Predator Behavior in the Wild Web World of Bugs: An Information Foraging Theory Perspective

Sandeep Kaur Kuttal, Anita Sarma, and Gregg Rothermel
Computer Science and Engineering
University of Nebraska–Lincoln
Lincoln, NE, USA
{skuttal,asarma,grother}@cse.unl.edu

***Abstract*—Web active end users often coalesce web information using web mashups. Web contents, however, tend to evolve frequently, and along with the black box nature of visual languages this complicates the process of debugging mashups. While debugging, end users need to locate faults within the code and then find a way to correct them; this process requires them to seek information related to web page content and behavior. In this paper, using an information foraging theory lens, we qualitatively study the debugging behaviors of 16 web-active end users. Our results show that the stronger scents available within mashup programming environments can improve users' foraging success. Our results lead to a new model for debugging activities framed in terms of information foraging theory, and to a better understanding of ways in which end-user programming environments can be enhanced to better support debugging.**

I. Introduction

A "web-active end user" is a person who engages in many internet activities, but lacks programming expertise [34]. For many such end users, the web has become a vital part of day-to-day life. It has been estimated that 34.3% of the world's population and 78.6% of the North American population use the internet [20]. The web, however, is content rich, and finding ways to effectively and efficiently access the information it provides can be challenging. Thus, web-active end users would often like to be able to "cobble together" various sources of data, functionality and forms of presentation to create new services [34]. Web mashups provide one approach to facilitate this. Web mashup environments such as Yahoo! Pipes [33], IBM mashup maker [12], and Deri pipes [5] allow non-programmers and programmers alike to compose various sources of data by taking advantage of dataflow concepts and visual interfaces. These environments ease the task of application development by end users.

Most web mashup programming environments deal with information that is collected at a given point in time. However, web contents change frequently [6], and this can cause mashups to fail, as the sources of data on which those mashups depend change. In fact, in one study of one particular mashup domain [15], it was noted that 64.1% of a large set of mashups had become erroneous due to the evolution of information on which they depended. Complicating this need is the fact that mashups tend to utilize various components in "black-box" manners, and this adds layers of abstraction to the problem of understanding and debugging them.

Faced with a program failure, programmers must "forage" through code and related information to identify and correct the fault responsible for it. In an effort to better understand this activity, information foraging theory has been studied and applied in connection to the process of "foraging" for information on the web [4], [8], [23], navigating through programs [22], and debugging [16], [17], [18].

During a study of users focusing on their "sensemaking" behavior while debugging, it was found that they spent up to two-thirds of their time foraging [10]. Studies of end users creating mashups [1] have also shown that they spend a significant portion of their time (76.3%) in debugging. To date, however, there has been no attempt to examine end user foraging behavior in the context of mashup debugging.

We believe that studying end users' foraging behavior in the context of mashup debugging will help us better understand the processes that end users follow in debugging, and better support those processes. We thus conducted a study of the behavior of 16 web-active end users, focusing on foraging activities observed during mashup debugging activities.

Based on an analysis of the data from our study, we derive a new model of debugging activities based on information foraging theory (IFT). Our model considers foraging behavior relative to both fault localization and fault correction. Our analysis reveals various cues that end users use while foraging, and several different strategies they use during localization and correction, and frames these in terms of our model. In doing this, we discover several ways in which mashup programming environments, and we suspect, end-user programming environments generally, can be enhanced to provide better cues, and to better support end users' debugging strategies.

II. Background and Related Work

A. End-user Debugging

Debugging is an integral part of programming. Studies have shown that professional developers as well as students [7] spend significant portions of their time debugging. In fact, Rosson et al. [27] suggest that even professional programmers often "debug into existence" their programs; that is, they create their programs though a process of successive refinements.

There has been some work directed at end-user programmers engaged in debugging. Grigoreanu et al. [9] have developed a classification framework for characterizing end-user programmers' debugging strategies. They identify several debugging strategies including code inspection, following data

978-1-4799-0369-6/13/$31.00 ©2013 IEEE

flow, following control flow, testing, feedback following, seeking help, following spatial layout and specification checking. In a subsequent study Grigoreanu et al. [10] examined the sensemaking process; that is, the process by which end users understand bugs and their causes (based on prior work by Priolli and Card [25]). Sensemaking consists of an information foraging loop followed by attempts to understand the foraged information. They found that the foraging loop dominated the sensemaking process. As noted in Section I, Cao et al. [1] observed that end users creating mashups spend a significant portion of time debugging. There have been no studies, however, of the use of foraging by end users debugging mashups.

B. Information Foraging

Information foraging theory (IFT) is based on optimal foraging theory, developed by Pirolli and Card [24] to understand how humans search for information. Optimal foraging theory is rooted in the biological sciences, in studies and theories of how animals hunt for food; this led Pirolli and Card to find similarities between users' information search patterns and animals' food foraging strategies. Human "predators" searching for information "prey" look at various information sources. By following "cues" in the environment they look for "scents" i.e., indicators of the relation of information sources to prey. They then look for prey in "patches" and sometimes engage in "enrichment" (modifying the environment) of a patch to increase the chances of success.

Patch models and diet models are important for IFT [24], [30]. A patch model predicts the amount of time that a predator will spend foraging in a patch before leaving for another patch. A predator will spend time foraging for prey within a patch until resources are depleted, and then will move to a new patch. A diet model deals with the tradeoff when a predator forages in a habitat that contains a variety of prey. If the predator's diet is too narrow it will spend more time searching for prey, and if it is too broad it will hunt for unprofitable prey.

Information foraging has helped researchers understand how users interact with the web. Navigational models have been developed based on the foraging behaviors exhibited by users; these help predict the navigational patterns of users and enhance the usability of websites [4], [8], [23]. Information foraging has also helped inform principles of design of websites [21], [29] and user interfaces [31]. Information foraging theory has also been helpful in accurately predicting the navigational behavior of programmers in general [22]. Lawrance et al. have mapped foraging theory to the debugging domain and developed models to describe the evolving goals of a programmer [16], [17], [18] when debugging.

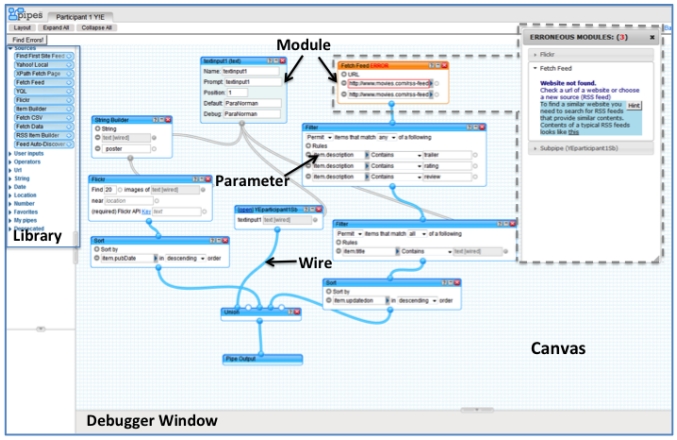

Fig. 1: Yahoo! Pipes with debugging support

III. Empirical Study

To investigate the information foraging behavior of end-users we conducted a new analysis of debugging behavior from an IFT perspective, based on results of a previous study [15] of debugging support for mashup programmers. Additional study details can be found in [15].

A. Participants

We emailed several departments in our university inviting students to participate in the study, promising a $20 gratuity. To participate, we required respondents to have experience with at least one web language. Background in computer science was not allowed beyond the rudimentary requirements of their majors. We selected 16 participants of varying backgrounds, including engineering, science, and arts. We used stratified sampling to categorize participants based on their experience with the web, programming languages, gender, and Yahoo! Pipes into a Control Group and an Experimental Group, each containing five males and three females. We chose to include data from participants who debugged with and without debugging support, because without debugging support, participants spent most of their time localizing faults and very few fixed bugs. Therefore, to study fault-fixing behavior that involves different actions such as foraging across multiple patches, we needed to analyze data from participants with debugging support.

B. Environment

Our study focused on mashup programming using Yahoo! Pipes [33], and on an extension to that environment providing debugging help to end users. Figure 1 provides a snapshot of the Yahoo! Pipes programming environment and a sample pipe; the pane inset at upper-right is information provided by our extension.

Yahoo! Pipes "programs" combine modules using wires that transfer data between them. The programming environment consists of three components: the library, canvas, and debugger. The library is located to the left and consists of a list of modules categorized according to functionality. The canvas is the central area in which users create pipes by placing modules and connecting them together. The debugger helps users see the output of specific modules as well as the final output of the pipe.

Inputs and outputs to pipes can be HTML, RSS, JSON, KML, and other formats. Inputs and outputs between modules are primarily RSS feed items. Yahoo! Pipes modules provide manipulation actions that can be executed on these RSS feed parameters. Yahoo! Pipes also allows users to define various datatypes such as URL, text, number, and date-time.

We added debugging support to the environment in the form of a "To-fix list". The To-fix list provides information on bugs that need to be resolved and their properties. The To-fix list is populated when the user saves or executes a pipe or

TABLE I: Details on Seeded Bugs

Task	Class	Bugs	Details
Yahoo! Error	Top	B1	API key Missing
	Level	B2	Website not found
	Nested	B3	Website not found
Silent Error	Top	B4	Website contents changed
	Level	B5	Parameter missing
	Nested	B6	Parameter missing

clicks a "Find Error" button. The To-fix list is overlaid on the top, right-hand side of the canvas so that users can view both the pipe and the list of bugs. Erroneous modules are listed in the order in which they appear on the canvas (top to bottom, left to right).

We provide information on the context of each To-fix item in the list; that is, we link each bug in the list to the faulty module so that when a user clicks on a bug in the list, that module is highlighted (marked in orange) and parameters implicated in the bug (if any) are marked in red. We also provide reverse functionality; that is, when an erroneous module is selected (or hovered over) the To-fix list expands to reveal the bug(s) in that module.

We followed Shneiderman's guidelines [28] to design error messages that are clear, concise, and use plain language. We provide constructive steps to help users arrive at solutions to bugs. We also provide a "Hint" button that can be expanded to provide further details for resolving the problem.

C. Procedure

The study used "think-aloud" protocol [19]. Verbal dialogues and onscreen interactions of participants were screen captured. The participants began by completing self efficacy questionnaire, and then were given a 10-minutes tutorial on Yahoo! pipes, which included information on how to create pipes and the functionalities of modules. The Experimental Group also received instructions on how to access our debugging support extension. We asked users to create a sample pipe to gain further familiarity with Yahoo! Pipes. We then asked users to complete two debugging tasks. The total time required for completion of a session per participant was approximately 80 minutes, which included an average of 50 minutes for task completion.

D. Tasks

Participants completed two tasks, which were counterbalanced to compensate for possible learning effects. The tasks were designed to understand the behavior of participants in the case in which feedback is provided by the Yahoo environment or absent. We also wished to examine the effect of bug nestedness. We seeded three bugs into each of the two pipes; for details see Table I. The participants were given the buggy pipes and were asked to find the bugs and correct the pipes.

Our first task, related to errors for which Yahoo! Pipes provides feedback, was "the pipe should display (1) a list of the top 10 rated movies (from rottentomatoes.com) along with their ratings (in descending order), (2) a poster of a selected movie (from Flickr) and (3) a review of the selected movie". Our second task, related to silent errors, was "the pipe should display (1) a list of theaters around a given area, (2) a list of movies in each theater along with their show times and (3) trailers of the top 10 movies (from rottentomatoes.com)".

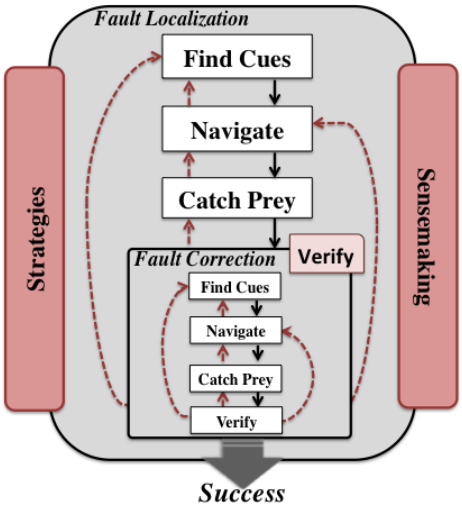

Fig. 2: Foraging Behavior during Debugging Activities

E. Analysis Methodology

For the study discussed in this paper, we transcribed all verbalizations and actions performed by our participants. We analyzed the data using codes from Pirolli and Card [23] (IFT in web navigation) and Lawrance et al. [18] (debugging behavior among professional programming through IFT). However, on analyzing the data we realized that we were observing different phenomena and needed to add new codes. Through iterative analysis of the data we created these by following the tenets of grounded theory [3].

IV. Results

Debugging involves three overall steps: (1) a user must identify a failure in program behavior (typically in output), (2) then they must localize the fault that causes the failure (typically incorrect or missing code or a problem involving program inputs) and (3) then they must identify and implement a solution to the fault. Where end users are concerned, these three steps are typically interleaved.

We model end-users' debugging behavior by drawing on IFT. We do this because debugging inherently involves foraging for cues to identify a fault and foraging for cues that lead to a solution. Although users may not separate these two debugging stages, we do so in order to better model debugging behaviors through the IFT perspective. In this study our participants were given information directly relevant to identifying that a failure has occurred, so we restrict our attention to steps 2 and 3, fault localization and fault correction.

We posit that a user's foraging behavior during debugging involves the following steps: (1) find cues and process them into scents, (2) navigate through patches, (3) catch prey, and (4) "verify" the viability of the prey. These are the basic debugging steps proposed by Lawrance et al. [17]. Figure 2 depicts this process. In the figure, the outer box represents the fault localization task, and the inner box represents the fault correction task. The fault correction task itself serves as

TABLE II: Redefining Information Foraging Terms for Debugging by End Users

IFT Extended	Definition	Fault Localization (Example)	Fault Fixing (Example)
Prey	Potential fault during fault localization; Potential fix during fault correction	Finding `Fetch Feed` module that contains bug B2 (website doesn't exist)	Finding the correct url and putting it in the `Fetch Feed` module that contains B2
Information Patch	Localities in the code, documents, examples, webpages and displays that may contain the prey [18]	Yahoo! Editor	Websites, e.g. rottentomatoes.com
Cues	Words, links, error messages, or highlighted objects that suggest scent relative to prey.		
Clear	Cues that are easy for end users to understand	API Key Missing error message for bug B1	Key link to the Flickr page to collect the API key
Fuzzy	Cues that are difficult for users to understand	"Root cause: org.xml.sax.SAXParseException: root element must be well-formed"	"Error fetching [url]. Response: Not Found (404)"
Elusive	Cues that are difficult for users to find	Finding bug B3 and B6	Finding the RSS feed while correcting bug B2
Navigate	Navigation by users through patches	To find bug B2 the user navigated through Yahoo! Pipes editor to external website	To correct bug B2 participant navigated to various websites to find the required url

the verification step for fault localization, as it is the ability to correct the fault that establishes that a fault has been localized.

While debugging, users can backtrack to different navigation paths or look for different cues if they are unsuccessful in catching the prey, and backtracking is represented in the figure by dotted edges. We further refine this view by suggesting that (1) users employ a set of strategies to identify cues and navigate through patches and (2) sensemaking is a step that users mentally perform as they create scents from cues and navigate through the patches. These two elements in the model occur throughout the information foraging process and therefore are shown as crosscutting concerns in the model.

The key elements of this model, their definitions, and an example of each for the localization and correction steps are shown in Table II. A user is the "predator" and their "prey" is a potential fault or a potential fault correction. A predator forages through patches (information areas) and based on cues from the environment picks up scents that may lead them to their prey. Simply capturing the prey is not enough, however; the predator must also ensure (verify) that the solution is appropriately formatted (we call this a "diet constraint").

We further identify different strategies that a user can incorporate while seeking and acting on cues, navigating, and capturing prey, which we explain in detail later. Another key element of IFT are cues in the environment. We found that cues differ in how well they: 1) add conceptual clarity to error messages, 2) promote detectability, 3) connect users with relevant debugging information, and 4) narrow the search space for users. Based on these criteria we classified cues as clear, fuzzy, and elusive. *Clear cues* contained direct links to a fault or correction, or text that could be clearly attributed to a piece of potentially faulty code or a potentially valid correction. *Fuzzy cues* contained information adequate to lead to a particular prey, but were difficult for participants to comprehend. *Elusive cues* were cues that existed in the environment, but were particularly difficult for participants to find.

Finally, as mentioned earlier, fault localization and correction steps are intertwined in practice. In our model, the Fault Correction (FC) step is initiated after the user catches a prey in the Fault Localization (FL) step.

We begin our discussion of our study results by describing the overall strategies that participants used to hunt for faults and corrections, followed by a description of the fault localization and correction steps. Note that we do not delve into the sensemaking process in this study.

A. Hunting Strategies

Salient versus directed goals. Even though user choices are affected by the salience of possible actions and goals. For example, we found that participants in the Control Group largely seemed to have a salient goal of looking for cues in the output of the program that could lead them to a fault. In contrast, participants in the Experimental Group were more directed in their search for cues for a particular fault (following the order in which bugs were listed in the To-fix list).

First available cue versus easiest prey to catch. When deciding which prey (bug) to hunt for, a participant could select the first scent thought to be relevant to any prey and pursue it, or strategize about and seek the easiest (lowest cost) prey to pursue. We found that participants in the Control Group did not look around for the easiest prey to catch, but rather, began pursuing whichever prey they found the first scent of (failure) in the output. In the Experimental Group, however, because of the presence of the To-fix list, participants began considering bugs in the order in which they appeared in the list.

Persistence versus obsession. While persistence is a virtue, obsession is not a good problem solving strategy. Wickelgren [32] propose that when stuck on a problem, it is better to take a step back and analyze the problem further and not focus on the immediate action. In the Control Group, participants pursued a single bug until they found a fault related to it and fixed it; none switched to another bug when they were stuck. In contrast, participants in the Experimental Group did switch to the next bug when stuck, and then returned to the problematic bug later. For example, participant E.P1 spent 5 minutes on bug B4, but on realizing that she was stuck she commented: *"Ok, I will come back to that later"* and moved on to bug B5. She returned to bug B4 after correcting bugs B5 and B6, during which she was able to gain a better overview of the pipe and its operation. We posit that having a list of bugs to fix helped participants move on to the next task as well as providing an overview of the problem space.

B. Fault Localization

1) Finding Cues

The starting point for fault localization is noting failures (incorrect or unexpected output). As noted above, our participants were provided with sample "correct" outputs. Participants typically executed a pipe and compared their output to the sample provided. These two types of output, along with system-generated error messages, the mashup code, and

debugger output related to a module provided the initial set of environmental cues. Here we discuss cues present only in the unenhanced Yahoo! Pipes, and cues provided by our debugging enhancements.

Clear Cues. In the case of the Control Group, clear cues existed when a bug resulted in a clear Yahoo! error message that could be directly connected to a particular module, allowing participants to quickly navigate to that potentially faulty module. For example, in the case of bug B1, the error message included: "API Key Missing", which caused participants to look for modules that included this scent. When participants navigated to the "`Flickr`" module, they found that it included the keyword "API key"; thus the "API Key Missing" message was a strong enough scent to allow participants to reach the faulty module. All eight participants in the Control Group were able to follow this scent and find the faulty module.

In the case of the Experimental Group, faulty modules were highlighted in orange, which served as clear cues that allowed participants to easily navigate to faulty modules. Thus, for the Experimental group, all cues were clear.

Fuzzy Cues. Participants had difficulty obtaining a strong scent from these cues, or the cues were difficult to understand. For example, participant C.P4 encountered the following error message when he executed the pipe:*"Error fetching [URL]. Response: OK (200). Error: Invalid XML document. Root cause: org.xml.sax.SAXParseException. The markup in the document preceding the root element must be well-formed*". He could not understand what this message meant, as evident from his comment *"What is this?"* and he returned back to the editor (patch) to look for another cue.

Elusive Cues. These cues were those that were difficult for a participant to even unearth. There were two main instances in which cues were elusive. The first instance occurred when a fault was nested – that is, the fault was present in a subpipe. In these cases, Yahoo! Pipes generated an error message when the pipe was executed. However, this message was not thrown in the debugger window if they clicked on the module that contained the subpipe, because errors are not propagated to higher levels. Participant C.P3 found the error message related to the nested module and after several attempts to identify the source of the message commented: *"Where is this [error message] coming from?"*.

The second case in which elusive cues occurred was when the pipe failed "silently", generating no error message. In this case, the only way to identify the failure was by comparing the correct output (provided by us) to the incorrect output (generated by the pipe). In these cases, participants typically tried to understand the functionality of the pipe one module at a time. Without any error message, it was extremely difficult for participants to localize these types of faults. They often picked up incorrect scents ending in dead ends, from which they needed to backtrack multiple times. For example, participant C.P1 could not identify which module was incorrect after multiple attempts and commented: *"I am not sure what I am doing, I am not lost but don't know what I am doing"*.

2) Strategies

We categorize the strategies that participants employed to localize faults as enrichment and navigation strategies.

a) Enrichment Strategies

A primary strategy that users follow when foraging is modifying their environment (or information patch) to optimize affordances for their foraging activity [17]. We observed the following enrichment strategies.

Rearranging modules. Participants created better affordance in the environment by realigning or regrouping modules so that they could better understand the connections between them. For example, participant C.P3 arranged the modules such that (1) all modules were aligned per their data flow path, and (2) all wires, modules and connections were clearly visible without scrolling.

Side-by-side comparison. Participants frequently had to forage for cues between patches; for example, participants had to switch between the Yahoo! Pipes editor and documentation or tutorials. To make it easier to glance through and compare patches, some participants (5 out of 16) rearranged the screen so that both patches could be compared side by side.

b) Navigation Strategies

Finding negative evidence. Participants tried to maintain a mental list of correct modules, as verified by them by checking module output in the debugger window. This helped them create regions of negative evidence that they could safely ignore in their future foraging efforts.

Carving out regions. Participants leveraged the visual affordances of the Yahoo! Pipe environment, which, for one of our tasks, presented three independent data-flow paths. Participants foraged for cues down each path separately. For example participant C.P1 focused on the first path and commented *"it [the module at the end of one independent path] is showing me output that means this part is working fine"*, after which he moved onto the next path.

Backtracking. When participants could identify only weak scents, they followed these hoping they would lead to stronger scents. When this was not the case they needed to backtrack to the original patch (incorrect output) to look for stronger scents. For example, participant C.P1 followed weaker scents such as the color of the wires connecting modules (some wires were blue while others were gray). The participant assumed that wires that were gray contained faults and began to investigate this hypothesis. When he realized that he had picked up a wrong scent, he backtracked to the initial module and resumed looking for cues.

C. Fault Correction

Participants entered the fault correction loop when they attempted to verify that a potential fault they had uncovered was indeed a fault, by finding a correction for it.

1) Finding Cues

Participants obtained cues about potential fixes from error messages, code (e.g., a missing parameter or connection), and help that was provided to them (including contextualized help through the Yahoo! Pipe interface, external web pages, and experiment tutorials). We categorize the cues into three groups.

Clear Cues. In the fault correction context, clear cues allow participants to obtain a clear scent to a potential solution. For

example, bug B1 was caused by the faulty `Flickr` module and caused the error message *"API key missing"* to appear. The `Flickr` module contained the label (*(required) "Flickr API Key"*), with the word "Key" hyperlinked to the Flickr website from where participants could generate the API key. This clear cue enabled most participants (seven in the Control Group and eight in the Experimental Group) to correct this fault.

Despite this clear cue, some participants were hesitant to navigate to a new patch (the Flickr website). For example, when they clicked on the link from the `Flickr` module and found a new website without any links to Yahoo! or any keywords related to the API, they became uncomfortable and closed that page to return to the editor. Thus, participant C.P4, while trying to learn about Flickr and API keys from the help available through Yahoo! Pipes, opened the Flickr website four times before learning enough to generate and use the API key from the website. This showed that in spite of the presence of a clear scent in a patch, participants may still have been wary of leaving that patch. This hesitation increased if the new patch did not resemble their current patch or had no obvious relevance to the cue that led to the scent. Therefore, we posit that understanding a new web page (new patch) and its relevance to the current task by visiting that page incurs a higher cost than remaining in a current page.

Fuzzy Cues. In the fault correction context, fuzzy cues were difficult to use to arrive at a possible solution to a fault. For example, even when participants had found the `Fetch Feed` module to be the faulty module causing incorrect review formats (*"Error fetching [url]. Response: Not Found (404)"*), they did not know how to find the correct solution (the RSS feed) from the error message.

The Experimental Group faced some fuzzy cues too, especially when hints did not directly lead to a solution. However, the hints that were available usually provided cues sufficient to allow participants to arrive at a solution by foraging through patches. For example, participant E.P8, when attempting to correct bug B4, found the hint (*"Please look at page source of website and provide a correct parameter value"*). This message was fuzzy, since he did not know what "page source" meant. He searched for *"How to search a page source code in Firefox"*, and hence was able to access the page source and later correct the fault.

Elusive Cues. In the fault correction context, elusive cues were cues to solutions that were embedded in error messages or results, but were difficult for participants to identify. For example, bugs B2 and B3 contained faults because of incorrect RSS feeds and even though participants identified the correct web page content they were unable to determine how to extract the RSS feed from the website. This shows that although the participants might reach a patch containing a potential fix, they could not always consume the prey due to a diet constraint (a specific way to extract the RSS feed). For example, when attempting to correct bug B3, participant C.P5 was able to obtain the list of "top 10 movies" on the Rotten Tomatoes movie review website, but could not find the RSS feed for them. He spent 18.5 minutes attempting to find the RSS feed without success.

In the Experimental Group, despite most cues being made explicit through hints, two participants still faced problems because of diet constraints. For example, participant E.P3 investigated 41 patches to identify the RSS feed needed to correct bug B3.

2) Strategies

We categorize the strategies that participants used to correct faults as enrichment, navigation, and verification strategies.

a) Enrichment Strategies

Search. When looking for information, participants often searched for possible cues to solutions, and aggregated them on a single page from which they foraged further. For example, while attempting to correct bugs B2 and B3, which required participants to identify the correct RSS feed, depending on their expertise in RSS, participants searched (Google) for information on them and on how to extract them from web pages. The success of this strategy depended, however, on the appropriateness of search terms. We found that participants in the Control Group had trouble seeing that the incorrect output produced by bugs B2 and B3 was due to an incorrect RSS feed. For example, participant C.P5 searched for different terms related to movie reviews 19 times without success. In the Experimental Group, the hints mentioned that *"To find a similar website you need to search for RSS feeds that provide similar contents"*, and due to this, most participants in this group began their search by studying how to use RSS feeds. Even the one participant in the Experimental Group (E.P3) who was unable to correct the bug knew that the solution involved finding the correct RSS feed as evident from his comment: *"Why it [the URL he found and pasted in the parameter box] is not RSS I am not sure...I will Google and see"*.

Temporary collection. To reduce the effort involved in moving between patches or to keep track of information, participants in both treatment groups used external applications such as Notepad or MS Word to keep a record of URLs that they had already visited or text that they wanted to investigate. For example, participant C.P7 copied the original URL from the parameter field to Notepad before making changes to the pipe so that she could undo his changes if necessary. Similarly, participant C.P3 used MS Word, but in this case he copied the tags from the module he was investigating so that he could compare the tags in the module to those in the page source of the website.

Side-by-side comparison. As during fault localization, participants also arranged windows to allow side-by-side comparison when foraging for solutions. For example, participant C.P3 (as just explained) compared the tags in the Word document and page source side by side.

b) Navigation Strategies

Skimming through patches. When looking for solutions, participants first quickly skimmed through patches (web sites) looking for cues that stood out. For example, when working on the first task, which involved finding the top 10 movies, participant E.P4 skimmed through the web pages and commented: *"I am looking for top 10"*.

Finding negative evidence. Users often remembered patches they had already visited and the type of information available in them. If these patches appeared again as a result of a new search, participants did not spend further time in them.

For example, E.P2 opened the webpage related to "Top box office list" and closed it immediately when she realized that she had already visited it.

Backtracking. When participants picked up an incorrect scent and navigated to a patch in which they could not find a solution, they backtracked to the Yahoo! Pipes editor (the location of the fault) and began looking for cues in the output or error messages. Participants in the Experimental Group backtracked fewer times than those in the Control Group, because the hints provided by the implementation allowed them to create stronger scents. We also found that while some participants remembered the patches they had visited, many participants reapplied the same solution multiple times. For example, participant E.P3 visited the same website nine times when looking for the RSS feed. He repeated steps because he did not know the correct solution and hoped that he could reach the correct solution by trial and error.

c) Verification Strategies

After finding a potential correction for a fault, participants needed to verify that the correction was appropriate. As noted earlier, we found that participants preferred to employ less effort and preferred to remain in the same patch. While in the Control Group, participants needed to rerun the pipe and compare the output (presented in a new web page) to the given solution, participants in the Experimental Group simply used the "Find Errors" widget and let the tool determine whether the error had been corrected (and removed from the To-fix list); this let them remain in their current patch.

V. Discussion

A. Implications for Theory

We have extended the debugging model from the perspective of IFT as originally proposed by Lawrance et al. [17] In our model, we specifically separate the Fault Localization (FL) and Fault Correction (FC) steps because the information that is foraged for during these steps differs and different foraging strategies are used. We found that the overall steps in debugging from the IFT perspective involved finding cues, navigating to the correct patch, catching the prey, and verifying the prey. The fault correction step is part of the verification stage in fault localization; once users find a potential fault, they need to verify that they have correctly localized the fault by finding a solution and applying it. If the solution is incorrect, it might be that the solution was incorrect (and users can stay in the FC loop) or they may find that they were mistaken about the fault and need to go back to the FL loop (Figure 2). This process continues until users correct the fault.

We also categorized the different types of cues that are present in a mashup environment into clear, fuzzy, and elusive and discussed participant behavior when they face issues related to these cues. This refinement of cue types allows designers to understand how to *strengthen* cues (discussed later) and allows researchers to focus on behaviors relative to these cue differences.

We also articulated the different strategies that users can employ when debugging. We began by noting the overall hunting strategies we saw participants use. We noted that these strategies are highly dependent on user preference and to an extent prompted by the environment (e.g., a To-fix list promotes a "sleep on the problem" strategy). We noted slightly different strategies when participants foraged for cues and navigated through patches when localizing a fault versus when looking for solution. We believe these differences arose because in the former case participants were more focused on staying within the Yahoo! Pipes editor, whereas in the latter case they needed to peruse more web content to find a way to correct the fault. Overall, we saw that participants preferred to stay within the patch (the editor) to the extent possible. Finally, we found that diet constraints (where information had to be formatted in a specific manner) affected debugging capacities. We found that participants could easily forage to the right web content (reviews by Rotten Tomatoes) but had a significant difficulty finding the RSS feed for the reviews. This is a large difference between foraging for information on the web and foraging for information when debugging a program. Oftentimes these diet restrictions (only RSS feed allowed in the module) and the diet format available in web content (where the RSS feed is located in the website) is not clear, especially to end users.

B. Implications for Design

We now discuss ways in which our findings can help improve interface design for black-box oriented or visual programming environments, especially those that are geared towards end users.

1) Strengthening scents through cue clarity: We found participants to have the greatest difficulties when cues were fuzzy or elusive. Of course, environment designers should strive to use simple, clear error messages that describe the problem and provide hints to help solve it instead of simply providing an error code or error message with technical jargon. This is particularly important for end users who are not well versed in debugging techniques. In cases where it is not possible to directly point to a solution, we found that providing hints to the reason for a problem (e.g., wrong format of URL or missing URL) or providing pointers to web development tools (e.g., Firebug which allows inspection of web page elements) helped enable participants to forage for a correct solution. The black box nature of Yahoo! pipes further complicates the process, because users can no longer see the underlying code or the functionality of a module. Faults in modules that fail silently are even more difficult to locate and correct. We posit that visual language environments should (1) ensure that there are no silent failures and (2) provide stack traces of failures so users can at least locate faulty modules through keyword searches. In the Experimental Group, when faults were visually marked, participants quickly identified faulty modules, even those that were nested. Similarly, when error messages had distinct cues (e.g., *"API Key missing"*) participants in the Control Group were able to easily search for these keywords and locate the faulty module.

2) Make diet constraints explicit: The fault that proved the most difficult to resolve involved identifying the RSS feed for reviews. In this case, while most participants were able to arrive to the right content, getting the content into the right format proved difficult. This was a big problem for the Control Group since participants even did not know that they were supposed to look for RSS feeds from the error message.

3) Multi-context views: We found that a majority of participants needed to alternately switch between the editor and other patches. Many users organized their windows to be able to perform side by side comparisons. Environments should thus support multi-context views, something proposed by Ko et al. [13] for professional developers, where users can view different dimensions of the code space (e.g., see the error output and the code canvas at the same time) and manipulate their placements as they prefer.

4) To-fix list: We found that having a To-fix list helped participants focus on the faults to fix and at the same time not get stuck on a problem. The importance of support for to-do lists has already been researched [11], [26]. We populated the To-fix list based on the order in which errors occurred, and found that participants did not perform any cost analysis and followed the list. A better option may be to populate the list based on estimates of the difficulty of problems so that users are guided to resolve the easier problems first.

5) Fine-grained backtracking: We found that participants needed to backtrack and revert their changes. To help with this process they often copied and pasted parameter values onto a notepad so that they did not lose their initial state. Environments should facilitate this process by providing backtracking support, a concept that has already been proposed for end users [2], [14]. We posit that the backtracking support should be fine-grained and (in the case of black box programming) available at the level of modules.

VI. Conclusions

We have presented data from a study of end-user foraging behavior when debugging mashups. To gain a better understanding of that behavior we modeled it using information foraging theory. This model allows us to obtain a refined understanding of the debugging behavior by separately focusing on the localization and correction of faults. Further, we categorized the different types of cues and strategies that users could employ when debugging mashups. Our contributions include: (1) extending IFT to model end-user debugging behavior, (2) classifying types of cues, (3) identifying debugging strategies, and (4) discovering several implications for the design of end-user programming environments to facilitate end-user debugging.

Studies have shown that there are differences behaviors of professional and end-user programs, and differences in programming behaviors across genders. Therefore, we intend to perform further studies to investigate the effect of these factors on debugging behavior.

References

[1] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck. A debugging perspective on end-user mashup programming. In *VLHCC*, pages 149–156, 2010.

[2] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu. End-user mashup programming: Through the design lens. In *CHI*, pages 1009–1018, 2010.

[3] K. Charmaz. *Constructing Grounded Theory: A Practical Guide through Qualitative Analysis*. SAGE Publishing, 2006.

[4] E. H. Chi, P. Pirolli, K. Chen, and J. Pitkow. Using information scent to model user information needs and actions and the web. In *CHI*, pages 490–497, 2001.

[5] Deri Pipes: http://pipes.deri.org/.

[6] D. Fetterly, M. Manasse, M. Najork, and J. Wiener. A large-scale study of the evolution of web pages. In *WWW*, pages 669–678, 2003.

[7] S. Fitzgerald, R. McCauley, B. Hanks, L. Murphy, B. Simon, and C. Zander. Debugging from the student perspective. *TOE*, 53:390–396, 2010.

[8] W.-T. Fu and P. Pirolli. SNIF-ACT: A cognitive model of user navigation on the world wide web. *HCI*, 22(4):355–412, 2007.

[9] V. Grigoreanu, J. Brundage, E. Bahna, M. M. Burnett, P. ElRif, and J. Snover. Males' and females' script debugging strategies. In *IS-EUD*, pages 205–224, 2009.

[10] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan. End-user debugging strategies: A sensemaking perspective. *TOCHI*, 19(1):5:1–5:28, 2012.

[11] V. I. Grigoreanu, M. M. Burnett, and G. G. Robertson. A strategy-centric approach to the design of end-user debugging tools. In *CHI*, pages 713–722, 2010.

[12] IBM Mashup Maker: http://ibm.com/software/info/mashup-center/.

[13] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *TSE*, 32(12):971–987, 2006.

[14] S. K. Kuttal, A. Sarma, and G. Rothermel. On the benefits of providing versioning support for end-users: An empirical study. In *Technical Report TR-UNL-CSE-2012-0008*, 2012.

[15] S. K. Kuttal, A. Sarma, and G. Rothermel. Debugging support for end-user mashup programming. In *CHI*, pages 1609–1618, 2013.

[16] J. Lawrance, R. Bellamy, and M. Burnett. Scents in programs: Does information foraging theory apply to program maintenance? In *VLHCC*, pages 15–22, 2007.

[17] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. Fleming. How programmers debug, revisited: An information foraging theory perspective. *TSE*, 39(2):197–215, 2013.

[18] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, and C. Swart. Reactive information foraging for evolving goals. In *CHI*, pages 25–34, 2010.

[19] C. H. Lewis. Using the "Thinking Aloud" method in cognitive interface design. RC 9265, IBM, 1982.

[20] Internet Usage Statistics: http://www.internetworldstats.com/stats.htm.

[21] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *CHI*, pages 249–256, 1990.

[22] N. Niu, A. Mahmoud, and G. Bradshaw. Information foraging as a foundation for code navigation. In *ICSE (NIER)*, pages 816–819, 2011.

[23] P. Pirolli. Computational models of information scent-following in a very large browsable text collection. In *CHI*, pages 3–10, 1997.

[24] P. Pirolli and S. Card. Information foraging. *Psychological Review*, 106:643–675, 1999.

[25] P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *IA*, pages 2–4, 2005.

[26] J. E. Robbins. Software architecture design from the perspective of human cognitive needs. In *California Software Symposium*, 1996.

[27] M. B. Rosson and J. M. Carroll. The reuse of uses in smalltalk programming. *TOCHI*, 3:219–253, 1996.

[28] B. Shneiderman. Designing computer system messages. *CACM*, 25:610–611, 1982.

[29] J. M. Spool, C. Perfetti, and B. David. *Designing for the Scent of Information*. User Interface Engineering, 2004.

[30] D. W. Stephens and J. R. Krebs. *Foraging Theory*. Princeton University Press, 1986.

[31] L.-H. Teo, B. John, and M. Blackmon. Cogtool-explorer: a model of goal-directed user exploration that considers information layout. In *CHI*, pages 2479–2488, 2012.

[32] W. A. Wickelgren. *How to Solve Problems: Elements of a Theory of Problems and Problem Solving*. W. H. Freeman, first edition, 1974.

[33] Yahoo! Pipes: http://pipes.yahoo.com/pipes/.

[34] N. Zang and M. Rosson. What's in a mashup? And why? Studying the perceptions of web-active end users. In *VLHCC*, pages 31–38, 2008.