RC24783 (W0904-064) April 15, 2009
Computer Science

IBM Research Report

How People Debug, Revisited: An Information Foraging Theory Perspective

Joseph Lawrance[1,2], Christopher Bogart[1], Margaret Burnett[1,2], Rachel Bellamy[2], Kyle Rector[1]

[1]Oregon State Unversity

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

How People Debug, Revisited: An Information Foraging Theory Perspective

Joseph Lawrence[1,2], Christopher Bogart[1], Margaret Burnett[1,2],
Rachel Bellamy[2], Kyle Rector[1]

[1]Oregon State University
{lawrance, bogart, burnett, rectorky}
@eecs.oregonstate.edu

[2]IBM TJ Watson Research Center
rachel@us.ibm.com

ABSTRACT

Many theories of debugging have been proposed over the years, but most were formulated using very small programs, in an era of programming environments that were much simpler than today's environments. Further, because these theories rely on complex mental constructs such as hypotheses, they offer little practical advice to builders of software engineering tools. In this paper, with a practical perspective in mind, we reconsider the notion of how people go about debugging in today's large collections of source code using a modern programming environment. Specifically, we present an information foraging theory of debugging-focused navigation that describes behavior in terms of environmental cues. The theory treats programmer navigation relative to a bug as being analogous to an animal using scent and topology when navigating in the wild in search of prey. It further proposes that these constructs of scent and topology provide enough information to describe and predict programmers' navigation during debugging, without reference to mental states such as hypotheses. To evaluate this premise, we used an executable model of the theory to predict ten professional programmers' navigation of real-world open source programs while they debugged, and analyzed the results in light of what these programmers talked aloud about while they debugged. We found that the programmers' verbalizations as they debugged were overwhelmingly about following scent, pervasively across six debugging modes, much more than their verbalizations of (non-scent) hypotheses. We also found that, although the programmers used numerous resources, source code was the artifact in which most scent processing took place, suggesting that relatively inexpensive algorithms based on static analysis of source code alone are viable in supporting programmers' information seeking activities as they debug. In fact, scent and scent plus topology were both more effective predictors of programmers' navigation than the programmers' stated hypotheses, suggesting that reasoning about artifacts in the environment is not only a more practical mechanism in practice than drawing from verbalizations of mental state, it may also be more accurate. Finally, we discuss our results' implications for ways to enhance software engineering tools according to information foraging principles.

Author Keywords

Information foraging theory, debugging, software maintenance, programmer navigation, information scent, empirical software engineering

ACM Classification Keywords

D.2.5 [Software Engineering]: Testing and Debugging; H.1.2 [Information Systems]: User/Machine Systems—Human factors

1. INTRODUCTION

One of the advantages of theories of programmer behavior is that some can be used to make predictions. Such predictions can be used to guide choices about potential benefits of new tool features, or can inspire new software engineering practices. These predictions are rarely 100% correct, but even the missed predictions can help shape the theory so that it makes more valuable predictions the next time around. In this way, software engineering researchers and tool developers can build on each others' foundations in a principled manner.

This paper presents a theory of programmer navigation when debugging. The programmer navigation aspect of debugging is important, as recent research has shown that programmers spend 35% of their time navigating [Ko et al. 2006]. Existing theories of program debugging do not explicitly consider navigation; instead, they rely primarily on in-the-head constructs such as mental models, comprehension, and hypotheses, to explain the behavior of programmers during program comprehension and debugging. While such theories have been used to explain specific programmer behaviors observed when programmers are working on particular programs, it is hard to make practical use of them to guide the design of software engineering tools and practices for three reasons. First, in-the-head constructs are not accessible to tools. Further, there is no general theory of hypothesis formation that would allow tools to infer entities such as hypotheses, and thus these theories do not directly provide practical guidance for tool developers. Finally, these theories were mostly developed in an age in which programming environments were relatively simple. Modern programming environments provide a plethora of visualizations, clickable links, animations, and other aids, but the use of these devices are not accounted for by the older theories.

Rational Analysis [Anderson 1990] may hold the key to solving all three of these problems. It suggests a basis for theories of programmer behavior that does not need knowledge of what is inside a programmer's head. Rational Analysis assumes that expert behavior is optimally adapted to the structure of the *environment*. It allows researchers to infer what a person adapted to an environment will do, subject to (1) the person's goals, (2) the costs and benefits of actions in the environment, and (3) a modest set of resource constraints known to apply to the brain's computational capacities. Applied to debugging, it implies that experts will make the best possible navigational choices, given the information the environment makes available to them at each moment. Anderson has shown good results in this approach to human behavior involving problem solving, categorization, and causal inference.

Information foraging theory is an example of a rational analysis that has emerged in the last decade as a way to explain how people seek, gather, and make use of information. Like all rational analyses, information foraging theory assumes that humans have evolved to be well adapted to the excessive information in the world around them, and that they behave accordingly in information spaces. The basic idea is that, given the plethora of available information that is not relevant to their current needs, humans have evolved strategies that allow them to efficiently find

information relevant to their greatest needs without processing everything—in essence, minimizing the mental cost to achieve their goals.

Information foraging theory [Pirolli and Card 1999] is based on optimal foraging theory, a theory of how predators and prey behave in the wild. In the wild, predators sniff for the prey, and follow the scent to the patch where the prey is likely to be. Applying these notions to the domain of information technology, predators (people in need of information) sniff for the prey (the information itself), and follow the scent through cues in the environment to the information patch where the prey seems likely to be. Information foraging theory has been shown to mathematically model which web pages human information foragers select on the web [Chi et al. 2001], and as a result has become extremely useful as a practical tool for web site design and evaluation [Chi et al. 2003], [Nielsen 2003], [Spool et al. 2004].

We believe that information foraging theory should apply to how programmers navigate when searching for a bug as well. Specifically, we propose that if a bug is treated as the "prey," words in the environment and the source code are treated as "cues" pointing the way to possible relationships to the prey, and modern programming environments' navigational affordances (such as the ability to mouse over a method call to see its definition) are treated as "topology," then the theory of information foraging can be mapped to the domain of software maintenance and debugging. If information foraging theory applies to programmers' debugging behavior, it has the potential to provide a parsimonious and easily understood model to guide the efforts of tool builders. Thus, a tool designer would have a theoretically well-grounded way of evaluating design choices about the navigational devices and cues in a proposed software debugging tool.

In previous research, we have contributed executable models inspired by this theory, and have shown that these models can predict effectively where expert programmers navigate when debugging [Lawrance et al. 2008b] and can also be used to predict where expert programmers *need* to navigate in order to fix bugs [Lawrance et al. 2008a]. In this paper we present the theory itself. We then use this theory to provide a new perspective on how programmers debug in modern programming environments, and empirically validate this perspective.

The contributions of this paper therefore are: (1) A theory of information foraging for programmer navigation behavior while debugging; (2) an empirical comparison of the prevalence of scent-following versus (non-scent) hypotheses processing in debugging; (3) a detailed empirical analysis of how information foraging activity pertains to six debugging "modes"; (4) an empirical analysis of which artifacts are needed the most by tools attempting to capitalize on information foraging theory; (5) an empirical evaluation of two variants of information foraging models compared to stated hypotheses as predictors of where programmers navigate; and (6) an analysis of the relationship between information foraging theory for debugging and other theories of debugging in software engineering literature.

2. AN INFORMATION FORAGING THEORY OF PROGRAMMER BEHAVIOR DURING DEBUGGING

Because the application of information foraging theory to debugging is novel, it was necessary to map the concepts of information foraging theory to debugging.

A central aspect of developing a theory is choosing the right constructs. A small number of constructs may improve the theory's parsimony, but at the expense of explanatory power or scope. Further, the potential utility of the

theory to future tool builders requires a manageably small set of intuitive constructs that together produce reasonably accurate results.

The constructs of our theory of information foraging theory in the domain of debugging are:

- *Prey*: What the programmer seeks to know: in general, the changes that need to be made to fix the bug.
- *Predator*: The programmer.
- Information *patches*: The places in the source code, related documents or displays in which the prey or proximal cues might hide.
- *Proximal cues*: Words, objects, and perceptible behaviors in the programming environment that suggest scent relative to the prey. Cues are signposts. Unlike scent, cues exist only in the environment.
- *Information scent*: The perceived likelihood of this cue leading to prey, either directly or indirectly. Includes the property that it can be compared to the scent of other cues; i.e., perceived "relatedness." Unlike cues, scent exists only in the programmer's head. This definition is equivalent to Chi, Pirolli, et al.'s definition of information scent as "the subjective sense of value and cost of accessing [information] based on perceptual cues" [Chi et al. 2001].
- *Topology*: The collection of paths through the source code and related documents or displays through which the programmer can navigate efficiently.

In order to test the validity of the theory, we used an executable model, PFIS (Programmer Flow by Information Scent). The PFIS algorithm and its predictive power have been presented in previous work [Lawrance et al. 2008b]. This section summarizes the PFIS model and its relationship to the theoretical foundation. In Section 5 we include PFIS-produced data in our answers to our research questions.

Some of the theoretical definitions above are sufficiently concrete to make obvious the operational definitions needed for modeling purposes. For the domain of debugging, we operationalize our definition of a patch as a group (such as a package) of (Java) classes, and we operationalize the definition of prey to be the places where code changes are to be made. Of course the predator is the programmer(s) being modeled.

However, the notions of topology, cues, and scent do not map so obviously to operational definitions. The operational definitions of these constructs that we implemented in PFIS are:

- *Topology*: A directed graph through the source code and its related documents or displays to which it is possible to navigate by following a series of links. Its vertices are elements of the source code (e.g., classes, methods) and environment (e.g., class labels enumerated in a class hierarchy browser), and its edges are links.
- *Link*: A connection between two nodes in the topology that allows the programmer to traverse the connection at a cost of just one click. (Environment and context-dependent.)
- *Proximal cue*: Words appearing near the head of a link.
- *Scent*: Word similarity between the issue text (description of the prey) and the proximal cue.

The measure of scent currently used by PFIS warrants further explanation. Since information scent is the programmer's (imperfect) perception of the value (relatedness) of information, as in Pirolli's information foraging research for web searching [Pirolli 1997], our measure of information scent is word similarity between the description of the prey (e.g., issue text) and the proximal cues in the source code files, measured by cosine similarity in a vector

space. Note that this operational definition is the model's approximation of scent; the "true" measure of scent is only in the programmer's head.

To implement this approximation, first, we developed a special tokenizer for words in source code, so that camel case identifiers (e.g., "NewsItem.getSafeXMLFeedURL()") would be split into their constituent words ("news item get safe xml feed url"). We also employed a standard stemming algorithm on the constituent words. Second, terms in files of source code were weighted according to the commonly-used term-frequency inverse document frequency (TF-IDF) formula [Baeza-Yates et al. 1999].

$$w_{i,j} = f_{i,j} \times idf_i$$

where $f_{i,j} = \frac{freq_{i,j}}{\max_{\forall v} freq_{v,j}}$ and $idf_i = \log \frac{N}{n_i}$

Here, $f_{i,j}$ is the frequency of word i in document j (normalized with respect to the most frequently occurring word v in a document), idf_i is the inverse document frequency, and $w_{i,j}$ is the weight of word i in document d_j.

Finally, the interword correlation between proximal cues in source files and the text of a bug report is computed via cosine similarity, a measure commonly used in information retrieval systems, shown below [Baeza-Yates et al. 1999].

$$sim(d_j, q) = \frac{d_j \cdot q}{\|d_j\| \times \|q\|} = \frac{\sum_{i=1}^{t} w_{i,j} \times w_{i,q}}{\sqrt{\sum_{i=1}^{t} w_{i,j}^2} \times \sqrt{\sum_{i=1}^{t} w_{i,q}^2}}$$

where $w_{i,q}$ is the weight of word i in the bug report text (i.e., the query in the terminology of [Baeza-Yates et al. 1999]).

Given these operational definitions of constructs and measures, the basic idea behind PFIS is to start by gather-

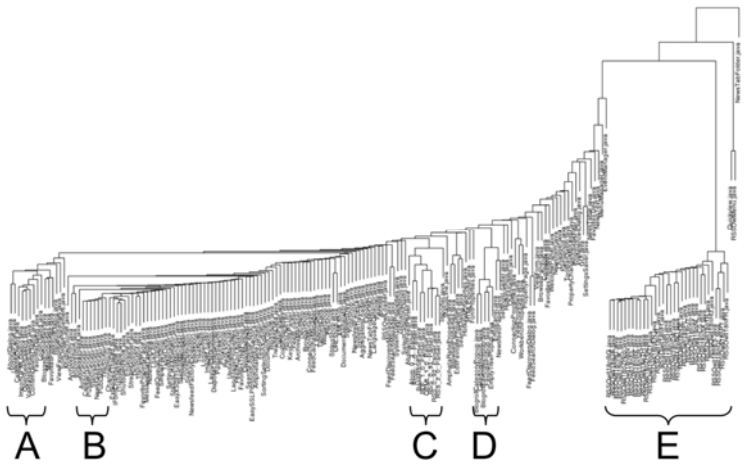

Figure 1. Dendogram of patches grouped by scent using cluster analysis. For example, "A" shows the clustering by scent of the dialog box code files, "B" the sorting code files, "C" the parsing code, "D" the popups, and "E" the internationalization.

ing topological information, then to calculate scent (relative to the prey) of each procedure/method's cue. Finally, in order to simulate the proportion of programmers that will follow each link in the topology, PFIS propagates the scent through the topology using the spreading activation technique [Anderson 1983], [Crestani 1997].

Using these operationalizations, PFIS predicts programmers' visits during debugging to "reachable" source code with the highest scent in relation to the bug report. Our results have been encouraging. We have empirically seen that our operational definitions of scent and cues resulted in related source code files clustering together, indicating that information patches were indeed present in this source code [Lawrance et al. 2007]. The dendogram in Figure 1 points out several examples from that study. Further, our results suggest that PFIS' performance is close to aggregated human decisions as to where to navigate, and is significantly better at predicting a programmer's navigation than the navigation log of a second programmer would be [Lawrance et al. 2007], [Lawrance et al. 2008b], [Lawrance et al. 2008a].

3. EMPIRICAL STUDY

PFIS predicts programmer behavior, yet previous research (detailed in Section 7) emphasizes the role of hypotheses in governing programmer behavior. One view that brings these seemingly contradictory perspectives together is that a scent pursued is the equivalent of a low-level hypothesis: searching for something entails a hypothesis that that thing exists and will be worth pursuing. In these cases, scent amounts to a low-level hypothesis.

Figure 2 shows an abstract model of how this and a similar sort of hypothesis may be related to scent. In the case described above, the scent-related behavior triggers hypothesis formation, modification, confirmation, or abandonment. In the other case, a hypothesis triggers (further) scent-seeking. The underlying assumption is that, when programmers are working with code they did not personally write, their hypothesis formation and/or follow-up relies heavily on the immediate environmental cues of scent. Scents gained and lost influence scents sought, and much of the hypothesizing sandwiched between these scent behaviors amounts to either choosing among a limited number of high-scent options or generating new pursuits of scent. If this model is correct, that would imply that analyzing and predicting scent-following behavior should be sufficient for predicting programmers' behavior during debugging, because traversing scents' in- and out-edges also cover the hypothesis-related edges.

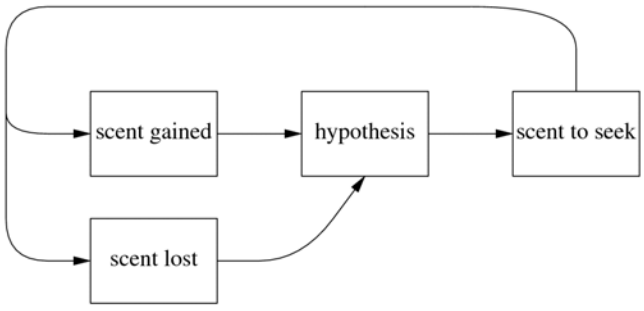

Figure 2: An abstract model of how hypotheses and scents may be related. The edges denote "leads to." There is no start point to this graph; the assumption is that a programmer might begin at any of these nodes.

This study investigates the validity of this premise through analysis of verbal protocols collected from our participants, focusing on what those verbalizations can tell us about the roles of hypotheses and scent in our theory and

in operational derivatives of it such as PFIS. Our first research question is therefore how scent processing activities compare to and relate to hypothesis processing activities. Our second research question then adds a "when" perspective, considering how these activities relate to six different debugging modes. Our third research question adds a "where" perspective, considering the types of artifacts through which programmers navigate when debugging, and in which artifacts programmers are in when they seek scent. Our final research question then compares two ways one might predict programmer navigation behavior based on information foraging theory against predictions that might be made from programmers' stated information needs and hypotheses.

3.1 Design, Participants, and Materials

The study design used the think aloud method. Participants were instructed to talk aloud as they worked, and we automatically captured their screens and their actions at the same time. Our goal was to gather enough rich data from participants to be able to perform an in-depth qualitative analysis of their words and actions. The length of the study period was designed to allow a high enough number of actions to quantitatively consider our model's predictions in terms of precision and recall.

We recruited twelve professional programmers from IBM. We required that each had at least two years experience programming Java, used Java for the majority of their software development, were familiar with Eclipse and bug tracking tools, and felt comfortable with searching, browsing, and finding bugs in code for a three-hour period of time.

To realize our design goals, we searched for a program for participants to work on that met several criteria: it needed to be a real-world program, we needed access to the source code and to descriptions of its bugs, it needed to be written in Java, it needed to involve enough files to allow enough navigation for useful analysis, and it needed to be editable and executable through Eclipse, a standard Java IDE. We selected RSSOwl, an open source news reader that is one of the most actively maintained and downloaded projects hosted at Sourceforge.net. RSSOwl's source code consists of 193 class files. The popularity of newsreaders and the similarity of its UI to email clients meant that our participants would understand the functionality and interface after a brief introduction, ensuring that our participants could begin using and testing the program immediately.

Having decided upon the program, we also needed issue reports for our participants to work on. In making our selections, we wanted to ensure that our participants would likely require more time to fix the issues than we gave them. We also decided that one issue should be about fixing erroneous code and the other about providing a missing feature. From these requirements, we selected two issues: 1458101: "HTML entities in titles of atom items not decoded" and 1398345: "Remove Feed Items Based on Age." We will refer to the first as Issue B ("Bug") and the second as Issue MF ("Missing Feature"). Each participant worked on both issues, and we counterbalanced the ordering of issues among subjects to control for learning effects. The former involves finding and fixing a bug, and the latter involves inserting missing functionality, requiring the search for a "hook," i.e. a part of the code to copy, modify, or link in the new functionality.

The issues we assigned to developers were open issues in RSSOwl. We considered looking at closed issues whose solution we could examine, but this would have meant locating an older version of RSSOwl for participants to work on, and would have required us to ensure that participants would not find the solution accidentally by browsing the web. Therefore, we decided that our participants would work on open issues, cognizant of the risk that

RSSOwl's own developers could close the issues during the study, updating the web-available solution with the correct code in the process. Fortunately, this did not happen.

3.2 Procedure

Upon their arrival, after participants filled out initial paper work, we briefly described what RSSOwl is, and explained to our participants that we wanted them to try to find and possibly fix issues that we assigned to them. We set up an instant messenger client so that participants could contact us remotely. Then we excused ourselves from the room. We observed each participant remotely for two hours.

We recorded synchronized audio, video, and screen captures, which we were able to replay together, allowing us to hear what they said, while observing their facial expressions and the screens they were viewing. We also logged their events. Participants had been instructed to think aloud, and were reminded to do so if they fell silent for an extended period. We archived the changes they made, if any. The electronic transcripts of their actions, videos with screen captures, and source code served as the data sources we used in our analysis.

4. ANALYSIS METHODOLOGY

We discarded data from two of the twelve participants: one was used as a pilot, and the data was corrupted for another. We performed three stages of analyses of the verbal protocol data. The first stage was performed on the ten participants' explicit verbalizations. Its purpose was to categorize the participants' explicit verbalizations about hypothesis processing and scent following. The second stage was also performed on the ten participants using both the verbalizations and the videos. Its purpose was to categorize the artifacts at which participants were looking when they made certain types of statements. The third stage was an extremely fine-grained analysis, and was performed on two of the participants' verbalizations and videos. The details of each analysis methodology are described below.

For the first stage of analysis, we coded all ten participants' verbal protocols in the following manner. Initially, codes for this stage were developed by four researchers as they jointly coded two of the protocols. They cooperatively refined the initial code sets and developed norms about how to apply them. All of the remaining protocols were coded by two researchers working independently and then comparing results. Thus, every protocol was coded by at least two researchers. The total agreement value was 97%, which indicates extremely high coding reliability. Reliability was even higher (99%) when we excluded from the calculations the code "no code," i.e., verbalizations deemed by the coders to be neither about hypotheses nor scent.

For hypotheses, we coded when participants formed hypotheses, modified hypotheses, confirmed hypotheses, or abandoned hypotheses. We used the hypothesis codes on verbalizations that were explicitly about the part of the code or application behavior implicated in the bug or the fix. These hypothesis codes are defined in the first four rows of Table 1. For scent, we coded participants' statements about what scent to look for, when they gained scent, and when they lost scent. Only three codes were needed here, instead of four, because we did not try to discriminate scents that were and were not modified variants of previous scents sought. These codes are defined in the last three rows of Table 1.

Code	Definition	Example
hypothesis-start	Suggesting what part of the code or application behavior is implicated in the bug.	82: “So it’s kind of—the apostrophes are not making it in.”
hypothesis-modify	Changing or refining their original hypothesis.	87: “So this newsfeedtitle string data value needs to be escaped.”
hypothesis-confirm	Deciding their hypothesis was correct.	911: “So that’s definitely the right spot.”
hypothesis-abandon	Deciding their hypothesis was not correct.	85: “And actually, now we don't need to figure out that it’s valid.”
scent-to-seek	Stating a need for certain kind of information.	96: “So I need to find something about feeds.”
scent-gained	Finding a scent (that they may or may not have been seeking)	98: “Let’s figure out—oh, here’s gui i18n translation.”
scent-lost	Losing the scent.	84: “This is all just date stuff.”

Table 1. Main codes identifying hypotheses and scent, and examples of participant verbalizations.

For the second stage, categorizing artifacts, we developed a set of “trigger” codes to identify artifacts that triggered participants’ decisions to pursue a hypothesis (hypothesis-start) or a scent (scent-to-seek). The trigger codes are listed in Table 2. Since the trigger codes did not require *interpretation* of the videos, we used a simpler coding process than on the main code set. We devised the codes by simply enumerating the artifacts/assets we expected (the plaintext entries in the table). Coding of the first two videos identified two more categories (italicized in the table). We also allowed “other” in case a coder ran across any more, but there were very few uses of that code. We then tested the scheme’s robustness by having two researchers code two participants for each of the bugs, i.e., 20% of the data set. At this point we reached an agreement of 93%, indicating that the codes were robust. At that point, one of the researchers applied the verified codes to the rest of the videos.

Code	Definition
Debugger-menus	Looking at debugger menus. (Debugger menus enumerate data field names, etc.)
Input-file	Looking at the input file.
Issue-text	Looking at the bug report or feature request.
Runtime	Looking at runtime behavior.
Source-code	Looking at source code.
UI-static-inspection	Looking at the UI “statically” to identify pieces of functionality, the structure of the UI, etc., (as opposed to runtime behavior).
Web	Browsing or searching web pages.
Other	Looking at anything else.

Table 2. Trigger codes: Artifacts at which participants were looking when they expressed hypothesis-start or scent-to-seek verbalizations.

The third stage of analysis was very fine-grained, so we restricted it to two participants’ performance of both tasks, a total of four tasks. We selected one participant (85) for whom our first stage of analysis revealed that hy-

pothesis activity and scent activity peaks often coincided and one (98) for whom hypothesis processing and scent processing activity peaks did not coincide often. We also wanted participants who tackled the bugs in different orders, and Participant 85 worked on the Issue MF first and then Issue B, whereas Participant 98 worked on Issue B first and then Issue MF. Our purpose for this analysis was to understand in a detailed manner why and when participants were processing hypotheses and acting upon scents. For each of these participants, three of the researchers watched the videos, taking into account what the programmers said their goals were, what actions they took, what artifacts they interacted with and how, and their voice inflections and facial expressions. The researchers worked in tandem, watching the videos together and discussing them in detail moment by moment. The three researchers worked together through three of the four tasks, and two researchers finished up the last task together.

5. RESULTS

We have pointed out that programming environments such as Eclipse enrich the programming environment topologically in order to reduce the time cost for programmers to navigate through the code and find related information. This could shift the cost/benefit balance for the programmer away from heavy reliance on forming hypotheses up front about how to fix the bug, toward navigating within the program in a more information-directed way. In previous results, we have shown that our information foraging model does indeed predict programmers' navigation behavior well [Lawrance et al. 2008b].

We suggested in Section 3 that the reason for these results is that in many cases, scent processing behavior may be in part the way that new hypotheses are formed, existing hypotheses are modified, and unsuitable hypotheses are abandoned. In order to investigate this premise, in this section, we present empirical analyses aimed at understanding exactly how programmers used the results of their information foraging to guide their debugging navigation behaviors. Specifically, we (1) analyze the extent of information foraging navigation behavior and the extent of verbalized hypothesis processing; (2) analyze *when* information foraging behavior and hypotheses arose by considering each of six observed debugging "modes" that arose in our participants' data; (3) analyze *where* the participants sought scent; and finally, (4) compare the extent to which verbalized hypotheses vs. scent alone vs. the combination of scent plus topology accounted for the observed behaviors.

5.1 Hypothesis Activity and Scent Activity

Can a scent-based model of debugging navigation succeed if it does not explicitly handle people's hypothesis processing? We propose that the answer is yes if two conditions are met: (1) some hypothesis content closely parallels scent content and therefore is accounted for in our model, and (2) the remaining amount of hypothesis-based processing is relatively small compared to the amount of scent processing that is going on.

Regarding the first condition, we did find that participants' statements sometimes showed a close relationship between hypotheses and scent. When there was a close relationship, it often reflected a bottom-up hypothesis modification that was based on a scent gained in the code, or sometimes reflected a hypothesis that led top-down to a new scent to look for. For example, after a while working on Issue B, Participant 82 gained scent on getTitle:

> 82: "*GetTitle, there we go!*," followed by a hypothesis modification: "*so this is the problem…we need to turn this into HTML*," followed directly by a scent to seek: "*so now the question is how do we test this to see if it's HTML?*"

Note that not all scent processing verbalizations led to hypotheses. There were numerous examples of scent verbalizations without hypothesis verbalizations. For example, Participant 82's early work on Issue B involved scent alone as he tried to gain a better understanding of the way RSSOwl was organized:

82: "GUI, where is that? Okay, GUI RSSOwl tab folder. So is it something in this directory?"

As another example, as Participant 99 browsed through code while working on Issue B, he discovered the class "entity resolver." This discovery triggered a scent-seeking diversion unrelated to any expression of hypotheses, starting with the statement:

99: "I have no idea what an entity resolver is."

These results make clear that although some instances of hypothesis processing and scent processing were intertwined (and therefore are accounted for explicitly by our model), not all of them were intertwined.

Regarding the second condition, whether the remaining amount of hypothesis processing is relatively small compared to the amount of scent processing, we found that this was indeed the case for our participants. Table 3 compares the numbers and patterns of hypothesis and scent verbalizations over time, and as the table shows, verbalizations about scent occurred about four times as often as verbalizations about hypotheses: about 1,000 verbalizations of scent, compared to about 250 verbalizations of hypotheses. This was true not only in aggregate—it was true of every participant for both issues. Profiles of each participant's hypothesis and scent verbalizations are graphed in Figure 3.

In fact, our participants expressed few new hypotheses: sometimes only one or two original hypotheses per issue. Half the hypothesis starts were later followed by hypothesis abandons. This effect was not the same for Issue B as for Issue MF, however. In fact, participants confirmed hypotheses *only* when working on Issue B, but they modified hypotheses more frequently when working on Issue MF. Potential differences in participant talkativeness did not account for this, since the differences in hypothesis verbalizations did not occur with scent verbalizations. For both Issue B and Issue MF, although participants started only a few new hypotheses, they made a lot of modifications to the hypotheses they started. Thus, it appears that once they had a hypothesis, these participants explored that initial hypothesis in depth.

	B	MF	Total
Hypothesis start	19	13	32
Hypothesis modify	75	116	191
Hypothesis confirmed	12	0	12
Hypothesis abandon	10	7	17
Scent to seek	235	242	477
Scent gained	220	204	424
Scent lost	71	63	134

Table 3. Total number of statement instances by type.

Unlike hypotheses, which our participants abandoned more often than they confirmed, with scent, participants gained it more frequently than they lost it. Participants lost the scent they were tracking only about a quarter as often as they sought scent. On the other hand, scents gained were almost as frequent as scents sought. However these were not necessarily the same scents; scent gains were often serendipitous. Another difference between scent and hypotheses activity was that each participant made roughly the *same* number of each kind of scent-related verbaliza-

tions for Issue B as for Issue MF. Scent processing thus seemed far more similar for Issue B and Issue MF than was the case for hypothesis processing.

Eisenstadt's data on real-world programmers' debugging experiences [Eisenstadt 1993] are consistent with our data regarding the prevalence of scent following activity over hypothesis-oriented activity. (Eisenstadt's work is one of the few classic works in which the researchers did not have a hypothesis-oriented focus when investigating how people debug.) He harvested 78 real programmers' self-reports (anecdotes) of how they had gone about debugging recently in their real-world lives. Eisenstadt categorized these anecdotes' "bug-catching techniques" into four categories, one of which amounts to scent-like pursuit of information. He called this one "gather data," which he defined as a bottom-up activity in which "informants may have had a rough idea of what they were *looking for* (emphasis added), but were not explicitly testing any hypotheses in a systematic way." He distinguished this from "controlled experiments," which were hypothesis oriented. He reported 27 occurrences of "gather data," compared to only 4 of "controlled experiments"; a similar result to ours. More recent works focused on software maintenance tasks such as debugging, although they have not actually measured scent, have explicitly emphasized the proportion of such tasks spent browsing through code in pursuit of relevant information (e.g., [Robillard et al. 2004], [Singer et al. 2005], [Ko et al. 2006]).

5.2 When: A Fine-Grained Analysis by Debugging Mode

Given that the data showed such a large amount of scent following, we wanted to find out why. To gain insights

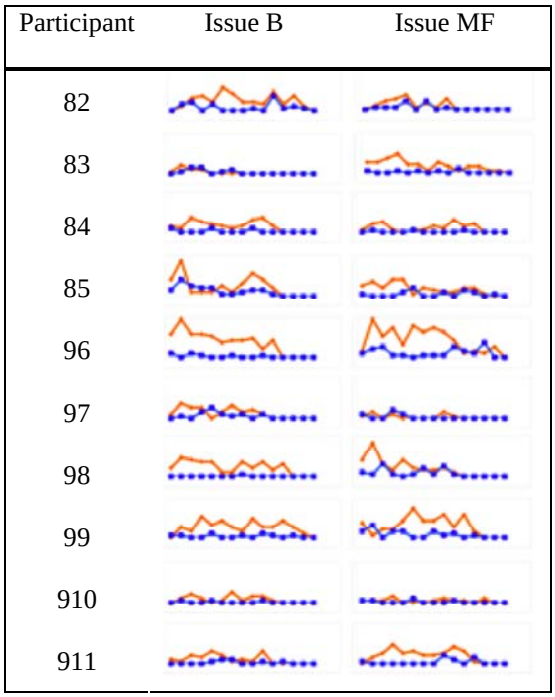

Figure 3. Thumbnails of each participant's hypothesis (dark blue) and scent (light orange) verbalizations over time. The x-axes are time (0 to 70+ minutes, in 5-minute intervals), the y-axes denote counts of verbalizations of each type (0 to 18).

into the "why," we examined when in their debugging activity programmers followed scent and when they worked on hypotheses. This was to see if scent following pervaded all kinds of debugging activities and how it compared to hypotheses activity during different modes of debugging.

To consider this question, we performed a very fine-grained analysis of Participant 85 and Participant 98. Recall from Section 4 that we selected these two participants in part because of the apparent differences in the relationships between their hypothesis and scent activities; these differences can be seen in Figure 3. The fine-grained analysis had two goals: to scrutinize the videos closely to identify instances of hypotheses and scent that were not apparent from the verbalizations alone, and to characterize the context in which participants' hypotheses and scent following occurred.

For this fine-grained analysis, we took into account not only the participants' explicit words, but also their actions, facial expressions, and so on. This led to the identification of additional instances of both hypotheses and scent that had not been present in participants' explicit words alone. The most prevalent of the additional instances were scent following through use of the pop-up (in tool tip form) method documentation. Whereas in older programming environments, looking up documentation for a method would have required opening another file and scrolling around, in Eclipse a mere mouse-over quickly brings up documentation. Our participants rarely even mentioned this functionality in their words, but used it extensively in their actions to follow up on scents. We did not use these actions in the verbalization-based analyses, which are more conservative since they relied solely on utterances in which participants made their intent explicit. However, the data from both types of analyses are consistent: although the fine-grained analyses increased the raw counts of both hypotheses and scent, it did not change the relative proportions of hypotheses to scent.

We categorized the major contexts in these participants' videos into six categories that emerged from the data. These contexts were major activities, covering almost the entire data set, and we therefore characterize them as debugging *modes*. The modes, which are summarized in Table 4, were:

(1) The *Mapping* mode*:* Program understanding is widely understood to be an important part of debugging (e.g., [Ko et al. 2006], [Nanja and Cook 1987], [Vans and von Mayrhauser 1999]). The type of program understanding that stood out in our participants' data was their attempts to build a mental model ("map") of the program. Both participants started off by mapping, not only in their first task, but also in their second one. This mode was charac-

Mode	Issue B	Issue MF
Mapping	Getting a whole-program overview.	
Drill-down mapping	Getting a detailed understanding of one particular portion of the code.	
Observing the failure	Trying to witness the bug or misbehavior at runtime.	Running the application to see where and how the new feature would manifest itself at runtime.
Locating the fault	Trying to find the location in the code that needs to be changed	
Fixing the fault	Fixing the bug.	Adding the new feature, including necessary refactoring.
Verifying the fix	Evaluating the changes just made to determine if the fix was a good solution.	

Table 4. The six primary modes observed in the bug and feature tasks.

terized by participants scanning proximal cues, noticing beacons [Brooks 1983], and taking inventory of the information patches. Participant 98 summarized this mode succinctly when he said:

> *98: "Let me see what the project is made out of."*

We believe that this mode was necessary to even begin to follow scent, because it provided the information participants would need in later modes to interpret proximal cues' scent, meaning, and relative importance.

(2) The *Drill-down mapping* mode: Participants worked harder to build a detailed mental model of some patches in the source code than others, drilling down into the details, and we termed those periods of time as "drill-down mapping." For example, during this mode, Participant 98 looked at constructors, reading the detailed comments and trying to figure out exactly what happened when a new instance was created. As with mapping, both participants used drill-down mapping in both tasks.

(3) The *Observe-the-failure* mode: It is common in debugging for programmers to try to replicate the failure, and we termed this mode "observe the failure." (Eisenstadt also reported this mode in his "gather data" category [Eisenstadt 1993].) In our study, an example failure had been included in Issue B's bug report, but detailed instructions for actually making the code fail were not present, and we noticed that some participants invested a great deal of effort in figuring out how to re-create the failure so as to observe the program carefully. In our fine-grained analysis, this occurred in three of the four task instances we analyzed. Observing the failure included not only figuring out exactly what kind of data to provide to make the failure happen, but also figuring out where to install breakpoints and the like to be able to observe the failure. From an information foraging perspective, at this point the (interim) prey was the failure, not the fault. For example, when trying to get the software to display the garbled character described in Issue B, Participant 98 said:

> *98: "I'm looking at CrookedTimber and I click on some of the items. I'm not seeing any examples of what they are talking about in the bug."*

(4) The *Locate-the-fault* mode: We termed the participant to be in the "locate the fault" mode when the prey they were currently seeking was the precise location in the code of the fault. In the case of a feature request, we assigned this mode when the prey was the "hook" in the code where changes should be made, even though there was no fault as such. Participant 98 was in drill-down mode when he found a properties page, and went into *locate-the-fault* mode as he started exploring the code on the theory that he should copy it and use it as a hook to add the requested feature:

> *98: "So this looks good; this is a properties page. Is there a view tab? Here's a view one. Yeah. Which one of these is different? I would have to add one of those."*

(5) The *Fix-the-fault* mode: Once a participant located the fault, it was not always obvious how exactly to fix it. In fact, for the three of the four task instances in which a participant got to this stage, it occupied far more time than all other modes combined. We termed a participant's efforts in actually devising a fix to a fault they had already located to be in "fix the fault" mode.

Why did the fix mode take so long? These tasks were reasonably challenging, and there were many decisions to be made about how exactly to implement a reasonable solution, any one of which could lead down a path that ultimately had to be undone. For example, Participant 85's fix to implement the new feature for Issue MF required extensive refactoring of existing code. The procedure was supported by the environment's refactoring tools, but these

tools also introduced new problems. In all, Participant 85 spent about an hour on the refactoring aspects of the fix, including solving bugs introduced by the refactoring. He finished the refactoring aspect just before time ran out, so was never able to spend much time on remaining aspects of the fix. Participant 98, on the other hand, spent considerable time deciding how to proceed, then began entering new code to implement the fix. After about 15 minutes invested in this new code, however, he decided he had been going about it the wrong way, and removed most of his new code and then started a different approach. He ran out of time before being able to pursue the second approach very far.

(6) The *Verify* mode. It has long been known that, after making a fix, novice and expert programmers alike evaluate whether their fix actually did correct the problem [Nanja and Cook 1987]. In our study, only one of our participants got his fix complete enough to evaluate it, which he proceeded to do, and this caused him to then change his fix, and then evaluate it again.

Using these modes, the graphs in Figure 4 show activity of hypotheses and scent for each of these participants and tasks, mode-by-mode, with each mode on a separate row. The thickened horizontal gray bars show the length of time each participant spent in that mode. The blue sticks below the gray bars denote hypothesis verbalizations, and the red sticks above the lines denote scent seeking (verbalizations and actions to request the pop-up explanations). Longer sticks indicate two or more events of the same type in quick succession.

The final row, “other,” was used to account for activities that could not be classified into one of the six modes. For example, Participant 85 spent about five minutes on “enrichment,” an information foraging concept in which

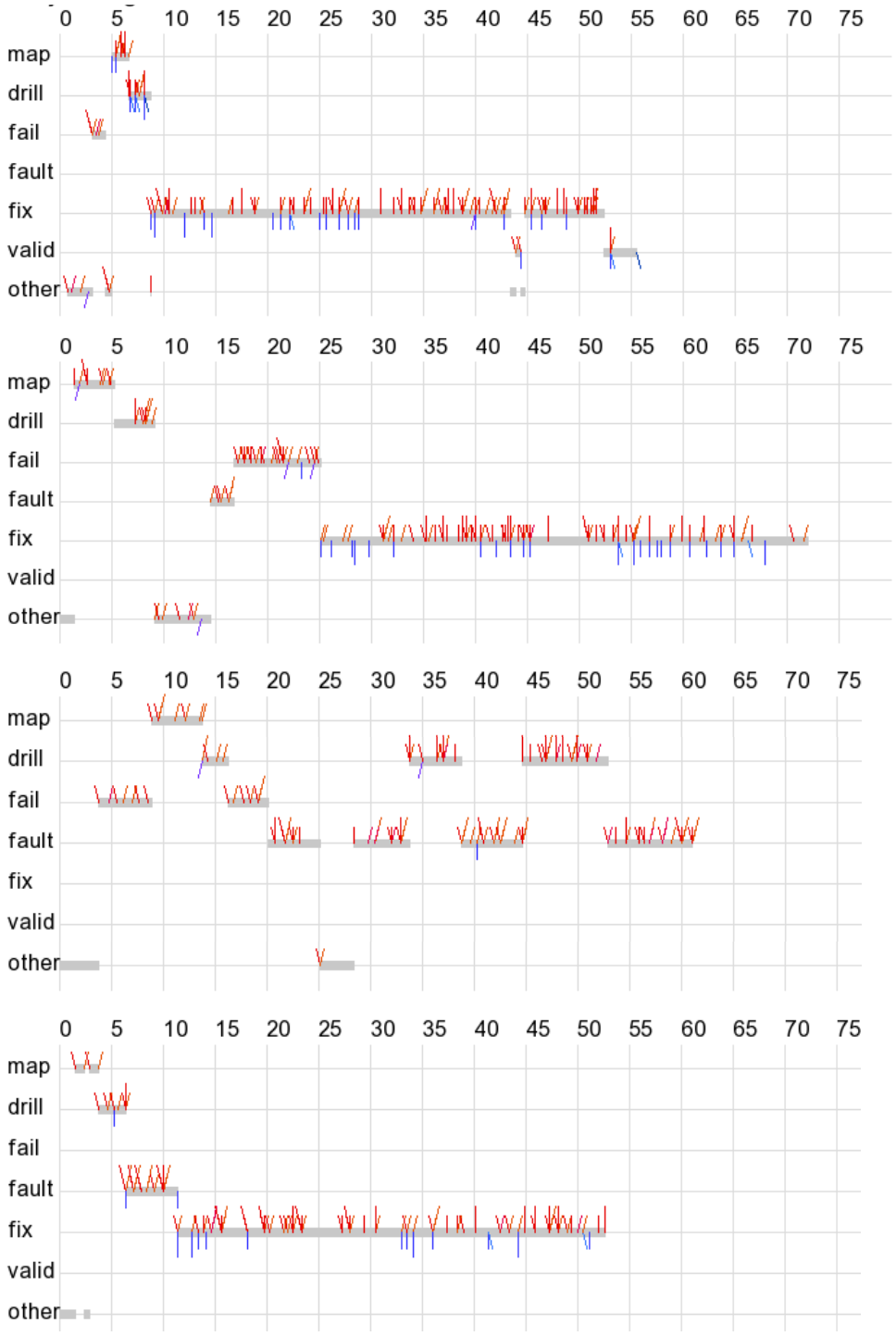

Figure 4: Scent and hypothesis events observed for each mode. Top to bottom: Participant 85 Issue B (done as task #2), Participant 85 Issue MF (done as task #1), Participant 98 Issue B (done as task #1), Participant 98 Issue MF (done as task #2). Gray horizontal bar shows mode is in effect: . Red sticks above the mode bar: Scent to seek: \, combined scent sought/gained via tooltip: |, scent gained/lost /. Blue sticks below the mode bar: Hypothesis start: /, hypothesis modify: |, hypothesis confirm/abandon: \. Longer sticks denote multiple events at one timestamp.

one modifies one's environment in order to make optimizing around the environment's affordances more viable. In this case, Participant 85 was working on making the debugging tool behave in a particular way so that he could proceed with the strategy he had in mind. Numerous other examples of enrichment, sometimes very short such as rearranging windows, were pervasive among all of the modes. (Enrichment was categorized as "other" only if it was relatively independent of any of the six debugging modes.)

As the figure shows, scent seeking was pervasive in all six modes. Recall that Table 3 showed that scent following was expressed much more often than hypotheses, and that Table 3 showed that this phenomenon occurred consistently over time. Figure 4 shows how consistently the phenomenon occurred in all six debugging modes. This was true of both participants, consistently for both issues, regardless of the order in which the issues were tackled.

Hypothesis verbalizations occurred mostly in the "Fix" mode. (Recall that for the purposes of our analysis, hypotheses were defined as being hypotheses about where the bug was lurking or how to fix it.) As we have pointed out, Fix dominated the time spent when it occurred, and this may be why more hypotheses occurred in that mode. It may also help explain why Participant 98 had such a low hypothesis count for Issue B (see Figure 4). Many hypothesis modifications in Fix mode represented the participant's evolving programming plans as they worked on their fix. For example Participant 98 verbalized two contradictory hypothesis modifications in a short span of time:

98: "Let's take some of this stuff out." Removed some code he believed had been obsoleted by his changes. "[…] You know, wait, this stuff has to be there." Used undo to add it back.

Interestingly, the hypotheses initiated and scents sought were not "bookended." In the analysis process, we tried to track hypotheses from initiation through modifications until the hypothesis was confirmed or abandoned, and likewise to track the length of a scent from the time a participant started pursuing it until they found or lost it. However, such nice bookending of hypotheses and scents were not present in our data. Participants often did not find what they sought, but instead some even more interesting scent "found" them, sending them off in a different direction than planned. Participants did *try* to make plans (initiate hypotheses and try to confirm or refute them)—but they were willing to change these plans to adapt to the cues and scents that arose along the way.

This is consistent with Activity Theory [Leontjev 1978], where plans are like high level descriptions of activities that guide behavior but do not specify exact actions or operations; rather the actions or operations are determined by the context in which the action is taking place. It is also consistent with Suchman's theories of situated cognition, in which plans are inherently vague, and the structure of the environment has far more effect on particular actions [Suchman 1987].

The fact that scent processing dominated across all debugging modes is also what Hollan et al.'s thesis of Distributed Cognition would predict. Hollan, Hutchins, and Kirch [Hollan et al. 2000] say that "the organization of mind […] is an emergent property of interactions among internal and external resources." They argue that a large part of cognition is triggered by interaction with the environment, rather than happening predominantly in the head.

Given the consistency of our data with these previous works, the scent orientation participants had across all six debugging modes may be the root of why information foraging theory appears to apply in the domain of debugging.

5.3 Where: In Which Artifacts Participants Sought and Found Scent

Knowing *which* artifacts in the environment triggered participants to decide to pursue a bug is needed to reveal which kinds of artifacts a predictive model needs to analyze in order to predict programmers' navigation behavior.

To investigate this issue for scent, we defined as the scent trigger the artifact at which participants were looking when they expressed a scent-to-seek verbalization (recall Table 2). The results were that, for both Issue B and Issue MF, participants' pursuits of scent were triggered in the source code far more than anywhere else, as Figure 5 shows. Other noticeable triggers included web resources, the runtime behavior of RSSOwl, and the issue itself (Issue B or MF).

The distribution of the scent-seeking triggers was reasonably consistent between Issues B and MF, as can be seen by comparing the light and dark bars in Figure 5. When starting a new hypothesis, however, trigger patterns were less clear because of the low number of hypotheses. However, it appears that, although the issue description itself triggered the hypothesis over a fourth of the time, no trigger dominated the others as source code had for scent seeking. Other than the issue text triggers, the distribution of triggers varied greatly between Issue B and Issue MF (light and dark bars in Figure 6, respectively). For the bug, Issue B, hypothesis triggers were mostly in the issue description, source code, and the program input; for the feature, Issue MF, hypothesis triggers were mostly in the issue and web resources. Comparing Figure 5 to the big graph in Figure 6 shows that the influence of these artifacts on participants' behaviors was dominated by influences on scent-seeking.

These results suggest that, although non-code artifacts did affect participants' navigation behavior to some extent, it is reasonable for a model or tool to rely solely on analysis of source code relative to the issue report to obtain predictions, without incurring the expense of the more costly analyses of runtime data, the web, and so on. This supports the validity of the PFIS modeling approach, which relies solely on static analysis of source code relative to the issue report.

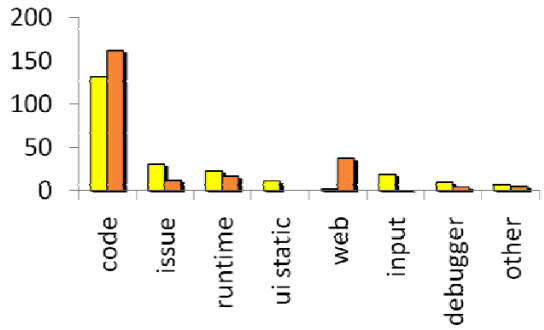

Figure 5. Scent to seek verbalizations: what triggered participants to seek scent. Light bars represent scent triggers for Issue B; dark bars represent scent triggers for Issue MF.

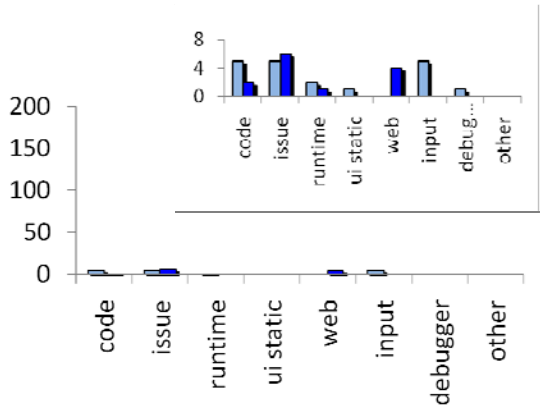

Figure 6. Hypothesis start verbalizations: what triggered participants to form new hypotheses. Light bars represent scent triggers for Issue B; dark bars represent scent triggers for Issue MF. The small inset graph blows up the data to allow comparison of the most influential artifacts on hypotheses. The big graph is at the same scale as Figure 5, showing that the influence of artifacts on hypotheses was overwhelmed by these artifacts' influences on scent.

5.4 Predictors of Where Participants Sought and Found Scent

We now consider four factors as predictors of where programmers *did* navigate and *should* navigate. The first three predict using only the factor and the programmer's location, without taking into account cues programmers may have seen along the way; conceptually, these can be characterized as *distal scent*. These three are: (1) *verbal scent-to-seek* stated by the participants; (2) *verbal hypotheses* stated by the participants; and (3) the *issue report*. The fourth factor is (4) *proximal scent + topology* as computed the PFIS algorithm using scent of each proximal cue relative to the issue report and spreading activation over the topology. For all four of these predictors, we measured "similarity" to the locations in code using cosine similarity via the method explained in Section 2. See Section 6.3 for more discussion of the distinction between distal and proximal scent and their relationship to topology.

First we consider these predictors' ability to predict the source code file to which participants *went* next, at each moment that a participant verbalized an information need (i.e., at each verbalization coded "scent to seek"). This sheds light on these constructs' potential as *behavior predictors*. Second, we consider the quality of each predictor as an "advisor" of where the participant *should go*, by considering each predictor as a measure of where participants pronounced themselves to have found something useful (i.e., at each verbalization coded "scent gained"). This sheds light on the potential benefits of including these predictors in future debugging tools.

The appropriate measure for evaluating how well a predictor chooses among different alternatives is the area under the receiver operating characteristic (ROC) curve [Herlocker et al. 2004]. The ROC curve plots correct predictions versus incorrect predictions, and the area under the curve (AUC) then measures the effectiveness of these predictions. In Figure 7 we show an ROC curve for *verbal scent-to-seek*, reflecting the degree to which scent verbalizations would predict the files participants navigated to (as measured by cosine similarity to distal scent as described in Section 2). We also generated ROC curves for *issue report's distal scent* and *proximal scent + topology*. The area under these curves can be interpreted as the probability that the ROC's predictor will assign a higher rank to a randomly selected visited class than to a randomly selected unvisited class. Thus, a value of 0.5 represents random chance, whereas an AUC of 1 represents an ideal predictor.

Using these measures, Figure 7 shows the predictions of where participants *navigated*, and Table 5 shows the predictions of where they *found* information. Unlike the figure, the table compares the artifact-based predictors (*issue report's distal scent* and PFIS's *proximal scent plus topology*) but not the verbalization predictors because the table's perspective is whether a debugging tool would benefit from incorporating these predictors, and verbalizations would be impractical for this purpose. In the figure and the table, three phenomena stand out.

First, the *issue report* was better at predicting participants' navigation than their verbalizations were. This was the case for both *verbal hypotheses* (see the left figure) and *verbal scent-to-seek* (right figure). This is probably due to two factors: the phenomenon of "invisibility" of hypotheses in verbal protocols in which participants neglect to state their hypotheses (c.f. [Shrager and Klahr 1986]), and the richness of natural language in which hypotheses are not stated explicitly enough for automated analysis. For example, our participants' expressions of hypotheses and scents were often very context-dependent, and also contained deixis ("Where is that?"), interjections ("Ahhh"), and disambiguations using non-explicit mechanisms, such as saying a word or phrase as a question ("getTitle?" indicating continuing search for information) or as an exclamation ("getTitle!" indicating having found it). These results suggest that future models and tools have more potential for accuracy using the artifacts of the environment than attempting to somehow harvest programmers' verbalizations.

Second, for Issue B, PFIS's use of *proximal scent plus topology* ("PFIS" in the figure) was better at predicting participants' navigation than the *issue report's distal scent*. In the best case, PFIS's recall approached 100% at a false positive rate of 25-30% for the bug.

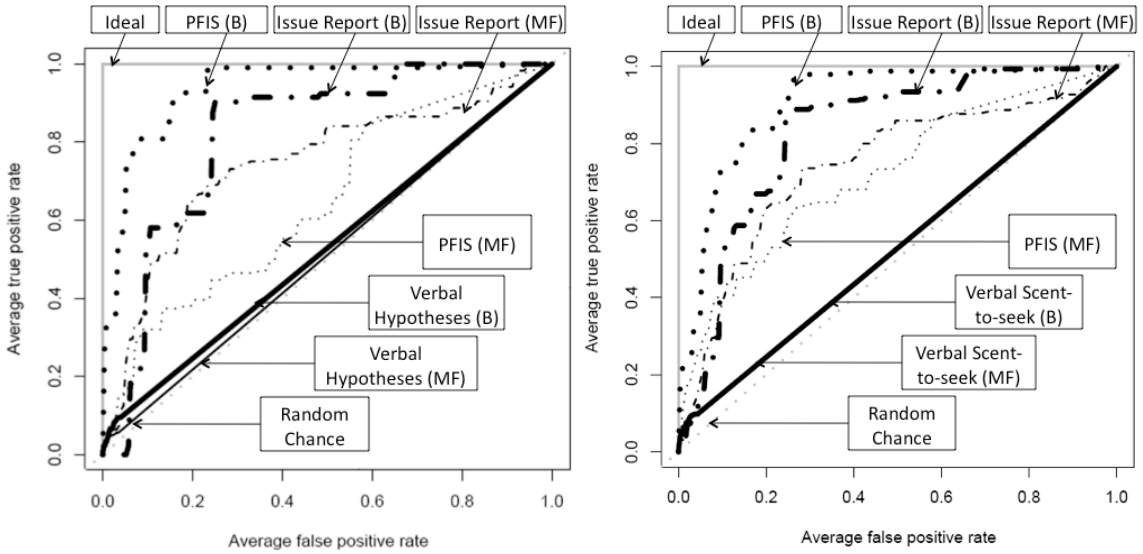

Figure 7. (Left): *Verbal hypothesis* ROC curves: Ability to predict classes to which participants navigated directly after they expressed a hypothesis (i.e., just after "hypothesis start" or "hypothesis modify" verbalizations). Issue B has thick lines; Issue MF has thin lines. (Right): *Verbal scent-to-seek* ROC curves: Ability to rank high the classes to which participants navigated directly after they expressed a need for information (i.e., just after "scent to seek" verbalizations).

Third, for both navigation (Figure 7) and information found (Table 5), the performance of PFIS in predicting navigation from the issue report on Issue B was quite different than its performance on Issue MF. For Issue B,

proximal scent plus topology was better as a predictor than the *issue report's distal scent*, which suggests the overall suitability of this combination to model programmer navigation during debugging. However, for Issue MF, the *issue report's distal scent* was more effective than *proximal scent plus topology* in predicting where programmers should go (Table 5), as was also the case for predicting where participants did go (Figure 7). A previous study involving 228 bugs and 25 feature requests from a different Open Source project (jEdit) has also suggested differences between bug reports' versus feature requests' ability to predict where programmers should navigate on the basis of distal scent [Lawrance et al. 2008a]. The differences regarding which of these factors best predict (Figure 7) and advise (Table 5) navigation during Issue B versus Issue MF suggest the need for future work to determine whether different variants of the theory or its models may be necessary for modeling navigation during debugging (corrective maintenance) versus modeling navigation during adding new features (perfective maintenance).

	Issue report's distal scent		PFIS proximal scent + topology	
Participant	B	MF	B	MF
82	0.820	0.734	0.870	0.362
83	0.641	0.737	0.942	0.570
84	0.592	0.923	0.945	0.945
85	0.870	0.741	0.952	0.788
96	0.765	0.811	0.934	0.761
97	0.466	0.670	0.891	0.569
98	0.657	0.687	0.851	0.567
99	0.906	0.625	0.851	0.973
910	0.914	0.854	0.978	0.585
911	0.870	0.818	0.806	0.717
Overall	0.777	0.743	0.881	0.659

Table 5. Issue text distal scent's ability to rank high the classes in which participants said they gained scent, i.e., whether these factors predicted where they *should* go to get the information they sought. The units are the areas under the ROC curves (AUCs) at the moments participants said they gained scent.

6. THREATS TO VALIDITY

Every experiment has threats to the validity of its results, and these threats must be considered in order to assess the meaning and impact of results. ([Wohlin et al. 2000] provide a general discussion of validity evaluation and a classification of validity threats.) This section discusses potential threats to the validity of our experiment and, where possible, how we attempted to mitigate the impact of these threats on our results.

6.1 External Validity

Threats to external validity limit the extent to which results can be generalized.

If the particular source code did not represent that of real software projects, our results may not generalize. To reduce this threat, we obtained source code from an actively maintained Open Source project. Even so, it is possible that this project's source code had characteristics that are unusual, and would not generalize to other projects. Also, because we assigned the particular project, participants did not have prior familiarity with the project, and therefore we cannot generalize from these results to other types of experiences with software, such as with code they wrote themselves or that they have been working with for a long time.

The ability to generalize our results may be limited by our selection of issue reports. We attempted to address this issue by choosing one issue in the corrective maintenance category and one in the perfective maintenance category. However, our study included only these two issues, and it seems unlikely that only two issues alone can reasonably represent corrective and perfective maintenance generally. One other study we performed analyzed a number of other issues in other Open Source projects [Lawrance et al. 2008a], but that study did not examine the questions in the current paper, only the predictiveness of issue reports alone to the files people ultimately changed, so further work is needed to generalize the findings of the current paper.

The limited size of our sample also limits our study's generalizability. A small number of participants is necessary with qualitative analyses of the sort used in our study, but it limits the extent to which findings can be generalized. To reduce this threat, we triangulated our data, analysis methods, and interpretations. Specifically, we used statistical quantitative methods on participant actions (in their activity logs), qualitative methods on their verbal data through dual coding of all verbal transcripts, and in-depth review of two participants' full videos for two tasks each. In this way, we incorporated multiple sources of data, multiple analysis methodologies, and multiple researchers' perspectives.

Finally, our experiment was conducted in Java in the Eclipse environment. As we have pointed out, our theory suggests that programmers' navigation behavior is context-dependent, and therefore programmers' navigation behavior is expected to be different in another programming language or environment. Whether navigation behavior in that different environment would then be predicted by the theory remains an open question.

All of these external validity threats can be addressed only through repeated studies, using different source code, different issues, and/or different programming environments.

6.2 Internal Validity

Threats to internal validity are other factors that may be responsible for an experiment's results.

We imposed a two-hour time limit, and this is somewhat artificial. Also, it is possible that imposing a time limit of this sort was not a good fit for some participants' preferred style of familiarizing themselves with new code.

The think-aloud protocol is a widely accepted method of getting an approximation of what a participant is thinking, but it is also well established that use of the think-aloud method can impact participant behavior. For example it could be that participants more readily verbalized scent-related thoughts because scents are connected to concrete actions in the world, whereas hypotheses were less verbally accessible [Shrager and Klahr 1986]. If that were the case, our data showing scent processing to be more prevalent than (non-scent) hypothesis processing would be undermined somewhat. However, this threat does not change the essence of the result that scent processing was widespread, nor would it suggest that tools should use hypotheses instead, because the difficulty in accessing hypotheses would still make them unavailable to tool builders in practice.

6.3 Construct Validity

Threats to construct validity question whether the measures in an experiment's design adequately capture the effects that they were intended to capture.

Recall that in our analysis methodology, we coded hypotheses only if the statements made were clearly hypotheses about where the bug might be lurking or how to fix it. Some statements about promising directions, verbal-

ized during browsing, were explicit enough about scent to be coded as scent seek/gained/lost, but if no hypothesis was explicit in the verbalization, we could not code such statements as hypotheses. Some of those statements not coded as hypotheses could have been related to hypotheses. However, this actually is further support for the idea of relying upon information foraging instead of hypothesis processing as a way to understand programmer behavior, because any hypothesis processing that arises as part of scent processing will be accounted for in the foraging model.

The PFIS model that was used to measure the theory's constructs uses only static analysis, and thus simplifies the information space through which a programmer really navigates. For example, as a consequence of using Eclipse, our programmers had the ability to use full-text search to alter their navigation behavior in a way that PFIS does not model. (Despite this ability, few of our programmers were successful in using Eclipse's search feature, and even when successful, Eclipse returned results in alphabetical order by class, not in order of relevance.) A more complete model of "patches" would have to include all the information within a participant's gaze. Instead, PFIS approximates patches with locations in source code only. In the future, we plan to include elements of dynamic analysis and analysis of the changes to elements of source code, to empirically determine whether the additional implementation and runtime cost of these approaches will produce commensurate improvements in prediction accuracy.

PFIS is one way of operationalizing the constructs of our theory, but the specific choices we made in implementing PFIS may not be the correct choices. For example, recall that "scent" means "relatedness." PFIS looks for only linguistic relatedness (i.e., word similarity) and does not consider any other form of scent. Further, it measures relatedness using TF-IDF [Baeza-Yates et al. 1999], a widely used measure in the information retrieval community, but it is possible that other measures of relatedness such as Latent Semantic Analysis [Landaur and Dumais 1997] or Pointwise Mutual Information [Cover and Thomas 1991] would be more appropriate. In future work, we plan to compare the suitability to our model of these competing measures against TF-IDF empirically.

Note that our comparison between *distal scent* and *proximal scent plus topology* might on the surface appear to be confounded: the reader may wonder if the differences are due to the distal/proximal difference, or the topology. However, proximal scent is inseparable from topology. Proximal scent without topology is impossible, because proximality is defined by topology. Distal scent with topology is equally strange; by its very definition distal scent does not propagate, so it would not differ from distal scent without topology. Each has strengths the other does not. For example, if a patch had strong scent relative to the bug report but weak proximal scent from the perspective of its neighbors (for example a class whose interface made no mention of words in the bug report, but whose implementation did mention such words), then the issue report's distal scent would predict navigation to that class, while *proximal scent plus topology* would not. On the other hand, proximal scent and topology take into account the proximal cues' ability to "light the way" down the path to the fix, which distal scent does not. Therefore, we considered it important to measure the performance of both to compare them.

7. RELATIONSHIP OF INFORMATION FORAGING TO OTHER THEORIES OF DEBUGGING

Some researchers have proposed models of cognition during debugging, and have suggested ways that these models might influence navigation. Theories of debugging have evolved in parallel with changes to the task of debugging itself. Broadly speaking, earlier theories tended to focus on a person reading programs and forming hy-

potheses in the head, until thinking of a fix. Later theories depict a process of gathering and organizing information; the trend is from largely in-the-head debugging towards a distributed cognition perspective, which says that some of cognition is "in the world" as a replacement for being in the head.

Brooks proposed a top-down theory of program comprehension in which a hierarchy of hypotheses drives comprehension [Brooks 1983]. According to Brooks, high level hypotheses are notions about how the whole of the code might be structured; these spawn lower-level hypotheses about parts or aspects of the program; and the most specific, lowest-level hypotheses are ones that can be directly validated by being mentally bound to specific lines of code. While Brooks did not use the terminology of information foraging or scent, he did make predictions that are specifically relevant to why programmers' hypotheses may not be useful as predictors of navigation during the code comprehension portion of debugging:

- Hypotheses do not generally have names or named components; they are "just descriptions of the components in terms of the functions they perform." This implies that there is no simple mechanism, such as keyword matching from verbalized hypotheses to code, that could be incorporated into a predictive model.
- In Brooks's theory, the "primary hypothesis," i.e. the top-level one about the structure of the code, is "global and non-specific," and thus "the programmer will almost always find it impossible to verify his hypothesis directly against the program code." Instead, it prompts "construction of subsidiary hypotheses," with the lowest ones "adding specific detail and concreteness." This suggests that the high-level hypotheses will be the least likely to match up in any way with the code.
- "Information collection during the search will be broad, rather than focused only on the hypothesis at hand." This suggests that a programmer's current hypothesis is unlikely to correspond to the code that the programmer visits during search.

Brooks's theory also helps us understand why scent may be a more useful construct to use as a predictor of code navigation. In his theory, he notes that the most concrete hypotheses are verified by the identification of "beacons": "sets of features that typically indicate the occurrence of certain structures or operations within the code." For example a swap operation within nested DO loops is a possible beacon for a sort operation. Variable names and other names used in code are other examples of beacons. Beacons are closely related to our *cues*: for the programmer, they give off a *scent* because they indicate that the code is related to some notion they have in their head (in this case, a hypothesis about what the code represents).

Letovsky proposed a cognitive model that involves both bottom-up and top-down hypothesis ("conjecture") formation [Letovsky 1986]. Analyzing the initial comprehension phase of some programmers' efforts to make program changes, he describes *what* and *why* hypotheses (and questions) as drawing connections bottom-up from the program ("implementation") to the program's domain ("specification"); and *how* hypotheses (and questions) as going top-down. Letovsky claims that these bottom-up hypotheses trigger a search through code or documentation, or a reasoning process within the mental model, to produce an answer, and that the programmer "pick[s] a method which is likely to yield an answer with little effort." Unlike Brooks' top-down hypotheses, verbalizations of Letovsky's bottom-up hypotheses might be expected to contain words found in the code itself, rather than in a bug report. However, they seem unlikely to be useful as a predictor of code navigation, as these hypotheses occur after a programmer has already navigated to a closely-related place in the code.

Code comprehension is of course just one aspect of debugging. A theory of debugging needs to consider all aspects of this complex task. Katz and Anderson, in experiments examining students' LISP debugging strategies [Katz and Anderson 1988], do just this. Working with students who were debugging code they had just written, they identify four distinct tasks that comprise debugging: comprehension, testing, locating the component containing the error, and repairing the component. Locating the component containing the error was the task that students had most difficulty with. Katz and Anderson found three general strategies used by the students for locating the buggy component: mapping directly from program behavior to the bug, causal reasoning, and hand-simulation. They speculate that causal reasoning as a strategy becomes less frequent over time as learners build up a set of situations that they recognize with easy remedies. They also found that causal reasoning was more frequent when debugging one's own code than someone else's. This would appear to be consistent with our results: relatively little causal reasoning (i.e. few hypotheses), among expert programmers, working with unfamiliar code. While the Katz and Anderson study has been criticized because the authors assume that comprehension precedes debugging [Gilmore 1991], programmers' use of these strategies has been corroborated in a study that does not make such an assumption [Romero et al. 2007].

We have already pointed out that the difference in today's environments from those of the early studies suggest that we must tread carefully in generalizing the results of these studies to programming today. Further, the programming tasks studied by Brooks, Letovsky, and Katz and Anderson were very different from those typically encountered by today's professional programmers. Brooks and Letovsky developed their theories using short programs presented on paper. When working with a short program on paper, a programmer could attempt to completely comprehend the whole program before making changes. In fact, navigation in the paper medium was fairly costly because no navigation tools are provided. But today, professional programmers must deal with large source code libraries containing thousands of lines of code. For such large programs, the only realistic strategy is to focus on relevant parts of the code and to navigate among them, and today's programming environments include affordances and tools that aim to support these capabilities.

Contemporary software development environments have already been found to affect debugging strategies. In an experiment verifying the debugging strategies observed by Katz and Anderson using contemporary software development environments, Romero et al. identify an additional forward-reasoning strategy that they termed "following execution" [Romero et al. 2007]. This strategy involves step-by-step tracing of the execution for one particular input example, and observing which lines of code change the data and affect the program's output. Such a strategy requires an environment supporting it, and was certainly not possible in the paper-based experiments of earlier years. Thus the richness and flexibility of modern development environments make it possible for a programmer to substitute informational manipulation for some cognitively intensive activities, if he or she is so inclined, such as in the "following execution" example supported by debugger's stepping functionality rather than performing hand simulations and calculating the results by hand.

The large size of today's programs also seem implicated; in fact, we believe today's large programs *necessitate* that programmers use foraging strategies in debugging. To reduce the cost in time and effort of debugging a large program, we conjecture that programmers choose to understand only parts of the program, just as our participants did, taking the risk that incomplete comprehension might lead to an incorrect fix. When following such a strategy,

programmers would have to make the hard choice of letting the irrelevant bottom-up what-does-this-variable-do kinds of questions go. We expect that in programming situations involving large programs and limited time to spend, programmers suppress hypotheses that are not related to the bug report, or at least choose not to spend time trying to confirm them. Instead, our premise is that programmers follow scents that involve words related, directly or indirectly, to the bug report. This premise is the basis of our theory.

Ko et al. have also argued the importance of seeking information as a central mechanism in debugging strategy [Ko et al. 2006]. They propose a model of program understanding during software maintenance in which the strategy chosen depends on seeking relevant information in the environment. Their model consists of three stages: *seeking*, *relating*, and *collecting* information. *Seeking* involves looking through the code for information relevant to the task at hand. *Relating* involves connecting the different found regions of code to each other to see how they relate. *Collecting* involves the designation of some of these regions as relevant to the task. Examples of *relating* and *collecting* include programmers repeatedly returning to regions of code they had visited before, and using affordances of the programming environment to speed up that repeated navigation: for example by creating bookmarks, leaving scrollbars strategically placed, and keeping multiple tabs or windows open for fast switching.

Actions such as these allow the programmer to improve the density of useful material in a patch, or to speed travel between patches. In turn, these actions change the cost/benefit trade-offs of different debugging strategies. As we have pointed out before, in information foraging theory such actions are referred to as *enrichment.* Through enrichment, the information seeker can deliberately modify the environment to either improve the density of useful material in a patch, or speed travel between patches. In future work, we plan to expand the current implementation of PFIS to take into account enrichment that programmers do on the fly.

8. DISCUSSION AND CONCLUSION

In this paper, we have presented an information foraging theory for how programmers navigate information artifacts when debugging. The results show that the scent and topology constructs of this theory are valuable constructs in a predictive model. In particular, we found that:

- Scent: The way participants worked with scent was highly consistent with information foraging theory. The participants verbalized activities related to scent about four times as often as (non-scent) hypotheses. In addition, scent was a significant predictor of programmer navigation. The relationships between scent and some types of hypotheses may have contributed to the effectiveness of scent as a predictor.
- Debugging modes: In the six debugging modes in our participants' data, scent following was pervasive in all six of them, whereas (non-scent) hypotheses were mostly concentrated in just one of them. This finding also helps to explain why scent was so effective at predicting programmer navigation.
- Where scent came from: The biggest "trigger" for searching for scent was the source code itself, but other triggers were the bug report, runtime behavior, and additional resources such as web pages and input files. These findings suggest, first, that operationalization of the scent construct using static analysis of source code alone can produce reasonably high quality predictions and, second, that even higher quality predictions may be possible if a model includes these additional data sources.

- Verbal data versus artifact data: Computing scent using issue reports and source code was more useful at predicting programmer navigation behavior than verbal sources. (This is fortunate, as in-the-head data would be hard to gather during real-world debugging.)
- Corrective versus perfective maintenance: The topology construct, which is explicit in our information foraging theory, was a powerful contributor in predicting which classes programmers visited for Issue B. For Issue B, programmers went to a class "linked" directly to the current one 90% of the time. But for Issue MF, the percentage dropped to only 57%. These and other differences suggest the need for future work investigating how to apply information foraging theory to corrective maintenance (debugging) versus perfective maintenance (adding new features).

Turning our attention to practical implications of the theory, we now consider how it could be used as a basis for software engineering tools. (Note that the PFIS model is an executable model of the theory for use in validating the theory, not a tool in its own right.)

First let us consider the implications for design of a debugging tool that includes interaction and the need for programmers to navigate. For this kind of tool, guidelines for the designer of such debugging tools derive directly from the theory's constructs: given the presence of prey (bug), a predator (programmer), and information patches (methods, screens, web pages, etc.), the theory implies that a tool should support scent following through the supported patches by leveraging proximal cues and topology. For example, a tool designer can first identify the proposed tool's goal in terms of the kinds of information the tool aims to help programmers find when debugging. Suppose the tool aims to support debugging by providing access to runtime information. Given the tool's goal, the tool designer's next step is to identify scent (relevance) by deciding what is "related" to that information. For example, some tools have the premise that the runtime stack is relevant to debugging, and provide visualizations of the runtime stack at the time range the programmer declares to be of interest; the information displayed for each activation record provide the cues, whose salience determine the usefulness of the visualization to a programmer trying to follow the scent to the bug. If a programmer using the tool spots a potentially useful cue, he or she will potentially want to navigate to it, so the tool will need a topology that allows that navigation to be efficient. Allowing the user to easily enrich the cues and topology to further enhance their working environment's ability would further support quick navigation. Although these implications might seem to simply say "employ good tool design," they make explicit theory-based critieria for doing so.

One challenge is that much of the quality of cues is outside the control of the tool designer, because they often occur in the source code itself, which was written by a programmer, not the tool designer. Here, information foraging theory can help too. Tools based on information foraging theory could evaluate and promote "forageability." For example, tools based on information foraging theory could evaluate and suggest improvements to names, labels, pictures, and explicit links in source code, hypertext documentation of source code, issue reports, and so on, so that cues in these artifacts would emanate stronger and more precise scent.

There are many open questions, for debugging tool development, for the PFIS model, and for the theory itself. For example, our speculations about the implications for practical tool building have not been tried out, let alone evaluated empirically. The PFIS model has been evaluated empirically, but in our empirical work with the PFIS model, we have experimented with only one variant of computing scent, and have pursued only linguistic scent at

that, not other kinds of relationships. Our model has not taken enrichment into account, nor the use of other artifacts in the environment. It is an open question whether increased predictive power from incorporating these factors would outweigh the costs of doing so. Further, both the current study and a previous one suggested differences in predictiveness of the PFIS model on Issue B (debugging, i.e., corrective maintenance) versus Issue MF (perfective maintenance). Further research is needed to determine the scope of information foraging theory across different forms of software maintenance beyond debugging.

In the future, we believe information foraging theory can provide a fundamental understanding as to why some software maintenance tool features are or are not useful to human programmers. Information foraging theory's principles are few in number, lending to its comprehensibility by tool builders. This parsimonious theory can therefore provide practical guidance to tool designers toward ways to increase practical support for programmers, such as by making clear that tools need to consider all three of the information foraging factors of scent, topology, and human attempts at optimality in an environment. Because of these attributes of information foraging theory, we hope this theory of human programmers' information seeking needs during maintenance can help make obsolete practices of building interactive software maintenance tools ad hoc.

ACKNOWLEDGMENTS

Much of this work was performed during M. Burnett's sabbatical stay at IBM TJ Watson Research Center. This work was also supported in part by the Air Force Office of Scientific Research FA9550-09-1-0213, by the EUSES Consortium via NSF ITR-0325273, by an IBM International Faculty Award, and by J. Lawrance's IBM PhD Scholarship.

REFERENCES

[Anderson 1983] J.R. Anderson, "A spreading activation theory of memory," *Verbal Learning and Verbal Behavior*, vol. 22, pp. 261-295, 1983.

[Anderson 1990] J.R. Anderson, *The Adaptive Character of Thought*. Lawrence Erlbaum Associates, 1990.

[Baeza-Yates et al. 1999] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval.* Addison Wesley Longman, 1999.

[Brooks 1983] R. Brooks, "Towards a theory of the comprehension of computer programs," *Int. Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.

[Chi et al. 2001] E. Chi, P. Pirolli, K. Chen and J. Pitkow, "Using information scent to model user information needs and actions on the web," *ACM Conf. Human Factors in Computing Systems*, pp. 490-497, 2001.

[Chi et al. 2003] E. Chi, A. Rosien, G. Supattanasiri, A. Williams, C. Royer, C. Chow, E. Robles, B. Dalal, J. Chen and S. Cousins, "The Bloodhound project: Automating discovery of web usability issues using the InfoScent simulator," *ACM Conf. Human Factors in Computing Systems*, pp. 505-512, 2003.

[Crestani 1997] F. Crestani, "Application of spreading activation techniques in information retrieval," *Artificial Intelligence. Rev.*, vol. 11, no. 6, pp. 453-482, 1997.

[Cover and Thomas 1991] T. Cover and J. Thomas, *Elements of Information Theory*. John Wiley & Sons, 1991.

[Eisenstadt 1993] M. Eisenstadt, "Tales of debugging from the front lines," *Empirical Studies of Programmers: Fifth Workshop*, Ablex Publishing Corporation, pp. 86-112, 1993.

[Gilmore 1991] D.J. Gilmore, "Models of debugging," *Acta Psychologica*, vol. 78, pp. 151-172, 1991.

[Herlocker et al. 2004] J. Herlocker, J. Konstan, L. Terveen and J. Riedl, "Evaluating collaborative filtering recommender systems," *ACM Trans. Information Systems*, vol. 22, no. 1, pp. 5-53, 2004.

[Hollan et al. 2000] J. Hollan, E. Hutchins and D. Kirsh, "Distributed cognition: Toward a new foundation for human-computer interaction research," *ACM Trans. Computer-Human Interaction*, vol. 7, pp. 174-196, 2000.

[Katz and Anderson 1988] I.R. Katz and J.R. Anderson, "Debugging: An analysis of bug-location strategies," *Human-Computer Interaction*, vol 3, pp. 351-399, 1988.

[Ko et al. 2006] 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," *IEEE Trans. Software Engineering*, vol. 32, no. 12, pp. 971-987, 2006.

[Landaur and Dumais 1997] T.K. Landauer and S.T. Dumais, "A solution to Plato's problem: The latent semantic analysis theory of the acquisition, induction, and representation of knowledge," *Psychological Review* 104, pp. 211-240, 1997.

[Lawrance et al. 2007] J. Lawrance, R. Bellamy and M. Burnett, "Scents in programs: Does information foraging theory apply to program maintenance?" *IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 15-22, 2007.

[Lawrance et al. 2008a] J. Lawrance, R. Bellamy, M. Burnett and K. Rector, "Can information foraging pick the fix? A field study," *IEEE Symp. Visual Languages and Human-Centric Computing*, pp. 57-64, 2008.

[Lawrance et al. 2008b] J. Lawrance, R. Bellamy, M. Burnett and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," *ACM Conf. Human Factors in Computing Systems*, pp. 1323-1332, 2008.

[Leontjev 1978] A. Leontjev, *Activity, Consciousness, and Personality.* Englewood Cliffs NJ: Prentice-Hall, 1978.

[Letovsky 1986] S. Letovsky, "Cognitive processes in program comprehension," in *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, Eds. Ablex Publishing Corporation, pp. 58-79, 1986.

[Nanja and Cook 1987] M. Nanja and C. Cook, "An analysis of the on-line debugging process," in *Empirical Studies of Programmers: Second Workshop*, E. Soloway and S. Sheppard, Eds. Ablex Publishing Corporation, pp. 172-184, 1987.

[Nielsen 2003] J. Nielsen, "Information foraging: Why Google makes people leave your site faster," June 30, 2003. [Online] Available: http://www.useit.com/alertbox/20030630.html [Accessed: March 16, 2009].

[Pirolli 1997] P. Pirolli, "Computational models of information scent-following in a very large browsable text collection," *ACM Conf. Human Factors in Computing Systems*, pp. 3-10, 1997.

[Pirolli and Card 1999] P. Pirolli and S. Card, "Information foraging," *Psychology Review*, vol. 106, no. 4, pp. 643-675, 1999.

[Robillard et al. 2004] M. Robillard, W. Coelho and G. Murphy, "How effective developers investigate source code: An exploratory study," *IEEE Trans. Software Engineering*, vol. 30, no. 12, pp. 889-903, 2004.

[Romero et al. 2007] P. Romero, B. du Boulay, R. Cox, R. Lutz and S. Bryant, "Debugging strategies and tactics in a multi-representation software environment," *Int. J. Human-Computer Studies*, vol. 65, no. 12, pp. 992-1009, Dec. 2007.

[Shrager and Klahr 1986] J. Shrager and D. Klahr, "Instructionless learning about a complex device," *Int'l, Journal Man-Machine Studies*, vol. 25, pp. 153-189, 1986.

[Singer et al. 2005] J. Singer, R. Elves and M.A. Storey, "NavTracks: Supporting navigation in software maintenance," *Int. Conf. Software Maintenance (ICSM05)*, pp. 325-334, 2005.

[Spool et al. 2004] J. Spool, C. Profetti and D. Britain, "Designing for the scent of information," *User Interface Eng.*, 2004.

[Suchman 1987] L. Suchman, *Plans and Situated Actions: The Problem of Human-Machine Communication*. Cambridge University Press, 1987.

[Vans and von Mayrhauser 1999] A. Vans and A. von Mayrhauser, "Program understanding behavior during corrective maintenance of large-scale software," *Int. Journal Human-Computer Studies*, vol. 51, pp. 31-70, 1999.

[Wohlin et al. 2000] C. Wohlin, P. Runeson, M. Host, B. Regnell, and A. Wesslen, *Experimentation in Software Engineering*. Kluwer Academic Publishers, Boston, Massachusetts, 2000.