This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.
IEEE TRANSACTIONS ON SOFTWARE ENGINEERING

How Programmers Debug, Revisited: An Information Foraging Theory Perspective

Joseph Lawrance[1,2,3], Christopher Bogart[2], Margaret Burnett[2,3],
Rachel Bellamy[3], Kyle Rector[2], Scott D. Fleming[2]

[1]Wentworth Institute of Technology
lawrancej@wit.edu

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

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

ABSTRACT

Many theories of human debugging rely on complex mental constructs that offer little practical advice to builders of software engineering tools. Although hypotheses are important in debugging, a theory of navigation adds more practical value to our understanding of how programmers debug. Therefore, in this paper, we reconsider how people go about debugging in large collections of source code using a modern programming environment. We present an information foraging theory of debugging that treats programmer navigation during debugging as being analogous to a predator following scent to find prey in the wild. The theory proposes that constructs of scent and topology provide enough information to describe and predict programmer navigation during debugging, without reference to mental states such as hypotheses. We investigate the scope of our theory through an empirical study of ten professional programmers debugging a real-world open source program. We found that the programmers' verbalizations far more often concerned scent-following than hypotheses. To evaluate the predictiveness of our theory, we created an executable model that predicted programmer navigation behavior more accurately than comparable models that did not consider information scent. Finally, we discuss the implications of our results for enhancing software engineering tools.

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

An advantage of some theories of programmer behavior is that they can be used to make predictions about how programmers will use software engineering tools. Such predictions can be used to guide choices about potential

Digital Object Indentifier 10.1109/TSE.2010.111

benefits of new tool features, or can inspire new software engineering practices. In this way, software engineering researchers and tool developers can build on each other's work in a principled manner.

This paper presents a theory of programmer navigation when debugging. The programmer navigation aspect of debugging is important: recent research has shown that programmers spend 35% of their time navigating [Ko et al. 2006]. Older theories of program debugging (e.g., [Brooks 1999], [Vans and von Mayrhauser 1999]) do not explicitly consider navigation; instead, they rely primarily on in-the-head constructs, such as mental models and hypotheses, to explain the behavior of programmers during program comprehension and debugging. Although such theories revealed phenomena that have influenced software tools, they have not revealed concrete behavioral phenomena that tools can directly observe and respond to appropriately. Furthermore, 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 understanding programmer navigation during debugging in modern programming environments, with potential concrete implications for programming tools. It suggests a basis for theories of programmer behavior that do not involve 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 [Pirolli and Card 1999] 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 irrelevant information in the environment, humans have evolved strategies to efficiently find information relevant to their needs without processing everything—in essence, minimizing the mental cost to achieve their goals.

Information foraging theory is based on optimal foraging theory, a theory of how predators and prey behave 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 that contains the prey. 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 and fixing a bug as well. That is, we propose that if a bug is treated as the "prey," words in the environment and the source code are treated as "cues" that suggest the "scent" of 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 debugging. (We detail this mapping in Section 2.) 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 the conclusion of this paper, we briefly suggest a few such implications; [Scaffidi et al. 2010] provides a more in-depth treatment of the implications for tool design.)

In this paper, we present a theory of programmer navigation during debugging based on information foraging theory. Consistent with Sjøberg et al.'s theory-building process [Sjøberg et al. 2008], we define the constructs and propositions of the theory; we empirically investigate the scope of the theory (i.e., the circumstances in which the theory is applicable); and we empirically evaluate the theory's predictive power.[1] To perform the evaluation, we developed an executable model, PFIS, which operationalizes the theory constructs and can predict where expert programmers navigate during debugging [Lawrance et al. 2008b].[2] As a point of comparison, we also explored a variant of our theory that bases predictions on programmer hypotheses, rather than scent.

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

2. AN INFORMATION FORAGING THEORY OF PROGRAMMER BEHAVIOR DURING DEBUGGING

A central aspect of developing a theory is choosing the right constructs [Sjøberg et al. 2008]. A small number of constructs may improve the theory's parsimony; however, too few or overly simple constructs may limit the theory's explanatory power or scope. Information foraging theory [Pirolli and Card 1999] defines a manageably small set of intuitive constructs that has demonstrated utility for explaining and predicting how humans navigate web pages [Chi et al. 2001].

To handle the realm of debugging, we refined the original information foraging constructs as follows:

- *Predator*: The programmer who is debugging.
- *Prey*: What the programmer seeks to know to reveal the changes that must be made to fix the bug. Furthermore, any information that the programmer seeks to achieve the goal also constitutes a form of prey.
- Information *patches*: Localities in the source code, related documents, and displays that may contain prey.
- *Proximal cues*: Words, objects, and perceptible runtime behaviors in the programming environment that suggest scent relative to the distal prey. Cues act as signposts to prey. For example, words in the source code, including comments, constitute a type of cue.

[1] The empirical evaluations (described in Section 5) are original and unique to this paper, but are based on previously collected navigation data and think-aloud protocols (See Section 4 and [Lawrance et al. 2007]).

[2] PFIS's implementation is described in Appendix A and in [Lawrance et al. 2008b].

- *Information scent*: The perceived likelihood of a cue leading to prey, either directly or indirectly. Scent is a measure, and scent from one cue can be compared to the scent of other cues. Unlike cues, scent exists only in the programmer's head. This definition is consistent with 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, related documents, and displays through which the programmer can navigate.

To apply the theory to real-world debugging, we developed the following operational definitions. Some constructs have self-evident operationalizations. We operationalize the prey construct as places in the code where changes must be made to fix the bug, the predator construct as a programmer, and the patch construct as localities in the source code, such as Java methods, classes, and packages.

The notions of topology, cues, and scent do not map obviously to operational definitions. We operationalize these constructs as follows:

- *Topology*: A directed graph with vertices representing elements of the source code (e.g., classes, methods) and of the environment (e.g., class labels enumerated in a class hierarchy browser), and with edges representing navigable links between the elements.
- *Link*: A connection between two nodes in the topology that allows the programmer to traverse the connection at a cost of just one click. For example, in the Eclipse editor, a method invocation can be clicked on to open the associated method definition. Hence, Eclipse provides links from method calls to definitions that the predator can navigate with one click. In this case, a method call is the *source* of a link and the associated definition is the *destination*. (Note that links are environment and context dependent.)
- *Proximal cues*: Words located near the source of a link. For example, consider the following line of code opened in the Eclipse editor: "`System.out.println(someData);`". The identifier `println` is the source of a method-invocation link, and the words `system`, `out`, `println`, and `someData` are proximal cues that engender scent about potential prey at the other end of the link.
- *Scent*: Word similarity between the bug report (description of the prey) and proximal cues. That is, a set of cues comprising words that appear frequently in the bug report will engender strong scent in the mind of the predator.

The measure of scent warrants further explanation. Information scent is the programmer's (imperfect) perception of the value (relatedness) of information (as in Pirolli's information foraging research on web searching [Pirolli 1997]). To computationally approximate information scent, we compute word similarity between the description of the prey (e.g., bug-report text) and the proximal cues in the source code by applying cosine similarity to a vector space IR model. Note that this operational definition is the model's *approximation* of scent; the true measure of scent exists only in the programmer's head.

Computing this approximation of scent is a three-step process. First, we preprocess the source-code text and bug report. We tokenize words so that camel case identifiers (e.g., "`NewsItem.getSafeXMLFeedURL()`") are split into their constituent words (e.g., "news," "item," "get," "safe," "xml," "feed," and "url"). Furthermore, we apply

the Porter stemming algorithm on the constituent words[3] and filter out Java reserved words (e.g., `public`, `static`, and `void`) and English stopwords (e.g., "the," "to," "be," "or," and "not").

Second, we weigh terms in files of source code according to the commonly used *tf-idf* formula [Baeza-Yates et al. 1999], which we compute as follows:

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

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

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.

Third, we compute the inter-word correlation between proximal cues in source files and the text of a bug report using cosine similarity, a measure commonly used in information retrieval systems [Baeza-Yates et al. 1999]. We compute cosine similarity as follows:

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

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]).

To evaluate the predictive power of the theory, we developed an executable model, PFIS (Programmer Flow by Information Scent) based on the above operationalization. We introduced PFIS in previous work [Lawrance et al. 2008b]. To implement PFIS's approximation of scent, we leveraged the Apache Lucene search-engine library[4], which provides functionality for computing word similarity.

To make predictions, PFIS propagates its computation of scent throughout the foraging environment (i.e., program source code). It starts by gathering topological information, then calculates scent of each procedure/method's cue. 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]. Appendix A provides the details of the spreading-activation algorithm. In Sections 4–6, we describe a study in which we applied PFIS to test the theory's predictive power.

3. 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 (e.g., [Brooks 1983], [Letovsky 1986]) tended to focus on a person

[3] Porter stemming is simple and efficient, but, like all stemming algorithms, it can stem words erroneously (i.e., producing different stems for different forms of the same words, or producing the same stem for different words) [Baeza-Yates et al. 1999].

[4] http://lucene.apache.org/

reading programs and forming hypotheses in his or her head, until he or she found a fix. Later theories (e.g., [Ko et al. 2006], [Romero et al. 2007]) 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 cognition is "in the world" as a replacement for being in the head [Hollan et al. 2000].

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. The pursuit of these high-level hypotheses spawns lower-level hypotheses about parts or aspects of the program. The programmer can validate the most specific, lowest-level hypotheses by inspecting specific lines of code. Brooks observed several patterns in programmers' hypothesis-processing behavior:

- Programmers generally do not give hypotheses names or named components; their hypotheses are "just descriptions of the components in terms of the functions they perform."
- In Brooks's theory, the "primary hypothesis"—that is, 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 directly against the program code." Instead, it prompts "construction of subsidiary hypotheses," with the lowest-level ones "adding specific detail and concreteness." This suggests that the high-level hypotheses will be the least likely to match up directly with the code.
- "Information collection during the search will be broad, rather than focused only on the hypothesis at hand." This observation suggests that a programmer's current hypothesis is unlikely to directly correspond to the code that the programmer visits during debugging.

Brooks' 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. Other examples of beacons include variable and other names used in code. Beacons are closely related to *cues* in information foraging theory. The programmer can register *scent* from beacons because beacons relate the code to some notion the programmer has in his or her head (e.g., a hypothesis about what the code represents).

Letovsky proposed a cognitive model that involves both bottom-up and top-down hypothesis[5] formation [Letovsky 1986]. His work looked at programmers' initial phase of comprehension during a small-scale change task. He described *what* and *why* hypotheses (and questions) as drawing connections bottom-up from the program code ("implementation") to the program's domain ("specification"); and *how* hypotheses (and questions) as going top-down. Letovsky claimed that these bottom-up hypotheses trigger a search through code or documentation, or a reasoning process within the mental model. Moreover, programmers tend to "pick 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 contain words found in the code itself, rather than in a bug report. However, the bottom-up hypotheses 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.

[5] Letovsky used the term *conjecture*.

Program comprehension is one aspect of debugging; however, there are other aspects to consider. Katz and Anderson conducted experiments examining students' LISP-debugging strategies [Katz and Anderson 1988]. The study revealed four distinct debugging subtasks: comprehension, testing, locating the component containing the error, and repairing the component. The students had the most difficulty with locating the component that contained the error. The study also found three general strategies for locating the buggy component: mapping directly from program behavior to the bug, causal reasoning, and hand-simulation. The researchers speculated that causal reasoning becomes less frequent over time, as learners build up a set of familiar problem situations and remedies. The study also found that students engaged in more causal reasoning when debugging their own code as opposed to someone else's. Although the Katz and Anderson study has been criticized because it assumed that program comprehension precedes debugging [Gilmore 1991], subsequent studies, which made no such assumption, corroborated the finding that programmers use these debugging strategies [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. Furthermore, 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 were 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. Today's programming environments include affordances and tools, such as the ubiquitous find utility, that aim to support these capabilities.

Contemporary software-development environments have been found to affect debugging strategies. In an experiment using such environments, Romero et al. identified 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 environmental support, and was 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. For instance, a programmer can use a debugger's stepping functionality when following execution rather than performing hand simulations and calculating results by hand.

The large size of today's programs also seems to impact debugging strategies; 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, accepting the risk that incomplete comprehension might lead to an incorrect fix. When following such a strategy, programmers may have to ignore many bottom-up hypotheses. We expect that in programming situations involving large programs and limited time, programmers tend to ignore hypotheses that are not related to the bug report. Instead, the programmers follow scents that involve words related, directly or indirectly, to the bug report. Our theory is based on this premise.

Ko et al. have also argued the importance of seeking information as a central mechanism in debugging behavior [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 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 reduce the cost of navigating between patches. In turn, these actions change the cost/benefit trade-offs of different debugging strategies. Recall that information foraging theory refers to such actions 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.

4. EMPIRICAL STUDY

This study investigates the use of information foraging theory to explain and predict the navigation behavior of programmers performing debugging. To better understand and evaluate the theory in this context, we compared it with a variant based on programmers' hypothesis processing, rather than scent following. In conducting the study, we addressed four primary research questions. The first research question explored the relationship between scent-following and hypothesis-processing behavior:

- RQ1: The literature emphasizes the importance of programmer hypotheses in understanding debugging behavior. Is there reason to expect a scent-based model of debugging navigation will succeed if it does not explicitly handle programmers' hypothesis processing?

The second and third research questions served to elaborate the scope of the theory:

- RQ2: Debugging involves a variety of activities, such as locating the fault, fixing the fault, and verifying the fix. When do developers engage in scent and hypothesis processing with respect to these activities?
- RQ3: Programmers navigate through a variety of artifacts, such as source code, documentation, and email. Where do programmers navigate during debugging with respect to these artifacts, and in which artifacts do programmers exhibit scent seeking and hypothesis processing?

The fourth question evaluated the predictive power of the theory:

- RQ4: Is the PFIS approach to predicting programmer navigation significantly more accurate than an approach based on programmers' stated information needs and hypotheses?

To answer these questions, we made detailed observations of programmers engaged in debugging. We analyzed verbal protocols collected from our participants, focusing on what those verbalizations could tell us about the roles of hypotheses and scent in our theory and in operational derivatives of it such as PFIS.

4.1 Design, Participants, and Materials

The study design used the think-aloud method [Ericsson and Simon 1993]. Consistent with the method, we observed participants performing programming tasks and prompted them to "think aloud" as they worked. We recorded video of their sessions as well as a log of their actions. Using a grounded-theory approach [Corbin and Strauss 2008], we performed an in-depth qualitative analysis of their words and actions. Furthermore, we quantitatively analyzed the action logs to evaluate our model's predictions.

We recruited twelve professional programmers from IBM to participate in the study. All participants had at least two years experience programming in Java, used Java for the majority of their software development, and were familiar with Eclipse[6], a standard Java IDE, and with bug-tracking tools, such as Bugzilla[7].

As the subject program for our study, we selected RSSOwl[8], an open-source news-feed reader that is one of the most actively maintained and downloaded projects hosted at Sourceforge.net. The program met several key criteria: it was a real-world program; we had access to its source code and bug reports; it was written in Java; it was sufficiently large to allow enough navigation for useful analysis; and it was editable and executable through Eclipse. RSSOwl's source code consisted of 193 class files (80,520 lines of code). The popularity of newsreaders and the similarity of its user interface to email clients helped ensure that our participants would understand the functionality and interface after a brief introduction, and that they could quickly begin using and testing the program.

As tasks for our participants, we selected two bug reports from RSSOwl's bug-tracking database. We were more interested in source-code navigation than the actual bug fixes; therefore, we wanted to ensure that the issues could not be solved within the duration of the session. We also decided that one bug should be about a code-level defect, whereas the other bug should be a specification-level defect. The code-level defect involved a broken feature: bug #1458101: "HTML entities in titles of atom items not decoded." The specification-level defect involved a missing feature: bug #1398345: "Remove Feed Items Based on Age." We refer to the first bug as BF ("Broken Feature") and the second as MF ("Missing Feature"). Although MF might be a less traditional bug than BF (e.g., MF could be characterized as a feature request), this kind of bug report is common. Including it in our analysis adds diversity to the tasks studied and thus improves our ability to assess the general applicability of our findings. These two bugs were still open when we ran the study—that is, no RSSOwl developers had fixed them yet. We considered looking at closed bugs with solutions we could examine; however, we would have been forced to restrict participants' use of the web to ensure they did not find the existing solutions. Such a restriction would have diminished the realism of the task context, so we went with open bugs to preserve realism. Fortunately, the bugs remained open for the duration of the study.

4.2 Procedure

Initially, participants filled out paper work, and we briefly described RSSOwl. Next, we individually instructed them to find and fix the defects described in the bug reports. We set up an instant messenger client so that participants could contact us, then we excused ourselves from the room. Each participant worked on both bugs. We also

[6] http://www.eclipse.org/

[7] http://www.bugzilla.org/

[8] http://www.rssowl.org/

asked participants to "think aloud" as they worked. If a participant fell silent for an extended period of time, we prompted the participant over instant messenger to "please, keep talking." We counterbalanced the ordering of tasks among the participants to control for learning effects. We met with each participant for three hours, and observed each participant remotely for two hours.[9] We allowed each participant to spend only one hour per bug.

We recorded synchronized audio, video, and screen captures, which we were able to replay together, allowing us to hear what participants said, while observing their actions and the screens they were viewing. We also logged their events, and archived any changes they made to the program. The electronic transcripts of their actions, videos with screen captures, and source code served as the data sources we used in our analysis.

5. ANALYSIS METHODOLOGY

We discarded the data for two of the twelve participants: one was used as a pilot, and the data for the other was unreadable (a recording-tool failure). Our analyses of the verbal protocol data comprised three stages. In the first stage, we categorized the participants' explicit verbalizations about hypothesis processing and scent following. In the second stage, we categorized the artifacts that participants were looking at when they made certain types of statements. In the third stage, we performed a fine-grained analysis of two of the participants' verbalizations and videos. We describe the details of each analysis below.

For the first stage of analysis, categorizing verbalizations, we coded all ten participants' verbal protocols in the following manner. Initially, four researchers developed codes while jointly coding two of the protocols. After the four researchers cooperatively refined the initial code sets and developed norms about how to apply them, two researchers coded the remaining protocols, working independently and then checking for agreement. Thus, at least two researchers coded every protocol. The total rate of agreement was 97%, which indicates extremely high coding reliability. The 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. The first four rows of Table 1 define these hypothesis codes. 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 between scents that were and were not variants of previous scents sought. The last three rows of Table 1 define these codes.

[9] We devoted one hour to overhead: paperwork, briefing participants, a break between issues, and debriefing.

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 a hypothesis.	87: "So this newsfeedtitle string data value needs to be escaped."
hypothesis-confirm	Deciding a hypothesis was correct.	911: "So that's definitely the right spot."
hypothesis-abandon	Deciding a 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 a certain kind of information.	96: "So I need to find something about feeds."
scent-gained	Finding a scent (that may or may not have been sought)	98: "Let's figure out—oh, here's gui i18n translation."
scent-lost	Losing a 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 derived a set of "trigger" codes by identifying the artifact in use when a participant decided to pursue a hypothesis (hypothesis-start) or a scent (scent-to-seek). (We call them "triggers" because we considered these artifacts as potentially triggering the participants' decisions to pursue hypotheses or scent.) Table 2 lists these trigger codes. Because 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 unitalicized entries in the first column of 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 tested the scheme's robustness by having two researchers code two participants for each of the bugs (i.e., 20% of the data set). After reaching an agreement of 93%, which indicated that the codes were robust, one of the researchers applied the verified codes to the remaining videos.

Code	Definition
Bug-text	Looking at the bug report.
Runtime	Looking at runtime behavior.
Source-code	Looking at source code.
UI-static-inspection	Looking at the application's UI "statically" to identify pieces of functionality, the structure of the application's UI, etc., (as opposed to its runtime behavior).
Web	Browsing or searching web pages.
Other	Looking at anything else, e.g., looking at portions of the Eclipse UI itself.
Debugger-menus	Looking at debugger menus. (Debugger menus enumerate data field names, etc.)
Input-file	Looking at the input file, e.g., "So in an xml document, we have some title, we have some html entities…"

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 (i.e., four tasks in total). 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 *context*—

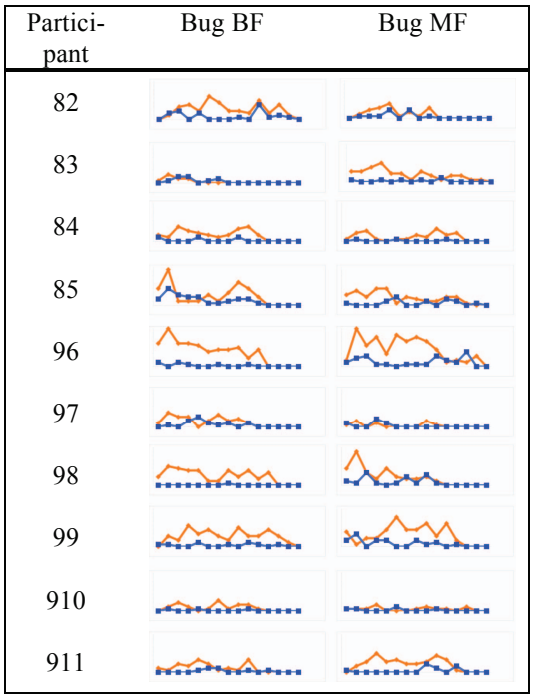


Figure 1. Thumbnails of each participant's hypothesis (dark blue) and scent (light orange) verbalizations over time. The x-axes are time (0 to 70+ minutes, in 5-minute intervals), the y-axes denote counts of verbalizations of each type (0 to 18).

why and when participants' hypotheses and scent following occurred. Since such fine-grained qualitative analyses are extremely time-consuming, we selected just four tasks. Our criteria were, first, that we were interested in extremes; second, task selection should be paired (i.e., two tasks from each of two participants); and third, participants must have performed tasks in different orders. We thus selected one participant (85) for whom our first stage of analysis revealed that hypothesis activity and scent activity peaks seemed to coincide. In particular, of all the sessions, participant 85's bug BF session had the highest correlation between hypothesis and scent counts, aggregated over 5-minute periods ($r(13)=0.7031$; see Figure 1). Participant 85's bug MF had one of the lowest correlations ($r(13)=-0.0825$), so it provided a good contrast. For the second participant we chose 98, because of the large distance between the hypothesis and scent curves in Figure 1 (a total count of 65 more scents than hypotheses for bug BF) and because Participant 98 worked on BF first, whereas 85 worked on MF first. For the two participants, three of the researchers watched the videos, taking into account what the participants said their goals were, what actions the participants took, how the participants interacted with artifacts, and what voice inflections and facial expressions the participants exhibited (such as surprise, or puzzlement). The researchers worked in tandem, watching the videos as a group and discussing each segment of video in detail. The three researchers worked through three of the four tasks, and two researchers finished up the remaining task.

6. RESULTS

In this section, we present empirical analyses aimed at building and evaluating our theory of information foraging during debugging. Specifically, we first analyzed the relationship between information-foraging navigation behavior and verbalized hypothesis-processing behavior (RQ1); second, we analyzed *when* information-foraging and

hypothesis-processing behavior occurred by considering each of the six debugging subtasks (or *modes*) that arose in our participants' data (RQ2); third, we analyzed *where* the participants sought scent by considering their interactions with various artifacts (RQ3); and fourth, we evaluated the predictive power of the theory by comparing the predictions produced by our PFIS model with those produced by variant models, including one based on hypotheses (RQ4). We did not analyze whether participants' utterances were "correct" in any way.

6.1 The Relationship between Scent and Hypotheses

The literature has long held that the debugging behavior of programmers is driven by hypothesis processing. We have reason to believe that our theory based on information foraging sufficiently accounts for hypothesis processing to explain and predict programmer navigation during debugging. In particular, we hypothesize that two conditions hold: (1) some hypothesis content closely parallels scent content and therefore is accounted for by our theory, and (2) the remaining amount of hypothesis-based processing (i.e., that is unaccounted for by our theory) is relatively small compared to the amount of scent processing. In this section, we check for these conditions with a qualitative analysis of all ten participants' scent-following and hypothesis-processing behavior.

Regarding the first condition, we found that when there was a relationship between hypotheses and scent, 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 working for some time on Bug BF, Participant 82 gained scent from 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 Bug BF 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, Participant 99 discovered the class "entity resolver" as he browsed through code while working on Bug BF. 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. We found strong evidence that verbalizations about scent occurred more often than verbalizations about hypotheses (paired t-test of log of counts: $p<.000004$, df=9, mean of 4.14 times as many scents as hypotheses). This finding not only held in the aggregate—it was true of each individual participant for both issues. Figure 1 depicts a profile of each participant's hypothesis and scent verbalizations over time.

In fact, the participants expressed few new hypotheses: sometimes only one or two original hypotheses per bug. Half the hypothesis starts were later followed by hypothesis abandons. Interestingly, participants confirmed hypotheses *only* when working on Bug BF. For both Bug BF and Bug MF, participants modified more hypotheses than they started (paired t-test of log counts: BF: $p<.00005$, df=9, mean of 5.57 times more modifications than starts, excluding one participant who started no hypotheses; MF: $p<.000006$, df=9, mean of 7.16 times more modifications than starts). Thus, it appears that once they had a hypothesis, these participants explored that initial hypothesis in depth. Apparent differences between BF and MF hypothesis counts in the table were not statistically significant.

	BF	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.

Scent processing showed a very different pattern. Here, participants gained scent more frequently than they lost it (paired t-test of log of counts: $p<.0002$, df=9, mean of 3.7 times more gains than losses). Participants verbalized lost scent less often than sought scent (paired t-test of log of counts: $p<.00005$, df=9, mean of 4.4 times more seeks than losses). Furthermore, scents gained were almost as frequent as scents sought (paired t-test of log of counts, $p<.025$, df=9, mean of 1.2 times more seeks than gains). However, these were not necessarily the same scents; scent gains were often serendipitous.

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 identified four categories of "bug-catching techniques," one of which essentially amounts to scent following. 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." Although we were counting instances of programmers' utterances and Eisenstadt was counting reported debugging strategies, his results were similar to ours in showing a heavy bias towards information seeking over hypothesizing. More recent works focused on software maintenance tasks, such as debugging. Although these works have not actually measured scent, they 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]).

6.2 When: A Fine-Grained Analysis by Debugging Mode

To understand the scope of our theory, we investigated *when* during debugging the participants engaged in information foraging. In particular, we analyzed our data to see when in their debugging activity programmers fol-

lowed scent and when they worked on hypotheses. We checked whether scent following pervaded all kinds of debugging activities and how it compared to hypothesis activity during different modes of debugging.

To consider this question, we performed a fine-grained analysis of Participant 85 and Participant 98, as explained in Section 5. For this fine-grained analysis, we watched video recordings of the sessions in order to take 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 evident when coding directly from the transcripts. The most prevalent of the additional instances were scent following through use of the pop-up method documentation (in tool-tip form). 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 were more conservative because they relied solely on utterances in which participants made their intent explicit. However, the data from both types of analyses were consistent: although the fine-grained analysis 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) *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 characterized 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.

Mode	Bug BF	Bug 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.

(2) *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) *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 Bug BF'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 reproduce 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. 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 Bug BF, 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) *Locate-the-fault* mode: We termed the participant to be in the "locate the fault" mode when he or she was currently seeking the location of the fault in the code. In the case of bug MF, we assigned this mode when the prey was the "hook" in the code where changes should be made. 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 with the idea that he should copy some code 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) *Fix-the-fault* mode: Once a participant located the fault, it was not always obvious how exactly to fix it. In fact, for 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 devising a fix 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 Bug MF required extensive refactoring of existing code. The environment's refactoring tools supported the procedure, but these tools also introduced new problems. In all, Participant 85 spent most of the time 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 he was never able to spend much time on the 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. However, he ran out of time before he could make progress with the second approach.

(6) *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 the two participants produced a fix complete enough to evaluate. The results of his first evaluation led him to change his fix, and then evaluate it again.

Using these modes, the graphs in Figure 2 show activity of hypotheses and scent for each of these participants and tasks, mode-by-mode, with each mode on a separate row. The thick gray horizontal bars show the length of time each participant spent in that mode. The blue hash marks below the gray bars denote hypothesis verbalizations, and the red hash marks above the lines denote scent seeking (verbalizations and actions to request the pop-up explanations). Longer hash marks indicate two or more events of the same type in quick succession.

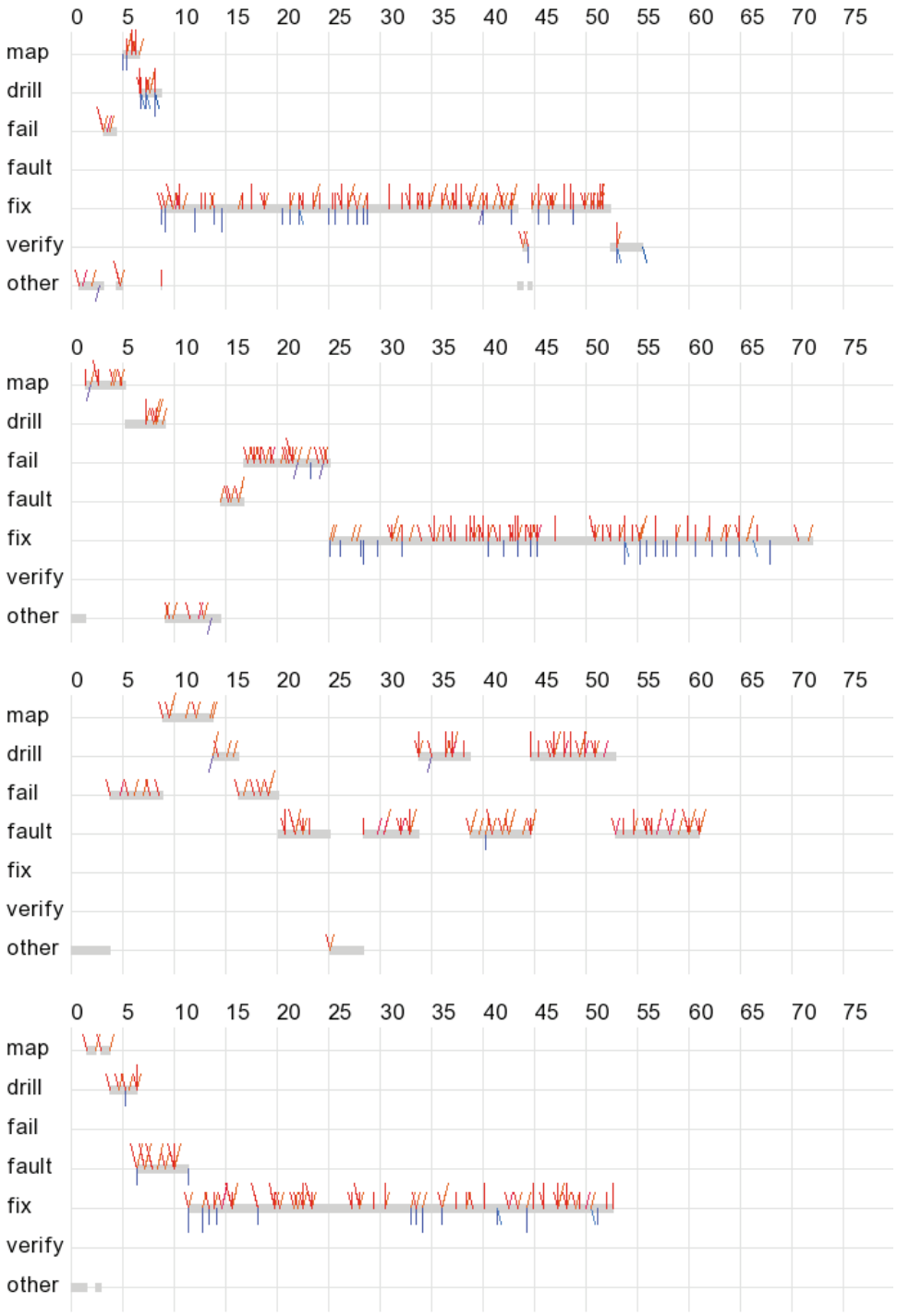


Figure 2: Scent and hypothesis events observed for each mode. Top to bottom: Participant 85 Bug BF (done as task #2), Participant 85 Bug MF (done as task #1), Participant 98 Bug BF (done as task #1), Participant 98 Bug MF (done as task #2). Gray horizontal bar shows mode is in effect: . Red hash marks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

The final row, "other," accounts 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 the predator modifies his environment to optimize the environment's affordances for foraging. 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 were pervasive among all of the modes, although sometimes they were very short, such as when a participant rearranged windows. (Enrichment was categorized as "other" only if it was relatively independent of any of the six debugging modes.)

As the figure shows, scent seeking permeated all six modes. Recall from Table 3 that participants expressed scent following much more often than hypotheses, and that this phenomenon occurred consistently over time. Figure 2 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.

76 of the 101 hypothesis verbalizations occurred 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 only three hypothesis verbalizations for Bug BF (see Figure 2). Many hypothesis modifications in Fix mode involved the participants evolving programming plans as they worked on their fixes. 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 rendered obsolete 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 he or she found or lost it. However, such bookending of hypotheses and scents was 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 behavior 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].

Also consistent with our observations, Hollan et al.'s thesis of distributed cognition [Hollan et al. 2000] would predict that scent processing dominated across all debugging modes. They assert that "the organization of mind … is an emergent property of interactions among internal and external resources." Moreover, they argue that a large part of cognition is triggered by interaction with the environment, rather than happening predominantly in the head.

6.3 Where: In Which Artifacts Participants Sought Scent and Formed Hypotheses

To further understand the scope of our theory, we investigated *where* the ten participants directed their attention as they engaged in information foraging. Studying which artifacts in the environment triggered participants to follow scent revealed the kinds of artifacts a computational model needs to analyze to predict programmers' navigation behavior. To investigate this issue, we analyzed the data for all ten participants to see which artifact a participant was looking at during each scent-to-seek verbalization (recall Table 1 and Table 2). For each such event, we refer to the artifact as the *scent trigger*. As a point of comparison, we performed a similar analysis for *hypothesis triggers* for each hypothesis-start verbalization.

Figure 3 depicts the results of our analyses. We found that, for both Bug BF and Bug MF, participants' pursuits of scent were triggered in the source code far more than anywhere else. Other noticeable triggers included web resources, the runtime behavior of RSSOwl, and the bug report itself (Bug BF or MF). These results were consistent across participants—each information source was used by at least seven participants (and no participant using fewer than four information sources), with source code being the dominant information source for every participant.

The distribution of the scent triggers was also reasonably consistent between Bugs BF and MF, as can be seen by comparing the light and dark bars on the left side of Figure 3. When starting a new hypothesis, however, trigger patterns were less clear because of the low number of hypotheses. However, it appears that, although the bug description itself triggered the hypothesis over one-fourth of the time, no trigger dominated the others as source code did for scent seeking. Other than the bug-report text triggers, the distribution of triggers varied greatly between Bug BF and Bug MF (light and dark bars on the right side of Figure 3, respectively). For Bug BF, hypothesis triggers were mostly in the bug report, source code, and the program input; for Bug MF, hypothesis triggers were mostly in the bug report and web resources. As expected, comparing the graphs in Figure 3 shows that these artifacts triggered far more scent-seeking behavior than hypothesis-processing behavior.

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 the analysis of source code relative to the bug report to obtain predictions, without incurring the expense of the more costly analyses of runtime data, the web, and so on. This concurs with the PFIS modeling approach, which relies solely on static analysis of source code relative to the bug report.

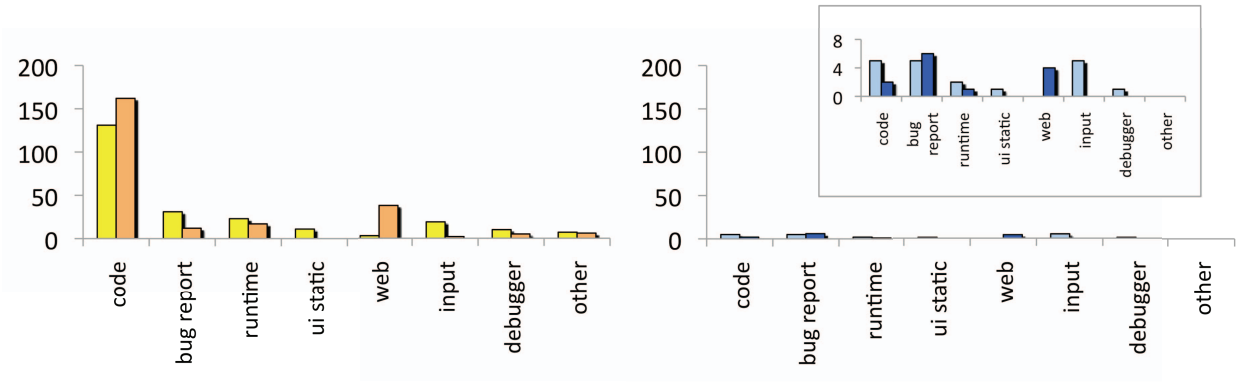


Figure 3. Scent to seek (left) and hypothesis (right) verbalizations: what triggered participants to seek scent or form new hypotheses. Light bars represent triggers for Bug BF; dark bars represent triggers for Bug MF. The influence of artifacts on hypotheses was overwhelmed by these artifacts' influences on scent. For that reason, the small inset graph blows up the data to allow comparison of the most influential artifacts on hypotheses.

6.4 Predictors of Where Participants Sought and Found Scent

Our fourth research question asked whether PFIS predicts programmer navigation better than programmers' stated information needs and hypotheses. To answer this, we evaluated aspects of PFIS in two stages, using the navigation data from all ten participants.

First, we compared three different sources of information as proxies for the programmers' information needs to see which of these was most predictive of navigation: verbal scent-to-seek, verbal hypotheses, and the bug report. We used as predictors the information's similarity to the method to which a programmer navigated, calculated via cosine similarity (explained in Section 2) between the text of the information source and the text of the method. We term these three predictors *verbal-scent-distal*, *verbal-hypothesis-distal*, and *bug-report-distal*. "Distal" emphasizes that these predictions are based only on the relevance of the place the programmer navigates to, rather than the proximal cues they follow to get there.

We considered each predictor's 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" for *verbal-scent-distal*, or "hypothesis start" for *verbal-hypothesis-distal*). Each predictor produced a rank-ordered list of predicted next navigations; ideally, the method to which the programmer actually navigated (the "true" navigation) would be ranked number 1. Table 5 shows the mean and median ranks that each predictor assigned to the observed navigations.

Paired Wilcoxon signed rank tests showed that the *bug report* was significantly better at predicting participants' navigations than their verbalizations. This was the case for both *verbal hypotheses* ($p<.00001$, $N=831$, $V=33150.5$, Wilcoxon, mean difference in ranks=50.3) and *verbal scent-to-seek* ($p<.00001$, $N=2171$, $V=250697$, mean difference in ranks=49.9) (see Table 5). This result is probably due to two factors: the phenomenon of "invisibility" of hypotheses in verbal protocols in which participants neglect to state their hypotheses (cf. [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 context dependent, and also contained deixis (e.g., "Where is that?"), interjections (e.g., "Ahhh"), and disambiguations using non-explicit mechanisms, such as saying a word or phrase as a question (e.g., "getTitle?" indicating continuing search for information) or as an exclamation (e.g., "getTitle!" indicating having found it).

Second, using the bug report as input, we compared two algorithms: simple cosine similarity (the same *bug-report-distal* as before) and the full PFIS algorithm (which we call *bug-report-proximal+topology*). "Proximal" emphasizes that PFIS predicts based on the relevance of nearby, visible cues, and "topology" is a reference to PFIS's spreading activation algorithm, which is detailed in Appendix A.

For Bug BF, *bug-report-proximal+topology* (PFIS) performed significantly better than *bug-report-distal* at predicting participants' navigations ($p<.00001$, $V=1062158$, $N=3002$, mean rank difference=22.3). In the best case, PFIS's recall approached 100% at a false-positive rate of 25-30% for the bug. This suggests the overall suitability of this combination to model programmer navigation during debugging.

However, for Bug MF, PFIS did not perform as well as the cosine similarity measure: *bug-report-distal* was more effective than *bug-report-proximal+topology* in predicting where participants went (Table 5; $p<.00001$, $V=534252$, $N=3002$, mean rank difference=15.3). We speculate that this may be related to different characteristics of BF and MF. A previous study involving 228 bugs and 25 feature requests from a different open-source project (jEdit) has also suggested differences in the utility of bug reports versus feature requests for predicting where programmers *should* navigate on the basis of distal scent [Lawrance et al. 2008a]. The decrease in PFIS's accuracy on the Bug MF task suggests that different types of bug reports and tasks may call for variants of the model.

The performance of different predictors can be visualized using a receiver operating characteristic (ROC) curve. In an ROC, the true positive rate is plotted on the vertical axis against the false positive rate on the horizontal axis, for all possible choices of decision threshold point. For example, in the left graph of Figure 4, the PFIS (BF) label pointer touches the curve at a point where true positives are greater than 0.9, and false positives are about 0.2. This point represents one possible threshold value that could be applied to PFIS to control the size of its list of predictions. This particular threshold choice would result in a 90% certainty of containing the one true next navigation, and a 20% certainty of containing any given wrong navigation. Thus if 1000 possible navigation locations were available to choose from, this predictor would have a 90% chance of containing the right choice within the top 200. The graphs support the statistical outcomes above, and in general show that bug reports were better than verbalizations at predicting subsequent navigation.

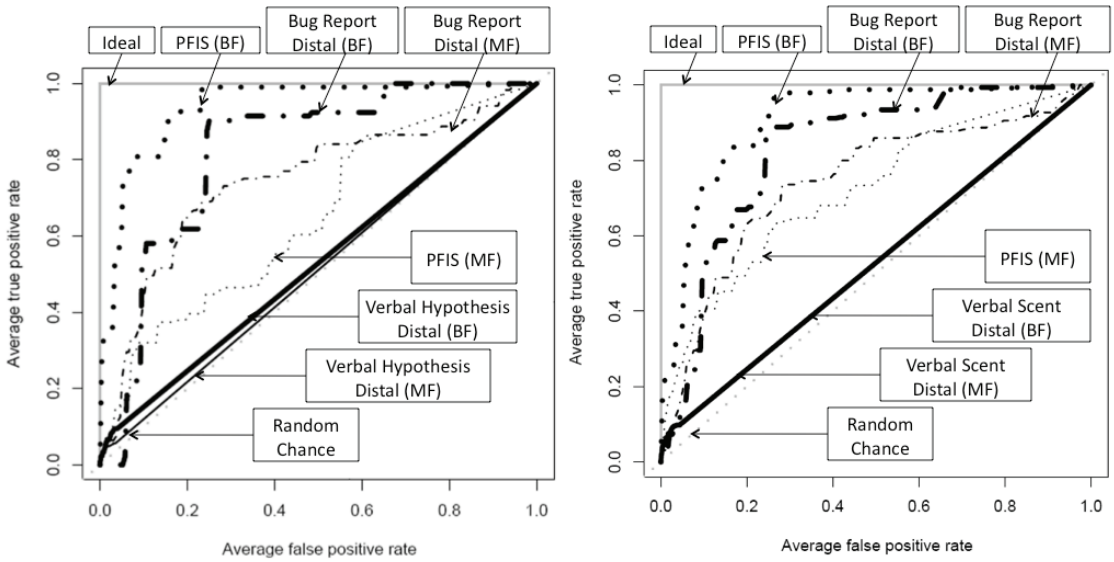


Figure 4. (Left): *Verbal hypothesis* ROC curves: Ability to predict classes to which participants navigated directly after they expressed a hypothesis (i.e., just after "hypothesis start" or "hypothesis modify" verbalizations). Bug BF has thick lines; Bug MF has thin lines. (Right): *Verbal scent-to-seek* 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).

Predictor	Median prediction rank		Mean prediction rank	
	Bug BF	Bug MF	Bug BF	Bug MF
verbal-scent-distal	97	97	91.3	91.6
verbal-hypothesis-distal	97	97	91.7	94.6
bug-report-distal	25	37	42.7	64.0
bug-report-proximal+topology (PFIS)	13	47	20.3	79.3

Table 5. Median and mean prediction ranks for each of the predictors shown in Figure 4.

7. 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.) Appendix B discusses three different types of threats to validity of our experiment: external, internal, and construct validity. Threats to external validity limit the extent to which results can be generalized. Threats to internal validity are other factors that may be responsible for an experiment's results. Threats to construct validity question whether the measures in an experiment's design adequately capture the effects that they were intended to capture. Appendix B also discusses how we attempted to mitigate the impact of these threats on our results.

8. DISCUSSION AND CONCLUSION

In this paper, we have presented an information foraging theory that predicts programmer's navigation choices when debugging source code in response to a bug report. The results show that the scent and topology constructs of this theory are valuable constructs in a predictive model. In particular, we found that:

- RQ1: The way participants worked with scent was consistent with information foraging theory. The participants verbalized activities related to scent about four times as often as (non-scent) hypotheses. The relationships between scent and some types of hypotheses may have contributed to the effectiveness of scent as a predictor.
- RQ2: 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, the predominant "fix" phase. This finding also helps to explain why scent was so effective at predicting programmer navigation.
- RQ3: The biggest "trigger" for scent following was the source code itself, but other triggers included 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 accurate predictions and, second, that even greater accuracy may be possible if a model includes these additional data sources.
- RQ4: The PFIS model, which operationalizes our theory, was more accurate at predicting programmer navigation behavior than comparable models that do not consider information scent.

Turning our attention to practical implications of the theory, we now briefly discuss how it potentially could be used as a basis for designing software engineering tools.[10] (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.) The constructs of information foraging theory can provide design guidelines for the makers of interactive debugging tools. 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 patches by leveraging proximal cues and topology. For example, suppose a tool designer aims to support debugging by providing visual access to runtime information. Absent a psychological theory, but familiar with the language interpreter's implementation details, the designer might be tempted to structure the topology of the visualizations in a way that mirrors the way the underlying data is structured, and label links between them with cues reminiscent of the variable names in the interpreter. Information foraging theory gives the designer a different way of approaching the problem. The designer should first identify the universe of information goals (prey) that the tool will support, and the cues that users are likely to associate with each of these goals. Once the designer has done this psychological research, IFT can guide him or her in laying out an information topology in which each screenful of information has embedded cues that will help bring the programmer a step closer to his or her goal, whatever that goal may be. The tool may also allow the programmer to manipulate and customize cues and the topology (i.e., to perform enrichment) to improve the efficiency of finding and navigating to particular types of information. Although these implications might seem to simply say "employ good tool design," they make explicit theory-based criteria for doing so, enabling designers to take a more principled approach.

[10] For an in-depth treatment of this topic, see [Scaffidi et al. 2010].

One design challenge is that much of the quality of cues is outside the control of the tool designer, because cues 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, in hypertext documentation of source code, in bug reports, and so on. Thus, 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 our theory itself. In future work, we plan to empirically explore and evaluate the use of the theory for the practical design of tools. Although we have empirically evaluated our theory's predictive power using PFIS, we only investigated one type of computing scent, linguistic similarity. In the future, we plan to explore other possible types of scent, such as structural relatedness and code complexity. We also have yet to consider enrichment activities and non-code artifacts in our model. It is an open question whether incorporating these factors into the model will impact its predictive power and whether any benefits will outweigh the computational overhead. Finally, both the current study and a previous one suggested differences in the ability of the PFIS model to predict program navigation for Bug BF versus Bug MF. Further research is needed to determine the scope of our theory across different types of debugging tasks. Moreover, we also plan to extend and evaluate information foraging for tasks beyond debugging, such as code refactoring and reuse tasks.

In the future, we believe information foraging theory can provide a fundamental understanding of why 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, for instance, by making clear that tools need to consider information-foraging factors, such as scent and topology. 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.

APPENDIX A: PFIS

PFIS is based upon the web user flow by information scent (WUFIS) algorithm [Chi et al. 2001], which combines information retrieval techniques with spreading activation. As WUFIS does for web path following, PFIS calculates the probability that a programmer will follow a particular "link" from one class or method in the source code to another, given a specific information need. The implementation of PFIS is similar to systems that derive relations in source code on the basis of source-code structure (e.g., [Long et al. 2009], [Robillard 2005], [Robillard 2008], [Saul et al. 2007], [Hill et al. 2007]). PFIS, as an executable model of a theory, is not a system for programmers; however, it was originally motivated by systems that derive useful relations from navigation and editing actions (e.g., [Shirabad et al. 2003], [Ying et al. 2004], [Zimmermann et al. 2004], [Cubranic et al. 2005], [DeLine et al. 2005], [Kersten and Murphy 2005], [Schneider et al. 2004], [Schummer 2001], [Singer et al. 2005]).

PFIS is summarized in Figure 5. We explain how each step was accomplished next.

Central to WUFIS is a description of the link topology of the web site, describing each link in terms of which page it is on, and which page it points to. For example, Figure 6 shows on the left four nodes, and the links between them. In WUFIS, the nodes are web pages; in PFIS the nodes are anything that is the destination of a link (e.g.,

Algorithm PFIS (Programmer Flow by Information Scent)

Given: Bug report, body of source code

Returns: A vector containing for each package, class, method, and variable, the probability that a programmer will visit that area of source code given the bug report.

Step 1. Determine link topology of source code and store them in matrix T.

Step 2. Determine set of proximal cues around each link.

Step 3. Determine information scent of each link to the bug report, and store the resulting similarity scores in matrix PS.

Step 4. Normalize matrix PS so that each column sums to 1.00 (i.e., generate a column-stochastic matrix).

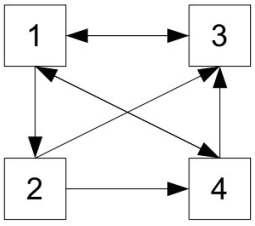
Step 5. Define the starting place (class or method) for the spreading activation, and create an entry vector E with the starting location given a 1.

Step 6. Calculate the probability of programmers going from the starting location to other documents by multiplying the entry vector E=A(1) by PS, to create an activation vector A(2).

Step 7. Repeat step 6 to simulate programmers traversing through the link topology. The final activation vector A(n), when normalized, contains for each location the expected probability that a programmer will visit that location given the bug report.

Figure 5. The PFIS algorithm.

method definitions, method invocations, and variable definitions). The link topology is described by the matrix on the right. For step 1 of the PFIS algorithm, to create the link topology of source code, we created an Eclipse JDT plugin to traverse each class and method in each compilation unit, and used the Java Universal Network/Graph Framework to construct the link topology (adjacency matrix) **T**, which gives us the beginning **(i)** and end points **(j)** for each link that a programmer can follow.



Matrix T

To \ From	1	2	3	4
1	0	0	1	1
2	1	0	0	0
3	1	1	0	1
4	1	1	0	0

Figure 6. Link topology (adjacency) matrix 'T'.

Steps 2 and 3 determine the information scent of each link relative to the bug report (Figure 7). Thus, PFIS treated the text of the bug report as the query, and the proximal cues of each link as a document. The scent of each link is determined by the cosine similarity of the words in the bug report to the text that labels the link and the text in close proximity to the link. Lucene determined the cosine similarity of each link in relation to the bug report to de-

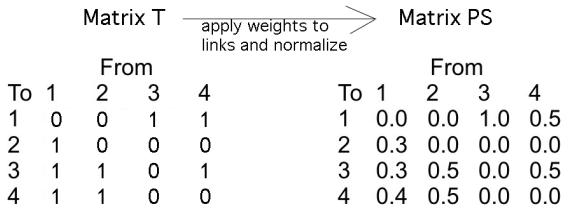


Figure 7. Create the normalized proximal scent matrix 'PS' by weighting edges in 'T' according to the cosine similarity scores computed using Lucene.

termine the scent of each link. We then used these results as weights for the edges in **T**, producing a proximal-scent matrix **PS**.

In step 4, PFIS normalizes **PS** so that each column sums to 1, thus producing a column-stochastic matrix. In effect, each column contains the probability that a programmer will follow a link from one location to another. Thus, at the end of step 4, the proximal scent relative to the bug report has been calculated, reflecting the information foraging premise that links in the source with proximal cues close to the important words in the bug report will smell more strongly of the bug, and are thus more likely to be followed.

Steps 5, 6 and 7 simulate programmers navigating through the source code, following links based on scent. Spreading activation is an iterative algorithm used widely by HCI theories in which phenomena spread, and by information foraging theory in particular. It calculates how widely the spreading emanates. For PFIS, spreading activation calculates the likely spread of programmers to locations in source code, which can be interpreted as the expectation that a programmer trying to resolve a particular bug report will navigate to those locations in the program.

Spreading activation takes an activation vector ***A***, a scent matrix ***PS***, an entry vector ***E***, and a scalar parameter α. The parameter α scales ***PS*** by the portion of users who do not follow a link. In the initial iteration, the activation vector equals the entry vector. The entry vector is derived from the location in the source code where each participant began navigation. Activation is updated (spread) in each iteration ***t*** as follows [3]:

$$A(t) := \alpha PS * A(t-1) + E$$

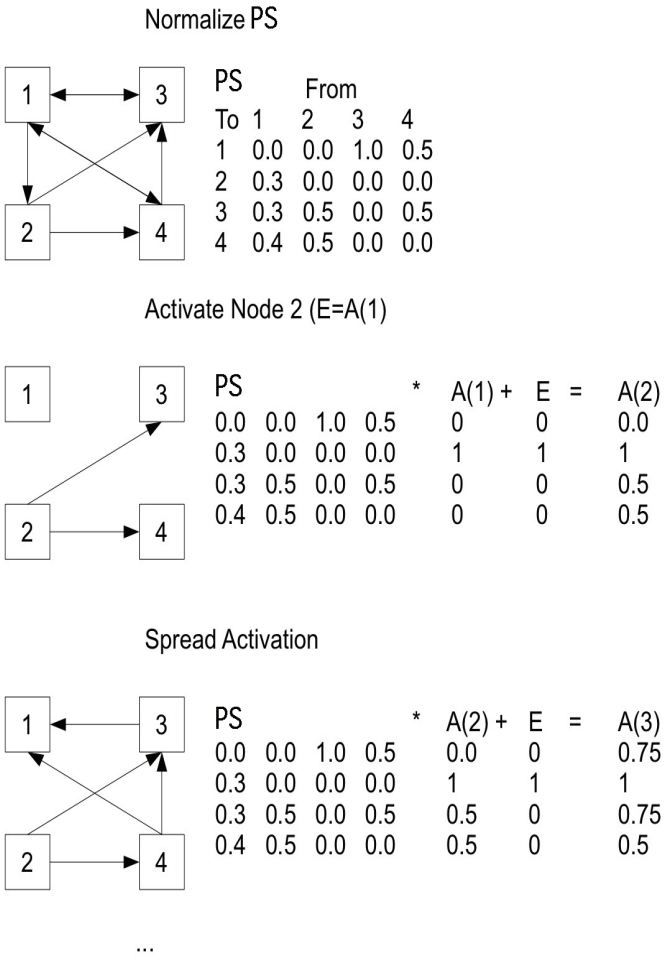


Figure 8. Example of application of spreading activation to matrix PS.

In each iteration, activation from the entry vector is spread out to adjacent nodes, and activation present in any node is spread to neighboring nodes according to the scent (i.e., the edge weights in PS). In the final iteration, activation vector A represents the activation of each node (package, class, method, field) in our topology T. Normalizing A, we interpret A as the probability of a hypothetical user visiting that node in T. See Figure 8. The final activation vector makes predictions about where programmers navigated. Although PFIS reasons at the granularity of method-to-method navigations, we observed programmer navigation at the class level. Therefore, we aggregated method predictions across classes to associate PFIS predictions with observed behavior.

APPENDIX B: THREATS TO VALIDITY

B.1 External Validity

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 this 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. As newcomers to the project, participants may have pursued a comprehension strategy different than that of a senior member of the project [Sim and Holt 1998].

The ability to generalize our results may be limited by our selection of bug reports. We attempted to address this issue by choosing one bug that involved a broken feature (BF) and one that involved a missing feature (MF). However, our study included only these two bugs, and it seems unlikely that only two bugs can reasonably represent bugs in general. One other study we performed analyzed a number of other bugs in other open-source projects [Lawrance et al. 2008a], but in contrast to the current study, that study investigated the use of bug reports for predicting the files programmers ultimately changed. Thus, further work is needed to generalize the findings of the current paper.

The limited size of our sample also limits our study's generalizability. When conducting in-depth qualitative analyses of the sort done in this study, practical constraints limit the number of participants, which in turn 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 programmer navigation behavior is context dependent, and therefore programmer 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 threats to external validity can be addressed through repeated studies, using different source code, different issues, and/or different programming environments.

B.2 Internal Validity

We imposed a time limit, which was 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.

B.3 Construct Validity

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, verbalized 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 the 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 information 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 costs 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. Furthermore, 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 *bug-report-distal* and *bug-report-proximal+topology (PFIS)* might on the surface appear to be confounded: the reader may wonder if the differences are due to the distal/proximal difference, or the fact that the latter involved a spreading activation process over the program's topology. However, proximal scent is inseparable from topology, 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 *bug-report-distal* would predict navigation to that class, whereas *bug-report-proximal+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.

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.

[Brooks 1999] Brooks, R. Toward a theory of the cognitive processes in computer programming, Int. J. Human-Computer Studies, vol. 51, 197–211, 1999.

[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.

[Corbin and Strauss 2008] J. Corbin and A. Strauss, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 2008.

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

[Cubranic et al. 2005] D. Cubranic, G. Murphy, J. Singer and K. Booth, "Hipikat: A project memory for software development," *IEEE Trans. Soft. Eng. vol. 31*, no. 6, pp. 446–465, 2005.

[DeLine et al. 2005] R. DeLine, M. Czerwinski and G. Robertson, "Easing program comprehension by sharing navigation data," In *Proc. IEEE Symp. Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE pp. 241–248, 2005.

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

[Ericsson and Simon 1993] K. A. Ericsson and H. A. Simon, *Protocol Analysis: Verbal Reports as Data*. MIT Press, 1993.

[Fritzson et al. 1991] P. Fritzson, T. Gyimóthy, M. Kamkar, N. Shahmehri: "Generalized Algorithmic Debugging and Testing," In *Proc. ACM SIGPLAN 2010 Conf. Programming Language Design and Implementation (PLDI)*, pp. 317–326, 1991.

[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.

[Hill et al. 2007] E. Hill, L. Pollock and K. Vijay-Shanker, "Exploring the Neighborhood with Dora to Expedite Software Maintenance." In *Proc. Int. Conf. Automated Software Engineering (ASE)*, pp. 14–23, 2007.

[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.

[Kersten and Murphy 2005] M. Kersten and G. Murphy, "Mylar: A degree of interest model for IDEs," In *Proc. Aspect-Oriented Software Development Conf. (AOSD)*, 2005.

[Ko et al. 2005] A.J. Ko, H. Aung, and B. A. Myers, "Eliciting design requirements for maintenance-oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks."In *Proc. Int. Conf. Software Engineering (ICSE)*, pp 126–135, 2005.

[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 (VL/HCC)*, 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 (VL/HCC)*, 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 (CHI)*, 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.

[Long et al. 2009] F. Long, X. Wang, Y. Cai: "API Hyperlinking via Structural Overlap." In *Proc. ESEC/ACM SIGSOFT Symp. Foundations of Software Engineering (FSE)*, pp. 203–212, 2009.

[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 (CHI)*, pp. 3–10, 1997.

[Pirolli and Card 1999] P. Pirolli and S. Card, "Information foraging," *Psychology Rev.*, 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.

[Robillard 2005] M. Robillard, "Automatic generation of suggestions for program investigation." In *Proc. ESEC/ACM SIGSOFT Symp. Foundations of Software Engineering (FSE)*, pp. 11–20, 2005.

[Robillard 2008] M. Robillard. "Topology Analysis of Software Dependencies." *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 4, 2008.

[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.

[Saul et al. 2007] Z. M. Saul, V. Filkov, P. Devanbu and C. Bird, "Recommending Random Walks." In *Proc. ESEC/ACM SIGSOFT Symp. Foundations of Software Engineering (FSE)*, pp. 15–24, 2007.

[Scaffidi et al. 2010] C. Scaffidi, S. Fleming, D. Piorkowski, M. Burnett, R. Bellamy, and J. Lawrance, "Unifying Software Engineering Methods and Tools: Principles and Patterns from Information Foraging," (under review).

[Schneider et al. 2004] K. Schneider, C. Gutwin, R. Penner and D. Paquette, "Mining a software developer's local interaction history," In *Proc. Intl. Workshop Mining Software Repositories*, 2004.

[Schummer 2001] T. Schummer, "Lost and found in software space," In *Proc. Hawaii Int. Conf. System Sciences (HICSS)*, 2001.

[Shirabad et al. 2003] J. Shirabad, T. Lethbridge and S. Matwin, "Mining the maintenance history of a legacy system," In *Proc. Int. Conf. Software Maintenance (ICSM)*, IEEE, 2003.

[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.

[Sim and Holt 1998] S. E. Sim and R. C. Holt, "The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize." In *Proc. International Conference on Software Engineering (ICSE)*, pp. 361–370, Apr. 1998.

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

[Sjøberg et al. 2008] D. I. K. Sjøberg, T. Dybå, B. C. D. Anda, and J. E. Hannay, "Building Theories in Software Engineering," *Guide to Advanced Empirical Software Engineering*, Springer, pp. 312–336, 2008.

[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.

[Ying et al. 2004] A. Ying, G. Murphy, R. Ng and M. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Software Engineering,* vol. 30, pp. 574–586, 2004.

[Zeller 1999] A. Zeller, "Yesterday, My Program Worked. Today, It Does Not. Why?" *ESEC/ACM SIGSOFT Symp. Foundations of Software Engineering (FSE),* pp. 253–267, 1999.

[Zimmermann et al. 2004] T. Zimmermann, P. Weissgerber, S. Diehl and A. Zeller, "Mining version histories to guide software changes," In *Proc. Int. Conf. Software Engineering (ICSE), IEEE*, 2004.

Joseph Lawrance is an Assistant Professor in the Computer Science and Systems Department at Wentworth Institute of Technology. His research interests include human factors in programming and debugging, particularly information foraging theory applied to debugging. He has a PhD in Computer Science from Oregon State University in 2009. Contact him at lawrancej@wit.edu.

Chris Bogart is a PhD student at Oregon State University's School of Electrical Engineering and Computer Science. His current research involves human factors in debugging, comprehension, and validation, particularly relating to scientific modeling.

Margaret Burnett is a professor in Oregon State University's School of Electrical Engineering and Computer Science. Her current research focuses on end-user programming, end-user software engineering, information foraging theory as applied to programming, and gender issues in those contexts. Burnett has a PhD in computer science from the University of Kansas. Contact her at burnett@eecs.oregonstate.edu.

Rachel Bellamy is the Manager of the Software Productivity group at IBM Research. She conducts research into the psychology of programming, expert debugging, parallel programmer debugging, end-user modeling, design tools for productivity assessments and is a practitioner of user-centered design. Before coming to IBM, she worked in Apple Computer's Advanced Technology Group and at The University of Cambridge. Contact her at rachel.us.ibm.com.

Kyle Rector received her bachelor's from Oregon State University in Electrical and Computer Engineering and Computer Science. Her research interests lie in Human Computer Interaction with respect to gender and Information Foraging Theory as it applies to debugging. Rector will be starting her PhD at the University of Washington in Fall 2010.

Scott D. Fleming is a research associate (postdoc) at Oregon State University's School of Electrical Engineering and Computer Science. His research interests include human aspects of software engineering, concurrent software, end-user software engineering, and design and usability of software engineering tools. Fleming has a PhD in computer science from Michigan State University.