In *Proceedings of the ICSE Workshop on Directions in Software Engineering Environments*, pages 83–88, IEE, May 2004

Program Navigation Analysis to Support Task-aware Software Development Environments

Martin P. Robillard and Gail C. Murphy
Department of Computer Science
University of British Columbia, Canada
{mrobilla,murphy}@cs.ubc.ca

Abstract

Performing a software modification requires a developer to investigate a program to find and understand the code relevant to the modification task. Although standard program investigation tools can help developers in this activity, developers often get lost in the complex web of information available about a program.

To address this problem we propose to use program navigation analysis, a technique to record and analyze the actions of a developer using a software development environment in order to infer the current task and the subset of a program relevant to this task. Our hypothesis is that we can use the results of program navigation analysis to dynamically configure the interface of a software development environment in a way that alleviates the problems of disorientation experienced by developers.

In this paper, we define program navigation analysis and present an overview of its underpinnings, summarize our experience with the technique, highlight important technical challenges, and discuss the benefits that can be reaped from use of the technique.

1. Introduction

Before performing a modification to a software system, developers must usually investigate the system to find and understand the source code relevant to the change task [2]. The large size of most production software systems, and the pressures inherent on development and maintenance tasks, render the program exploration activity a serious challenge to developers [3]. These factors make it unrealistic to expect developers to master the complete details of a system's design and implementation prior to undertaking a modification. Rather, a developer must efficiently discover a sufficient amount of the structure and behavior of the program relevant to a modification [21].

Many tools have been developed to support developers in the task of identifying information about a program that is relevant to a change task. In particular, cross-referencing tools, such as code browsers and program databases, allow developers to perform queries that elicit the structural relations between different elements in a program. The main purpose of cross-referencing tools is to provide developers with information that cannot be obtained easily through source code inspection [5, 8, 14, 16, 22]. Currently, support for performing cross-reference queries is a standard feature of most modern software development environments, such as the Eclipse platform [15].

Although cross-referencing tools allow developers to quickly identify related parts of a program that are not co-located, their use leads to non-sequential patterns of navigation through a program [4] that can lead to problems of "disorientation" that have been observed and extensively studied in the context of hypertext navigation [6, 7, 13]. For example, one problem often encountered by developers investigating a program is the *Embedded Digression Problem*:

> ...pursuing multiple paths and digressions leads to a lot of trouble such as: losing your place, forgetting to return from digressions, and neglecting to pursue digressions you intended to follow. [7, p. 408].

As an illustration of this problem, we can take the case of a developer investigating the code implementing a matrix calculation algorithm. At some point, the developer may realize that the global memory allocation strategy in the program investigated also needs to be understood [1], and then pursue a thread of investigation relevant to the memory allocation strategy. Once this aspect is understood, the developer might become "lost", and not remember how or where to pursue the investigation of the matrix calculation algorithm.

Various approaches have been proposed to alleviate the problem of orientation in complex data structures. Such approaches include history lists and similar features [7], visual displays that support the metaphor of physical navigation [9, 13], and tools that can record the paths taken by developers while investigating programs [12]. However, although they present program information in a way that may facilitate navigation, these approaches do not address the fundamental problem of helping developers focus on the information relevant to their task. We believe that task-specific support for program navigation is essential for developers to keep their bearing in what would otherwise be an overwhelming amount of information. Unfortunately, it is practically impossible to determine what is relevant to a task through standard program analyses. As Woods et al. have observed in a different context, the discovery of relevant information is inevitably a human-centric activity.

> Just as machine diagnosis can err, we cannot expect machine agents to consistently and correctly identify all of the data that is relevant and significant in a particular context in order to bring it to the attention of the human practitioner. It always takes cognitive work to find the significance of data. [24, p. 31]

Based on this fundamental observation, we are interested in developing software development environments that are sensitive to the activities of a developer, and that can contextualize their interface based on an interpretation of the task being performed. For this purpose, we define and motivate the need for a new type of analysis, called *program navigation analysis*, which focuses on the interpretation of the navigation activities of developers as they investigate a program, with the purpose of facilitating program investigation in the context of a specific task.

In the rest of this paper, we define program navigation analysis and present an overview of its underpinnings (Section 2), summarize our experience with the technique (Section 3), highlight important technical challenges (Section 4), and discuss benefits that can be reaped from use of the technique (Section 5).

2. Program Navigation Analysis

Generally speaking, we define *program navigation analysis* as the process of collecting data about the program investigation activities of a developer during a software engineering task, and analyzing this data to support the same or a different software engineering task.

The idea of using data about the navigation paths of tool users has long been a desirable goal, and research addressing this ideal finds its source in work on user interfaces and human-computer interaction:

> A nice outcome would be to have a personalized summary of what has been examined during a browsing session that could later be used for analysis and integration [7, p. 408].

An early example in the direction of program navigation analysis is the idea of recording how much different parts of a document have been respectively read and edited, and of displaying information about this *edit* and *read wear* as graphical annotations in the scrollbars of a text editor [10].

Later work based on this idea include *Footprints*, a project to help users navigate effectively through a web of information by analyzing a record of previous interaction between people and a system:

> Work done by users to solve problems in information systems should leave traces. These traces should be accessible to future users who could take advantage of the work done in the past to make their own problem-solving easier. [23, p. 270]

Research in the context of the *Footprints* project has led to the development of advanced web navigation tools that have been shown to help users perform information-gathering tasks more effectively.

In the context of software engineering, we propose the more ambitious goal of using program navigation analysis to automatically tailor software development environments to a software engineering *task* based on an inference of the code relevant to the *task*. This goal can be contrasted with the purpose of user-adaptive systems, which is to adapt to particular *users* [11].

With a mature program navigation analysis technology available, we hope to be able to produce task-aware software development environments that can reduce the problem of disorientation experienced by developers during program investigation activities by automatically determining and visually emphasizing the subset of a program likely to be related to the task.

3. Our Experience

As part of a research project, we developed an algorithm to automatically infer clusters of program elements potentially related to a task based on an analysis of the source code a developer examined during a program investigation session [18]. The motivation for this research was to find an inexpensive way to generate artifacts describing how and where different concerns[1] are implemented in a sys-

[1] We define the term *concern* as any high-level concept a developer needs to consider during a software engineering task, and that has a corresponding mapping in the source code of a system.

tem. With appropriate tool support [19], concern descriptions have been shown to help developers perform software evolution tasks more systematically [17].

The technique we developed is based on an analysis of all the code that becomes visible in the editor window of a software development environment during a program investigation session. Specifically, our technique comprises three phases: a transcription phase, an analysis phase, and a clustering phase.

Transcription Phase In this phase, a machine-readable transcript summarizing the investigation session is produced. Information about the code viewed by a developer can be captured in a variety of ways. We decided to capture a list of navigation events each comprising the following information:

- The set of all the methods completely or partially visible in the active editor window at any point in time (fields are ignored for reasons described in Section 4).
- How each event was produced, distinguishing between five potential actions: choosing an method from a browser, performing a keyword search, performing a cross-reference search, scrolling, and recalling a previously active window.
- The target of the action (e.g., in the case where a method is selected from a browser, we record which method was selected).

In our initial research, we produced the navigation transcripts based on a manual coding of screen movies recorded during empirical studies of program evolution. It is understood that a mature version of this technology should support the automatic production of navigation transcripts.

Analysis Phase In this phase, an inference program scans the navigation transcript and assigns a *correlation metric* to all pairs of methods present in the transcript based on how "related" the two methods in the pair were during the program investigation. A number of factors are considered in this analysis, but the general intuition is that methods that were heavily examined in direct or close sequence probably are related in the context of a *concern* (or task). The details of the algorithm can be found in a separate article [18].

Clustering phase In the final phase, the inference program selects a user-specified number of pairs in decreasing value of correlation and groups elements into concerns by taking the transitive closure of each pair. For example, if the pairs [A,B][B,C][D,E] are found to have the highest correlation metric, the inference program produces the concerns [A,B,C] and [D,E].

Empirical evaluation of this technique on two different program investigation tasks of 45 and 60 minutes involving a total of five different developers, showed that in every case the algorithm could infer a core set of elements representing important (and in some cases, critical) methods related to a task. In each case, the results were produced with a minimum amount of noise. As a typical example, for one subject, three useful concerns were found in a list of seven concerns proposed by the algorithm. This project thus showed the feasibility of inferring code relevant to a task based on program navigation activities.

With the algorithm summarized above, the performance of program navigation analysis is not an issue. For investigation sessions of one hour, the time required to generate results using a sub-optimal implementation was in the order of a few seconds. At this point we do not believe computational complexity to be an impediment to the adoption of program navigation analysis. As for the automatic generation of navigation transcripts, we are currently investigating potential ways to instrument the Eclipse platform to automatically produce the navigation transcripts, and we do not foresee this issue to be a fundamental hurdle.

4. Technical Challenges

In our experience, we found that the most complex issues related to program navigation analysis lie in generating precise results that require a minimum of user input to correct or filter. In this context, we now discuss the four important challenges we have identified when trying to improve the precision of our program navigation analysis technique: determining what a developer is *really* examining, accounting for time, identifying the start point of a task, and inferring the nature of a task. Although these challenges were identified as part our our work on concern inference algorithms, we believe that they are general challenges inherent to the concept of program navigation analysis.

Inferring the code a developer is examining When investigating a program, a lot of the information a developer accesses is irrelevant and mentally discarded. For example, when studying the code for a short method, a developer will typically have in his field of view the code for other methods above or below the method of interest in the file. In this case, can we be sure that the developer is only investigating one method? In other words, how can we determine with accuracy the subset of methods on the screen that is actively examined by a developer? This problem is exacerbated in the case of field declarations, which typically occupy a very small fraction of the source code on display in an editor. For this reason, our current algorithm ignores field declarations in most situations. In the case of methods, we use a

stochastic approach that involves assigning artificial probabilities to the different methods visible based of a number of factors such as whether the method was accessed explicitly or revealed through scrolling. We believe this type of stochastic approach to be generally more desirable than alternatives such as controlling the number of lines of source code displayed in an editor window. However, much work needs to be done to improve the parameters of the algorithm. Ideally, configurations for the probabilities involved in determining which method is examined should be based on empirical data, and should include self-training feedback.

Accounting for time Our current program navigation analysis algorithm does not account for the time a developer spends examining each method. However, we believe that important gains in precision can be obtained by factoring in such time data. Unfortunately, without specialized devices, it is generally not possible to determine, based on an instrumentation of a software development environment, whether or not a developer is actually looking at the code on the screen. Specifically, important interferences can be caused just by having a developer leave to get a coffee, or turn away from the screen to discuss with a colleague. In such cases, a method that was on the screen for five minutes could have been actually examined for only five seconds. After numerous unsuccessful attempts at increasing the precision of our algorithm through the inclusion of time data, we decided to develop our initial prototype without this factor. In a fully-integrated approach, one possibility could be to have an explicit way for developers to "pause" the recording of their activities. Unfortunately, this alternative is disruptive and decreases the value of the approach. In any case, the inclusion of time data in program navigation analysis without user interference remains an open problem.

Identifying the start point of a task During the empirical evaluation of our program navigation analysis technique, we observed that, in the absence of any information about where to start a program investigation, developers began their task with very broad searches that often yielded few useful results [18]. We also observed that the results of the analysis were more precise for developers who were given an initial start point for their investigation. These observations indicate that broad searches such as the ones performed to identified a starting point for an investigation task are of little value to help determine the code relevant to a task. A simple solution to elide these broad initial searches is to allow developers to manually start the recording of their investigation activities once they become involved in a focused investigation. However, in addition to being disruptive, this alternative requires that developers be able and willing to identify when their investigation becomes focused. Instead, a more desirable outcome would be to infer this transition automatically. This inference may be possible through an analysis of how many search results are used versus how many are discarded. However, we have not yet investigated the feasibility of this idea.

Inferring the nature of a task The program navigation analysis we developed applies a set of general heuristics to the record of a developer's investigation actions. However, we believe the precision of program navigation analysis may be further improved by inferring the nature of the task being performed. For example, we expect the navigation behavior of developers to be different whether they are performing a debugging task, planning a refactoring, writing unit tests, or enhancing a feature. Characterizing the program navigation behavior of a developer in terms of the nature of a task performed may allow us to apply program navigation analysis heuristics that are specially tailored to the task, yielding more precise results.

5. Potential Uses for Program Navigation Analysis

The technique we developed is a proof of concept intended for a specific purpose: the production of artifacts describing the implementation of different concerns in source code. However, once program navigation analysis is a mature technology, we believe it will be possible to use it to support a number of software development environment features intended to mitigate the disorientation problems associated with program investigation activities. Foreseeable examples include support for streamlined program navigation, task-sensitive history lists, and task-aware environment configuration.

Streamlined program navigation By inferring the code relevant to a task, it will be possible to highlight it in code browsers, editors, and other views developers use to access program information. This feature should have a direct impact on the efficiency of developers. Indeed, during empirical studies of programmers [20], we noticed that developers often accessed unrelated elements above or below a related element in a code browser. For example, Figure 1 is a partial view of a code browser showing 11 of the 181 elements declared in a class accessed by a developer as part of a software change task we studied. As we observed during the study, in the process of accessing method `recoverAutosave`, which was previously identified as relevant, the developer mistakenly accessed method with similar-looking signatures (e.g., method `removeAllMarkers`).

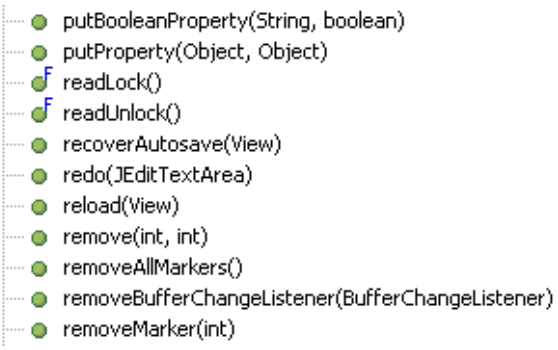

Figure 1. A typical code browser

By making elements related to a task more conspicuous in the software development environment as soon as they are identified through program navigation analysis, we expect that developers will access the program information with fewer false positives. For example, in the case described above, method `recoverAutosave` could be flagged in the browser after being identified as relevant to the task with program navigation analysis (see Figure 2), preventing the developer from mistakenly accessing irrelevant methods such as `removeAllMarkers`.

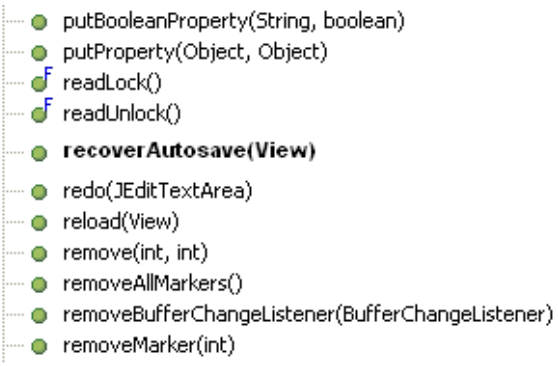

Figure 2. A code browser enhanced with program navigation analysis

Task-sensitive history lists History lists present in web browsers and the search engines of software development environments keep a record of all the searches performed by a user. However, because of the heuristic nature of program investigation, many of the search results are useless dead-ends. Using information obtained through program navigation analysis, we could make information relevant to a task more conspicuous in history lists.

Task-aware environment configuration The two features described above rely on program navigation analysis to identify the subset of a program relevant to a task. A more ambitious goal for the use of program navigation analysis is to also infer the nature of the task itself. Besides improving the precision of the analysis results (see above), a characterization of the task a developer is performing could be used to support a dynamic configuration of the software development environment. This type of dynamic environment configuration could be used to make the tools most relevant to the task conspicuous in the software development environment. Although previous work has pointed to the possibility of inferring the nature of a task from an analysis of navigation paths [4], we have not yet evaluated this possibility in the context of a prototype tool.

6. Conclusions

The discovery of information relevant to a software engineering task is inevitably a human-centric activity. Although standard program investigation tools can help developers find elements in a program that a relevant to a task, they can also cause developers to become lost in the web of information that the software engineering tools make available about a program.

To address this problem, we propose to use program navigation analysis to infer the task a developer is currently performing, and the subset of a program relevant to that task. Based on this analysis, we hope to be able to dynamically customize software development environments in order to help developers focus on their task as much as possible. We believe software development environments features leveraging off program navigation analysis have the potential to mitigate the problem of disorientation associated with program investigation activities.

A proof-of-concept project has shown that program navigation analysis is feasible. Even though technical challenges remain on the road to a mature and fully integrated version of this technology, we believe that the potential benefits warrant further investigation.

Acknowledgments

The authors are grateful to Brian de Alwis for interesting discussions on the topic of program navigation and for providing useful comments and references. Thanks also go to Andrea Bunt for additional pointers and to Reid Holmes for reviewing the paper.

References

[1] E. L. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: An inquisitive study. In *Proceedings of the 1st Conference on Aspect-Oriented Software Development*, pages 120–126. ACM Press, April 2002.

[2] B. W. Boehm. Software engineering. *IEEE Transactions on Computers*, 25(12):1226–1242, December 1976.

[3] S. A. Bohner and R. S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[4] D. Canter, R. Rivers, and G. Storrs. Characterizing user navigation through complex data structures. *Behavior and Information Technology*, 4(2):93–102, 1985.

[5] Y.-F. Chen, M. Y. Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[6] J. Conklin. Hypertext: An introduction and survey. *IEEE Computer*, 20(9):17–41, September 1987.

[7] C. L. Foss. Tools for reading and browsing hypertext. *Information Processing & Management*, 25(4):407–418, 1989.

[8] A. Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, MA, USA, 1984.

[9] W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274. IEEE Computer Society Press, May 2001.

[10] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit wear and read wear. In *Proceedings of the Conference on Human Factors and Computing Systems*, pages 3–9. ACM Press, May 1992.

[11] A. Jameson. Adaptive interfaces and agents. In J. Jacko and A. Sears, editors, *Human-computer interaction handbook*, pages 305–330. Erlbaum, 2003.

[12] D. Janzen and K. De Volder. Navigating and querying code without getting lost. In *Proceedings of the Conference on Aspect-Oriented Software Development*. ACM Press, March 2003.

[13] H. Kim and S. C. Hirtle. Spatial metaphors and disorientation in hypertext browsing. *Behaviour & Information Technology*, 14(4):239–250, 1995.

[14] M. Lejter, S. Meyers, and S. P. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1045–1052, December 1992.

[15] Object Technology International, Inc. Eclipse platform technical overview. White Paper, July 2001.

[16] P. D. O'Brien, D. C. Halbert, and M. F. Kilian. The Trellis programming environment. In *Proceedings of the Conference on Object-oriented Programming, Systems, and Applications*, pages 91–102. ACM Press, October 1987.

[17] M. P. Robillard and G. C. Murphy. Concern Graphs: Finding and describing concerns using structural program dependencies. In *Proceedings of the 24th International Conference on Software Engineering*, pages 406–416. ACM Press, May 2002.

[18] M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–234. IEEE Computer Society Press, October 2003.

[19] M. P. Robillard and G. C. Murphy. FEAT: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, pages 822–823. ACM Press, May 2003.

[20] M. P. Robillard and G. C. Murphy. A study of program evolution involving scattered concerns. Technical Report TR-2003-06, Department of Computer Science, University of British Columbia, Vancouver, BC, Canada, March 2003.

[21] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil. An examination of software engineering work practices. In *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*, pages 209–223. IBM Press, 1997.

[22] W. Teitelman and L. Masinter. The Interlisp programming environment. *IEEE Computer*, 14(4):25–33, April 1981.

[23] A. Wexelblat and P. Maes. Footprints: History-rich tools for information foraging. In *Proceedings of the Conference on Human Factors and Computing Systems*, pages 270–277. ACM Press, May 1999.

[24] D. Woods, E. Patterson, and E. Roth. Can we ever escape from data overload? A cognitive system diagnosis. *Cognition, Technology & Work*, 4(1):22–36, April 2002.