How End-User Programmers Debug Visual Web-Based Programs: An Information Foraging Theory Perspective [1]

Sandeep Kaur Kuttal

University of Tulsa

Anita Sarma

Oregon State University

Margaret Burnett

Oregon State University

Gregg Rothermel

North Carolina State University

Ian Koeppe

University of Nebraska at Omaha

Brooke Shepherd

University of Tulsa

Abstract

Web-active end-user programmers squander much of their time foraging for bugs and related information in mashup programming environments as well as on the web. To analyze this foraging behavior while debugging, we utilize an Information Foraging Theory perspective. Information Foraging Theory models the human (predator) behavior to forage for specific information (prey) in the webpages or programming IDEs (patches) by following the information features (cues) in the environment. We qualitatively studied the debugging behavior of 16 web-active end users. Our results show that end-user programmers spend substantial amounts (73%) of their time just foraging. Further, our study reveals new cue types and foraging strategies framed in terms of Information Foraging Theory, and it uncovers which of these helped end-user programmers succeed in their debugging efforts.

Keywords: Information Foraging Theory; End-user programming, end-user software engineering, Visual programming language; Debugging.

1. Introduction

As the Internet grows increasingly ubiquitous, many web-active "end users" (non-professional programmers) utilize the Internet as a vital part of their day-to-day lives. These users lack programming expertise [67], yet many create applications to sift through the content-rich web in order to access its information efficiently and effectively.

This can be challenging due to the Internet's ever-increasing size. Creating applications that optimize information access and utilize a Visually Distributed Language for the Web (VDLW) involves aggregating heterogeneous web APIs distributed across different platforms. Challenges in building such programs include software and hardware

[1]This is a revised and extended version of [37] and [38]. This paper replaces previous results, except for comparison purposes, with an extensive new analysis of the data from a new perspective.

dependencies not represented in traditional programming environments. Examples of such applications are the Internet of Things (Zenodys[2], Losant[3]), mobile development environments (Dojo[4], IBM Mobile Portal Accelerator[5]), cloud-mobile-web (LondonTube [58]), and web mashups.

Web mashups provide one approach for "cobbling together" various sources of data, functionality, and forms of presentation to create new services [67]. Professional and end-user programmers create these applications by combining web services and processing their outputs in various ways. Unfortunately, mashup programming is prone to bugs, especially those resulting from changes in a source's data [37]. Mashups' dependence on the complex ecosystem of the web, composed of evolving heterogeneous formats, services, protocols, standards, and languages [16], leaves them vulnerable to unexpected behaviors. A further complication arises from the black box nature of VDLWs, which obscures the code and thus the sources of bugs from the user [9]. In addition, end users prefer to learn from examples and code reuse, but this programming practice often propagates bugs to new mashups [37, 63]. The prevalence of bugs in mashups, the tendency for users to reuse mashups, and black box code abstraction create problems for mashup dependability.

Thus, end users spend a significant portion of their time debugging mashups [10] and "foraging" for information [24]. The problems encountered in mashup programming are also found in other VDLW programming environments. End users who encounter these unreliable mashups may struggle to debug them, as they must forage through a mashup program to find bugs and then forage through the web in order to identify and correct faults.

Foraging behavior can be understood more easily by utilizing Information Foraging Theory (IFT). In IFT, a predator (e.g., end-user programmer) forages for prey (e.g., bugs, while finding or fixing) by following cues (e.g., label on links) in patches (e.g., web pages, IDEs). IFT has been studied and applied in connection to the process of foraging by users on the web [13, 21, 51], to professional programmers navigating through programs [47], and to debugging [43, 41, 42, 48, 49, 50].

This paper is an extension and presents new analysis and results from our previous studies [37, 38]. In [37], we analyzed the Yahoo! Pipes repository, found the types of bugs that existed, and created an extension to Yahoo! Pipes to find those bugs. Based on each bug we designed a To-Fix list and hints for each error to support end-user programmers. We evaluated Yahoo! Pipes and our developed extension through a controlled lab study. The study quantitatively compared the performance of end-user programmers. In [38], we conducted a qualitative analysis of the study results using Information Foraging Theory. We created a debugging model based on end user's debugging behavior and found three cues and strategies related to navigation, enrichment, and hunting. In this paper, we present an extension of our previous studies by performing a different analysis. We reanalyzed the transcripts in depth using Information Foraging Theory, and, by using grounded theory, we identified a new set of cues and their effects on the performance of end-user programmers' debugging behavior.

We make the following contributions:

- We investigate, from an IFT perspective, the behaviors of end-user programmers using mashups in a visual programming environment.
- We identify and elaborate on a hierarchy of cues followed by end-user programmers in mashup programming environments, that can be generalized to visual programming IDEs, distributed programming environments, and the web.
- We compare the amount of time end-user programmers spend foraging while debugging with the time professional programmers' spend foraging while debugging.
- We identify the types of foraging cues that help end users the most while debugging.

This article is structured as follows. Section 2 describes the background on visually distributed languages, IFT, and Yahoo! Pipes. Section 3 describes the debugging extension to Yahoo! Pipes. Section 4 describes our empirical study design, followed by Section 5 on Results and Section 6 on Threats to Validity. Section 8 discusses related work, and Section 9 concludes.

[2]https://www.zenodys.com/
[3]https://www.losant.com/
[4]https://dojotoolkit.org/reference-guide/1.10/dojox/mobile.html
[5]http://www-03.ibm.com/software/products/en/ibmmobiportacce

2. Background

2.1. Visually Distributed Languages for the Web

Visually Distributed Languages for the Web (VDLW) are languages that allow web applications to be created using visual languages in distributed system environments.

Distributed programming: Distributed programming environments are similar to web mashup environments in that they are both complex ecosystems of heterogeneous formats, services, protocols, standards and languages. One example of a distributed programming environment is Dojo mobile[6], which allows users to create mobile applications to facilitate reuse of application code across devices with a simple style sheet change. These applications allow users to create new native smart phone builders to support build-once-deploy-anywhere systems, e.g., on portals, web app servers, mobile devices, kiosks, and webTVs. Another example is the IBM Mobile Portal Accelerator[7], which allows mobile web site developers to support rendering on over 8000 types of basic to smart devices utilizing over 600 attributes. Similarly, API aggregation tools create applications either by automation or creative production using If This Then That (IFTTT)[8], Zapier[9], and RunMyProcess[10]. Hence, these environments are very similar to web mashup environments, except that these environments build (stand-alone) apps, not web pages.

Visual programming: Visual programming environment languages like Yahoo! Pipes aim to allow easy creation of programs by abstracting the underlying code as black boxes. Most visual programming languages like App Inventor[11], LabVIEW[12], LondonTube [58], IFTTT[8], and IBM Mobile Portal Accelerator[7] could integrate the debugging features discussed in this paper for fault localization and correction into their programming environments.

2.2. Information Foraging Theory

Pirolli and Card developed Information Foraging Theory (IFT) in order to understand the various ways humans search for information [52]. IFT, based on optimal foraging theory, finds its inspiration in the similarities between how animals forage for food and how humans search for information. In IFT, humans are "predators" searching for "prey," or information. Like animal predators, IFT predators have a specific "diet," i.e., the ability to use certain information is dependent on the predator's knowledge. Predators follow "cues" within the environment which lead them to indicators of prey, or "scents." Prey and scents can be found in "patches" which may contain "information features" that may serve as cues for other prey. Occasionally, predators modify patches via "enrichment" to increase the chances of prey acquisition. Some examples of these terms in the context of our study can be found in Table 1.

IFT has improved the understanding of user interaction with the web, especially by revealing design principles for web sites and user interfaces. Navigational models identified by IFT have enhanced website usability by predicting users' navigational patterns [13, 21, 51]. Engineering tools for professional developers have also been analyzed using IFT to model navigational and debugging behavior. Using IFT, Niu et al. have modeled professional programmers' navigational behavior [47], and Lawrance et al. and Piorkowski et al. have modeled the evolving goals of a programmer while debugging [43, 41, 42, 48, 49, 50].

2.3. The Yahoo! Pipes Mashup Environment

Yahoo! Pipes was a VDLW programming environment popular with professional and end-user programmers alike [28]; it was one of the most popular mashup programming environments at the time we conducted our study, but was discontinued in September 2015.

As seen in Figure 1, the Yahoo! Pipes programming environment consisted of three major components: the Canvas, Library, and Debugger Window. (The area in Figure 1 surrounded by dashed lines contains our debugging tool, which is described in Section 3.) The central area composed the Canvas in which users placed Modules from the Library. The Library, located to the left of the Canvas, contained modules categorized by functionality. Users organized the data flow of the Modules by connecting them with Wires. The Debugger Window beneath the Canvas displayed output from specific modules as well as the final output of the program, which was known as a Pipe.

[6]https://dojotoolkit.org/reference-guide/1.10/dojox/mobile.html
[7]http://www-03.ibm.com/software/products/en/ibmmobiportacce
[8]https://ifttt.com/
[9]https://zapier.com/
[10]https://www.runmyprocess.com/
[11]ai2.appinventor.mit.edu
[12]http://www.ni.com/labview/

Table 1: IFT Terminologies from the Yahoo! Pipes Perspective [37].

IFT Terminologies	**Definitions**	**Fault Finding (Examples)**	**Fault Fixing (Examples)**
Prey	Potential fault during fault finding; Potential fix during fault correction	Finding Fetch Feed module that contains a bug B2 (web site 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, web-pages and displays that may contain the prey [18]	Yahoo! Pipes Editor, Help documents, help examples	Web pages
Information Feature	Words, links, error messages, or highlighted objects that suggest scent relative to prey	API Key Missing error message for bug B1	"Error fetching [url]. Response: Not Found (404)"
Cues	Proximal links to patches	"about this module" link to the example code related to specific module	"Key" link to the Flickr page to collect the API key
Navigate	Navigation by users through patches	To find bug B2 the user navigated through Yahoo! Pipes editor to external web site	To correct bug B2 participant navigated to various web sites to find the required url

Inputs to pipes were HTML, XML, JSON, KML, RSS feeds and other formats, and outputs were RSS, JSON, KML and other formats. Inputs and outputs between modules were primarily RSS feeds. RSS feeds consisted of item "Parameters" and descriptions. Yahoo! Pipes modules provided manipulation actions that could be executed on these RSS feed parameters. Yahoo! Pipes also allowed users to define various data types such as url, text, number, and date-time

3. Debugging Support for Yahoo! Pipes: an IFT Perspective

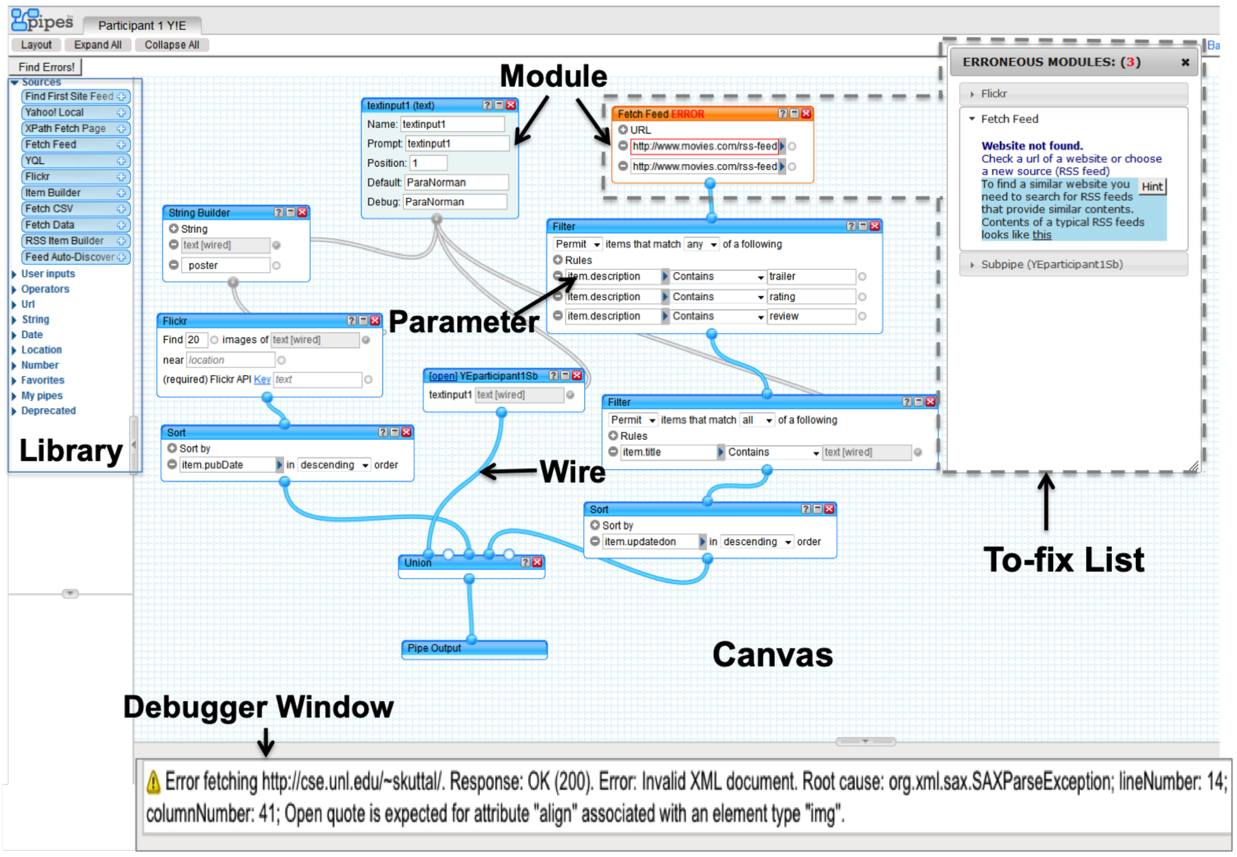
Drawing on IFT, we developed a tool to integrate into the Yahoo! Pipes environment. The tool contained an Anomaly Detector that detected bugs identified by Yahoo! Pipes and bugs occurring silently whenever a user saved or executed the pipe. A To-fix list of bugs and a set of straightforward Error Messages constituted the Tool's user interface. The To-fix list enriched the development environment by providing additional cues for the user to follow. By simplifying the semantics of the error messages, we adapted them to fit into the diet of end users and improved the quality of the cues in the error message patch. We largely followed Nielsen's Heuristics [46] in designing the interface with the main goal of reducing cognitive load on users [59], and Schneiderman's guidelines [60] for designing error messages. Figure 1 shows our user interface extensions.

3.1. To-fix List of Bugs

To facilitate bug localization, we wanted to present information contextualized to our users' current task. According to prior studies [25], end users consistently utilize to-fix lists to reduce cognitive load, and this strategy was employed by users regardless of their personal differences or debugging habits. Professional environments, such as Eclipse IDE[13], provide to-fix lists (stack traces of failures) within their UI. However, this functionality is not commonly available in many end-user environments, such as Yahoo! Pipes or Microsoft Excel. Thus, our UI included a To-fix list populated with information on bugs and their properties. The To-fix list can be seen in Figure 2.

The To-fix list, which is overlaid on the top, right-hand side of the Canvas, displays erroneous modules from top to bottom and from left to right. In this way, users can view the pipe and the list of bugs simultaneously. We believe

[13]An open-source Integrated Development Environment (IDE) for Java that is available for download at: `http://www.eclipse.org/downloads`.



ERRONEOUS MODULES: (3)

Flickr

Fetch Feed

Website not found.
Check a url of a website or choose a new source (RSS feed)
To find a similar website you need to search for RSS feeds that provide similar contents. Contents of a typical RSS feeds looks like this

Subpipe (YEparticipant1Sb)

```
Error fetching http://cse.unl.edu/~skuttal/. Response: OK (200). Error: Invalid XML document. Root cause: org.xml.sax.SAXParseException; lineNumber: 14; columnNumber: 41; Open quote is expected for attribute "align" associated with an element type "img".
```

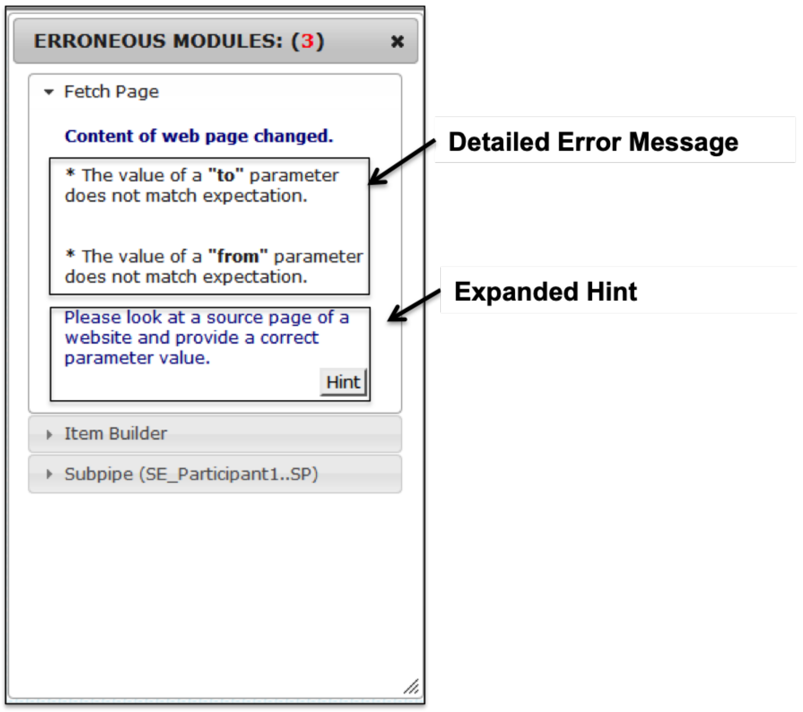
Figure 1: Yahoo! Pipes with debugging support with the Debugger Window showing a specific error. The (dotted) marked areas are our extension to Yahoo! Pipes interface.

users may prefer this approach since spatially following the data flow of a program is a common debugging strategy [25]. An alternative that could reduce the likelihood that the user is overwhelmed with too many alternatives would be to group bugs by type, which would enable them to focus on bugs of the same type [15].

The To-fix list represents a patch with cues organized on it. Whenever a user saves the pipe, executes the pipe, or clicks on the "Find Error" button on the left-hand side of the canvas, the To-fix list is populated by an Anomaly Detector with information on bugs that need to be resolved and their properties. In IFT terms, "Find Error" is a cue to activate anomaly detection. The erroneous module associated with a bug in the list is highlighted in orange whenever a user clicks on a bug in the list, and the selected module's potentially incorrect parameters are marked in red. In IFT terms, highlighted modules and colored parameters are cues. The same follows in reverse: when a user clicks on a faulty module then the bug or bugs in the To-fix list related to that module are expanded. After a bug is resolved, it is removed from the list. Hence, only relevant cues are kept while irrelevant cues are removed from the To-fix list patch. In this way, we reduce the cognitive load on the user by directly associating the bug in the To-fix list with the faulty module.

3.2. *Error Messages*

We followed Nielsen's Heuristics [46] and Schneiderman's guidelines [60] to design error messages. Error messages were designed to be clear and concise, to use plain language, and to help users recognize, diagnose, and recover from bugs. The error messages were formulated for list of errors for Yahoo! Pipes (a comprehensive list types of errors for Yahoo! Pipes can be found in [37]). Yahoo! Pipes only detected errors related to web pages that no longer existed, had any page access restrictions or server error. For example, a Yahoo! Pipes error message of the form:


ERRONEOUS MODULES: (3)

Fetch Page

Content of web page changed.

* The value of a "to" parameter does not match expectation.

* The value of a "from" parameter does not match expectation.

Please look at a source page of a website and provide a correct parameter value.

Figure 2: To-fix List. The list shows the number and name of modules containing errors. Error message, detailed error message and Hints (expanded on a need basis).

“Error [url] Response code. Error: [Invalid XML document.] Root cause: org.xml.sax.SAXParseException: [root element must be well-formed]”, was translated to “Website does not contain RSS feed.” Another example was “Error Fetching [url]. Response: Not Found (404)” translated to “Website not found.”. Yet another example is shown in Figure 1 (bottom panel). Figure 2 shows how these error messages were modified to show to end users.

3.3. Feedback

We also provided incremental assistance through a “Hint” button in order to help users resolve bugs. Users can adjust the amount of information visible in the hint by expanding the hint to provide additional details (refer to Figure 2). The template for Hints were formulated for every error message of Yahoo! Pipes and rendered for the current error. Hints for the example in Section 3.2 provided hints to find web sites for missing RSS feeds and external documentation describing the required structure of RSS feeds. Hints for other bugs provided references to third party web applications, such as FireBug[14], to enable the user to inspect elements of the web page. In IFT terms, the Hint button is another cue that provides contextualized help and adjusts the cue to be included in the user’s diet. The hints not only catered to users with different skill levels but also provided users with constructive steps to create solutions for bugs.

4. Empirical Study

To investigate our tool design and to understand the debugging behavior of end users using VDLWs from an IFT perspective, we conducted a new analysis of our previous studies of end users debugging Yahoo! Pipes [37, 38]. The goal of this new analysis was to provide additional insights into IFT regarding Visually Distributed Languages for the Web and end-user foraging behavior while debugging.

[14]FireBug was discontinued in 2017, but many of its features have been incorporated into the built-in developer tools in Firefox. See `https://getfirebug.com/` for additional details.

4.1. Participants

We invited students at the University of Nebraska-Lincoln with experience with at least one web language, but no background in computer science beyond their major's requirements, to participate in our study for a gratuity of \$20. We selected 16 participants from a variety of fields including mathematics, finance, engineering, and the natural and social sciences. We categorized participants using stratified sampling according to their gender as well as their experience with the web, programming languages, and Yahoo! Pipes. Based on these categories, we divided the participants into two groups, control and experimental, with five male and three female participants per group. The two participants with prior experience with Yahoo! Pipes were assigned to different groups, one as control participant 3 (C.P3) and the other as experimental participant 2 (E.P2). Paired t-tests on questionnaire data revealed no statistically significant differences (p=0.731) between the groups based on grade point average, experience with Yahoo! Pipes, experience with web languages, or experience in other forms of programming.

4.2. Environment

Our study focused on mashup programming using Yahoo! Pipes, and on our extension to that environment, providing debugging help to end users, as described in Section 2.3 and Section 3.

4.3. Procedure

We employed a between-subjects study design [65] to avoid learning effects. One half of the participants, the experimental group, completed two debugging tasks with our extension to Yahoo! Pipes, and the other half, the control group, completed the tasks with the ordinary Yahoo! Pipes interface. Since the same participant was never exposed to the same independent variable (the same system) precludes a participant from learning on one system and then carrying that learning to the other system. Because we needed to obtain insights into the participants' barriers, problems, and thought processes as they performed their tasks, we employed a think-aloud protocol [44] by asking them to vocalize their thought processes and feelings as they completed each task. We administered our study to each participant on an individual basis in a usability lab at the University of Nebraska-Lincoln.

Participants were asked to complete a brief self-efficacy questionnaire (formulated as a Likert scale between 1 and 10) at the start of each task [14]. A ten-minute tutorial before each task on Yahoo! Pipes, which described how to create pipes and the functionality of various modules, included a short video describing think-aloud studies in order for users to understand the process. The experimental group also received instructions on how to invoke our debugging support tool (see Section 3). Having completed the tutorial, we asked participants to create a sample pipe to familiarize themselves with Yahoo! Pipes via hands-on training. We began the experiment only after users told us that they were comfortable using the environment.

Participants completed two tasks (see Section 4.4) designed to understand our participants' behavior when the Yahoo! Pipes environment provided feedback or when feedback was absent. The tasks were counterbalanced to mitigate possible learning effects. Each session was audio recorded, and the participants' on-screen interactions were logged using a screen capture system Morae [15]. The participants were allowed as much time as they wanted to complete each task, which took participants 50 minutes on average and a maximum of 80 minutes. We conducted interviews of the participants once they had completed the tasks to collect feedback or any additional thoughts from them.

4.4. Tasks

For the first task, Task Y!E, we required participants to debug a pipe seeded with bugs for which Yahoo! Pipes provides error messages. In the second task, the pipe contained bugs with no Yahoo! Pipes error messages ("silent" errors). We counterbalanced the tasks to compensate for possible learning effects. Three bugs were seeded into each pipe (see Table 2 for details). All bugs were located on separate data flow paths to avoid interaction effects. We also included one bug related to subpipes in each pipe to help us study the effects of nested errors.

We told participants that they had been given pipes that were not working correctly and were required to make them work correctly (that is, to find the bugs and correct them). To guide them, we gave them specifications of the pipes and of the output each pipe was intended to produce.

[15]http://www.techsmith.com/morae.asp.

Table 2: Details on Seeded Bugs [37]

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

Table 3: Cue-types Used in Our Study, Taken From the Literature [48]. * are the New Codes we Observed in our Study.

Codes	Cues
Output-Inspired	Cues based on how output looks in the Debugger Window or running a pipe
Reusability	Cues that allow the participant to reuse elements from either an example or other places in their code (pipe)
Understandability	Cues based on a participant seeking to understand something about the pipe
Source-Code Content-Inspired	Cues based on looking at source code
Source-Code Appearance-Inspired	Cues based on the appearance of the code or development environment
Domain Text	Cues based on words or phrases related to the task
Documentation-Inspired (External)*	Cues based on content in documentation outside the Yahoo! Pipes editor, usually on the Web
Documentation-Inspired (Internal)*	Cues based on content in documentation within the Yahoo! Pipes editor
	Operations
Comparison*	Performing a comparison based on two or more patches
Backtrack (Code)*	Returning the code to a previous state
Backtrack (Web)*	Returning to a web site after visiting a different patch

For Task Y!E, we required participants to fix a pipe which was supposed to display the following: (1) a list of the top 10 rated moves according to rottentomatoes.com and their ratings in descending order, (2) a poster of the selected movie from Flickr.com, and (3) a review of the movie. A previous study of a large corpus of pipes [37] found that two of the most common bugs in pipes were “Link” bugs and “Deprecated module” bugs, which were found in 31.6% and 22.9% of pipes, respectively. Link bugs occur when a web page no longer exists, web page access is restricted, or a server error is encountered. Deprecated Module bugs indicate a module is no longer supported within the Yahoo! Pipes environment. Thus, the first bug seeded into the pipe was a Link bug, t he second was a Deprecated module bug, and the third was a Link bug embedded in a subpipe.

For Task SE, we required participants to fix a pipe which was supposed to display the following: (1) a list of theaters in a given area, (2) a list of movies in each theater with their show times, and (3) trailers of the top 10 movies (again, based on rottentomatoes.com). Based on the study of the corpus of the pipes [37], the two most prominent silent bugs were Missing: Content and Missing: Parameter bugs, which were found in 11.3% of pipes and 60.5% of pipes, respectively. Missing: Content bugs occur when web page contents are missing, and Missing: Parameter bugs occur when a parameter value in one of the Yahoo! Pipes modules is missing. We seeded one of each of these two bugs into the pipe, and we also included a Missing: Parameter bug in a subpipe.

4.5. Analysis Methodology

We transcribed all actions and verbalizations performed by our participants for analysis. We used a baseline code set inspired by previous research [42, 48, 51]. However, our analysis revealed that we needed new codes in addition to the existing ones to fully describe our participants' behavior. Thus, we created new codes by content analysis after analyzing the participants' transcripts over several iterations. We also noted cases where differences between control and experimental group participants' activities shed light on debugging behavior.

We coded the transcripts of participants' think-aloud videos with two code sets. One code set (Table 3) is based on the cue types the participants used and the type of operations they performed. The second code set is in the form of a hierarchy based on how easily a participant could predict where the cue would lead (Figure 3 and Table 4). We

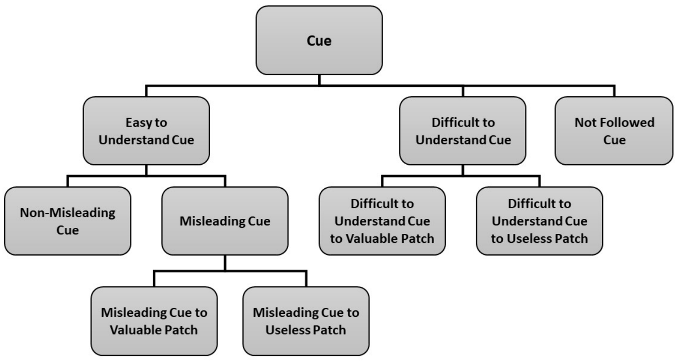

Cue
Easy to Understand Cue
Difficult to Understand Cue
Not Followed Cue
Non-Misleading Cue
Misleading Cue
Difficult to Understand Cue to Valuable Patch
Difficult to Understand Cue to Useless Patch
Misleading Cue to Valuable Patch
Misleading Cue to Useless Patch

Figure 3: Cue hierarchy

then segmented the transcripts into segments whose lengths depended on a participant's actions or words, allowing multiple codes per segment. Two researchers independently coded 20% of the transcripts. We then calculated inter-rater reliability using the Jaccard measure [27], resulting in a rate of agreement of 85% on 20% of the data. Given this high rate of inter-rater reliability, the two researchers split up the coding of the remaining code set.

5. Results

5.1. Behavior of End-User Programmers While Debugging

We observed that our participants' behavior depended on whether they were foraging for faults, were foraging for solutions to the faults, or were fixing the faults. The control group had to find and fix the bugs (refer to Table 5), whereas the experimental group had access to our tool and only needed to fix the bug (refer to Table 6).

5.1.1. Foraging While Finding Faults

The control participants spent, on average, 73% of their time foraging in order to locate the errors over the 1.5 hours of the user study (Table 7). Past studies of professional programmers' foraging behavior found that they tend to spend up to 50% of their time foraging [50] to debug or to navigate between code fragments [32]. Our results coincide with the findings from Grigoreanu et al. [24], in which end-user participants spent two-thirds of their time in the foraging loop of sensemaking while debugging. Hence, our participants spent higher amounts of time foraging compared to professional programmers in past studies.

To understand the behavior of end-user programmers while localizing bugs, we investigated their forging behavior using IFT. Since the control group did not have the debugging support and had to forage to localize bugs, we discuss the foraging of only control participants while analyzing the transcripts for localizing the bugs. For an in-depth analysis, we randomly selected two participants each from the sets of the most successful and most unsuccessful participants in Task Y!E and Task SE (Figure 4). For Task Y!E we selected C.P5 and C.P7, and for Task SE we selected C.P2 and C.P3. In their respective tasks, participants C.P7 and C.P3 were successful, and participants C.P5 and C.P2 were unsuccessful.

The most unsuccessful participants mentioned above foraged more than the most successful participants. Spool's position is that the scarcity of scent leads to foraging [61]. Spool et al. found that when users run out of scents they often backtrack in hopes of picking up a better scent. However, this rarely works because they chose the best scent to start with. Consistent with Spool's position, the most unsuccessful participants in our study foraged more than the most successful participants in the control group while locating the errors. As can be seen in Figure 4, unsuccessful participants C.P5 and C.P2 tended to spend a greater amount of time foraging. They foraged more in the debugging window to inspect the modules and visited more unique patches. C.P2 visited 10 unique patches whereas successful

Table 4: Descriptions and Examples of Cues in Our Cue Hierarchy. Top level cues and their definitions are colored in grey.

Cue	**Description**	**Example**	**Remarks**
Easy to Understand Cue	A cue such that a forager can easily determine the direction in which it will lead. A forager may find these cues the least costly to utilize.		
Non-Misleading Cue	An Easy to Understand Cue that leads to a patch the forager expected		
Tried and Worked in the Past Cue	Non-Misleading Cue a forager follows/looks for again	C.P5 [Goes back to Rotten Tomatoes home page to open the Box Office list again] "It should show in descending order."	The participant had been to this patch before, and he anticipated it would provide the same information as when he foraged there in the past.
Misleading Cue	Easy to Understand Cue that leads to a patch other than what the forager expected		
Misleading Cue to Valuable Patch	Misleading Cue that still leads to a Valuable Patch	E.P1 [Googles 'Dave White RSS Feeds' and then tries 'RSS Feeds Dave White Rotten Tomatoes'. Opens the first link] "I think it's not RSS." [Looks at the modules and goes back to the web site of Dave White. Sees the RSS feed symbol where it's written 'RSS Feeds' and clicks and sees the contents as RSS feed] "Yeee!"	The participant erroneously presumed that this patch did not contain an RSS feed, but this was exactly what the participant needed in order to fix the bug. Once the participant realized this was indeed what she needed by looking at the RSS feed symbol, she returned to the patch to forage for the correct information.
Misleading Cue to Useless Patch	Misleading Cue that leads to a Useless Patch	E.P1 "I am trying to find the top 10 movies" [Clicks on menu option from URL of movies. It is blank, so she returns to the previous page].	The participant was uncertain as to where the patch would lead, but she decided to forage there since it contained a potentially helpful scent. Unfortunately, the patch yielded no information useful to the participant.
Difficult to Understand Cue	A cue such that a forager may find it difficult to determine where it may lead. A forager may find these cues more costly to follow and less beneficial		
Difficult to Understand Cue to Valuable Patch	Difficult to Understand Cue that leads to a Valuable Patch	C.P7 [Clicks on the Link to the API Key Application Page. [Logs into Flickr web site] "What is Flickr?" [Goes back to the original pipe and looks at the Flickr Module. Goes back to the Flickr API Key Application and reads about the different applications. Clicks to apply for a non-commercial key, then goes back to check the original pipe and returns to fill out the application].	The participant is uncertain as to whether or not this patch contains information about the missing API key needed to fix bug B1. Once she gains an understanding of what she needs by foraging in the modules and the application patch, she is able to use the information garnered by the non-commercial API key application patch.
Difficult to Understand Cue to Useless Patch	Difficult to Understand Cue that leads to a Useless Patch	E.P3 [Clicks on the link to the Example RSS Feed in the hint, then returns to the pipe].	Unable to reuse anything from the example to fix his own pipe, the participant returns to his pipe. The participant ventured to this patch to gain an understanding of RSS feeds, but the example, in the participant's view, did not provide any information applicable to solving the bugs in his pipe.
Not Followed Cue	A cue that is not followed by a forager because it is difficult to identify or locate where it may lead. A forager may find these cues to be highly expensive.		

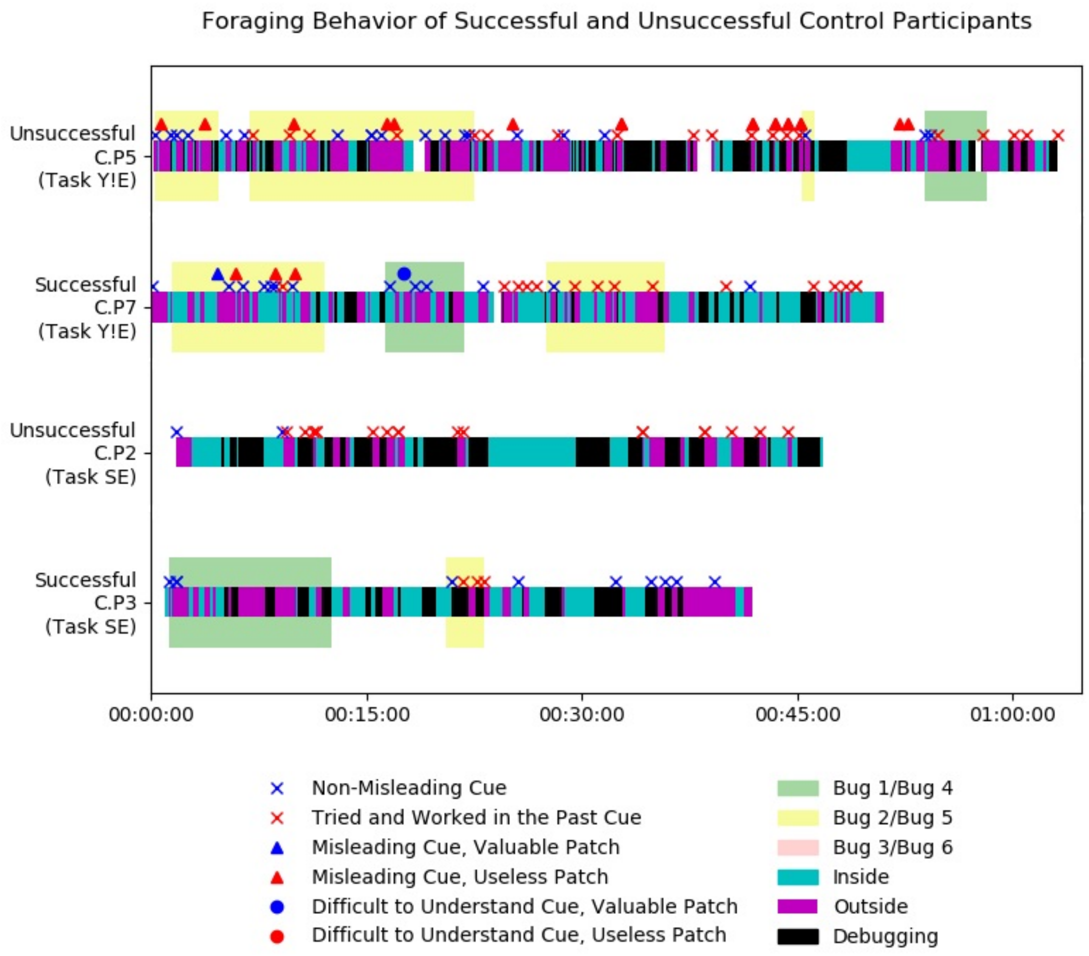


Figure 4: Foraging of four participants while performing Yahoo! Errors task (Y!E) and Silent Errors task (SE). Regions labeled "Inside" refer to users foraging inside the Yahoo! Pipes environment, and regions labeled "Outside" refer to foraging outside the environment. Regions labeled "Debugging" refer to when the user was fixing the bug. For the labels Bug X/Bug Y, Bug X refers to bugs in Task Y!E, and Bug Y refers to bugs in Task SE. Bug 1/Bug 4 (shown in green) and Bug 2/Bug 5 (shown in yellow) designate when the participant's focus was on the respective bug, and Bug 3/ Bug 6 (shown in red) designate when the participant's focus was on a bug in a nested subpipe. The X's, triangles, and circles above the bar show the cue a participant followed at a specific time.

Table 5: Bugs Finding and Fixed per Control Group Participant [37]. * represents a participant with prior knowledge of Yahoo! Pipes.

Participant	**Yahoo! Errors**						**Silent Errors**					
	B1		**B2**		**B3**		**B4**		**B5**		**B6**	
	L	**F**	**L**	**F**	**L**	**F**	**L**	**F**	**L**	**F**	**L**	**F**
P1	1	1	-	-	-	-	-	-	1	-	-	-
P2	1	1	1	-	-	-	-	-	-	-	-	-
P3*	1	1	1	1	-	-	1	1	1	1	-	-
P4	1	1	1	-	-	-	1	-	1	1	-	-
P5	1	1	1	-	1	-	1	-	-	-	-	-
P6	1	1	1	-	1	1	1	-	1	-	1	-
P7	1	1	1	-	1	1	1	-	-	-	-	-
P8	1	-	1	-	-	-	1	-	-	-	-	-
Total	**8**	**7**	**7**	**1**	**2**	**1**	**6**	**1**	**4**	**2**	**1**	**0**

Table 6: Bugs Fixed per Experimental Group Participant [37]. * represents a participant with prior knowledge of Yahoo! Pipes.

Participant	**Yahoo! Errors**			**Silent Errors**		
	B1	**B2**	**B3**	**B4**	**B5**	**B6**
	F	**F**	**F**	**F**	**F**	**F**
P1	1	1	1	1	1	1
P2*	1	1	1	-	-	1
P3	1	-	-	1	1	1
P4	1	1	1	1	1	1
P5	1	1	1	1	1	1
P6	1	-	1	1	1	1
P7	1	1	1	1	1	1
P8	1	-	1	-	-	1
Total	**8**	**5**	**7**	**6**	**6**	**8**

participant C.P3 visited 2 unique patches. The unsuccessful control participants could not find better scents and frequently went to `Useless Patches`. As can be seen in the following excerpt, unsuccessful participant C.P5 (Task Y!E) ran out of scents, so he continued to switch back and forth between Google searches and the Yahoo! Pipes editor (22 times over the course of 30 minutes), backtracking to previous locations while attempting to find promising leads.

Excerpt from C.P5 Yahoo! Error Transcript:

15:51 [Searches 'Fetch Feed' on the pipes web site]
...
16:13 [Changes search to 'help']
16:22 [Opens the first search result, but soon returns to search]
16:32 [Starts a new Google search, 'How to Fetch IMDB in Yahoo! Pipes']
16:53 [Opens the last search result on the page. Looks briefly at the page and then closes the window]
17:05 [Returns to the pipe]
17:07 [Saves and runs the pipe. There is still an error and warning message]
17:24 "Nothing's happening"
17:34 [Goes back to the pipe]
...
19:03 [Opens Google search 'How to Fetch IMDB in Yahoo! Pipes' and clicks on one of the top links]

We postulate that the entire control group, not just the most successful and most unsuccessful, struggled with Task SE because they were attempting to understand the entire pipe without any knowledge of where to begin looking

Table 7: Control Participants Spent a Large Fraction of Their Time, Ranging From 65% to 87%, Foraging for Information. * represents a participant with prior knowledge of Yahoo! Pipes.

Participant	C.P1	C.P2	C.P3*	C.P4	C.P5	C.P6	C.P7	C.P8	Mean
Time Foraging	80%	70%	65%	74%	71%	70%	87%	68%	73%
Y!E Foraging	91%	68%	73%	82%	67%	70%	84%	67%	75%
SE Foraging	71%	71%	60%	70%	82%	70%	90%	71%	73%

for errors. Our tool, however, which highlighted faulty modules, reduced the cognitive load on the experimental participants, who could then focus on trying to comprehend individual modules rather than the program in its entirety. From an IFT perspective, Spool et. al [61] suggests that users resort to search only when they cannot find their trigger words or enough cues on the web page. For example, participant E.P2 spent 7 minutes fixing a bug in Task Y!E, whereas C.P4 required 27 minutes just to locate a bug in Task SE. This example helps to explain why participants in the experimental group spent less time fixing the bugs than the control group spent locating bugs.

During Task SE, the relatively few cues provided by the control environment for silent errors led to increased foraging both within the canvas and to unique patches. Both the most successful and most unsuccessful control participants struggled with Task SE, and silent errors were a huge foraging time sink. The control participants in Task Y!E visited 38 unique patches when localizing the bugs, whereas in Task SE they went to 55 unique patches. Additionally, our participants, both the most successful and the most unsuccessful, followed very few `Non-Misleading Cues` (see Figure 4). We conjecture that the lack of these `Non-Misleading Cues` led them to spend most of their time foraging for any potential bugs within the Yahoo! Pipes environment, other web pages (patches), and the debugging window. For example, unsuccessful C.P2 in Task SE spent all of his time tinkering in the Yahoo! Pipes Canvas as the error messages did not contain `Non-Misleading Cues`. Thus, foraging was a major time sink for control participants working on Task SE. Both the most successful and the most unsuccessful participants in the control group were unable to localize nested errors, as they were difficult to determine. Successful participant C.P3, on obtaining an error message in the subpipe during Task Y!E (bug B3), commented: *"Where is this coming from?"* In Yahoo! Pipes, to debug a subpipe a user must open and execute it; C.P3 did not realize this and spent 10.44 minutes investigating the bug by clicking on other modules in the pipe. He then moved onto the next task after commenting: *"I don't know what it's saying."*

One reason for the lack of `Non-Misleading Cues` for the nested errors (bugs B3 and B6) may be that the cue for the subpipe was on just the top banner portion of the pipe's output along with the sub-program name. We suspect this behavior was because of "banner blindness," which is often found in web sites. Banner blindness occurs if links appear in the top few rows of pixels of a web page as users ignore the links at the top of the page [61]. Another reason for this can be lack of understanding of concepts related to modularity or nested programs.

The entire control group used `Tried and Worked in the Past Cues` more when localizing bugs than when fixing bugs. The transcripts revealed that while localizing bugs, 65.7% of the cues followed by the control participants were `Tried and Worked in the Past Cues`, whereas these uses accounted for only 43.4% of the cues followed while fixing bugs (Section 5.1.2). For example, Participant C.P4, while foraging in Task SE, followed 26 `Tried and Worked in the Past Cues` while localizing faults and 9 while fixing bugs. Participants were likely motivated to rely on familiar cues in order to reduce cognitive load. It is widely acknowledged among psychologists that the more a person attempts to learn in a shorter amount of time, the more difficult it is to process that information in short-term memory [5].

The above observations are summarized as follows:

- Control participants spent more time foraging compared to their professional counterparts in past studies.
- Successful control participants (C.P3 and C.P7) and unsuccessful control participants (C.P2 and C.P5) foraged differently. Unlike successful participants, unsuccessful participants spent more time foraging, and they frequently followed cues and scents that led them to `Useless Patches`.
- All Eight control participants struggled with Task SE (Silent Error) because there were few `Non-Misleading Cues`, which inhibited their ability to know where to begin looking for errors. The experimental group, however, knew where to begin foraging by using our tool.

Table 8: Details on Time Spent by Control Participants Foraging for Information.

	Non-Misleading Cues	**Tried and Worked in the Past Cues**	**Misleading Cues to Valuable Patch**	**Misleading Cues to Useless Patch**	**Difficult to Understand Cues to Valuable Patch**	**Difficult to Understand Cues to Useless Patch**	**Total**
Navigations per Bug Fixed							
Experimental	3.80	4.03	0.10	1.03	0.20	0.20	**9.20**
Control	1.58	1.42	0.00	0.17	0.08	0.33	**3.58**
Navigations per Bug Not Fixed							
Experimental	5.75	6.75	0.00	2.13	0.00	1.63	**16.25**
Control	1.08	1.25	0.03	0.36	0.06	0.00	**2.78**
Navigations per Bug Fix Attempt (Total)							
Experimental	9.55	10.78	0.10	3.16	0.20	1.83	**25.45**
Control	2.66	2.67	0.03	0.53	0.14	0.33	**6.36**

- Due to the lack of `Non-Misleading Cues`, the control group reduced their cognitive load by relying on `Tried and Worked in the Past Cues`, which were familiar to them.

5.1.2. Foraging While Finding Solutions to Faults

To understand the foraging behavior of participants while fixing bugs, we measured whether the navigations made by them while attempting to fix a bug were successful or not.

Compared to the entire control group, experimental group participants navigated more often between patches by following cues and were more successful in fixing the bugs. As seen from the data in Table 8, the experimental group followed a significantly greater number of cues per bug (25.45 navigations) than the control group (6.36 navigations), showing that they were more apt to forage. For example, E.P1, while doing Task Y!E, used the information features supplied by the To-fix list, so he quickly found relevant, useful cues that led to prey, fixing 2 bugs in less than 5 minutes. Therefore, our tool helped participants navigate more often as they followed more cues per bug than participants in the control group.

As can be seen in Table 8, the control group followed fewer cues per bug (3.58 navigations per bug fixed and 2.78 navigations per bug not fixed) than their experimental group counterparts (9.20 navigations per bug fixed and 16.25 navigations per bug not fixed). The experimental group, however, was willing to spend more time foraging for prey, so they continued to navigate for them. Conversely, control group participants gave up sooner even if they were on the right track. For example, participant C.P6, while fixing bugs in Task SE (see Figure 5), spent less than 2 minutes foraging to find/fix a bug before moving on to find/fix a different bug. Figure 5 reveals this behavior. The participant oscillated between brief periods of foraging (cyan and magenta regions) and debugging (black regions). In addition, from the the beginning of Task SE until around minute 30, C.P6 briefly worked on a specific bug before moving on to another (note the green and yellow regions in Figure 5). In most cases, had participants persevered a short while longer, they would have fixed the bug.

The foregoing results suggest that our tool impacted our participants' level of persistence. While the experimental group was given the precise location of, and additional information related to, the bug, the control group was not provided such assistance. It is possible that the control group participants questioned whether the prey they were following was really the correct prey because of a relative lack of scent. Due to uncertainty, the control group may have abandoned certain available scents because the cost of pursuing potential prey was greater than the perceived benefit. For example, Participant C.P6, despite missing the link to the Flickr API Application right within the Yahoo! Pipes environment, managed to navigate there by Google search. Unfortunately, his lack of understanding caused him to abandon his efforts and leave the application for the API key. In contrast, every experimental group participant fixed the bug with the API key, even those uncertain of an API key's purpose.

On the other hand, the experimental group followed more cues to `Useless Patches` (4.98 navigations) than the control group (0.86 navigations). As the experimental group foraged more and had greater success despite visiting more `Useless Patches`, we posit that our tool assisted the experimental group in foraging and recovering

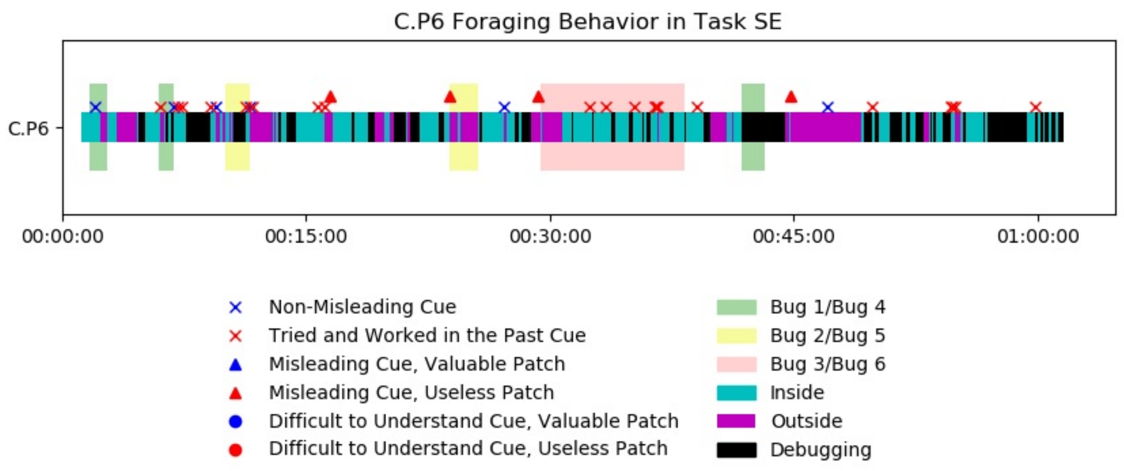


Figure 5: Foraging and Debugging Behavior of Participant C.P6. Note the periods in which the participant oscillates between debugging (black regions) and foraging (cyan and magenta regions). The participant spent brief periods working on Bugs 4 and 5 (green and yellow regions, respectfully) but spent an extended period working on Bug 6 (red regions). The participant primarily foraged Inside the Yahoo! Pipes environment while working on Bug 6.

from navigations to `Useless Patches` by reducing navigation cost and providing more scent to the experimental group. It did this by supplying useful information features related to the prey, which assisted participants in producing more accurate estimates of the value of cues. This is similar to the behavior of professional programmers discussed in similar studies as they persistently pursue specific information in spite of low returns coupled with high cost [50]. As can be seen in the transcript excerpt below, participant E.P4, while working on Task Y!E, followed cues from our tool, which led the participant to a `Useless Patch`, the erroneous URL.

Excerpt from E.P4 Yahoo! Error Transcript:

5:21 [Reads about the FF Error. Clicks on the Hint]
5:52 [Clicks on Link to View Typically RSS Feeds]
6:07 [Copies the erroneous URL from the {FF}, and pastes it in the browser. Web site says that the page doesn't exist].

In this case, although the patch was useless to the participant, our tool provided additional scents and aided the participant in foraging for information.

Both groups followed a significant number of `Tried and Worked in the Past Cues` (an average of 43%) and `Non-Misleading Cues` (an average of 40%) while fixing pipes (see Table 8). This demonstrates end users' heavy reliance on `Easy to Understand Cues`, especially those that they have followed in the past, when determining which cues to follow in the future. We postulate that `Tried and Worked in the Past Cues` were popular for end users because they made it easier to determine the cost/benefit of similar cues based on past success. Participant E.P2, as can be seen in the transcript excerpt below, spent 7 minutes during Task SE making progress since we can observe him developing ideas about how to fix the problem. If these hints were useful in the past, he was able to create more accurate cost/benefit analyses regarding whether or not he wanted to follow those cues in the future. Since he followed many of these cues in a row, we can conclude that he found them to be helpful. The participants varied the types of cues they followed only if `Tried and Worked in the Past Cues` were not available or useful-looking for the case at hand.

Excerpt from E.P2 Silent Error Transcript:

11:23 [Returns to the pipe and looks at the second error and reads the Help about the {IB} error]
12:35 [Starts reading the {IB Help}. Then opens the IB example pipe and looks at the {IB Attributes}]
13:20 [Returns to the main pipe and examines the {IB Module}]
14:06 [Goes back to the example pipe and examines the {IB Module}]
14:30 [Runs the {IB Example Pipe} to see how IB works]
14:58 [Opens the {Link} in the example IB link attribute]
15:27 [Goes to the {Learn More About Item Builder} and reads the information and compares it to the pipe]
17:16 [Clicks on {View Source Link} and opens the example pipe] "Yeah, that's the same one."
17:23 [Returns to the main pipe and examines the pipe to see how it works]
18:28 "I'm just wondering what this thing is doing right here." [Refers to the {IB and SB} then looks through the task again]

Few participants from both groups followed `Difficult to Understand Cues`, nor did they tend to have problems with `Difficult to Understand Cues`. Experimental group participants followed only 2.03 `Difficult to Understand Cues` per bug fix attempt (Successful and Unsuccessful), and control group participants followed only 0.47 per bug fix attempt (Successful and Unsuccessful). We postulate that they did not follow these cues as they were difficult to understand or determine their benefit. This can be seen in the behavior of Participant C.P4 while doing Task Y!E: *[Clicks in the API Key box. Clicks on the Key Link. Page Opens. Returns to pipe].* Participant C.P4 was unable to determine the exact value of the patch, so he left the patch even though it was necessary to fix bug B1.

The above observations are summarized as follows:

- Experimental participants followed cues between patches more often than control participants. Thus, our tool helped our participants to navigate between patches more easily.
- Our tool enabled the experimental participants to persevere and find solutions to bugs even if they were uncertain of where to find solutions.
- Our tool reduced navigation costs between patches, which can be seen by the increased number of experimental participants' navigations to `Useless Patches`.
- Participants in both groups preferred `Tried and Worked in the Past Cues` because their cost-benefit analyses were more easily determined.
- Neither group tended to follow `Difficult to Understand Cues` because their cost-benefit analyses were difficult to determine.

5.1.3. Backtracking While Finding and Fixing Faults

The most unsuccessful participants in Task Y!E tended to backtrack more often than the most successful participants. In past studies related to professional programmers, it was observed that the programmers backtracked at least 10.3 times per hour [66]. Some users were frustrated because they could not "undo" their changes and some examples available through Yahoo! Pipes were not a close match or did not execute correctly. Figure 4 shows the prominence of similar behavior for end-user programmers. During Task Y!E, successful participant C.P7 backtracked 7 times. C.P7 commented: *"Many times the examples don't work . . . if those examples worked I could have done better."* At other times, the available examples were too complex for users to follow. Conversely, unsuccessful participant C.P5 backtracked 40 times in Task Y!E between the main pipe and various help/example pipes, resulting in little progress and an inability to fix most of the bugs. This resonates with our previous results on end-user programmers where they backtracked numerous times to their previous strategies [11, 39].

In general, all control participants backtracked their code edits more (228 instances) than the experimental group (35 instances). Usually, control participants were uncertain as to how to fix the errors, so they were forced to reverse their edits if their changes did not fix the bugs in ways they expected. This may also be reflected on the relatively

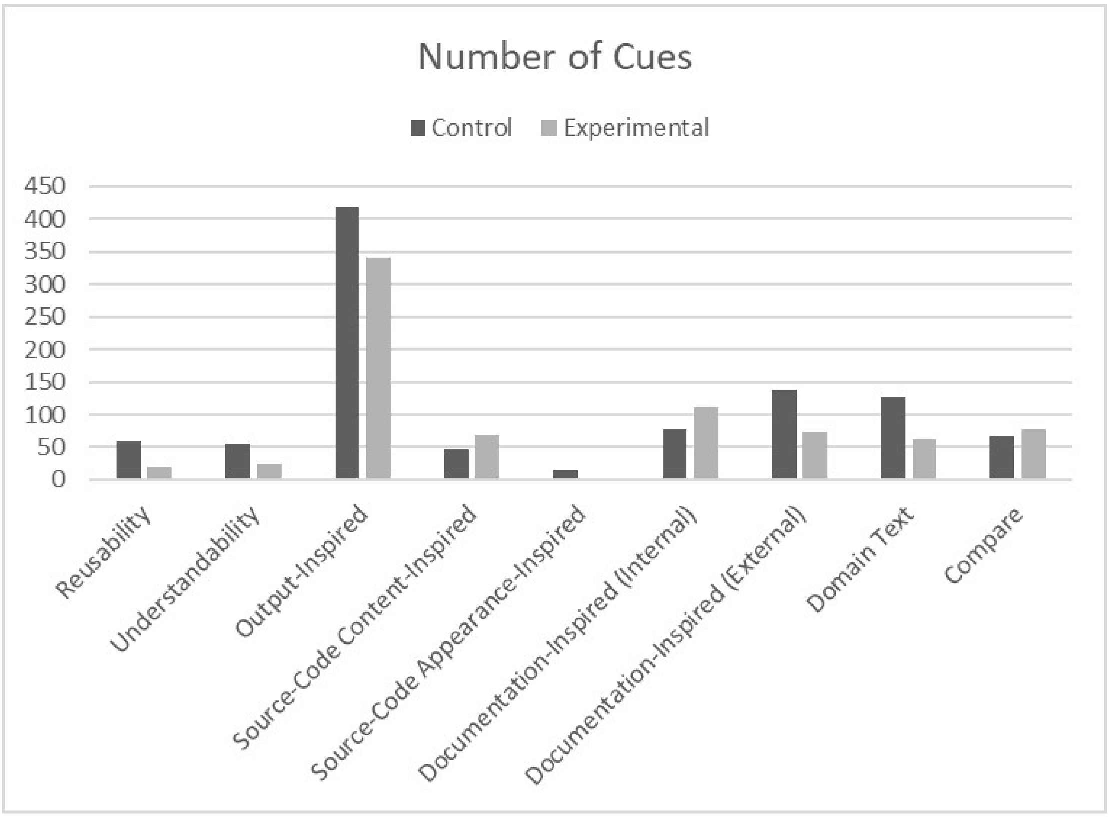


Figure 6: Number of cues followed per treatment group

higher number of `Output-Inspired Cues` they followed in order to verify their edits with output. However, both groups backtracked through web contents nearly the same number of times (control 51 and experimental 56).

Most participants in the control group explored alternative strategies to fix the bugs and needed to backtrack when they were not successful. For example, C.P4 spent 51 minutes in unsuccessful explorations, and then before continuing to the next task commented: *"I couldn't understand the error messages...if there were steps to solve I could have solved it. They expected you to know that this is an error...I am not familiar with how to solve them"*. 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 bug) 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 debugging support enabled 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 web site 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.

5.2. Cues While Debugging in a VDLW Environment

Using cues from the literature [48], we can discern that experimental participants followed cues to help them fix bugs, and control participants followed cues to localize them. To understand what additional cues participants tended to follow, we aggregated the occurrences of the cues found in Table 3. As can be seen in Figure 6, the experimental and control groups followed `Output-Inspired Cues` more often than other cue types, and of the other cue types, they followed cues that led them to patches specific to their goal.

Each group preferred different cues because they had different goals. The control group had to localize the bugs as well as fix them, and they did not have the tool that was available to the experimental group. Thus, since they are end users with limited experience with Yahoo! Pipes, more expensive foraging was required. The control group used cues that enabled them to more easily understand the pipe and glean information from more example programs than the development environment could provide. These cues included `Domain Text`, `Understandability`, `Documentation-Inspired (External)`, and `Reusability Cues`. The experimental group, however, was provided with hints from the debugging tool. Although they were also unfamiliar with Yahoo! Pipes, they did not need to learn about Yahoo! Pipes outside of the information readily provided to them by the tool. As such, the experimental group preferred `Documentation-Inspired (Internal) Cues` and made more `Comparisons` between their buggy pipes and working examples. Hence, the control group followed cues to more easily understand the pipe, and experimental participants followed cues available from our tool to fix bugs.

Users in both groups tended to follow `Output-Inspired Cues` more than any other cues consistent with past literature [48], as seen in Figure 6. Participants seemed to prefer to see what the pipe actually did in order to determine what errors may be present. Although experimental participants had a tool that localized the bugs for them, they still followed significantly more `Output-Inspired Cues` than any other type. `Output-Inspired Cues`, then, must also be `Easy to Understand Cues` (Table 4) to end-user programmers during both fault localization and fault correction. If the output is incorrect, then it is fairly obvious there is some sort of error that must be fixed. For example, C.P2, within the first two minutes of Task SE: *[Runs the pipe] "I want to see what's wrong with the pipe." [Looks at the output of the pipe and reads the instructions of the task].* In the case of experimental participants who knew what errors they needed to fix and where, participants likely wanted to see exactly how errors presented themselves in order to determine how to fix them. Experimental participants primarily followed `Output-Inspired Cues` provided by the debugging tool. For example, the first action E.P1 took in Task Y!E was: *[Clicks to find the errors] "Let me first click to get the errors."* So we posit that `Output-Inspired Cues` are `Easy to Understand Cues`.

The control participants followed `Documentation-Inspired (External) Cues` and experimental participants followed `Documentation-Inspired (Internal) Cues` (Table 4). Control participants likely wanted more guidance to fix the program than the VDLW environment could provide, so they looked elsewhere for assistance, namely the Internet. Unlike control participants, experimental participants preferred `Documentation-Inspired (Internal) Cues`, especially those found in the hints provided by our tool's To-fix list. These hints allowed participants to remain in the patch (Yahoo! Pipes environment) while foraging for solutions. The cost to use these hints was relatively low as the participants did not have to forage outside of the environment for patches. They could gather whatever information they needed within the development environment, and these patches were often valuable and pertinent to the bug the participants were trying to fix. For example, E.P4 consistently returned to the tool hints when she needed clarification or was stuck. Hence the experimental participants stayed in the Yahoo! Pipes environment because the cost to follow hints provided by our debugging tool was low.

As Figure 6 shows, experimental participants performed more `Comparisons` between example pipes and their own pipes. As mentioned above, the experimental group utilized more internal documentation in the form of hints, which often led to sample pipes. Due to the relatively low cost of finding and accessing these examples, we posit that our tool helped the experimental participants more accurately determine what information they needed from the sample pipe patches by comparing the information found in the hints, the information in the sample pipes and other documentation, and the bugs reported in the To-fix list. This phenomenon, in turn, lowered the overall cost of foraging in the example pipes by allowing the experimental participants to hone their focus while foraging. The following excerpt illustrates this relative to participant E.P2:

Excerpt from E.P2 Silent Error Transcript:

11:23 [Returns to the pipe and looks at the second error and reads the {Help} about the {IB} error]
12:35 [Starts reading the {IB Help}. Then opens the IB example pipe and looks at the {IB Attributes}]
13:20 [Returns to the main pipe and examines the {IB Module
14:06 [Goes back to the example pipe and examines the IB Module].

Control participants would often follow `Reusability Cues`, which they believed would allow them to reuse elements from other internal modules or external sample sources even if they did not fully understand how an example pipe worked. Uncertain of what information they truly needed from sample pipes, control participants likely followed `Reusability Cues` in an attempt to reduce the cost of finding which parameters (URLs, API keys, etc.) they needed to change in their own pipe. They attempted to incorporate a preexistent solution as it reduced the amount of time necessary to understand and create a solution themselves. For example, C.P4: *[Finds the IB and starts reading the help. Loads the IB Example pipe] [Types 'title' into the IB Attributes. Looks at the example pipe again].* The experimental group, by comparison, ventured to patches marked by `Reusability Cues` less often because they had better scents to follow and found information through other actions or cues, such as `Comparisons` or `Documentation-Inspired (Internal) Cues`.

Control participants followed more `Domain Text Cues` and `Understandability Cues` [48] than the experimental participants in an attempt to understand the pipe in its entirety. Thus, they searched through the domain text of the pipe in order to gain an understanding of the function of each module, a process that was often time-consuming. For example, Participant C.P8 wanted to understand how to edit the pipe, so she searched for answers to her questions: *"Can I add boxes?" [Begins to look through the list of modules].* The experimental group, on the other hand, had access to the hints provided by our tool, so they did not need to localize the bugs. The only cues they followed to understand the pipe were those that helped them understand the error itself. Hence, the To-fix list provided by our tool facilitated Minimalist Theory [12], i.e., helping participants learn in the context of their specific task and goal.

6. Discussion

6.1. Implications for Theory

We categorized the different types of cues that are present in a visual programming environment into a hierarchy of cues, namely `Easy to Understand`, `Difficult to Understand`, and `Not Followed`. We discussed participant behavior when they faced issues related to these cues and classified the patches as these cues led them to a `Valuable Patch` or `Useless Patch`. 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.

In addition, we articulated the different strategies that users can employ when debugging. We began by noting the overall hunting strategies participants used. 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 a 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, participants preferred to stay within the editor (the patch) to the extent possible. Finally, diet constraints (where information had to be formatted in a specific manner) affected debugging capacities. Participants could easily forage to the right web content (reviews by Rotten Tomatoes) but had significant difficulty finding the RSS feed for the reviews. Hence, there 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 feeds are allowed in the module) and the diet formats available in web content (where the RSS feed is located in the web site) are not clear, especially to end users.

Furthermore, the IFT model was traditionally designed to support dissimilar but joint topology. Unfortunately, most distributed programming environments such as mashups, mobile programming, and the internet of things tend to be topologies of disjoint patches. This disjoint topology problem is prominent during fault correction in distributed programming environments. For example, the APIs in the mobile environment are also disjoint and create challenges similar to those associated with the mashups used in our study. Hence, the cues and strategies observed in our study can be generalized to other similar programming environments, especially Visually Distributed Languages for the web programming environments.

6.2. Implications for Design

We now discuss ways in which our findings can help improve interface design for distributed, visual, and VDLW programming environments.

Bookmarking cues: The ability to manually or automatically bookmark cues has been determined to be useful in the past [29, 3] and makes cues easily accessible in the future. Such an ability could be implemented by marking cues similarly to bookmarking URLs in a web browser. This would allow the users to add, delete, and manage cues in the form of a list with references to `Valuable Patches` so that they can easily keep track of successful cues. In the context of IFTTT, a platform that allows end users to create customizable apps, enabling bookmarking of favorite services (as cues) can help end users who are looking to resuse these services in future apps (or help others find popular services).

Structural relatedness: Debugging tools need to provide structurally related solutions when providing the exact solution is not feasible. For example, providing information about the structure of RSS feeds, that a user is looking for, helps them find the relevant RSS feeds. In the context of IFTTT apps, "Connect RSS Feed" can utilize the hints with structured relatedness to atom web feeds[16] template.

Output-inspired tool designs: Debugging tools, especially those related to end-user programmers and mashups, need to be based on the output of the programs. High instances of `Output-Inspired Cues` for both control and experimental participants affirm that end-user programmers, especially when using VDLW programming environments, follow the outputs as cues for their future foraging ventures. In IFTTT, allowing searching based on displaying the output of the Applet would be useful.

Information feature/patch recommenders: VDLW programming environment developers should consider creating a recommender system that highlights useful information features or patches, which would lead to diminished cost. If participants have visited a patch, the interface should provide cues, specifically `Tried and Worked in the Past Cues`, along with feedback regarding their previous visits. Google search, for example, displays the date when a page was last visited and changes the color of previously visited links.

Patch layout is important: Tools need to have better information feature layouts that may help end users forage for errors without unnecessary cost. The foraging cycles within the tool environment led to frustrations and unsuccessful results. In the context of IFTTT applets, the services should have contents that are easily accessible.

7. Threats to Validity

Every study has threats to validity and here we explain threats related to our study and how we guarded against these threats.

External Validity: The primary threat to external validity is the generalizabity of our study participants. Our study participants were undergraduate and graduate students from different majors but had either little or no experience with web mashups. The students were selected through convenience sampling and past studies have shown that students are an appropriate sample [33] when representing end users. While, we cannot generalize our results to mashup programmers who may be more experienced in web programming, one can argue that students are representative of end-user programmers with some experience in programming. Further, our participants had created programs in a wide-variety of programming languages, including many that are popular with end users.

A second threat relates to the generalizability of the programming environment. We have studied only one mashup environment; however, it is representative of a broader class of web-development environments (e.g., Apatar [17], App Inventor).

A third threat relates to the generalizability of tasks. Our study considered only two tasks that built on only two types of pipes. Our participants were asked to use pipes that were provided, rather than pipes that they had created for themselves. End-user programmers frequently reuse as they learn by looking at or using code examples [7, 45]. A case study of Android Market found an average of 61% of two or more mobile app containing same code [56]. Similarly, another case study of Yahoo! Pipes found more than 56% of clones [64] and 27.4% of the programs containing subpipes [37]. While the reuse context is common and important, prior familiarity with pipes could lead to different results. Our tasks for control and experimental groups were counterbalanced to help reduce bias in the performance of the tasks.

[16]https://validator.w3.org/feed/docs/atom.html
[17]http://apatar.com/.

Internal Validity: The primary threat to this particular study relates to our choice of a between-subjects design. This study design helped minimize the effects of learning as the participants moved from initial to later tasks, but it may have led to individual differences in our use of a small pool of participants. We performed Wilcoxon rank tests on our time data to quantitatively study the effects of time; however, because our participants were performing in think aloud mode, timing measures may be affected. Further, it is also well-established that use of the think-aloud method can impact participant behavior although the think-aloud protocol is a widely accepted method of approximating a participant's thought process.

Construct Validity: To help assure the validity of our constructs, we examined both groups with respect to a broad range of IFT constructs (patch/cue/strategies). Our inter-rater reliability was 85% on all code sets, helping to assure construct validity. To help assure the extent to which a measure actually captures what it intends to measure, we supplemented qualitative analysis with quantitative analysis and carefully controlled for any confounding per-participant effects. The results of our study can be influenced by the depth of the tutorial provided to our participants. However, all participants were provided with the same tutorial and the knowledge in the tutorial was similar to that in the online Yahoo! Pipes environment. Further, threats may occur with the possibility that the complexity of our pipes was not high enough to allow measurements of effects, and that the pipes used were not representative of the complexity of real-time applications. We controlled for this by performing initial pilot studies on non-participants and using their feedback to adjust the pipes and the tasks. Additional studies are needed to examine other types of mashup environments, tasks, and usage contexts.

8. Related Work

Here, we discuss related work on mashups and Yahoo! Pipes, end-user debugging, and IFT in software engineering tools.

8.1. End-User Debugging

Debugging is an integral part of programming. Studies have shown that professional developers as well as students [19] spend significant portions of their time debugging. In fact, Rosson et al. [54] suggest that even professional programmers often "debug into existence" their programs; that is, they create their programs through a process of successive refinements.

There has been some work directed at end-user programmers engaged in debugging. Grigoreanu et al. [23] have developed a classification framework for characterizing end-user programmers' debugging strategies. They identify several debugging strategies including code inspection, following data flow, following control flow, testing, feedback following, seeking help, following spatial layout and specification checking. In a subsequent study, Grigoreanu et al. [24] examined the sense making process; that is, the process by which end users understand bugs and their causes (based on prior work by Priolli and Card [53]). Sense making consists of an information foraging loop followed by attempts to understand the foraged information. They found that the foraging loop dominated the sense making process. As noted in Section 6.2, Cao et al. [10] 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.

Most end-user programming environments support debugging only through consoles that print debugging output [22]. A few improvements, however, have been proposed. "What You See Is What You Test" (WYSIWYT) [55] supplements the conventions by which spreadsheets provide visual feedback about values by adding feedback about "testedness." The WYSIWYT methodology can also be useful for other visual programming languages (and VDLWs) [30] and for debugging of machine-learned classifiers by end users [35]. Our approach shares some aspects of WYSIWYT; we also use definition-use coverage to find some classes of faults. We differ from WYSIWYT as we provide a "To-fix" list of errors and guidance for fixing them.

Another approach for identifying faults is to check the internal consistency of the code instead of relying on the external specifications. Ucheck is an example of such an approach. Ucheck [1] allows users to statically analyze a spreadsheet and narrow the class of faults it may contain. Similar to this approach, our Anomaly Detector also identifies some types of faults by statically analyzing the code of mashups. Unlike Ucheck, we provide guidance to users for fixing the identified faults.

Lawrance et al. [40] developed ways to combine reasoning from Ucheck and WYSIWYT. The combined technique was helpful in locating faults and mapping the fault information visually to the user. They found that their combined technique was more effective than either technique alone. Our support is similar to their approach in that we also automatically locate faults and provide information regarding faults visually. We differ as we provide a “To-fix” list for viewing the list of faults and also provide guidance for fixing those faults.

GoalDebug [2] is a semi-automatic debugger for spreadsheets. When a user selects an erroneous value and provides an expected value, the debugger recommends a list of changes in the formula that would provide the expected value. The user can explore, apply, refine or reject changes suggested by the debugger. Our approach differs as users are not anticipated to know the expected value of the faulty code; rather, we provide guidance based on types of faults in the code.

Topes [57] is a data model that helps end users validate and manipulate data in spreadsheets. A similar strategy can be used to identify mismatches in data formats. Giuseppe et al. [17] presented the Task Process Model and the Landmark-Mapping model for describing the code search strategies of non-programmers. Our work also includes search strategies by non-programmers, but we use the lens of Information Foraging Theory.

Whyline [31] is a hypothesis-driven tool that allows programmers to ask “why did” and “why didn’t” questions about program output in visual programming languages such as Alice. It employs both static and dynamic analyses of programs to provide answers when a programmer selects a question. In our case, we employ static analysis for detecting different classes of faults. Our approach is different from Whyline [31] as we do not apply dynamic analysis of programs to provide answers to the programmer. Mashup architectures are collections of physically distinct components; this makes it difficult to dynamically analyze their results and hence mandates a different approach. Therefore, we have provided guidance based on types of faults in the code.

Assertions have been used to find faults in web macros [34], whose creation involves copying and pasting techniques. Once data is entered by the user it is saved in the clipboard, and before the data is reused, it is tested for existence and data types. Assertion-based approaches have also been effective in testing and debugging spreadsheets [8]. In our case, for one class of faults (Mismatch Range) we use assertions where we check the ranges of numbers used during program execution. We use other kinds of static analysis for identifying other types of faults.

Stolee and Elbaum [62] found that end-user mashups tend to suffer from several types of errors and deficiencies. They defined programming deficiencies in Yahoo! Pipes as “smells.” They developed mechanisms for identifying and classifying smells and refactoring techniques for removing smells from pipes. Their proposed techniques reduce the complexity of the pipes and also help apply software development patterns to standardize the design of pipes. Their studies showed that refactoring techniques can significantly minimize smells. Some of the smells they identified are similar to our fault classifications, e.g., Link, Missing: Parameter, Deprecated Modules, and Connectors. Our approach differs from their work as we automatically identify faults but do not attempt to fix them. Rather, we provide guidance to users for fixing faults.

Most mashup programming environments support debugging by providing a debugger console for the program. Yahoo! Pipes is the only such environment that provided a debugger console for every module. The only other debugging support for mashup environments that we are aware of is from our prior work [36], which allows mashup programmers to “tag” faulty and tested mashups. Faults can then be localized by “differencing” faulty and correct versions of a pipe.

8.2. Mashups and Yahoo! Pipes in EUP research

Jones and Churchill [28] described various issues faced by end users while developing web mashups using the Yahoo! Pipes environment. They observed discussion forum conversations in order to understand the practices followed, problem solving tactics employed, and collaborative debugging engaged in online communities of end users.

Zang and Rosson [67] investigated the types of information that users encounter and the interactions between information sources that they find useful in relation to mashups. In another study they found that, when asked about the mashup creation process, end users could not even describe it in terms of the three basic steps of collecting, transforming, and displaying data [68].

There has been research aimed at understanding the programming practices and hurdles that mashup programmers face in mashup building. Cao et al. [11] discussed problem solving attempts and barriers that end users face while working with mashup environments and described a “design-lens methodology” with which to view programming. Cao et al. later [10] also studied debugging strategies used by end-user mashup programmers. Gross et al. created a mashup model consisting of the integration layer, presentation layer, and UI components[26]. Aghaee et al. created a mashup a high level of expressive mashup environment, NaturalMash, that allows end-user programmers to create mashups [4].

Dinmore and Boylls [18] empirically studied end-user programming behaviors in the Yahoo! Pipes environment. They observed that most users sample only a small fraction of the available design space, and simple models describe their composition behaviors. They also found that end users attempt to minimize the degrees of freedom associated with a composition as it is built and used.

Stolee et al. [63] analyzed a large set of pipes to understand the trends and behaviors of end users and their communities. They observed that end users employ repositories in different ways than professionals, do not effectively reuse existing programs, and are unaware of the community.

Grammel and Storey [22] analyzed six mashup environments and compared their features across six different factors (Levels of Abstraction, Learning Support, Community Support, Searchability, UI Design and Software Engineering Techniques). They observed that support for various software engineering techniques like version control, testing, and debugging is limited in various mashup environments.

Research related to mashups motivated us to investigate our approaches in mashup environments. Hence, to implement and evaluate our approaches we have used the Yahoo! Pipes environment. Our work differs from the foregoing in several ways: (1) We have analyzed the code and execution traces of code to analyze and find faulty programs that exist in the Yahoo! Pipes repository. (2) We have provided debugging support for Yahoo! Pipes and our studies showed that this helped users debug their programs efficiently and effectively. (3) We have analyzed the behavior of users while debugging mashups from an IFT perspective.

8.3. IFT in Software Engineering Research

Researchers have applied IFT to software engineering to understand programmers' behavior. IFT has been applied to understand the navigational behavior of programmers by interpreting results from an empirically based model of program comprehension [32]. Another formative study observed that opportunistic programmers use documentation in a manner consistent with IFT [6], but these studies did not mention how any specific IFT constructs, such as cues, strategies, matched up with empirical observations.

Fleming et al. [20] investigated the applicability and utility of IFT for debugging, re-factoring, and reuse. They found that, in these successful tools, IFT mathematical models can be applied to help programmers and to identify recurring design patterns.

Lawrance et. al [43] studied the foraging behavior of programmers using programming environments while debugging large collections of source code. Based on these results the researchers found that IFT can be used to predict programmers' behavior with scents and topology without relying on their mental states such as hypotheses. The researchers developed an executable IFT model that accurately predicted programmers' navigations when compared with the non-IFT comparable models.

Piorkowski et al. [48] studied the foraging behavior of professional programmers while they forage for learning vs. fixing bugs. The researchers found different foraging behavior for both groups, and they also found differences in the type of information, patches, and tactics they used. Piorkowski et al. [50] also studied the diet of the programmers and found they need diverse and distinct dietary patterns. They found that programmers foraged using different strategies to fulfill each of their dietary needs and goals. Later, they studied how well a programmer can predict the value and/or cost of a navigation path and found that 51% of their programmers' foraging decisions led to disappointing values of information obtained, and about 37% of their programmers' foraging decisions resulted in higher costs than anticipated [49].

Our study allows us to understand the foraging behavior of end-user programmers especially in mashup programming environments. End-user programmers not only forage within the programming environment like professionals but also on the web to find the fixes for the bugs.

9. Conclusions

Our analysis of the behavior of 16 end-user programmers using IFT demonstrated how debugging support in a VDLW programming environment, a web mashup development environment in this case, can ease the difficulty of foraging for information while debugging. We identified new patches, `Valuable` and `Useless`, and a new hierarchy of cues that end users followed, including `Easy to Understand`, `Difficult to Understand`, and `Not Followed Cues`. End users preferred familiarity over novelty. They followed cues that have worked for them before (`Tried and Worked in the Past Cues`, a type of `Easy to Understand Cue`) and did not follow cues with higher navigation costs (`Difficult to Understand` and `Not Followed Cues`). This was due in part to the need of the end users in our study to learn the environment as they debugged the pipes. Thus, foraging for any information was costly to them since they were often uncertain as to what they should look for, but they were willing to forage through costly patches if they had exhausted their `Easy to Understand` scents.

During the debugging process, end users spend significantly more time foraging for information than professional programmers. While debugging, end users look for some of the same cues as professional programmers, namely those that enable them to observe the output of their program. Current debugging mechanisms (like runtime observations in Yahoo! Pipes) do not reveal bugs due to the black box nature of visual programming environments. Therefore, without the aid of debugging support, end users struggle with localizing bugs and verifying solutions. As a result, they tend to backtrack to previous states in the debugging process if they determine (sometimes erroneously) that they are not arriving at `Valuable Patches`.

As most VDLW environments such as mashups, mobile programming, and the internet of things tend to be topologies of dissimilar and disjoint patches, IFT was not intended to model such environments but those with dissimilar yet joint topology. The cues and strategies observed in this study reflect the disjoint nature of APIs that create challenges similar to mashup programming environments. Thus, the cues and strategies observed in our study may be generalized to other disjoint programming environments.

References

[1] R. Abraham and M. Erwig. Header and unit inference for spreadsheets through spatial analyses. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 165–172, Italy, 2004.

[2] R. Abraham and M. Erwig. GoalDebug: A spreadsheet debugger for end users. In *Proceedings of the International Conference on Software Engineering*, pages 251–260, UK, 2007.

[3] David Abrams, Ron Baecker, and Mark Chignell. Information archiving with bookmarks: Personal web space construction and organization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 41–48, USA, 1998.

[4] Saeed Aghaee and Cesare Pautasso. End-user development of mashups with naturalmash. *Journal of Visual Languages Computing*, 25(4): 414 – 432, 2014.

[5] V. G. Aukrust. *Learning and Cognition*. Elsevier, first edition, 2011.

[6] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer. Opportunistic programming: How rapid ideation and prototyping occur in practice. In *Proceedings of the International Workshop on End-user Software Engineering*, pages 1–5, Germany, 2008.

[7] J. Brandt, P.J. Guo, J. Lewenstein, S.R. Klemmer, and M. Dontcheva. Writing code to prototype, ideate, and discover. *Software, IEEE*, 26(5): 18–24, Sept 2009.

[8] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *Proceedings of the International Conference on Software Engineering*, pages 93–103, 2003.

[9] J. M. Cameron, E. F. Churchill, and M. B. Twidale. Mashing up visual languages and web mash-ups. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 143–146, Germany, 2008.

[10] J. Cao, K. Rector, T. H. Park, S. D. Fleming, M. Burnett, and S. Wiedenbeck. A debugging perspective on end-user mashup programming. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 149–156, Spain, 2010.

[11] J. Cao, Y. Riche, S. Wiedenbeck, M. Burnett, and V. Grigoreanu. End-user mashup programming: Through the design lens. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1009–1018, USA, 2010.

[12] J.M. Carroll. *Minimalism beyond the Nurnberg Funnel*. MIT Press, 1998.

[13] E. H. Chi, P. Pirolli, K. Chen, and J. Pitkow. Using information scent to model user information needs and actions and the web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 490–497, USA, 2001.

[14] D. R. Compeau and C. A. Higgins. Computer self-efficacy: development of a measure and initial test. *MIS Quarterly*, 19(2):189–211, 1995.

[15] L. L. Constantine and L. A. D. Lockwood. *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design*. ACM Press/Addison-Wesley, 1999.

[16] A. Cypher, M. Dontcheva, T. Lau, and J. Nichols. *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann, 2010.

[17] Giuseppe Desolda, Carmelo Ardito, Maria Costabile, and Maristella Matera. End-user composition of interactive applications through actionable ui components. *Journal of Visual Languages Computing*, 42, 08 2017.

[18] Matthew D. Dinmore and C. Curtis Boylls. Empirically-observed end-user programming behaviors in Yahoo! Pipes. In *Psychology if Programming Interest Group*, Spain, 2010.

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

[20] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. M. Burnett, R. K. E. Bellamy, J. Lawrance, and I. Kwan. An information foraging theory perspective on tools for debugging, refactoring, and reuse tasks. *ACM Transactions on Software Engineering and Methodology*, 22:14:1–14:41, 2013.

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

[22] L. Grammel and M.-A. Storey. An end user perspective on mashup makers. Technical Report DCS-324-IR, Department of Computer Science, University of Victoria, 2008.

[23] V. Grigoreanu, J. Brundage, E. Bahna, M. M. Burnett, P. Elrif, and J. Snover. Males' and females' script debugging strategies. In *International Symposium on End-User Development*, pages 205–224, Germany, 2009.

[24] V. Grigoreanu, M. Burnett, S. Wiedenbeck, J. Cao, K. Rector, and I. Kwan. End-user debugging strategies: A sensemaking perspective. *ACM Transactions on Computer-Human Interaction*, 19(1):5:1–5:28, 2012.

[25] 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.

[26] Paul Gross and Caitlin Kelleher. Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages Computing*, 21(5):263 – 276, 2010. Part Special issue on selected papers from VL/HCC'09.

[27] P. Jaccard. Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines. *Bulletin de la Société Vaudoise des Sciences Naturelles*, 37:241 – 272, 1901.

[28] M.C. Jones and E.F. Churchill. Conversations in developer communities: A preliminary analysis of the Yahoo! Pipes community. In *Proceedings of the International Conference on Communities and Technologies*, pages 51–60, USA, 2009.

[29] Shaun Kaasten and Saul Greenberg. Integrating back, history and bookmarks in web browsers. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Extended abstracts*, pages 379–380, USA, 2001.

[30] M.R. Karam and T.J. Smedley. A testing methodology for a dataflow based visual programming language. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 280 –287, Italy, 2001.

[31] A. J. Ko and B. A. Myers. Designing the Whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 151–158, 2004.

[32] 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. *Transaction of Software Engineering*, 32(12):971–987, 2006.

[33] Andrew J. Ko, Thomas D. Latoza, and Margaret M. Burnett. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Softw. Engg.*, 20(1):110–141, February 2015.

[34] A. Koesnandar, S. Elbaum, G. Rothermel, L. Hochstein, K. Thomasset, and C. Scaffidi. Using assertions to help end-user programmers create dependable web macros. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pages 124–134, 2008.

[35] T. Kulesza. Toward end-user debugging of machine-learned classifiers. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing* , pages 253 –254, Spain, 2010.

[36] S. K. Kuttal, A. Sarma, and G. Rothermel. History repeats itself more easily when you log it: Versioning for mashups. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 69–72, USA, 2011.

[37] S. K. Kuttal, A. Sarma, and G. Rothermel. Debugging support for end-user mashup programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 1609–1618, 2013.

[38] S. K. Kuttal, A. Sarma, and G. Rothermel. Predator behavior in the wild web world of bugs: An information foraging theory perspective. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 59–66, USA, 2013.

[39] S. K. Kuttal, A. Sarma, and G. Rothermel. On the benefits of providing versioning support for end-users: An empirical study. In *Transaction of Computer Human Interaction*, volume 21, pages 9:1–9:43, 2014.

[40] J. Lawrance, R. Abraham, M. Burnett, and M. Erwig. Sharing reasoning about faults in spreadsheets: An empirical study. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 35–42, UK, 2006.

[41] J. Lawrance, R. Bellamy, and M. Burnett. Scents in programs: Does information foraging theory apply to program maintenance? In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 15–22, 2007.

[42] J. Lawrance, M. Burnett, R. Bellamy, C. Bogart, and C. Swart. Reactive information foraging for evolving goals. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 25–34, 2010.

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

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

[45] Henry Lieberman, Fabio Paternó, and Volker Wulf. *End User Development*. Springer, first edition, 2006.

[46] J. Nielsen and R. Molich. Heuristic evaluation of user interfaces. In *ACM Conference on Human Factors in Computing Systems*, pages 249–256, 1990.

[47] N. Niu, A. Mahmoud, and G. Bradshaw. Information foraging as a foundation for code navigation: New Ideas and Emerging Results track. In *Proceedings of the International Conference on Software Engineering*, pages 816–819, USA, 2011.

[48] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath. To fix or to learn? how production bias affects developers' information foraging during debugging. In *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution*, pages 11–20, Bremen, Germany, 2015.

[49] D. Piorkowski, A. Z. Henley, T. Nabi, S. D. Fleming, C. Scaffidi, and M. Burnett. Foraging and navigations, fundamentally: Developers' predictions of value and cost. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 97–108, New York, NY, USA, 2016. ACM.

[50] D. J. Piorkowski, S. D. Fleming, I. Kwan, M. M. Burnett, C. Scaffidi, R. K.E. Bellamy, and J. Jordahl. The whats and hows of programmers' foraging diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3063–3072, New York, NY, USA, 2013. ACM.

[51] P. Pirolli. Computational models of information scent-following in a very large browsable text collection. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, pages 3–10, 1997.

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

[53] P. Pirolli and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In

Proceedings of International Conference on Intelligence Analysis, pages 2–5, USA, 2005.

[54] M. B. Rosson and J. M. Carroll. The reuse of uses in smalltalk programming. *ACM Transactions on Computer-Human Interaction*, 3: 219–253, September 1996.

[55] G. Rothermel, L. Li, C. DuPuis, M. Burnett, and A. Sheretov. A methodology for testing form-based visual programs. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, 2001.

[56] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan. Understanding reuse in the android market. In *20th IEEE International Conference on Program Comprehension*, pages 113–122, June 2012.

[57] C. Scaffidi, A. Cypher, S. Elbaum, A. Koesnandar, J. Lin, B. Myers, and M. Shaw. Using topes to validate and reformat data in end-user programming tools. In *Workshop on End-User Software Engineering at the International Conference on Software Engineering*, pages 11–15, Germany, 2008.

[58] S. Scaffidi, A. Dove, and T. Nabi. Londontube: Overcoming hidden dependencies in cloud-mobile-web programming. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2016.

[59] H. Sharp, Y. Rogers, and J. Preece. *Interaction Design: Beyond Human-Computer Interaction*. Wiley, 2007, Ch. 15.

[60] B. Shneiderman. Designing computer system messages. *Communication of the ACM*, 25:610–611, 1982.

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

[62] K. T. Stolee and S. Elbaum. Refactoring pipe-like mashups for end-user programmers. In *Proceedings of the International Conference on Software Engineering*, pages 81–90, USA, 2011.

[63] K. T. Stolee, S. Elbaum, and A. Sarma. Discovering how end-user programmers and their communities use public repositories: A study on Yahoo! Pipes. *Information & Software Technology*, 55:1289–1303, 2013.

[64] Kathryn T. Stolee, Sebastian Elbaum, and Anita Sarma. End-user programmers and their communities: An artifact-based analysis. In *International Symposium on Empirical Software Engineering and Measurement*, pages 147–156, Canada, 2011.

[65] C. Wohlin, P. Runeson, M. Hóst, M.C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Springer, 2000.

[66] Y. S. Yoon and B. A. Myers. A longitudinal study of programmers' backtracking. In *Proceedings of the 2014 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 101–108, 2014.

[67] N. Zang and M. Rosson. What's in a mashup? And why? Studying the perceptions of web-active end users. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 31–38, Germany, 2008.

[68] N. Zang and M. Rosson. Playing with information: How end users think about and integrate dynamic data. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 85–92, Spain, 2009.