Koala: Capture, Share, Automate, Personalize Business Processes on the Web

Greg Little[1], Tessa A. Lau[2], James Lin[2], Eser Kandogan[2], Eben M. Haber[2], Allen Cypher[2]

[1]MIT CSAIL
Building 32
77 Massachusetts Ave
Cambridge, MA 02139 USA
glittle@mit.edu

[2]IBM Almaden Research Center
650 Harry Rd
San Jose, CA 95120 USA
{tessalau, jameslin, eser, ehaber, acypher}@us.ibm.com

ABSTRACT
We present Koala, a system that enables users to capture, share, automate, and personalize business processes on the web. Koala is a programming by demonstration system that records, edits, and plays back scripts, which are represented as a list of pseudo-natural language instructions. Compared to previous programming by demonstration systems, Koala pioneers the use of *sloppy programming*—a natural language representation for scripts that is both human- and machine-understandable. Our initial experiences show that Koala's sloppy interpreter is also effective at interpreting instructions originally written for people. Koala scripts are automatically stored in the Koalescence wiki, where a community of users can share and collaboratively develop their "how-to" knowledge. Koala also takes advantage of corporate and personal data stores to automatically generalize and instantiate user-specific data, so that procedures created by one user are automatically personalized for others.

Author Keywords
end-user programming, programming by demonstration, wikis, collaboration, business process, automation, knowledge capture.

INTRODUCTION
Modern business is full of complex, multi-step procedures performed at irregular intervals by many people. For example, corporate cost cutting efforts often force end-users to perform their own travel arrangements, hiring, and purchasing, despite their lack of domain expertise. Salespeople who make their own travel arrangements may not be familiar with every detail of corporate travel policy, such as restrictions on hotels or daily meal allowances. Engineers wanting to order a new monitor may be confused by arcane terminology choices such as "Net parts Expense - Purchased from Vendor" versus "Purchased & Leased Equip.- PC/Workstation" in the procurement process.

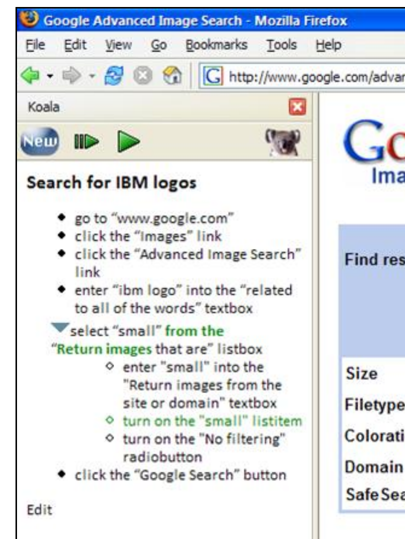

Figure 1. The Koala sidebar in Firefox.

We have created a system called Koala to address this problem by making it easy for end users to document and record the specifics of performing a business process, using a wiki to share their experience and allow others to automatically execute the process on their own desktops.

Koala enables end-users to record and automate their activities within the Mozilla Firefox web browser. Users can perform an activity once, and Koala records all the forms filled, links and buttons clicked, and menus selected. The procedure is saved in the form of pseudo-natural language text, which users can easily read and edit, and which Koala can interpret and execute. The Koala interpreter is sufficiently flexible that it is also quite successful in interpreting and automatically executing instructions written for people. Koala procedures are saved to Koala's wiki, called *Koalescence*, so that users can take advantage of working procedures created by their colleagues. By making pseudo-natural language human-readable instructions available on a wiki, Koala makes it

easy for co-workers to collaboratively edit and share procedures.

PRIOR WORK

Koala builds on a number of emerging technologies in the web space. Greasemonkey [1] enables users to make client-side modifications to the appearance and behavior of web pages on their computer. However, creating a Greasemonkey script requires detailed knowledge of JavaScript programming to alter the DOM of the web page.

Chickenfoot [2] eases client-side customization by providing a higher-level API for accessing and manipulating common web page elements, using information in the rendered DOM. For example, the Chickenfoot instruction `click('search button')` will click a button with the text "search" near it. However, the Chickenfoot interface is still very much a programming interface, in which users write statements in the Chickenfoot programming language.

The Keyword Commands approach [3] seeks to lower the language barrier even further with what is called *sloppy programming*. This approach can interpret expressions composed of keywords, such as `click search button`. However, the Keyword Command algorithm was specifically designed to handle detailed scripting languages and does not scale well to the longer textual inputs that are likely to appear in how-to documents written for humans.

Koala modifies the Keyword Command approach for application to the web domain by taking advantage of the fact that most web commands are flat: there is one verb, and one or two arguments. This assumption dramatically simplifies the algorithm, and makes it more robust to extraneous words. It can handle long expressions originally intended for humans, such as `scroll down to the "Shop by commodity" section, click "View commodities"` (where only the last 3 words are meaningful to the system).

Koala also exposes the synergy of a new paradigm in end-user programming, namely the combination of: 1) a sloppy interpreter; 2) programming by demonstration where recorded actions are human readable and editable; and 3) a wiki for procedures, which serves as an end-user version control system.

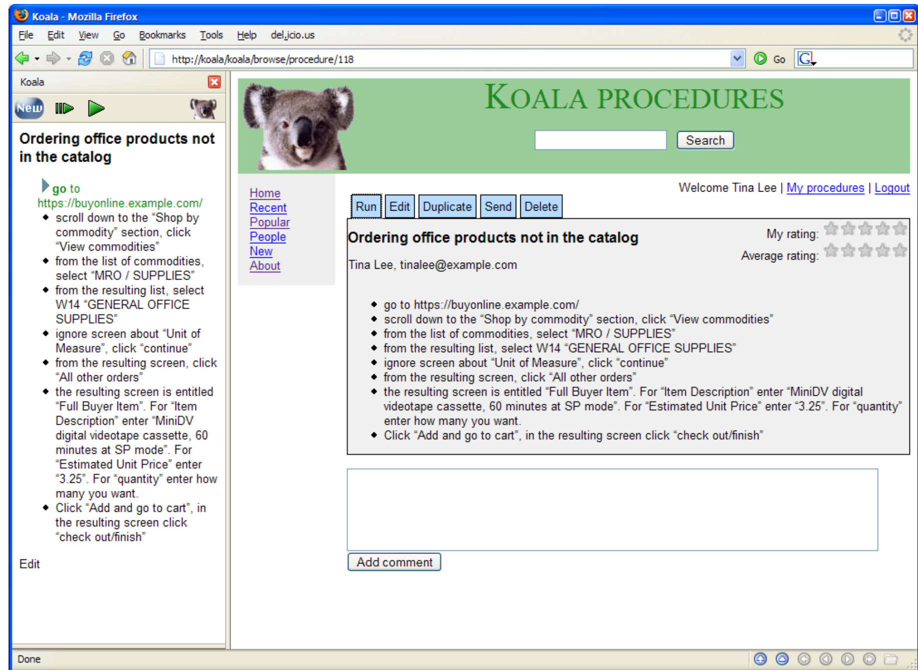

Figure 2. Koalescence is a wiki for sharing procedures among a community of users. Users can execute procedures as-is, or edit them to make improvements.

Koala consists of two parts: a wiki, where users can search for and interact with a collection of user-defined scripts (Figure 2); and a Firefox extension that adds a sidebar to the browser for recording and executing procedures (Figure 1).We illustrate the use of Koala with an example which has been drawn from experience (Figure 3).

AN EXAMPLE: ORDERING OFFICE PRODUCTS NOT IN THE CATALOG

One of our first actual uses of Koala occurred when Tina wanted to order a type of pen through our company's online ordering system (BOND), but the pen was not listed in the

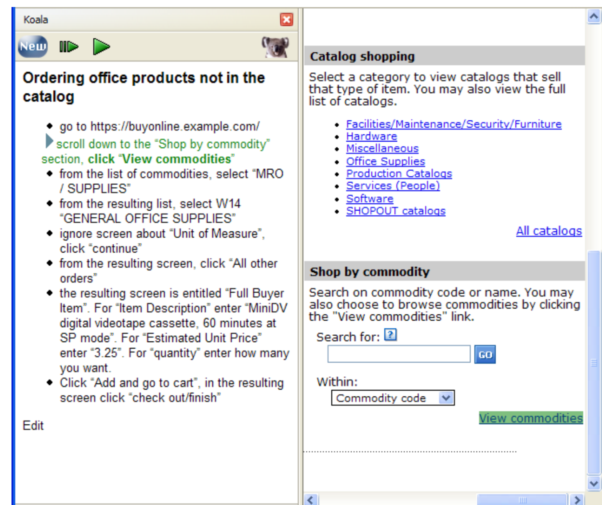

Figure 3. Koala use-case scenario: Tina first copies instructions sent to her in an email into the Koala sidebar.

online catalog. Tina didn't know how to proceed, so her colleague Edward sent her step-by-step instructions, based on how he had ordered video cassettes which were not in the catalog. The following example is based on that email.

Tina opens the Koala sidebar, which contains buttons for creating a new procedure, executing one step of the current procedure, executing all remaining steps, turning on recording, and showing the procedure in Koalescence. She pastes in the instructions and clicks Run. Koala highlights the first step in red, which says `login to BOND`. The red means that Koala does not know how to perform the step, so Tina clicks the Edit button below the procedure, which puts Koala into Record mode, and then she selects her personal bookmark for the BOND website. Koala records this menu selection in the script as: `go to https://buyonline.example.com/`. Tina deletes the line `login to BOND` and clicks the Run button again. Koala performs each step of the script automatically, pausing for each web page to load before continuing to the next step. Koala highlights the current step in green, and visually flashes the corresponding button or input field on the web page so that Tina can see what it is about to do.

When Koala gets to the next-to-last step, it highlights that line in red and does not attempt to execute it. There are two reasons for this. First, Koala does not currently attempt to split multiple steps listed on a single line. Second and more important, Koala does not attempt to interpret any line containing the word "you". Instead, it waits for the user to perform the step manually. This is a simple and effective way to enable *mixed initiative* execution of procedures: the computer performs some steps and the user performs others, usually the steps that are too complex to be worth automating. This is a staple of practical end user programming [4].

When she has finished running the procedure, Tina can view this procedure within Koalescence by clicking the Koala button in the sidebar. She can edit it further, rate it, comment on it, or use the Send button to email a message about it to her colleague.

SLOPPY PROGRAMMING

Each step in a sloppy program is interpreted by an interpreter that tries to evaluate the step relative to the currently displayed web page.

At its core, the sloppy interpreter is a function that takes two arguments: an HTML document, and some *slop* (a line of pseudo-natural language text). For instance, we could pass in the Google home page, and the slop *click search button*. The interpreter in this case would programmatically click the search button. Let's describe the algorithm in three basic steps, using the example instruction *type Danny into first name field*.

First, we enumerate all the possible actions in the webpage.

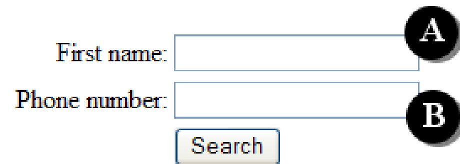

Figure 4. A simple web form.

For our purposes, all actions are associated with various HTML objects. In particular, we associate actions with links, buttons, text boxes, combo-box options, check boxes, and radio buttons. For each of these objects, the interpreter associates a variety of keywords, including the object's label, synonyms for the object's type, and verbs commonly associated with the object. For instance, the *First name* field of a web form (Ⓐ in Figure 4) would be associated with the words *first* and *name*, because they are part of the label. It would also associate *textbox* and *field* as synonyms of the field's type. Finally, the interpreter would include verbs commonly associated with text fields such as *enter*, *type*, and *write*.

Next, the system searches for the object that matches the greatest number of keywords with the slop. In the example, it would match the keywords *type*, *first*, *name* and *field*, which is a score of 4. In contrast, another field in the webpage with the label *Phone number* (Ⓑ in Figure 4) would only match the keywords *type* and *field*, resulting in a score of 2.

Finally, the system may need to do some post-processing on the slop to extract arguments. In our present example, we need to extract the word *Danny*, since this is the text that we want to place in the text field. The key idea of this process is to divide the slop into two parts, such that one part is still a good match to the keywords associated with the web object, and the second part is a good match for a string (e.g. has quotes around it, or is followed/preceded by a preposition). In this case, we divide the string into *type into first name field* (which still matches 4 keywords) and *Danny* (which gets points for being followed by the preposition "into" in the original slop).

Note that our sloppy procedures consist only of text. We do not save interpretations or any other additional information with the procedure. This is a radical departure from previous programming-by-demonstration techniques, and it is surprisingly effective since a web page provides a very small search space. As with Chickenfoot [2], the use of textual labels to identify web page elements makes Koala scripts more robust to changes in the page, since we expect that developers will keep constant this proximate text-labeling, to avoid confusing their users. Furthermore, the plain-text representation of Koala scripts facilitates wiki-style collaboration: users can simply edit the document and change the script. Finally, because Koala avoids a formal programming language and instead just sloppily interprets

natural language text, users can import how-to documents from existing help systems into Koala, and have good success executing them.

UNDERSTANDING AND CORRECTING KOALA

Because Koala's interpreter is sometimes wrong, we have implemented several techniques to help the user know what the interpreter is doing and make corrections. We'll illustrate these techniques with the following example:

Suppose the user has selected the highlighted line in Figure 3: *scroll down to the "Shop by commodity" section, click "View commodities"*. Koala interprets this line as an instruction to click a link labeled *View commodities*. At this point, we want to make two things clear to the user: what is the interpreter planning to do, and why?

We show the user what the system is planning to do by placing a transparent green rectangle around the *View commodities* link, which is also scrolled into view (see Figure 3).

We address the question of "why" by letting the user know which words in their line lead the interpreter to its selection. In this case, the words *click*, *view* and *commodities* were associated with the link, so we make these words bold: *scroll down to the "Shop by commodity" section,* ***click "View commodities"*** (see Figure 3).

If the interpretation was wrong, the user can click the triangle to the left of the line, which expands a list of alternate interpretations. These interpretations are relatively unambiguous instructions generated by the interpreter:

- click the "View commodities" link
- click the "View contracts" link
- click the "Skip to navigation" link

When the user clicks on any of these lines, the system places a green rectangle over the corresponding HTML control. If the line is the correct interpretation, they can click the Run or Step button to execute it. If not, they may need to edit the line. Failing that, they can add the keyword *you* (e.g., ***you*** *click the "View commodities" link*) so that the interpreter leaves interpretation to the user.

SCREEN SCRAPING BY EXAMPLE

Mashups are typically created by wiring together web service APIs from two or more web sites. However, there is a dearth of web sites offering web services, making them difficult to mash up. In contrast, there are lots of web sites that offer interactive interfaces to underlying data. A screen scraper effectively turns any interactive web site into a web service, but they are hard to write and often very brittle. Since we developed Koala to enable ordinary end users to interact with web pages, we decided to re-purpose Koala to enable end-user creation of screen scrapers.

In addition to the existing capabilities in Koala for recording the way a user navigates around web pages, we added two features for creating scripts for scraping: 1) a Select Similar operation (Figure 5), that lets the user identify an important element on a web page, and have Koala automatically locate all of the similar elements on the page, so that they can all be scraped, and 2) a Scrape operation, which lets the user select an element and give it a property name to be used in scraping. To make selection easy, the HTML element under the user's mouse is highlighted with a red border, so users can simply move their mouse over the page, and then click on an element that they want to scrape. When Koala runs a script that includes screen-scraping operations, it outputs XML, similar to what could be obtained from a web service, into a separate pane in the sidebar.

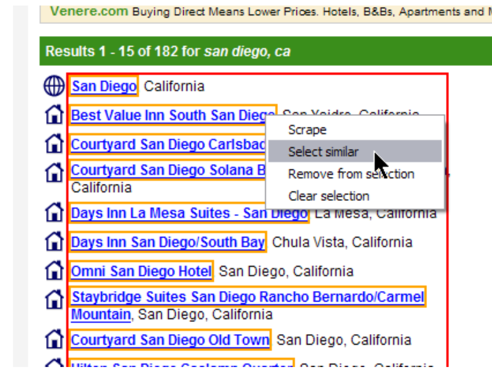

Figure 5. Screen-scraping operations in Koala.

CONCLUSION AND FUTURE WORK

Koala allows users to capture, share, automate, and personalize business processes on the web. With Koala, the barriers to capturing procedural knowledge are significantly lowered. Future work includes leveraging past contexts, adding data detectors, enriching the interface to support control flow, and enlarging the community of users to help evaluate it against a wider variety of real-world tasks.

REFERENCES

1. McFarlane, N. 2005. Fixing web sites with Greasemonkey. *Linux J*. 2005, 138 (Oct. 2005), 1.
2. Bolin, M., Webber, M., Rha, P., Wilson, T., and Miller, R. C. Automation and customization of rendered web pages. In *Proc. UIST 2005*, ACM Press (2005), 163–172.
3. Little, G., and Miller, R. C. Translating Keyword Commands into Executable Code. In *Proc. UIST 2006*, ACM Press (2006).
4. Horvitz, E. Principles of Mixed-Initiative User Interfaces. In *Proc. CHI '99*, ACM Press (1999). 159–166.