2016 IEEE/ACM 13th Working Conference on Mining Software Repositories

Interactive Exploration of Developer Interaction Traces using a Hidden Markov Model

Kostadin Damevski
Virginia Commonwealth University
Richmond, VA 23284
damevski@acm.org

Hui Chen
Virginia State University
Petersburg, VA 23806
huichen@ieee.org

David Shepherd
ABB Corporate Research
Raleigh, NC 27606
david.shepherd@us.abb.com

Lori Pollock
University of Delaware
Newark, DE 19350
pollock@udel.edu

ABSTRACT

Using IDE usage data to analyze the behavior of software developers in the field, during the course of their daily work, can lend support to (or dispute) laboratory studies of developers. This paper describes a technique that leverages Hidden Markov Models (HMMs) as a means of mining high-level developer behavior from low-level IDE interaction traces of many developers in the field. HMMs use dual stochastic processes to model higher-level hidden behavior using observable input sequences of events. We propose an interactive approach of mining interpretable HMMs, based on guiding a human expert in building a high quality HMM in an iterative, one state at a time, manner. The final result is a model that is both representative of the field data and captures the field phenomena of interest. We apply our HMM construction approach to study debugging behavior, using a large IDE interaction dataset collected from nearly 200 developers at ABB, Inc. Our results highlight the different modes and constituent actions in debugging, exhibited by the developers in our dataset.

CCS Concepts

•**Human-centered computing** → *Empirical studies in HCI;* •**Software and its engineering** → *Maintaining software;* •**Computing methodologies** → *Modeling methodologies;*

Keywords

field studies; IDE usage data; hidden-markov model

MSR'16, May 14-15 2016, Austin, TX, USA

ISBN 978-1-4503-4186-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2901739.2901741

1 Introduction

Modern IDEs provide an encompassing set of software development tools that capture the majority of development tasks, by including support for development, advanced software navigation, version control, debugging, and testing. Mining how developers interact with the IDE can aid researchers in understanding developer behaviors in the field, without the observational bias present in laboratory studies, supporting laboratory studies' findings and understanding them in wider contexts. In addition, field IDE data analyses can help in making the IDE more usable, by, for instance, exposing hidden, rarely used capabilities of the IDE in the contexts that are most useful to developers [13].

Large-scale IDE usage data has previously been collected and put to a variety of uses in software engineering research, including to understand developer behavior in feature location [4], refactoring [15,21], bug prediction [10], and general IDE usability [13]. Most such studies so far have learned from simple developers' interactions (i.e., clicks and key presses), while high-level behaviors of developers, exhibited by sequences of several IDE interactions, have rarely been modeled or studied.

Hidden Markov Models are adept at capturing higher-level intent using a large set of simple events as input, which is typical of IDE usage data. However, as they are typically used for prediction, HMMs often produce models that are difficult to interpret by humans, or are not focused on the phenomena of interest. In cases where the number of different events (or symbols) in the dataset is large, as in IDE interaction data, HMM interpretation is nearly impossible. The technique that we propose for IDE data analysis, adapted from previous work by Jarosewicz [7], interactively builds up an HMM by integrating expert feedback at each step, while producing several informative representations of the model and input dataset to guide the expert. The approach avoids building a model that is difficult to interpret post-hoc, as interpretation is in fact performed at each step of the process, using one new state at a time.

This paper presents the following contributions:

- an interactive approach of building an interpretable HMM model of developer behavior based on IDE usage data;

- formulation of the problem of analyzing debugging behavior as an HMM; and
- results from a large-scale study of using interactive HMM mining, applied to uncovering developers' debugging behavior.

The paper is organized as follows. We begin by describing the related work in Section 2, followed by a detailed description of the interactive approach for HMM construction in Section 3. We apply the algorithm to a large IDE dataset to discover developers' debugging behavior, and describe how the dataset was processed in Section 4 followed by the results of the analysis in Section 5.

2 Related Work

The work most similar to ours is that of Khodabandelou et al. [8], who first proposed a Hidden Markov Model (HMM) approach for mining IDE interaction datasets to understand high-level developer intent. This intentional process mining approach produces one HMM model, based on a set of heuristics. However, the mined model may not answer the research question, which another model, of similar quality, could. Our paper builds on this work, by proposing an interactive process of building HMMs that highlight a relevant aspect of developer behavior.

Other related work can be organized in two categories: other mining approaches that leverage IDE usage data to understand developers' behavior, and mining models from human-generated event data, in general.

Many other studies have analyzed developer behavior based on IDE interactions. Recently, Minelli et al. analyzed the time for different development activities (e.g., program comprehension) spent by developers in the field [11], highlighting the high amount of time for program comprehension and the high negative impact of interruptions on developer work. Corley et al. studied the impact of developer interruptions by also considering the web usage patterns of developers [3]. Several studies have measured the types of commands more frequently used (or not used) in the IDE, leading to the description of specific usability problems [12, 14].

A technique called process mining has been gaining traction in recent years as a means of capturing human (or business) behavior based on event data [22]. The mining techniques used in this domain differ mainly in the types of models used, including petri nets, state machines, and others. Recently, some of these approaches have been applied to software systems' logs [18], to help understand user behaviors. Most of these approaches, however, capture a direct representation of the logs, while HMMs aim to capture higher level hidden human behavior. Due to the direct representation of the logs, such approaches may also have difficulty scaling to a large alphabet of log messages.

3 Interactive HMM Construction

Hidden Markov Models (HMMs) are probabilistic state machines that have been shown to have a variety of uses, most notably, in speech recognition and signal processing [17]. Their predictive power comes from modeling two distinct, but tightly-coupled, stochastic processes, one that is observable and one that is hidden. The state of the hidden process is the output of the prediction at any particular moment in time, while the observable process directly tracks the input data.

Here, we are not interested in prediction, but rather in the descriptive power of HMMs. HMMs have previously been proposed as an appropriate form for inferring high-level human cognitive processes, represented by the HMM's hidden states and their transitions [8].

In the remainder of this section, we will describe how HMMs can be used for modeling human processes in software engineering, based on IDE interaction datasets, including how they can be built interactively, with interactive feedback, to best capture a particular hidden process. To ease the discussion, we first present a general discussion of the method, then proceed to the formal definition of the approach.

3.1 Overview: HMM for Software Developer Behavior Analysis

We illustrate how HMMs can be used for modeling human behavior and provide an overview of the modeling approach through a running example from the software engineering domain. Consider a set of two IDE interactions related to using a grep-like tool, Find in Files, to search within the IDE: `Find in Files Search`, `Search Result Click`. The first message indicates that a user issued a query with the search tool, while the second message indicates a click on a result retrieved by the search tool, which opens the IDE editor. These two interactions will form the HMM's observable symbol space.

Let's also assume that there are two hidden states in the HMM: Code Search Success and Code Search Failure. We consider a failed search sessionto be those interactions with the search tool where the developer issues a query, but does not click on any of the search results, because, presumably, they are irrelevant to the information need. An interaction sequence representative of a failed search may look like this:

```
1, Find in Files Search
2, Find in Files Search
3, Find in Files Search
```

Conversely, the following may be an instance of a successful search session:

```
1, Find in Files Search
2, Search Result Click
3, Search Result Click
```

When modeled via an HMM, the extracted HMM model with parameter values determined by a training algorithm, such as the Baum-Welch algorithm [17], on these interaction sessions may look like the one in Figure 1. Each of the two hidden states (Code Search Success and Code Search Failure) emits the two observable messages (`Find in Files Search` and `Search Result Click`) with a probability distribution. The model also expresses the transitional probability distribution between the hidden states as well as the probability distribution of initial states.

A common task in constructing such an HMM for prediction is to determine the sequence of states that the HMM is likely to be traversing for a new, previously unseen, sequence

of developer interactions. However, we are particularly interested in constructing such a model to account for developer interactions. In particular, to gain an understanding of developer behaviors that can be difficult to be observed directly, we determine the states and interpret their message emission and transition probability distributions.

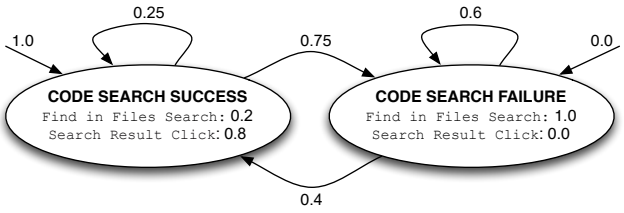

Figure 1: An example HMM for code search.

An HMM is characterized by two embedded stochastic processes among which one is on observable symbols emitted over time and one is on unobservable states of the model changing over time [17]. It is commonly expressed as a quintuple, i.e., $\lambda = (\mathbb{S}, \mathbb{V}, \mathbf{T}, \mathbf{E}, \boldsymbol{\pi})$ where $\mathbb{S} = \{S_1, S_2, \ldots, S_N\}$ is the set of N *hidden* states and $\mathbb{V} = \{V_1, V_2, \ldots, V_M\}$ is the set of M *observable* symbols. A stochastic process results in the model at state $q_t \in \mathbb{S}$ at time t, where $\mathbf{T} = \{T_{ij} : 1 \leq i, j \leq N\}$ is the state transition probability distribution and $T_{ij} = P\{q_{t+1} = S_j | q_t = S_i\}$ is the probability that the HMM transits from state S_i at time t to state S_j at time $t+1$. $\mathbf{E} = \{E_j(k) : 1 \leq j \leq N, 1 \leq k \leq M\}$ where $E_j(k) = P\{V_k | q_t = S_j\}$ is the emission (or observation symbol) probability distribution and $E_j(k)$ is the probability that symbol V_k is emitted at state S_j. $\pi = \{\pi_i, 1 \leq i \leq N\}$ where $\pi_i = P\{q_1 = S_i\}$ is the initial state probability distribution and π_i is the probability that the HMM starts at state S_i, i.e., the probability that the HMM is in state S_i at time $t = 1$.

A basic problem for HMM λ is, given $\mathbb{S}$ and $\mathbb{V}$, to determine the values of model parameters, $\mathbf{T}$, $\mathbf{E}$, and $\boldsymbol{\pi}$ that maximize the probability at which a set of sequences $\mathbb{O} = \{O_1, O_2 \ldots O_K\}$ are being observed, i.e., to solve the optimization problem,

$$\underset{\mathbf{T},\mathbf{B},\boldsymbol{\pi}}{\operatorname{argmax}} P\{\mathbb{O}|\lambda\} \tag{1}$$

where

$$\begin{aligned} P\{\mathbb{O}|\lambda\} &= P\left\{\{O^{(1)}, O^{(2)}, \ldots, O^{(K)}\}|\lambda\right\} \\ &= \prod_{k=1}^{K} P\left\{O^{(k)}|\lambda\right\} \end{aligned} \tag{2}$$

This optimization is a training process for the HMM. Among a number of methods developed for the training process are the Baum-Welch algorithm, the EM method, and various gradient techniques [17].

Following the definition above, we can define the HMM for the code search example illustrated in Figure 1 as $\lambda_{code\ search} = (\mathbb{S}, \mathbb{V}, \mathbf{T}, \mathbf{E}, \boldsymbol{\pi})$ where $N = 2$, $M = 2$, $\mathbb{S} = \{S_1, S_2\}$, $V = \{V_1, V_2\}$, $\mathbf{T} = \{T_{ij}\}$, $\mathbf{B} = \{b_j(k)\}$, $\boldsymbol{\pi} = \{\pi_i\}$, $1 \leq i \leq N$, $1 \leq k \leq M$, and

$S_1 =$ [Code Search Success]
$S_2 =$ [Code Search Failure]

$V_1 =$ `<Find in Files Search>`
$V_2 =$ `<Search Result Click>`

$$\mathbf{T} = \begin{Bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{Bmatrix} = \begin{Bmatrix} 0.25 & 0.75 \\ 0.60 & 0.40 \end{Bmatrix}$$

$$\mathbf{E} = \begin{Bmatrix} b_1(1) & b_1(2) \\ b_2(1) & b_2(2) \end{Bmatrix} = \begin{Bmatrix} 0.2 & 0.8 \\ 1.0 & 0.0 \end{Bmatrix}$$

$$\boldsymbol{\pi} = \begin{Bmatrix} \pi_1 & \pi_2 \end{Bmatrix} = \begin{Bmatrix} 1.0 & 0.0 \end{Bmatrix}$$

Training the above HMM $\lambda_{code\ search}$ involves determining the values of T_{ij}, $E_j(k)$, and π_i where $1 \leq i, j \leq 2$ and $1 \leq k \leq 2$, given the set of sequences, $\mathbb{O} = \{O^{(1)}, O^{(2)}\}$, such as,

$O^{(1)} =$ (`<Find in Files Search>`
`<Find in Files Search>`
`<Find in Files Search>`)
$O^{(2)} =$ (`<Find in Files Search>`
`<Search Result Click>`
`<Search Result Click>`)

such that the following probability is maximized.

$$\begin{aligned} P\{\mathbf{O}|\lambda_{code\ search}\} &= P\left\{\{O^{(1)}, O^{(2)}\}|\lambda_{code\ search}\right\} \\ &= \prod_{k=1}^{2} P\{O^{(k)}|\lambda_{code\ search}\} \end{aligned}$$

Throughout this work, we use the well-known Baum-Welch algorithm to train HMMs.

3.2 Building an Interpretable HMM Model

There are several challenges in building an interpretable HMM model for analyzing developer behavior. The Baum-Welch algorithm is an unsupervised method of training an HMM, given a set of input (observable) sequences and the number of hidden states in the model. It is difficult to estimate the number of hidden states for an unknown model, where no prior knowledge exists [1, 2]. One strategy for selecting this parameter is sensitivity analysis using metrics such as the Bayesian Information Coefficient (BIC). However, this technique and metrics are not definitive and may not apply to all problems.

Another issue is the interpretability of the inferred model. The Baum-Welch algorithm optimizes the predictive power of the model, but as the number of hidden and observable states grows, the complexity of the inferred model grows and interpreting it can become very difficult. This is due to the fact that each of the HMM's hidden states can emit many of the obervation symbols, and the same symbol is often emitted with similar probability from multiple states. Labeling individual hidden states is especially challenging

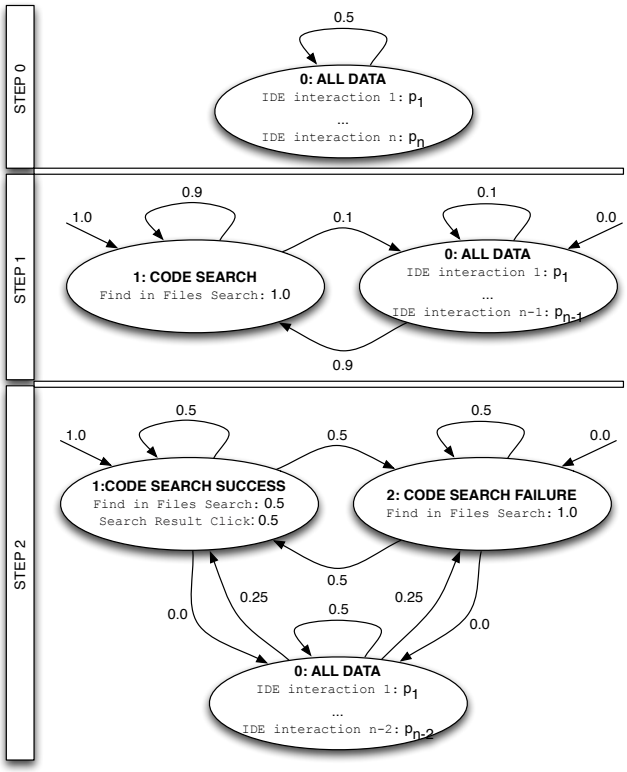

Figure 2: Interactive HMM construction, in 2 steps, for a code search example. The model is retrained between each step using the Baum-Welch algorithm.

when the probability distribution of the observation symbols is fat-tailed.

Interactive model building is an approach that overcomes these two difficulties [7]. In this approach, the HMM is built iteratively by a domain expert, by introducing one hidden state at a time and retraining the model with the Baum-Welch algorithm at each step. A special state, called the *sink state*, exists in the model to capture the remaining symbols/sequences that haven't been explicitly represented to that point. To illustrate this idea, we refer back to the HMM in Figure 1. We begin by training a model, using the Baum-Welch algorithm, from a single state, i.e, the sink state called All Data (step 0 in Figure 2). Based on the this first model, we proceed to add an additional Code Search state (step 1 in Figure 2). In training the model with the new state using Baum-Welch, we constrain the model so that the new state only outputs one message, i.e., `Find in Files Search`, while the All Data state no longer emits that message (or emits it with zero probability). In the next step, we can introduce another state (e.g., a Code Search Failure state) constrained to emit both `Find in Files Search` and `Search Result Click` messages, and retrain using Baum-Welch (step 2 in Figure 2). The process continues until we are satisfied with the set of modeled states.

The interactive model building process is therefore an iterative multi-step approach as shown in Algorithm 1. During the interactive construction of an HMM, the human expert that drives the process is provided with two types of intermediate output to help in constructing a model that may be more accurate and more interpretable. The information includes: (1) a set of *interesting* sequences found in the data, which are poorly captured by the current model, in order to suggest new states that could be created; (2) and model quality metrics and a set of sequences that are captured by the model, but do not exist in the data, both aimed to ensure the final model accurately reflects the input data. This intermediate ouput helps the expert to decide *whether an additional hidden state should be added*, and to control *which symbols the added hidden state should emit*, and to determine *the meaning of the added hidden state*.

Input: the initial number of hidden states, $N = 1$;
observation symbols $\mathbb{V}$; observation sequences $\mathbb{O}$
Result: interpretable and accurate HMM
initialize model with one All Data state
keep going ← true
while *keep going* **do**
  train model using the Baum-Welch algorithm
  find interesting sequences that are not in model
  calculate model quality
  if *user wants to expand model* **then**
    add states and constraints based on user input
  else
    keep going ← false
  end
end

Algorithm 1: Interactive Model Building

3.3 Generating Interesting Sequences

As described in Algorithm 1, a new set of *interesting sequences* [6, 7] informs the human expert at each iteration in answering the question of whether to create another hidden state, and which observable messages it should emit. The interesting sequences are selected to show occurrences that are present in the input IDE interaction dataset, but are not, or not well, expressed in the current model. Note that even though a special state may not yet exist for each observable message, the All Data state captures the expression of remaining messages, likely with less accuracy.

For instance, in Step 1 in Figure 2, based on the model consisting of two states All Data and Code Search, and their associated probabilities, the human expert is shown a set of interesting sequences from the IDE interaction dataset, perhaps containing the following set of two interesting sequences:

```
1. <Find in Files Search, Search Result Click,
    Search Result Click> -- score = 0.135
2. <Find in Files Search, Search Result Click,
    Search Result Click, Search Result Click>
    -- score = 0.087
```

Observing that sequences with several clicks after the query are not captured well in the current model, the human expert may decide to create a Code Search Failure state that emits those two messages, and rename the existing Code Search state to Code Search Success, such as in Step 2 of Figure 2.

The *interesting sequences* are ranked in descending order by the likelihood that they will appear in the dataset minus the likelihood that they will be generated by the current

HMM. This is the score shown at the end of each sequence in the above example. Sequences that rank highly, but have already been encapsulated in a state, are omitted and not shown to the human expert.

The *interesting sequences* are computed from two sets of computed observable sequences, called *θ-frequent sequences by data* and *θ-probable sequences by model*. We denote these two set of observable sequences as $\mathbb{O}^D(\theta, \mathbb{O})$ and $\mathbb{O}^M(\theta, \lambda)$, respectively. We follow [6, 7] to compute the two observable sequences with a minor variation.

3.3.1 θ-frequent Sequences by Data

As defined in Section 3.1, $\mathbb{V} = \{V_1, V_2, \ldots, V_M\}$ are M observation symbols, $\mathbb{O} = \left\{O^{(1)}, O^{(2)}, \ldots, O^{(K)}\right\}$ the set of K observation sequences where a sequence is $O \in \mathbb{O}$ and $O = O_1 O_2 \ldots O_T$ is a observation sequence of length T and O_1, O_2, …, $O_T \in \mathbb{V}$. A θ-frequent observation sequence is a subsequence that appears no less frequently than a frequency threshold θ where $0 \leq \theta \leq 1$ in the K observation sequences $\mathbb{O}$.

Following [6, 7], we compute $\mathbb{O}^D(\theta, \mathbb{O})$. In this paper, we only consider prefixes of sequences in $\mathbb{O}$ for the θ-frequent sequences, and consider more general forms of θ-frequent sequences that may not be prefixes in future work. We denote a prefix of sequence O as sequence $\widehat{O}$, i.e., $\widehat{O} \sqsubseteq O$, and define $\mathbb{X}(\widehat{O}) = \{O : \widehat{O} \sqsubseteq O \wedge O \in \mathbb{O}\}$ as the set of observation sequences that have the common prefix $\widehat{O}$. It is clear that $\mathbb{X}(O) \subseteq \mathbb{O}$. Then, the frequency of a sequence $\widehat{O}$, denoted as $\mathcal{F}^D(\widehat{O}, \mathbb{O})$ is defined as follows:

$$\begin{aligned}\mathcal{F}^D(\widehat{O}, \mathbb{O}) &= \frac{\left|\mathbb{X}(\widehat{O})\right|}{|\mathbb{O}|} \\ &= \frac{\left|\left\{O : \widehat{O} \sqsubseteq O \wedge O \in \mathbb{O}\right\}\right|}{|\mathbb{O}|} \\ &= \frac{1}{K}\left|\left\{O : \widehat{O} \sqsubseteq O \wedge O \in \mathbb{O}\right\}\right| \end{aligned} \tag{3}$$

while θ-frequent sequences *from* $\mathbb{O}$ become:

$$\mathbb{O}^D(\theta, \mathbb{O}) = \{\widehat{O} : \mathcal{F}^D(\widehat{O}, \mathbb{O}) \geq \theta\} \tag{4}$$

The above definition indicates that for any sequence $\widehat{O} \in \mathbb{O}^D(\theta, \mathbb{O})$, it is a common prefix of the set of observation sequences $\mathbb{X}(\widehat{O}) \subseteq \mathbb{O}$. Note that any prefixes of such a sequence $\widehat{O} \in \mathbb{O}^D(\theta, \mathbb{O})$ is also a sequence in $\mathbb{O}^D(\theta, \mathbb{O})$. Therefore, it is not useful to show the prefixes of $\widehat{O}$ to the human expert who is building an HMM interactively. *We are hence interested only in finding the longest θ-frequent sequences from* $\mathbb{O}$.

A longest common prefix $\widehat{O}^{(L)}$ is a sequence that satisfies the following condition: $\mathbb{X}(\widehat{O}^{(L)}) \neq \mathbb{X}(\widehat{O}^{(L)}o), \forall o \in \mathbb{V}$. *For ease of discussion, from this point onward, we use* $\mathbb{O}^D(\theta, \mathbb{O})$ *as the set of the longest θ-frequent sequences.* Following [6, 7], we use Algorithm 2 to generate the longest θ-frequent sequences $\mathbb{O}^D(\theta, \mathbb{O})$ and $\theta \geq \frac{2}{|\mathbb{O}|} = \frac{2}{K}$ from data set $\mathbb{O}$. Note that we enforce the condition of $\theta \geq \frac{2}{|\mathbb{O}|} = \frac{2}{K}$ since prefixes that appear only once are not relevant, and need not be included in $\mathbb{O}^D(\theta, \mathbb{O})$.

Input: observation sequences $\mathbb{O}$; frequency threshold θ where $\frac{2}{|\mathbb{O}|} = \frac{2}{K} \leq \theta \ll 1$
Output: longest θ-frequent sequences $\mathbb{O}^D(\theta, \mathbb{O})$

```
1  sort 𝕆 in lexicographical order into list L{𝕆} = (O^(1), ..., O^(K))
2  PushBack (L{𝕆}, ϵ) // for handling last sequence
   // T^(p), T^(c): previous & current prefix length
3  T^(p) ← 0
4  for i ← 1 to K + 1 do
5  |  T^(c) ← 1
6  |  while O_j^(i) = O_j^(i+1) ∧ j ≤ min (Len (O^(i)), Len (O^(i+1))) do
7  |  |  T^(c) ← T^(c) + 1
8  |  end
9  |  if T^(p) ≠ T^(c) then
10 |  |  Ô ← Prefix (O^(i), T^(p))
11 |  |  f ← ComputePrefixFrequency (L(𝕆), i, Ô, T^(p))
   |  |  SaveSeqAndFreq(𝕆^D(θ, 𝕆), Ô, f)
12 |  |  T^(p) ← T^(c)
13 |  end
14 end
15 Function ComputePrefixFrequency (L(𝕆), i, Ô, T)
16 |  n ← 1, j ← i − 1
17 |  while j > 0 ∧ Prefix (O^(j), T) ≠ Ô do
18 |  |  n ← n + 1, j ← j − 1
19 |  end
20 |  j ← i + 1
21 |  while j ≤ K ∧ Prefix (O^(j), T) ≠ Ô do
22 |  |  n ← n + 1, j ← j + 1
23 |  end
24 |  return n/K
25 end
```

Algorithm 2: Generating the longest θ-frequent sequences $\mathbb{O}^D(\theta, \mathbb{O})$ where $\theta \geq \frac{2}{K}$

3.3.2 θ-probable Sequences by Model

In Algorithm 1, the human expert building the model relies on a second set of sequences, which are likely predicted by an HMM. To obtain the sequences to aid the expert, we are in effect solving a generation problem, i.e., generate sequences with significant probability of being observed, given an HMM. We refer to the sequences as the set of θ-probable sequences generated by a model and define it as follows:

$$\mathbb{O}^M(\theta, \lambda) = \left\{O : P\{O|\lambda\} \geq \theta, O \in \mathbb{V}^+\right\} \tag{5}$$

where "+" is the Kleene-+ and $P\{O|\lambda\}$ is the probability of observing the sequence $O = O^{(1)}O^{(2)}\ldots O^{(T)}$ of length T given the model $\lambda = (\mathbb{S}, \mathbb{V}, \mathbf{T}, \mathbf{E}, \boldsymbol{\pi})$ [17].

To generate an observation sequence from HMM λ, the HMM starts at a state $q_1 \in \mathbb{S} = \{S_1, S_2, \ldots, S_N\}$ and goes through $T-1$ transitions. We denote the sequence of states visited as $Q = q_1 q_2 \ldots q_T$ where $q_1, q_2, \ldots, q_T \in \mathbb{S}$. The observation sequence O is of length T and is the result of emissions from visiting any state sequences in $\mathbb{Q} = \mathbb{S}^T = \{q_1 q_2 \ldots q_T : q_i \in \mathbb{S}, 1 \leq i \leq T\}$.

Following [17], the probability of observing sequence $O = O_1 O_2 \ldots O_T$ given HMM λ is:

$$
\begin{aligned}
P\{O|\lambda\} =& P\{O, \mathbb{Q}|\lambda\} \\
=& \sum_{\forall q \in \mathbb{Q}} P\{O, q|\lambda\} \\
=& \sum_{\forall q \in \mathbb{Q}} P\{O|q, \lambda\} P\{q|\lambda\} \\
=& \sum_{\forall q_1 \in \mathbb{Q}, \forall q_2 \in \mathbb{Q}, \ldots \forall q_T \in \mathbb{Q}} \pi_{q_1} E_{q_1}(O_1) \\
& T_{q_1 q_2} E_{q_2}(O_2) \\
& T_{q_2 q_3} E_{q_3}(O_3) \\
& \ldots \\
& T_{q_{T-1} q_T} E_{q_T}(O_T)
\end{aligned} \tag{6}
$$

Direct computation of $P\{O|\lambda\}$ using the definition in equation (6) needs $2TN^T$ calculations and is costly [17]. A more efficient algorithm to compute $P\{O|\lambda\}$ is to use the *forward* part of the *forward-backward* procedure for HMMs, a dynamic programming approach, and requires on the order of TN^2 calculations [17].

Following [6,7], we generate the set of θ-probable sequences by HMM λ in Algorithm 3 by taking advantage of the *forward* part of the *forward-backward* procedure.

Input: HMM $\lambda = (\mathbb{S}, \mathbb{V}, \mathbf{T}, \mathbf{E}, \boldsymbol{\pi})$; $|\mathbb{S}| = N$; $|\mathbb{V}| = M$; $\pi^{(0)} = \frac{1}{N}\left\{\pi_1^{(0)}\pi_2^{(0)} \ldots \pi_N^{(0)}\right\}$ where $\pi_i^{(0)} = 1$, $1 \leq i \leq N$
Output: θ-probable sequence set $\mathbb{O}^M(\theta, \lambda)$
// ϵ: $\mathbb{V}^0$, the empty sequence
1 $\boldsymbol{\alpha}(\epsilon) \leftarrow \pi^{(0)}$ // $\boldsymbol{\alpha}(\cdot)$ is a vector of length N
2 GenerateHMMSequences $(\epsilon, \boldsymbol{\alpha}(\epsilon))$
3 **Function** GenerateHMMSequences *(O, $\boldsymbol{\alpha}(O)$)*
4 $\quad p \leftarrow \sum_{i=1}^{N} \alpha(O, i)$
5 $\quad$ **if** $p \geq \theta$ **then**
6 $\quad\quad$ SaveSeqAndProb $(\mathbb{O}^M(\theta, \lambda), O, p)$
7 $\quad\quad$ **foreach** $o \in \mathbb{V}$ **do**
8 $\quad\quad\quad$ **for** $j \leftarrow 1$ **to** M **do**
9 $\quad\quad\quad\quad O' \leftarrow Oo$
10 $\quad\quad\quad\quad \alpha(O', j) \leftarrow \left[\sum_{i=1}^{N} \alpha(O, i) T_{ij}\right] E_j(o)$
11 $\quad\quad\quad\quad$ GenerateHMMSequences $(O', \boldsymbol{\alpha}(O'))$
12 $\quad\quad\quad$ **end**
13 $\quad\quad$ **end**
14 $\quad$ **end**
15 **end**

Algorithm 3: Generating θ-probable sequences $\mathbb{O}^M(\theta, \lambda)$ from HMM $\lambda = (\mathbb{S}, \mathbb{V}, \mathbf{A}, \mathbf{B}, \boldsymbol{\pi})$

3.3.3 Computing Interesting Sequences

We are now ready to define interesting sequences for interative modeling building and the sequences are the output at Line 5 in Algorithm 1. The expert will use the interesting sequences to decide states to add and additional contraints to apply at Line 7 in Algorithm 1. In a slight departure from [6, 7], we define *interesting sequences for interactive model mining* as:

$$
\mathbb{O}^{DM} = \left\{O : O \in \left(\mathbb{O}^D(\theta, \mathbb{O}) \cup \mathbb{O}^M(\theta, \lambda)\right) \wedge \mathcal{I}^{DM}(O) \geq \theta\right\} \tag{7}
$$

where $\mathcal{I}^{DM}(O|\mathbb{O}, \lambda)$ is the *interestingness* of a sequence,

$$
\mathcal{I}^{DM}(O|\mathbb{O}, \lambda) = \mathcal{F}^D(O, \mathbb{O}) - P\{O|\lambda\} \tag{8}
$$

Equation (8) can be divided into three cases as in equations (9a), (9b), and (9c). Two cases in equations (9a) and (9b) are for sequences that frequently appear in data, but are not well suggested by the model. These sequences are used at Line 7 in Algorithm 1 to decide states to add and additional constraints to apply. The difference of the two cases lies in the minor difference in how $P\{O|\lambda\}$ is calculated. In the later case, $P\{O|\lambda\}$ is already calculated in Algorithm 3 while in the former case, $P\{O|\lambda\}$ needs to be calculated by applying the *forward* part of the *forward-backward* procedure as described in [17].

The case in equation (9c) is for sequences that are well supported by the model, but do not appear in data. Note that the right hand side of equation (9c) is an approximation since we apply the fact that $\mathcal{F}^D(O, \mathbb{O}) < \frac{2}{K}$, $K \gg 2$, and $\mathcal{F}^D(O, \mathbb{O}) \approx 0$ to the equation. Sequences corresponding to this case will be displayed to the human expert to help assess the quality of the model at Line 5 in Algorithm 1.

$$
\mathcal{I}^{DM}(O|\mathbb{O}, \lambda) = \begin{cases}
\mathcal{F}^D(O, \mathbb{O}) - P\{O|\lambda\}, O \in \mathbb{O}^D(\theta, \mathbb{O}) - \mathbb{O}^M(\theta, \lambda) & \text{(9a)} \\
\mathcal{F}^D(O, \mathbb{O}) - P\{O|\lambda\}, O \in \mathbb{O}^M(\theta, \lambda) \cap \mathbb{O}^D(\theta, \mathbb{O}) & \text{(9b)} \\
-P\{O|\lambda\}, \qquad O \in \mathbb{O}^M(\theta, \lambda) - \mathbb{O}^D(\theta, \mathbb{O}) & \text{(9c)}
\end{cases}
$$

3.4 Controlling HMM Mining

The above discussion is laid out in general form, i.e., the HMM is assumed to be ergodic (or fully connected). For some applications, other types of HMMs have been found to be more accurate models than ergodic ones. For example, in speech recognition, left-right models (or Bakis models) in which the states proceed from left to right are commonly used [17].

In our approach for interactive building of the HMMs, we do not necessarily consider that the model being built is ergodic because we control whether transitions between states are possible. The HMM mining process is controlled by adding hidden states and by constraining two parameters $\mathbf{T}$ and $\mathbf{E}$ in HMM $\lambda = \{\mathbb{S}, \mathbb{V}, \mathbf{T}, \mathbf{E}, \boldsymbol{\pi}\}$. The mining proceeds until no interesting sequences are generated or the human expert chooses to terminate the process. By constraining parameter $\mathbf{T}$, we can control the topology of an HMM by fixing selected transition probabilities as 0. By constraining parameter $\mathbf{E}$, we can control what symbols, or IDE interaction messages in our case, to be emitted by a specific state.

Consider at iteration i, the HMM is $\lambda^{(i)} = \{\mathbb{S}^{(i)}, \mathbb{V}, \mathbf{T}^{(i)}, \mathbf{E}^{(i)}, \boldsymbol{\pi}^{(i)}\}$ where $\mathbb{S}^{(i)} = \{\mathfrak{S}_1, S_2, \ldots, S_{N^{(i)}}\}$ and $\mathfrak{S}_1$ is a special state, called the ALL DATA sink state (as in the example in Figure 2), $\mathbb{V} = \{V_1, V_2, \ldots, V_M\}$, $\mathbf{T} = \{T_{jk}^{(i)} : 1 \leq j, k \leq N^{(i)}\}$, $\mathbf{E}^{(i)} = \{E_j^{(i)}(k) : 1 \leq j \leq N^{(i)}, 1 \leq k \leq M\}$, and $\boldsymbol{\pi} = \{\pi_i^{(i)} : 1 \leq i \leq N^{(i)}\}$.

In the next iteration, by examining an interesting sequence $O = O_1 O_2 \ldots O_T$ that corresponds to the case either in equation (9a) or equation (9b), the expert may add a hidden state to the model, i.e., by incrementing the number of hidden states, $N^{(i+1)} = N^{(i)} + 1$, and by adjusting the transition probability matrix and the emission probability matrix accordingly. The added hidden state will emit a subset of the symbols that form O. For instance, provided that $O_2 \neq O_3$ and $O_2, O_3 \in \mathbb{V}$, the expert decides that the newly added hidden state S^{i+1} should emit these two symbols and the sink state $\mathfrak{S}_1$ should cease to emit the two symbols. Then, the emission probability matrix at iteration $i+1$ should become, $\mathbf{E}^{(i+1)} = \{E_j^{(i+1)}(k) : 1 \leq j \leq N^{(i+1)}, 1 \leq k \leq M\}$ and,

$$E_j^{(i+1)}(k) =$$

$$\begin{cases} 0, & V_k \notin \{O_2, O_3\} \wedge j = i+1 \quad (10a) \\ r_2, & V_k = O_2 \wedge j = i+1 \quad (10b) \\ r_3, & V_k = O_3 \wedge j = i+1 \quad (10c) \\ 0, & V_k \in \{O_2, O_3\} \wedge j = 1 \quad (10d) \\ \dfrac{E_j^{(i+1)}(k)}{\sum_{l=1}^{N} E_j^{(i+1)}(l)}, & V_k \notin \{O_2, O_3\} \wedge j = 1 \quad (10e) \\ E_j^{(i+1)}(k), & otherwise \quad (10f) \end{cases}$$

Equations (10a), (10b), and (10c) specify emission probability distribution at the newly added state S_{j+1}. They indicate that two symbols (or IDE interaction messages) $O_2 \in \mathbb{V}$ and $O_3 \in \mathbb{V}$ should be emitted at probabilities r_2 and r_3, respectively, and $r_2 + r_3 = 1$, $r_2 > 0$, $r_3 > 0$.

Equations (10d) and (10e) specify the change that should be made for the emission probability distribution at the sink state $\mathfrak{S}_1$, i.e., the state should not emit symbols O_2 and O_3 by setting their corresponding emission probabilities as 0, then the probabilities for the rest of symbols should be normalized such that $\sum_{l=1}^{N} E_j^{(i+1)}(l) = 1$.

Equation (10f) allows the probability distribution for any other state to remain the same as in the previous iteration before the training of the HMM for iteration $i+1$. During the training process at iteration $i+1$ (e.g. using the Baum-Welch algorithm) equations (10a) to (10d) must hold as a constraint.

3.5 Preserving Model Quality

As our interactive approach constrains the trained algorithm, it is possible that after several steps, the model quality will degrade. To prevent this from occurring, two pieces of information are presented to the expert: (1) a probabilistic rating of the accuracy of the model, and a comparison of this rating to a completely unconstrained model from the same data, also trained with Baum-Welch, with the same same number of hidden states; (2) a set of sequences that the current model can produce with high probability, but do not exist in the data, sorted in descending order of their probability scores. Both of these can aid the expert in deciding that a model's quality may have degraded to a point where it does not accurately represent the input data.

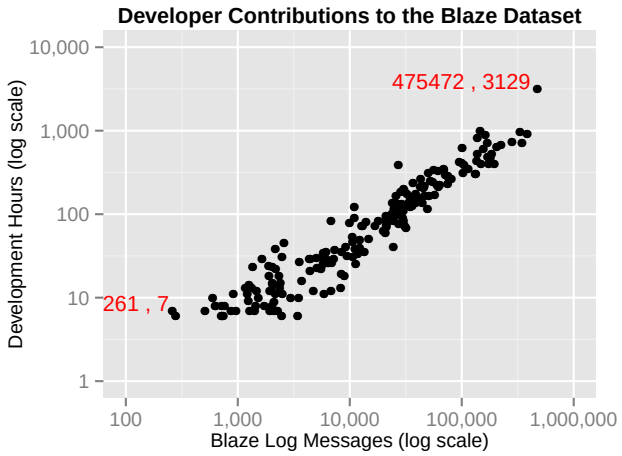

Figure 3: The collection interval and number of messages for each of the 196 developers.

The probabilistic rating of the accuracy of the model is measured using the optimization target of the Baum-Welch algorithm, i.e., the log-likelihood of observing the given sequences. Given a set of K observation sequences $\mathbb{O}$ and HMM λ, the probability of observing the sequences is equation (2) which has log-likelihood becomes,

$$\mathcal{L}\{\mathbb{O}|\lambda_{code\ search}\} = \log(P\{\mathbb{O}|\lambda_{code\ search}\}) \quad (11)$$

In general, during the model building iterations, we desire to observe $\mathcal{L}^{(1)}\{\mathbb{O}|\lambda_{code\ search}\} < \mathcal{L}^{(2)}\{\mathbb{O}|\lambda_{code\ search}\}\} < \ldots$, i.e., the log-likelihood of observing $\mathbb{O}$ should gradually increase during the model building process. In addition the log-likelihood of observing $\mathbb{O}$ with regard to an interatively built model should be no worse than unconstrained model.

Interesting sequences correponding to equation (9c) are the sequences that the expert may use to assess the quality of the interactively built model.

4 An HMM Model of Debugging Behavior

To illustrate the capabilities of our approach, we applied it on a real-world IDE interaction dataset collected from nearly 200 developers at ABB, Inc., during their daily work, for a period of up to 12 months. We focus our attention on developer behavior in using the debugging capabilities of an IDE as a case study for our interactive HMM approach of analyzing IDE interaction data.

4.1 Dataset Description

Our input dataset was collected using an IDE extension called Blaze [20] that, when installed, monitors and logs all available IDE clicks and events. The types of messages are very similar to the popular, but no longer collected, Eclipse UDC dataset [5], which has been used widely by researchers. As shown in Figure 3, the distribution of developers' participation in IDE data collection was broad, ranging from 7 hours (261 messages) to 3129 hours (475472 messages).

Each entry in the dataset consists of a tuple including a timestamp, the id of the developer that generated the interaction, and the type of interaction (e.g., `Debug.Start`). The breadth of messages in the dataset is large, consisting of

nearly 5000 unique entries. The messages are often reflective of developer actions, such as `Debug.Start` which indicates that the developer started debugging, though some messages only indicate IDE events, e.g., the debugger stopping on a breakpoint, via the `Debug.Debug Break Mode` message.

4.2 Data Preprocessing

To understand developers' high-level actions during debugging, we first process the Blaze dataset into a set of debugging sessions, each of which is a contiguous set of interactions that include frequent debugger commands or events. These sessions form the input to our interactive HMM mining process. They are constructed as follows.

1. **Filter extremely rare messages**. The dataset contains messages that occur only for individual developers, usually indicative of specific IDE extensions or configurations. We remove such messages, using a threshold of 3% of the developers, i.e. messages that occur in less than 3% of the developers are filtered out.

2. **Remove cursor movement messages**. The Blaze dataset contains certain messages that are not interesting, but very frequent. For instance, the message, `View.OnChangeCaretLine` indicates that the caret has moved in the editor window, while the Blaze log message `View.OnChangeScrollInfo` indicates that the user has scrolled up or down in the editor. We remove the few such messages from the dataset, as they have no impact on our further analysis.

3. **Form debug sessions**. We enumerated a set of 40 messages to represent all of the IDE interactions with the Visual Studio debugger in Blaze. Based on this set, we defined a debugging session to be a time-contiguous sequence of messages by one user that contains at least one of the messages in the debugging set in a 30-second window of time.

After the above pre-processing, we obtained 2,849 debugging sessions, consisting of on average of 186.1 messages, and with an average length of 179.7 seconds. Based on the extracted sessions, we executed the algorithm for interactive HMM construction, and present a case study of the results in the next section.

5 Mining Results

Using the large set of debugging log sessions, we performed 14 iterations of the HMM mining algorithm, resulting in many states (excluding the ALL DATA state) representing 39 message types in the Blaze input dataset. The quality of the mined model was satisfactory, and at each step in the interactive mining process, the quality of the model equaled or surpassed an unconstrained (and very difficult to interpret) model with the same number of states, which was used as a baseline. Overall, the quality of the model never dipped below 99% accuracy.

The mined model is displayed in Figure 4, where each state is represented by a graph node, and the strength of the transition probabilities are represented by the thickness of each directed edge. A description of the usage behavior expressed by each node is given in Table 1.

Table 1: Model states, the corresponding types of developer behaviors in the Visual Studio IDE, and a listing of the input messages in the Blaze dataset.

View Locals – Click on a view that provides a listing of the variables in a particular scope of the program, while debugging. Allows for examination of the values of each active program variable. (View.Locals)
View/Set Watch – Add specific variable(s) to a watch list, where they can be observed during debugging. Visual Studio allows for both a QuickWatch, single variable observed temporarily, or a watch list of several variables observed for the duration of the debugging session. (View.Watch;Debug.QuickWatch;Debug.AddWatch)
Debug Step/Move – Provide the ability to step through the program interactively, one statement at a time. Includes similar commands like `Run to Cursor`, which also advance the debugger to a specific point in the program. (Debug.StepOver;Debug.StepInto;Debug.StepOut; Debug.SetNextStatement;Debug.RunToCursor)
View Immediate – A window that allows the execution of commands and evaluation of expressions during the debugging session. (View.Immediate Window;Debug.Immediate)
View Call Stack – Provide a listing of the methods that are currently on the stack. A click on a specific method opens it in the Visual Studio editor. (View.Call Stack;Debug.CallStack)
View Autos – Similar to the VIEW LOCALS but only displays the values of variables defined on lines near to the current breakpoint. In most languages, this is the current and preceding line of the program. (View.Autos)
Output View – A click on the window listing of the output of compilation as well as console messages written during the execution of the program, or a keypress that makes this window visible. (View.Output;Debug.Output)
Stop Debug – Command that stops the current debugging session. (Debug.StopDebugging)
Start Debug – Commands that start a new debugging session. (Debug.Start;Debug.StartDebugTarget; TestExplorer.DebugSelectedTests;Debug.Restart;Debug.AttachtoProcess; TestExplorer.DebugAllTestsInContext;TestExplorer.DebugAllTests)
View Package Explorer – The Visual Studio package explorer provides a hierarchical view of the project files. Clicking a file opens it in the editor. (View.Solution Explorer)
Set Breakpoints – Set or enable a previously disabled (set of) breakpoints. (Debug.ToggleBreakpoint; Debug.EnableBreakpoint)
Code Search – Search for code using a query string and clicking on the retrieved results. Here we consider use of the `Find in Files` VS built-in search tool, as well as usage of the Sando IR-based code search tool [19]. (View.Find and Replace;View.Find Results 1; View.Sando Search;Edit.FindinFiles)
Structured Navigation – Navigate the call graph or the def-use chain, usually initiated by selecting specific mentions of a identifier in the IDE editor. Both the VS studio builtin commands and the `ReSharper` VS extension, when used for this purpose, were considered. (Edit.GoToDefinition;View.Find Symbol Results; Edit.FindAllReferences;ReSharper.ReSharper.GotoDeclaration; Edit.NavigateTo;ReSharper.ReSharper.FindUsages; Edit.GoToDeclaration;View.Call Hierarchy)
All Data – The state used in our mining approach that captures all remaining messages in the input dataset, which haven't been examined via a specific state.

Next, we discuss the model, the insights that it provides about developer debugging behavior in the field, and relation to previous lab studies of debugging behavior.

Observation 1. Debugging views and commands form three usage groups. In examining the mined HMM

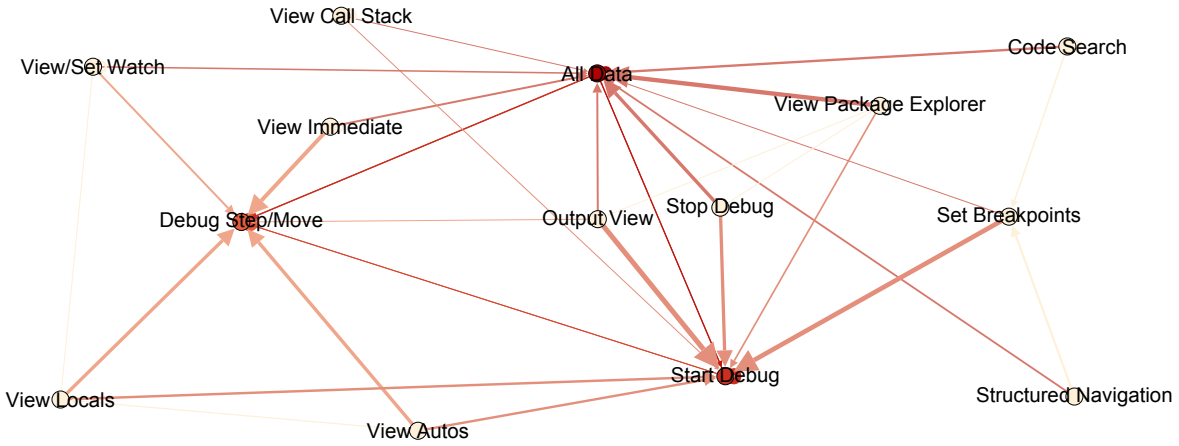

Figure 4: An HMM of debugging behavior in Visual Studio, mined by the technique described in this paper. Nodes represent specific behaviors and edges represent transitions between these behaviors. The color intensity of each node represents its in-degree, while the edge thickness represents the strength of the transition probability.

in Figure 4, we can observe that most of the commands and views can be grouped into three separate clusters: commands/views that primarily transition into the Debug Step/Move state, commands/views that primarily transition into the Start Debug state, and commands/views that primarily transition into the Set Breakpoints state. While some commands express an affinity to more than one of these groups (e.g., View Locals), the majority of views can be grouped easily into one of these three categories, indicating that there are three phases to debugging behavior: planning, starting (or restarting), and stepping. Each of these phases is supported by a set of relevant views, presenting specific types of information to the developer.

Observation 2. Variable state views are most relevant to developers when stepping through code. Many of the views and commands described in Figure 4 only appear when Visual Studio's interactive debugger starts operating and the environment switches to the debugging perspective. Within this perspective, a number of views are shown by default to the user, many of which are represented in our model. Most of the views that are used heavily during the stepping behavior, shown via a strong transition to the Debug Step/Move state, display information about the value of a (set of) variable at that point in the execution. The View Call Stack and View Output states are part of the debugging perspective, but are much more strongly associated with starting (or restarting) to debug (i.e., the Start Debug state). We believe that these two views are not useful to stepping, but more useful to observing a bug, shown via an exception in the View Output and assocated call stack in View Call Stack, prior to beginning a debug session.

Observation 3. Structured navigation and code search help developers in setting breakpoints. Setting breakpoints on relevant lines of code is commonly the first step in debugging. In doing so, according to our model, developers prefer to use structured navigation or code search, but not the hierarchical view of the project provided by the View Package Explorer. Hierarchial information is used for developers to learn the overall project structure, but this learning rarely transitions directly to debugging, unlike code search and structured program navigation.

5.1 Discussion

The overall theme of our findings based on the mined HMM model is that debugging is a multi-phase activity, requiring certain types of knowledge at each stage. Recent work has highlighted the information foraging theory of information seeking during debugging [9, 16], where developers follow a scent of information, based on a series of cues from the environment, in order to locate the code of interest. We believe that our findings support this line of work, by the diversity of cues that precede setting breakpoints, starting a debugging session, or stepping through code, indicative of a broad set of developer usage scenarios.

Our results indicate that several cues occur before the developer begins a debugging session in the IDE, including following the developer's examininations of the hierarchical structure of the project, as well as following a specific output or a specific cue in the call stack of (likely) a failing program. Similarly, several tools are likely to precede the setting of a breakpoint (i.e., code search and structured navigation). Overall, the IDE is not particularly well task optimized, in the sense that there is no natural transition between some of these tools, though it may make sense to provide it. For instance, starting to debug after looking at the call graph of a failing program is not in any way made easier or straightforward, though that seems to be a natural progression that many developers would follow in using the IDE.

Others research studies have also observed that the debugger is commonly used as a program comprehension tool and that many comprehension tasks involve the debugger in consort with a project-scope code search tool, and call graph navigation commands [4, 23]. However, these tools are very poorly integrated in the IDE, requiring a separate set of commands and without any easy transitions between them. We believe that clearer integration strategies would

help expert developers during program comprehension to operate more efficiently, while guiding novices into directions that are likely to yield faster and better answers to their information need.

5.2 Threats to Validity

The mining results have the potential of several internal and external threats to their validity. One internal threat comes from the mining technique itself. However, since HMMs are a well known and understood modeling approach and the final model quality surpassed 99%, we believe that this threat is significantly reduced. Externally, we collected usage data from only 196 professional developers at ABB, Inc. for a fixed period of time, which may not sufficiently represent a different population of developers or different time scales. Another external threat in the presented debugging model is the fact that we only investigated specific views and commands, while investingating other views may have brought forth other conclusions. A mitigating fact is that our technique is intended to reveal, via the set of interesting messages, frequently occurring sequences in the dataset, which were considered in the construction of the final model. In the future, we also aim to broaden our data collection and scope of our analysis.

6 Conclusions and Future Work

In this paper, we presented a novel approach to mining software developers' interaction traces with an IDE, which provides human-interpretable models that capture high-level developer behaviors of interest, using minor guidance from a researcher. The approach is based on interactively building Hidden Markov Models (HMMs) from IDE usage data, by adapting previous work by Jaroszewicz [6] to this domain and modifying to improve the interactivity of its feedback mechanism, which relies on capturing the differences between the HMM's prediction and the input dataset.

Using this approach, we mined a set of debugging sessions extracted from the IDE interactions of nearly 200 professional developers at ABB, Inc. The mined debugging model captures developers behaviors in setting breakpoints, starting to debug, and stepping through code, higlighting the most relevant commands and views for each of these activities commonly performed by the developers. From the mined model, we also observed that each task was supported by a clearly defined subset of IDE views and commands.

Future empirical work of this project includes the mining of new IDE models, relevant to specific development tasks (e.g. testing, program navigation). It also includes further data collection to provide greater strength to the mined models, as well as the collection of other relevant data to augment the IDE usage log with web behavior, specific program elements, or repository links.

7 References

[1] Silvia Bacci, Silvia Pandolfi, and Fulvia Pennoni. A comparison of some criteria for states selection in the latent markov model for longitudinal data. *Advances in Data Analysis and Classification*, 8(2):125–145, 2013.

[2] Gilles Celeux and Jean-Baptiste Durand. Selecting hidden markov model state number with cross-validated likelihood. *Computational Statistics*, 23(4):541–564, 2007.

[3] Christopher S Corley, Federico Lois, and Sebastian Quezada. Web usage patterns of developers. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 381–390. IEEE, 2015.

[4] Kostadin Damevski, David Shepherd, and Lori Pollock. A field study of how developers locate features in source code. *Empirical Software Engineering*, pages 1–24, 2015.

[5] The Eclipse Foundation Filtered UDC Data. http://archive.eclipse.org/projects/usagedata, 2016.

[6] Szymon Jaroszewicz. Interactive hmm construction based on interesting sequences. In *Proc. of Local Patterns to Global Models (LeGo'08) Workshop at the 12th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD'08)*, pages 82–91, 2008.

[7] Szymon Jaroszewicz. Using interesting sequences to interactively build hidden markov models. *Data Mining and Knowledge Discovery*, 21(1):186–220, 2010.

[8] Ghazaleh Khodabandelou, Charlotte Hug, Rebecca Deneckère, and Camille Salinesi. Unsupervised discovery of intentional process models from event logs. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 282–291. ACM, 2014.

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

[10] Taek Lee, Jaechang Nam, DongGyun Han, Sunghun Kim, and Hoh Peter In. Micro interaction metrics for defect prediction. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 311–321. ACM, 2011.

[11] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer – an investigation of how developers spend their time. In *Proceedings of ICPC 2015 (*23^{rd} *IEEE International Conference on Program Comprehension)*, pages 25–35, 2015.

[12] G.C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, July 2006.

[13] Emerson Murphy-Hill, Rahul Jiresal, and Gail C. Murphy. Improving software developers' fluency by recommending development environment commands. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 42:1–42:11. ACM Press, 2012.

[14] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. How We Refactor, and How We Know It. *IEEE Transactions on Software Engineering*, 38(1):5–18, January 2012.

[15] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining Fine-grained Code Changes to Detect Unknown Change Patterns. In *Proceedings*

of the 36th International Conference on Software Engineering, ICSE 2014, pages 803–813. ACM, 2014.

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

[17] L. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, Feb 1989.

[18] Vladimir A. Rubin, Alexey A. Mitsyuk, Irina A. Lomazova, and Wil M. P. van der Aalst. Process mining can be applied to software too! In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, pages 57:1–57:8. ACM, 2014.

[19] David Shepherd, Kostadin Damevski, Bartosz Ropski, and Thomas Fritz. Sando: an extensible local code search framework. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE, pages 15:1–15:2, 2012.

[20] Will Snipes, Vinay Augustine, Anil R. Nair, and Emerson M. Hill. Towards recognizing and rewarding efficient developer work patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1277–1280, 2013.

[21] Mohsen Vakilian and Ralph E. Johnson. Alternate refactoring paths reveal usability problems. pages 1106–1116. ACM Press, 2014.

[22] Wil Van Der Aalst. *Process mining: discovery, conformance and enhancement of business processes*. Springer Science & Business Media, 2011.

[23] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions. In *Software Maintenance, IEEE International Conference on*, pages 213–222. IEEE, 2011.