Using Activity Traces to Characterize Programming Behaviour Beyond the Lab

Gail C. Murphy,[†,‡] Petcharat Viriyakattiyaporn[†] and David Shepherd[‡]

[†]Department of Computer Science
University of British Columbia
{murphy, pviriya}@cs.ubc.ca

[‡]Tasktop Technologies Inc.
Vancouver Canada
david.shepherd@tasktop.com

Systematically improving the efficiency of programmers requires understanding what activities occur during programming, which activities are inefficient and then assessing languages, tools and processes proposed to improve the situation. Conducting the experiments required to support a systematic approach is difficult for many reasons, including the lack of availability of experienced programmers and the common belief that individual programmer effectiveness varies greatly. In this paper, we investigate whether generic activity traces of how a programmer interacts with a development environment can help bridge between results gathered in the lab with how programming occurs in the field. We describe the kinds of information that can be gleaned from activity traces, consider whether positive indication of a behaviour seen in the lab translates to data collected from the field, and discuss challenges with gathering appropriate data and with using gathered data appropriately.

1. Introduction

A systematic attack on improving programming efficiency requires understanding how programming is performed today, determining what programmers are finding inefficient and assessing languages, processes or tools proposed to improve the situation. The first step in this systematic attack requires experimentation, which may take any number of forms from controlled-lab experiments to case studies. Researchers face several challenges in conducting suitable experimentation, including:

- a lack of access to experienced programmers who have time available to participate in lab-oriented experiments and who often have concerns about data privacy when participating in field studies,
- difficulties in generalizing results from experiments containing small numbers of subjects because of the common belief in large differences in individual programmer efficacy [1], and
- difficulties in generalizing results because it is unknown how programming activities vary between short-term tasks and long-term tasks and between small-to-medium-sized source code bases and large code bases.

These challenges have limited the breadth of experimentation that has been conducted about existing programming activities. Researchers have typically been left with a choice between a more controlled experiment of an activity of interest on short-term tasks with programmers of varying ability (e.g., [5]) or a less controlled experiment involving more experienced programmers, often working on their own systems and tasks (eg., [4]). The former allows a more conclusive statement about behaviour for a particular population but leaves open the question of whether and how often the behaviour occurs across a larger population. The latter allows less conclusive statements about the precise nature of the behaviours occurring but more general statements about the prevalence of such behaviours.

In this paper, we consider whether it is possible, for some programming activities, to bridge between these end-points in the experimental spectrum using generic activity traces that record how a programmer works with an integrated development environment. The basic idea is to develop a set of *markers* of a behaviour of interest in a controlled lab setting and to then validate the occurrence and prevalence of the behaviour in a larger population by looking for the markers on data collected as experienced programmers work on their own tasks on their own systems. The markers are expressed as patterns in the generic activity traces collected as programmers work.

In this paper, we investigate a marker, summarized into a measure, for a code exploration pattern that has been of interest to researchers: the extent to which programmers explore the context of classes as they make changes to a system (e.g., [9]). We develop the marker and summarizing measure using activity traces collected from a controlled lab experiment. We then compute the measure on traces collected from a second controlled lab experiment and trian-

gulate the results using video from the programmers' sessions. Finally, we show how markers of the behaviour of local context investigation can be seen when we apply the measure to data gathered from programmers working in the field, adding support that this behaviour does extend beyond the lab and is likely an interesting target for improvement.

The main contribution of this paper is to show the *potential* of the overall approach. It is only through debate of proposed markers and their application to many kinds of data (more than one research group can collect) that we can arrive at a stable set of markers for a variety of behaviours.

2. Related Work

The capture of activity data into traces (or logs) is well-established in both human-computer interaction and software engineering studies. For example, Hilbert and Redmiles survey computer-aided techniques used to investigate usability information from user interface events [3], which range from counting the frequency of individual events to abstracting events and considering sequences of abstract events. In software engineering, there has been less discussion of the capture and use of traces in experiments conducted, although it is not uncommon to see reference to the capture of traces in experimental descriptions (e.g., [2]). Our use of activity traces in this paper differs in two ways. First, our primary focus is to show it is possible to use analysis of generic activity traces to help generalize from a small number of participants in a particular study to the behaviour of interest occuring over a larger population. We are not aware of the use of such generic activity traces for this purpose in existing literature. Second, the measure we investigate goes beyond the gathered activity logs to consider and infer how the programmer is interacting with the structure of the information on which they are working.

3. Generic Activity Traces

The generic activity traces we use in this paper are collected using the Mylyn Monitor framework[1] a plug-in for the Eclipse[2] integrated development environment. This monitor records each selection, edit and command performed by a programmer. For each interaction, the information reported by the monitor includes a timestamp, the kind of interaction, and the target of the interaction, such as a particular program element that was selected. An example of a fragment of a trace can be found elsewhere [6].

We refer to the traces as containing generic activity because we hook into the primary event mechanism of the integrated development environment to generate the trace. As such, it is straightforward to ensure a programmer's interactions with the many views in the integrated development environment are tracked.

As one might expect, the number of events in a trace file can grow quickly. To make analysis of these traces tractable for our investigations, we used subsets of the recorded generic activity traces, manually removing sections that were not directly related to exploring and editing Java code, which is the focus of the marker considered in this paper.

It would be a strong assumption to consider each interaction in a programmer's activity trace as equally purposeful and meaningful. To increase the likelihood our analysis of the traces focuses on meaningful parts of the recorded interaction, we use the concept of a *significant edit* as the occurrence of a number of consecutive edit events on the same class. In the traces we are using, an edit event occurs either due to an actual editing keystroke, such as adding a character or due to navigations in the editor, such as moving the cursor from one line to another. Our rationale in introducing the concept of a significant edit is to more likely identify points in the trace that correspond to a semantic investigation of the program.

4. A Marker for Local Context Investigations

For an initial investigation into markers, we chose to focus on a behaviour that several reserchers have targetted for improvement: programmers' exploration of a program element's local context—the interaction of the element with program elements in other classes—when they attempt to work with that element. We could not find any definitive evidence for the occurrence of this behaviour in the literature for this behaviour, only work to address it (e.g., [9]).

4.1. Developing a Marker

Developing a marker for the behaviour of investigating a local context requires careful consideration of the nature of the behaviour. We posit that to explore local context a programmer's actions must include visiting elements connected[3] to an edited element, either through the use of cross-reference tools or by other means of browsing. Presumably, this visiting of connected elements would need to occur within a finite time prior to the editing event for the programmer to be able to remember and process the information.

This analysis suggests the marker should consist of a graph of the connected elements visited near the point at which an element of interest was edited. We call such a

[1] `www.eclipse.org/mylyn`, verified 17/3/09

[2] `www.eclipse.org`, verified 17/3/09

[3] We consider two program elements connected if there is a calling, overriding, implementing, or referencing relationship between the elements.

graph a *Local Context Graph* (LCG). To characterize the kinds of LCG that arise in a set of traces, we need a measure. We define the *Local Context Measure* (LCM) as a measure associated with the marker for local context investigations. For each significant edit of a part of a class (C) in a trace, the LCM is a count of the number of classes that the programmer visited recently and that are structurally connected to C.

To compute the LCM for a significant edit, we form the LCG for that edit point and then compute the LCM as the number of classes transitively connected in the graph to the class containing the significant edit. As described above, we must determine what constitutes the recently visited classes in the trace' from which the LCG is formed. For the LCM to be meaningful, the definition of this concept should include those classes the programmer consulted to make the significant edit. We have chosen to use a window of approximately twenty minutes to encompass the elements that the programmer likely consulted to make a given edit.

4.2. Applying the Marker

To see what results the marker and its associated measure produce when applied to a known set of data, we computed the LCM for three activity traces collected during a controlled study about a tool for how to retroactively indicate the start of a programming task [7]. We refer to these traces as $\texttt{controlData}_1$. Each of these traces is from a separate subject, each of whom worked for over an hour total on four programming tasks on a slightly modified JHotDraw system. These subjects in this experiment had an average of five years of experience with Java and just under two years of experience with Eclipse. For these traces, we also have access to the program source, allowing us to compute the relationships between program elements using static program analysis facilities from Eclipse JDT.[4]

Figure 1(a) plots, for each significant edit in the traces of $\texttt{controlData}_1$, the number of classes used in the calculation of LCM for the edit against the LCM value. As the number of classes can influence the LCM upwards it is important to consider these numbers together. Each point in the plot is scaled to indicate the number of cases at that point. Overall, this figure shows that as a programmer's work involves more classes, there is need to understand the structural context of the classes involved in the work; programmers are not typically investigating isolated classes.

The values of LCM plotted in Figure 1(a) are based on a significant edit size of four and a translation of the twenty minute investigation window to 160 interactions.[5] We arrived at these values after experimentation with other choices; these values provide the best trade-off.

There are several threats to the validity of LCM. Most notably, some classes visited during the interval over which LCM is computed may have been visited prior to the programmer finding the focal point of the current task. working.

5. Validation

Determining if our approach has any merit requires considering two questions:

1. does LCM characterize the code navigation behaviours of interest, and
2. are similar characteristics of the phenomena seen in field data?

We investigate the first question by computing the LCM measure for a second known set of data ($\texttt{controlData}_2$) and then analyzing screen capture and think-aloud data captured for the participants who produced the traces in that data set to see if it confirms the computed measures. The $\texttt{controlData}_2$ data set consists of five traces collected during a different controlled study than for $\texttt{controlData}_1$ [8].These traces include work on tasks involving two software systems, JHotDraw and ProGuard.[6] Each subject in this experiment was a professional programmer with between nine months and seven years of experience with Java and between nine months and seven years of experience with Eclipse.

To answer the second question, we computed the LCM measure for traces collected from programmers in the field (`fieldData`). This data set comprises five traces collected during a field study [6]. The subjects in this study were professional programmers working on tasks on the systems on which they normally work. We do not have information about years of experience in Java and Eclipse available for these participants. Similar values for LCM computed for the field data as compared to the controlled study data provide positive evidence that our approach may generalize to field data. Dissimilar values cast doubt that the approach can be beneficial.

In this section, we use the parameter choices determined from the $\texttt{controlData}_1$ data set: a significant edit definition where the number of edit events is four and an interaction window of 160 events.

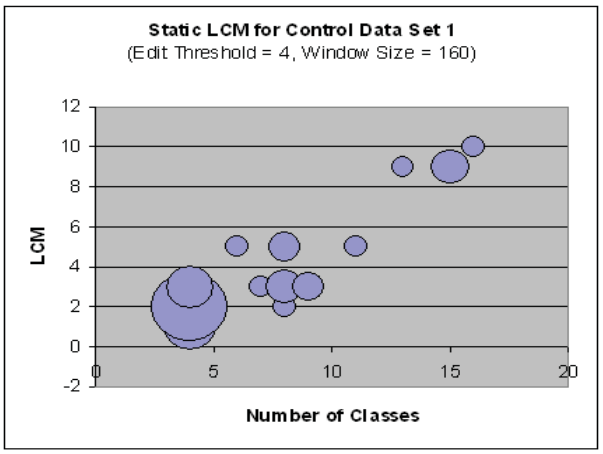
5.1. Triangulation

Figure 1(b) shows the LCM values for the $\texttt{controlData}_2$ traces. While there is some similarity in rise of LCM values as the number of classes that

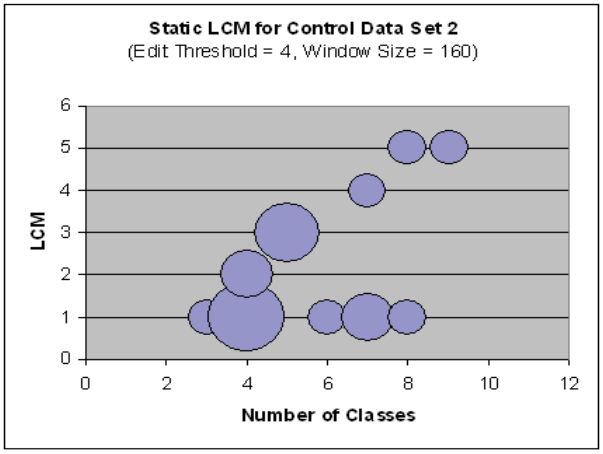
[4] `www.eclipse.org/jdt`, verified 17/3/09

[5] Given that we have full information about the experimental setup for $\texttt{controlData}_1$, we were able to analyze the traces and determine that twenty minutes of activity corresponded on average to 160 interactions in the activity trace.

[6] `proguard.sourceforge.net`, verified 17/3/09



(a) `ControlData`$_1$



(b) `ControlData`$_2$

Figure 1. Static LCMs for `ControlData`$_1$ and `ControlData`$_2$

are explored increases compared to the `controlData`$_1$ plot (Figure 1(a)), there are a number of bubbles moving out horizontally in the graph. Figure 2(b) includes a comparison of the LCM values for the two control data sets using logarithmic trend lines. More similarity appears in this interpretation of the data, lending some support that similar behaviour may be occurring across the data sets. To support our claim that the measures we compute are representative of the programming behaviours they are intended to characterize, we randomly sampled ten significant edit points in the traces from `controlData`$_2$, watched the video sequences and read the transcripts of the events leading up to that point and then rated whether the programmer was consulting local context, assigning a rating of low, medium, or high. We then computed ranges to represent low, medium and high from the data for `controlData`$_2$ based on the lowest third ([0,2]), middle third ([3,5]) and highest third (>=6) of the values for LCM. We were able to accurately predict LCM in 70% of the cases. A major threat to the validity of these results is the lack of precise objective criteria for our rankings from the video and transcript data and the lack of correlation statistics with another coder of that data.

Overall, the trend similarity in the graphs and the degree to which our measures seem to capture phenomena of interest when we watch the video of the sessions suggests that there is some validity to our proposed approach, enough to warrant future effort towards developing this form of validation.

5.2. Generalization

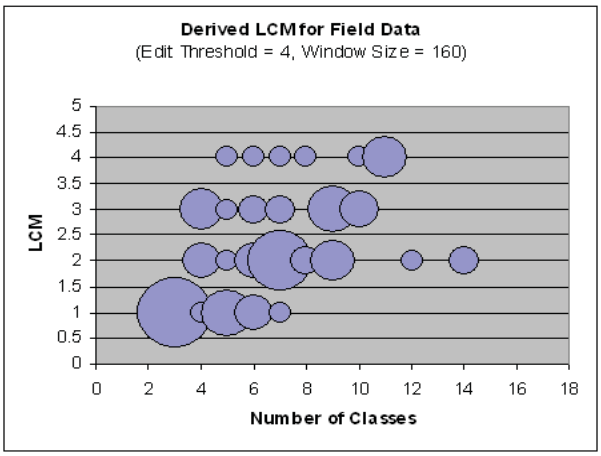
Figure 2(a) shows the LCM values for the `fieldData` traces. The computation of the LCM values is difficult for the field data as we do not have access to the source code on which these subjects worked. As a result, we derived relationships from the kinds of interactions recorded in the trace by traversing the trace, tracking when any kind of structural query was triggered and noting the most recently visited program element, V, at that point. When we see an event that corresponds to the query result Q being viewed, we consider there to be an edge between V and Q. This algorithm produces a lower bound on the number of relationship between program elements. We verified that this approach produces correct edges manually and through comparison to those produced using static analysis; for `controlData`$_1$ this approach produced 59% of the edges surrounding signficant edits.

On the surface, Figure 2(a) does not look similar to the LCM plots shown earlier for the control data sets. To investigate whether there is some deeper similarity, Figure 2(b) compares the logarithmic trend lines for all three data sets: `controlData`$_1$, `controlData`$_2$ and `fieldData`. In this plot, we see that there is similarity in the upward trend of the LCM plots, lending some support that programmers in the field investigate more structural context information as the number of classes they investigate increases.

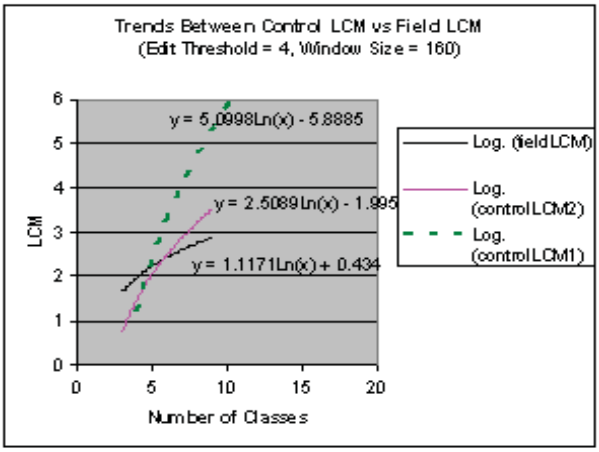
Major threats to the validity of the results include the small number of traces in each data set and the lack of triangulation data for the analysis of the `fieldData` traces.

6. Discussion

The approach we are advocating in this paper is fraught with potential dangers if not applied carefully. The approach is limited in that it cannot be used to prove or disprove the presence or absence of programming behaviours, but rather is intended to help ensure behaviours of interest occur across a wider population of programmers.



(a) Derived LCM for `FieldData`



(b) LCM Trends

Figure 2. `FieldData` **LCM and Comparison of LCMs across Data Sets**

The characteristics of the trace data available affect what kinds of markers can be defined and thus, which kinds of behaviours can be investigated. This limitation might be reduced if the scope of information captured in a generic activity trace is expanded, such as including checkpoints of the structure of the target software system.

A benefit of our approach is that it might enable the sharing of typically hard to obtain traces of experienced programmers. If traces from such subjects could be shared between researchers, it would be possible to investigate the generality of behaviours across a much wider population.

7. Summary

In a perfect world, all researchers would have access to experimental subjects from a target population of interest. In reality, for studies of new languages, tools and processes developed to improve program activities it is hard to get access to professional programmers. In this paper, we have described how activity trace data might be used as a means of determining whether programming behaviour seen in lab experiments with often less experienced programmers might also be occurring in the field. To show that our approach might have merit, we have introduced a marker for one programming behaviour—the investigation of local structural context of a programming element being changed. We have shown that there is evidence to support that this marker both captures the behaviour of interest and that the marker has similar values across multiple data sets. We see this result as a step towards improving our approaches to experimentation in software engineering.

Acknowledgements

We thank Ducky Sherwood for help in interpreting the results. This work was funded in part by NSERC and in part by IBM.

References

[1] B. Curtis. Substantiating programmer variability. In *Proc. of the IEEE*, volume 69, pages 846–846, 1981.

[2] R. DeLine, M. Czerwinski, and G. Robertson. Easing program comprehension by sharing navigation data. In *Proc. of VLHCC*, pages 241–248, 2005.

[3] D. M. Hilbert and D. F. Redmiles. Extracting usability information from user interface events. In *Proc. of ACM Comput. Surv.*, volume 32, pages 384–421, 2000.

[4] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of the 14th ACM SIGSOFT Int'l Symp. on Foundations of Soft. Eng.*, pages 1–11, 2006.

[5] 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. of ICSE*, pages 126–135, 2005.

[6] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? *IEEE Software*, 23(4):76–83, 2006.

[7] I. Safer and G. C. Murphy. Comparing episodic and semantic interfaces for task boundary identification. In *Proc. of CASCON*, pages 229–243, 2007.

[8] K. D. Sherwood. Path exploration during code navigation. Master's thesis, University of British Columbia, August 2008.

[9] V. Sinha, D. Karger, and R. Miller. Relo: Helping users manage context during interactive exploratory visualization of large codebases. In *Proc. of VLHCC*, pages 187–194, 2006.