned by CoCo beyond the success/failure result code: explicit clipping and auto-clipping. These are described in the following two sections.

Explicit output clipping

The first clipping mechanism is support for an explicit `clip` command in scripts, which instructs CoCo to extract a region of the current web page. The result is the concatenation of the outputs of all the clip commands within the script. For example, this script retrieves the operating hours and address for a local library by clipping two HTML table cells that contain the relevant information:

- go to http://sjlibrary.org/about/locations/almaden/index.htm
- clip the element that contains “Almaden Branch Library”
- clip the element that contains “Regular Hours:”

Note that the `clip` command is a part of the CoScripter language, first added to support the creation of Highlight [10] applications and further refined for the TrailBlazer system [3].

The result of running this script consists of two clipped outputs, separated by a horizontal rule, extracted from the web page:

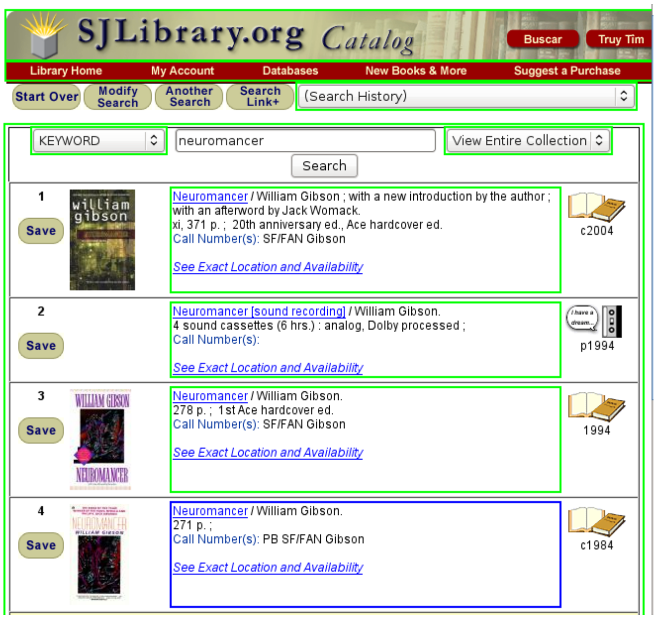
```
Almaden Branch Library
```

[1] Note that scripts mined from CoScripter Reusable History do not contain variable references. CRH logs are a literal recording of actions performed in a web browser, with no generalization.

```
6445 Camden Ave.
San Jose, CA 95120
(408) 808-3040
Email: ab.sjpl@sjlibrary.org
--------------------------------
Regular Hours:
Monday 2:00 PM - 7:00 PM
Tuesday - Wednesday 11:00 AM - 8:00 PM
Thursday - Saturday 10:00 AM - 6:00 PM
Sunday Closed
Holiday Closures
```

Unfortunately, explicit clip commands are rare in the CoScripter repository. In addition, CoScripter Reusable History does not currently record any information about the content that a user looks at on a page, which means that scripts mined from CRH will not include any explicit clip commands. Therefore, we created a novel automatic clipping feature described in the next section.

Automatic output clipping



1 Save
Neuromancer / William Gibson ; with a new introduction by the author ; with an afterword by Jack Womack.
xi, 371 p. ; 20th anniversary ed., Ace hardcover ed.
Call Number(s): SF/FAN Gibson
See Exact Location and Availability
c2004

2 Save
Neuromancer [sound recording] / William Gibson.
4 sound cassettes (6 hrs.) : analog, Dolby processed ;
Call Number(s):
See Exact Location and Availability
p1994

3 Save
Neuromancer / William Gibson.
278 p. ; 1st Ace hardcover ed.
Call Number(s): SF/FAN Gibson
See Exact Location and Availability
1994

4 Save
Neuromancer / William Gibson.
271 p. ;
Call Number(s): PB SF/FAN Gibson
See Exact Location and Availability
c1984

Figure 3: Output of our clipping algorithm on a library website. Each green region represents a clip; the blue region is the best clip relative to the command “search sjlibrary for book neuromancer”.

We formulate auto-clipping as a search problem. We assume that the most relevant response will be a portion of the last web page reached after script execution. For all possible regions on this page, which region is most representative of the “result” of the script execution? We use the user’s command text as a clue.

Our approach is as follows. First, we use the geometric clustering algorithm from CSurf [9], which uses geometry information in the rendered web page to group together DOM nodes into maximal clusters that all have the same alignment. For example, a box whose children are all left-justified is considered to be a region. These regions form the candidate set of clips for this page. Figure 3 shows an example of all the regions identified on a library website. Each region is outlined in either green or blue.

From this set of regions, we implemented a ranking feature that scores regions relative to a query (e.g., the text of the user’s command). For each region, we compute the bag of words in that region. We compare this against the bag of words in the query, and score each region using the relative size of the intersection between those bags of words. A region that contains exactly the words in the query will score 1, a region that has no words in common with the query will score 0.

The output of the auto-clipping algorithm is the region that has the highest score relative to the user’s command. For example, in Figure 3, the blue-outlined region at the bottom is considered the best match for the command “search sjlibrary for book neuromancer”. Note that all four of the books on the web page match the query, but the fourth book region is returned because it has the shortest title and therefore scores the highest on overlap with the query.

If the automatic clipping algorithm fails to produce the desired result, a user always has the option of creating a CoScripter script with explicit clip commands, to override the automatic mechanism.

Incremental auto-clipping

For some scripts, the most relevant clip is not located on the final page. One example is a script that logs you out after performing a transaction. To improve the auto-clipping algorithm for these scripts, we enhanced the basic auto-clip feature to incrementally clip from each page encountered during execution of the script.

For each page, we calculate the set of candidate regions, and then we score those regions relative to the text of the step that got us to that page (after removing stopwords such as “click” and “button”). For example, after executing the step `click the Library Hours link`, this algorithm identifies the region titled “Library Hours” on the resulting web page.

We collect these intermediate clip regions as the script progresses, and at the end, we combine these candidates with all the ones identified on the final page, as above. We then score that entire set of candidate clip regions against the original command text, and return the highest-scoring region.

In practice, this algorithm has enabled us to return clips extracted from the middle of the script’s execution. The phone-forwarding example in Bob’s usage scenario ends with a final “logout” step, so incremental auto-clip enables CoCo to report the forwarding status from an intermediate page during the script’s execution.

IMPLEMENTATION

We have implemented CoCo as a web service following the architecture described in the previous sections.

Input/output transports

Input and output are handled by a component that polls various I/O transports and manages a request/response queue.

The I/O transports we currently support include:

- SMS,
- Twitter,
- email,
- a native Android application with speech input,
- a Unix command-line interface,
- a lightweight web UI,
- and a REST API.

Extending the system to incorporate additional transports is straightforward, using the existing REST API.

Parsing

We implemented CoCo's parser using the ANTLR[2] parser generator, which generates program code given a grammar specification in a format similar to BNF. A subset of the grammar we have implemented is shown in Table 1.

Accessing user scripts and web history

In order to formulate plans, CoCo relies on having access to the user's scripts in the CoScripter repository, which are retrieved by using an API call to the CoScripter server. If the user wants CoCo to be able to synthesize scripts using CoScripter Reusable History log data, she has the option of uploading this log data to the CoCo server. Similarly, if she wishes to have her scripts parameterized using data from her CoScripter personal database, she can also upload this to CoCo. We provide a Firefox extension that makes this upload process trivial. In the future, we imagine that platforms such as Mozilla Weave[3] will allow services such as CoCo to securely access data stored in users' Firefox browsers (including both CRH logs and the personal database). We explored using Weave for our implementation, but the current release was not mature or stable enough for our needs.

Browser automation

Once a script has been found and the necessary parameter values filled in, CoCo needs to execute the script on the web. Automating the web is a challenging problem. Client-side automation tools such as CoScripter [6], iMacros[4], and Automation Anywhere[5] all require the user to be running a desktop client in order to automate a task, which is inappropriate for CoCo. Server-side automation tools such as Perl's WWW::Mechanize[6] and HtmlUnit[7] only simulate a browser on the server. Unlike full-fledged browsers such as Firefox or Internet Explorer, these simulated browsers cannot reproduce the full client-side JavaScript that today's web applications employ to provide a rich user experience.

```
command ::= {please} , command_core , {","}
    param_list , {","} , transport
command_core ::= word , {word}
param_list ::= "using" , value , "as" , var |
    "for" , var , value |
    "with" , value , "as" , var
transport ::= "via" , (sms | web | twitter |
    email)
```

Table 1: Portion of CoCo's grammar (in EBNF)

To ensure that the CoCo server interacts with a web application in the same way a human would, we leverage Highlight [10], a server-side proxy for mobile web application development and deployment. Highlight instantiates a number of Firefox browsers that can be remotely controlled to perform web-based tasks as instructed. A Highlight application consists of JavaScript code that defines what web actions to take, and what result should be returned to the user at each point.

To connect CoCo with Highlight, we developed a new Highlight "application" that takes as input a script in the CoScripter syntax, executes each step in a browser, and returns a result back to CoCo with the result of the automation. Highlight uses a copy of the CoScripter extension in each Firefox browser to parse those steps and perform the automation.

The clipping algorithms described above are implemented inside the CoCo Highlight application, to automatically extract regions of the web pages being automated and return those regions to the user.

EVALUATING COCO CONFIRMATION MESSAGES

One of the key questions for real-world usability of CoCo is whether users will trust that an intelligent agent is correctly performing tasks on their behalf. Web automation is already in widespread use through systems such as CoScripter, Automation Anywhere, and iMacros. What makes CoCo different is that CoCo is performing the automation on a server, without a user visually verifying that the automation is proceeding correctly.

One way that CoCo tries to increase trust in its automation is to confirm its intended actions with the user prior to executing them. Since the steps in CoScripter scripts are human-readable, the user could read the script and determine whether it is plausible that it should reach the goal.

However, scripts contain much less information than actual web pages. Is reading just the script text sufficient to determine whether a proposed set of actions accomplishes the desired goal? We devised an experiment, using Mechanical Turk, to answer this question.

Our intent was not to show that either condition is perfect – no automation system is flawless. On the contrary, what we wanted to understand was the *difference* between textual and visual feedback. In other words, does the reduced information in a textual representation hamper users' understanding of the effects of the automation? If the two conditions are comparable, then CoCo's interface should be no worse than existing web automation systems in use today.

Study Design

We designed our study to mimic a user's interaction with an automation system. For a given task, a user of an automation system could see the automation being performed (if using a desktop automation system) or see a textual representation

[2] http://www.antlr.org

[3] https://services.mozilla.com/

[4] http://www.iopus.com

[5] http://www.automationanywhere.com

[6] http://search.cpan.org/dist/WWW-Mechanize

[7] http://htmlunit.sourceforge.net

of the steps that would be performed (if using CoCo). After seeing this information, could a user determine whether the automation accomplished the intended task? If the automation was flawed and performed the wrong task, would the user notice this discrepancy?

To conduct this experiment, we devised 20 tasks. Each task included the task description (a few words describing the task, using language similar to what a user might provide CoCo with as input) and a script that would accomplish the task. We then captured screenshots and video of the task being executed in a web browser and recorded these as visualizations.

When automation goes awry, it often fails in subtle ways such that a user might be fooled into thinking that the automation is executing the intended task when in fact it is doing something different. To explore this effect, we created tasks in pairs, chosen either from the same site or representing the same task on different sites. For instance, one pair was looking up a zip code and calculating postage on *usps.gov*. Another pair consisted of checking flight status on *aa.com* and *united.com*. We used these tasks in the study to simulate the effect of failed automation. For example, if the user asked to check in on American Airlines but the automation showed checking in for United Airlines, would the user notice?

Using the screenshots and task information, we created three different visualizations of each task. The first visualization is a video, designed to mimic today's web automation systems (iMacros, Selenium, CoScripter), which visually highlight the interaction target for each step in the automation. The second visualization is a sequence of static screenshots representing the execution of the script, including a green highlight around each interaction target when applicable. We included this condition as a slower-paced version of the video, in case the automation playback speed adversely affected user understanding; however, we expected this condition to perform similarly to the video condition. The third visualization is simply the textual description of the steps for each task, displayed as a CoScripter script. An example of a step in such a script is `click the ''Search'' button`. This condition was designed to mimic the information that CoCo would provide to a user.

Each study participant was given one of the 20 task descriptions and one or more accompanying visualizations, and asked whether the visualizations matched the task. If multiple visualizations were displayed, they all came from the same task.

Half of the participants were given matched pairs of task/visualization information, simulating a correct automation run. The other half were given mismatched task/visualization information, simulating the case when automation failed to execute the intended task. The mismatched visualizations came from the alternate task in each task pairing (e.g. if the command was to check in on American, the visualizations showed checking in on United).

A trial was successful if, for matching task/visualization information, the participant clicked the button saying that the task matched the visualization. In the mismatched task/visualization condition, the trial was successful if the participate indicated that the task and visualizations did not match.

	Video	ScreenShots	Script	Success Rate
7.	Y	Y	Y	**77.2%**
6.	Y	Y	N	**74.4%**
5.	Y	N	Y	**77.2%**
4.	Y	N	N	**85.1%**
3.	N	Y	Y	**75.9%**
2.	N	Y	N	**75.1%**
1.	N	N	Y	**76.0%**

Figure 4: Success rate of participants under the different conditions

The study was conducted on Mechanical Turk using the TurKit API [1]. Mechanical Turk workers (Turkers) completed a total of 1219 trials over 3 days and were paid $0.02 for each successful answer. The study web page explicitly said that Turkers would only be paid if their answer was correct, providing additional motivation for them to answer correctly. Each individual Turker was allowed to complete only one assignment for each task.

Results and discussion

For each visualization condition, we averaged the results for the matching/non-matching conditions, as shown in Figure 4.

Overall, participants were correct in 77.3% of cases. The video-only condition (#4) was significantly or marginally significantly better than the other conditions. Clearly more experimentation is required to discover which factors in the video make it more informative than the other formats; one hypothesis is that participants in the video-only condition were not distracted by comparing the video against additional visualizations to make sure they all matched.

More importantly, however, we observe that with the exception of condition #4, all conditions had similar success rates. Therefore, the script-only condition is comparable to almost all of the other conditions. Thus, a user presented with only the textual representation of a script should be no worse at evaluating its correctness than a user who sees visual evidence of the automation. From these results, we conclude that CoCo's interface for confirming scripts prior to execution is likely to provide enough information for users to determine whether the reported script is the one they want.

DISCUSSION AND FUTURE WORK

We have used CoCo to complete all the tasks mentioned above in the Usage Scenarios section. All work as described except for Alice's scenario with the timecard application. While CoCo successfully locates this scenario in Alice's CoScripter Reusable History logs, and creates a script to automate the task, script execution fails to complete because of limitations in the underlying CoScripter platform. The field where Alice needs to enter her punch-out time has a label that changes every minute; CRH logs it as:

- enter "17:30" into the "OVERRIDE (04/02 18:11)" textbox

Unfortunately when CoCo tries to automate this previously-recorded step at a later date, the textbox label is different so it fails to execute this step. Note that this is was a prior limitation of the CoScripter platform, not a problem specific to CoCo.

Users are able to "teach" CoCo about new tasks simply by recording a new CoScripter script to perform that task using the existing CoScripter system. We saw this happen multiple times in our own use as we experimented with the system's capabilities and discovered new tasks that could be done with CoCo's help. We believe this is not an unreasonable requirement; even the best human assistant often needs to be instructed how to perform tasks correctly. CoScripter provides a natural way for CoCo users to specify new behaviors for the system.

While our simple syntax for allowing users to specify parameters in a command worked well for the most part, we discovered that a few commands would be parsed incorrectly. For example, one CoScripter user has a script named "log 30 minute run for fitness rebate". The keyword `for` in the command is interpreted by CoCo as the beginning of a parameter specification (fitness=rebate), when actually it should be part of the command. CoCo's confirmation feature should mitigate this problem, because it should be apparent when a command is being interpreted incorrectly. We also plan to improve the syntax to reduce the occurrence of these mistakes. Also, although our system assumes that all parameters are named, we would like to extend the syntax to support unnamed parameters, to support commands such as "get list price for cats soundtrack".

Although we provided the ability to automatically mine new plans from the logs created by CoScripter Reusable History, in practice this feature has not seen much use. One drawback of CRH-mined scripts compared to CoScripter scripts is that they lack the descriptive titles that make them easily and uniquely searchable. CoCo's search algorithm is only able to use the words in the CRH logs, which include link text and button labels – hardly high-level task descriptions. One area for future work will be to improve recall of CRH-mined scripts.

We also discovered a more subtle problem with reusing CRH-mined scripts in conjunction with auto-clipping. Since our algorithm for retrieving scripts from CRH does a search over the text of the step, the best queries for retrieving CRH scripts refer to the *process* being performed. However, the auto-clipping algorithm uses the same query to decide which *content* to extract from the web page. The same words don't always describe both the process and the content for a task, therefore the same command may not always produce both a reasonable CRH-mined script and a reasonable auto-clipped result. Solving this problem is an open area for future work.

While we were developing CoCo, Twitter released a new feature that allows you to add your location to each of your tweets. Although we have not yet taken advantage of that feature in CoCo, we see many possibilities for its use. A user asking CoCo for weather conditions could get the forecast for her location, after CoCo converted the geographic coordinates provided by Twitter into a parameter for a weather retrieval script. A traveler could ask CoCo for Yelp reviews of restaurants near his location, delivered to his email.

Preventing mistakes

One open problem for CoCo is addressing users' trust that it will perform the right actions on a user's behalf, and that it will not do anything malicious. The act of creating a CoScripter script seems to be a good indicator that a task is "safe" for automation. However, the use of arbitrary scripts mined from CoScripter Reusable History introduces more serious concerns. If CoCo is capable of repeating any sequence of actions you have previously done on the web, what stops it from purchasing another item of furniture, cancelling your electricity service, or uploading the wrong PDF to a conference paper submission system?

The conversational interface we have designed into CoCo is the first step towards increasing users' trust, and our Mechanical Turk study shows that users can comprehend scripts presented textually. However, users can make mistakes and we would like to be able to protect them from the results of improper automation.

One feature we plan to develop is the ability to automatically recognize web actions that have side effects in the real world: steps that involve financial transactions, or require confirmation. We might be able to use machine learning to recognize such "side-effecting" actions. If such an action were detected, we could then open a dialogue with the user to confirm that they really want to take this action before proceeding.

Securing private information

One possible area of user concern is the security of their private information. Because CoCo is accessible via public communications channels such Twitter and email, a malicious attacker could hijack one of these communications channels and potentially request CoCo to perform dangerous actions on an honest user's behalf. Currently CoCo trusts in third party authentication systems (e.g. it requires that you be logged in to Twitter), though a future implementation could require a second layer of authentication beyond what the third party services provide. These extra authentication steps must be balanced against ease of use, however; if communicating with CoCo requires too much overhead, the system may not be usable.

RELATED WORK

CoCo builds on many previous systems in the space of intelligent web assistants. In many ways it follows in the footsteps of the original Internet Softbot [5], which could synthesize plans to perform file-manipulation tasks over FTP given high-level goals such as "print all of mitchell's papers". Rather than attempting to apply a general-purpose planner and synthesizing all plans from scratch, CoCo severely restricts the domain by leveraging a personalized plan repository created by each user (in the form of CoScripter scripts and CRH logs). As a result, CoCo is able to perform a richer variety of tasks, at the cost of users having to teach the system how to do the tasks they want it to do.

CoCo's natural language interface is similar to previous question answering systems on the web such as SmartWeb [12], GOOG 411[8], or IBM's Watson DeepQA system[9]. Unlike question answering systems whose goal is to perform general purpose information retrieval and return an answer, CoCo is capable of actually *performing* actions on a user's behalf in order to interact with the web. However, CoCo is limited to only tasks that have previously been done, compared to general-purpose QA systems which are designed to answer any question, even ones that have not yet been seen.

CoCo is also similar to Siri[10], an intelligent assistant for the iPhone that allows delegation of certain tasks such as making restaurant reservations, getting movie tickets, and hailing a taxi. However, CoCo is designed to be extensible to user-defined tasks, rather than limited to functionality already programmed in to the system. More recently, HP's SiteOnMobile[11] allows end users to create "tasklets" to scrape data from websites and make it available via SMS or voice calls. CoCo goes beyond SiteOnMobile by adding a natural-language interface and an auto-clipping algorithm to automatically summarize the results of a task.

Web automation systems such as Chickenfoot [4], PLOW [2], and Ubiquity[12] all enable users to automate the web and define new scripts for automation. However, all of these services expect users to be interacting with the web on a full-fledged browser. In contrast, CoCo aims to hide the complexities of the web behind a simple command-driven interface. An item for future work is to incorporate scripts sourced from any of these systems into the CoCo framework so they can be leveraged as plans for automation.

Finally, CoCo's intelligent assistant-style interface to the web was inspired by past work on web accessibility, including CSurf [9], TrailBlazer [3], and aDesigner [11]. In future work we would like to explore uses of CoCo within the accessibilty community.

CONCLUSIONS

In summary, we have presented CoCo, an intelligent conversational assistant for the web that is capable of performing tasks on users' behalf as instructed through a simple textual interface. Given a short textual command from a user, CoCo locates a plan to meet the user's goal, fills in parameters as necessary to customize the plan, automatically executes it on the web, and returns a result back to the user representing the result of the automation. Our experience with CoCo has shown that it is capable of performing a variety of web tasks, and returns useful results to let us know what it has done.

REFERENCES

1. TurKit, 2009. http://groups.csail.mit.edu/uid/turkit/.

2. J. F. Allen, N. Chambers, G. Ferguson, L. Galescu, H. Jung, M. D. Swift, and W. Taysom. Plow: A collaborative task learning agent. In *AAAI*, pages 1514–1519. AAAI Press, 2007.

3. J. P. Bigham, T. Lau, and J. Nichols. TrailBlazer: Enabling Blind Users to Blaze Trails Through the Web. In *IUI '09: Proceedings of the 13th international conference on Intelligent user interfaces*, pages 177–186, New York, NY, USA, 2009. ACM.

4. M. Bolin, M. Webber, P. Rha, T. Wilson, and R. C. Miller. Automation and customization of rendered web pages. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 163–172, New York, NY, USA, 2005. ACM.

5. O. Etzioni and D. Weld. A softbot-based interface to the Internet. *Commun. ACM*, 37(7):72–76, 1994.

6. G. Leshed, E. M. Haber, T. Matthews, and T. Lau. CoScripter: Automating & Sharing How-To Knowledge in the Enterprise. In *Proceedings of the 26th SIGCHI Conference on Human Factors in Computing Systems (CHI '08)*, pages 1719–1728, Florence, Italy, 2008.

7. I. Li, J. W. Nichols, and T. Lau. Here's What I Did: Sharing and Reusing Web Activity with ActionShot. In *Proceedings of the 28th SIGCHI Conference on Human Factors in Computing Systems (CHI '10)*, 2010. To appear.

8. G. Little, T. A. Lau, A. Cypher, J. Lin, E. M. Haber, and E. Kandogan. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *CHI '07: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 943–946, New York, NY, USA, 2007. ACM.

9. J. U. Mahmud, Y. Borodin, and I. V. Ramakrishnan. CSurf: a context-driven non-visual web-browser. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 31–40, New York, NY, USA, 2007. ACM.

10. J. Nichols, Z. Hua, and J. Barton. Highlight: a system for creating and deploying mobile web applications. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 249–258, New York, NY, USA, 2008. ACM.

11. H. Takagi, C. Asakawa, K. Fukuda, and J. Maeda. Accessibility designer: visualizing usability for the blind. In *Assets '04: Proceedings of the 6th international ACM SIGACCESS conference on Computers and accessibility*, pages 177–184, New York, NY, USA, 2004. ACM.

12. W. Wahlster. Smartweb: multimodal web services on the road. In *MULTIMEDIA '07: Proceedings of the 15th international conference on Multimedia*, pages 16–16, New York, NY, USA, 2007. ACM.

[8] GOOG 411: http://www.google.com/goog411/

[9] DeepQA: http://www.ibm.com/deepqa

[10] Siri: http://siri.com

[11] SiteOnMobile: http://siteonmobile.com

[12] Ubiquity: https://mozillalabs.com/ubiquity/