Supporting Bug Investigation using History Analysis

Francisco Servant
University of California, Irvine, U.S.A.
fservant@uci.edu

Abstract—**In my research, I propose an automated technique to support bug investigation by using a novel analysis of the history of the source code. During the bug-fixing process, developers spend a high amount of manual effort investigating the bug in order to answer a series of questions about it. My research will support developers in answering the following questions about a bug: *Who is the most suitable developer to fix the bug?*, *Where is the bug located?*, *When was the bug inserted?* and *Why was the bug inserted?***

I. Proposed Research

All software projects are affected by software bugs. New bugs in software are found and reported every day. As an example, the Eclipse open-source project can receive an average of 29 new bug reports per day [2]. Fixing bugs is a highly expensive process in terms of developer effort. Bug-fixing is the most important task of software maintenance, which previous studies have shown to take more than 90% of the development effort [6]. Also, an important part of the effort that developers spend in fixing bugs involves their investigation to answer multiple questions about them.

The goal of my research is to provide automated support for bug investigation in the process of answering: ***Who** is the most suitable developer to fix the bug?*, ***Where** is the bug located?*, ***When** was the bug inserted?* and ***Why** was the bug inserted?* In order to provide automated support for bug investigation, I will propose a novel analysis for the history of source code. Since the code involved in a bug may be composed of contiguous or disparate lines of code from one or multiple methods and files, I will propose a history-analysis technique that operates at the granularity of a line of code.

Current revision-control systems allow users to process code history at the granularity of files (e.g. Subversion, Git). Some researchers proposed techniques to obtain code history at the granularity of methods, e.g. [10]. Other researchers proposed techniques to track the history of single lines of code, but they still require this history to be queried for whole source code files [23]. I will propose a novel technique to obtain the minimum complete history of any set of lines of code.

By using this novel history analysis, my research will process in different ways the history of the lines of code involved in a bug to provide automated support for answering the previously mentioned questions for bug investigation.

II. Motivation and Related Work

My research proposes to provide developers with automated support to answer four questions in their investigation of bugs:

Who is the most suitable developer to fix the bug? With high numbers of new bugs being found every day and large numbers of developers working in the same software project, development teams need to dedicate a non-trivial amount of time to decide who is the most suitable person to fix each one of them. Currently, development teams perform the developer-to-bug assignment by manually analyzing very limited information, such as: the contents of the bug description in the bug report, the error message provided by the bug, and their experiential knowledge about the expertise of each developer.

The problem of finding the most suitable developer to fix a bug has been mainly approached in three different ways. First, some authors use a machine-learning classifier to predict which developer should fix a bug according to its natural-text description inside the bug report, e.g. [3]. Anvik et al. [3] evaluated the performance of six machine-learning classifiers — Naive Bayes, Support Vector Machines, C4.5, Expectation Maximization, Conjuctive Rules, and Nearest Neighbor. Second, other authors build expertise profiles by extracting terms for the bug description, e.g. [21]. Shokripour et al. [21] build expertise profiles from bug descriptions, commit messages, and comments in the source code. Third, other authors build expertise profiles by extracting topics from the source code that each developer modified, e.g. [15].

However, all these approaches require a natural-text description of the bug, and therefore cannot be applied in situations where the only representation of the bug available is a failing execution or test case. My proposed research will be applicable as soon as the bug can be reproduced by a failing execution.

Where is the bug located? Once the bug has been assigned to a developer, she needs to debug the code in order to fix it. Debugging is one of the most time-consuming tasks within the efforts of reducing the number of bugs in software [4]. Within the debugging process, locating the errors in the source code was found to be the most difficult task [22].

Multiple techniques have been proposed by researchers to localize the area of the source code that is correlated with a bug, e.g. [11], [14]. For example, Jones et al. [11] analyze the correlation between executed lines of code and failing test cases. Each line of code is more correlated with the bug if it is executed by more failing test cases and it is less correlated with the bug if it is executed by more passing test cases.

The fault-localization techniques proposed to-date are designed specifically to provide an indication of where in the source code the bug is located. My proposed research will use fault-localization techniques to identify the lines of code that are involved in a bug, but it will also adapt them to combine their results with my new code-history analysis.

When and why was the bug inserted? After the bug has been located within the source code, developers need to make an effort to understand it. Von Mayrhauser and Vans [24] found that good program understanding was of central importance in the debugging task. Additionally, Gilmore [7] found that the success of a debugging session was more attributed to better comprehension than to better debugging skills. However, Ko et al. [12] found that one of the most time consuming questions to answer about the source code was why code had been implemented in a certain way. Also, LaToza and Myers [13] found that among the hard-to-answer questions about source code were questions about when was some code changed.

Some authors have proposed techniques to visualize the evolution of source code in order to provide insights about when and why code was implemented in a certain manner. Some visualizations show the evolution of the source code at a granularity of source code files, e.g. [8], [25]. Other visualizations show more details, such as the evolution of methods, e.g. [9], [10]. And other techniques use a granularity of the line of code to visualize the evolution of code, e.g. [23].

However, neither of these visualizations allow the exploration of the history of an arbitrary set of lines of code, such as the lines of code that are involved in a bug. Even the techniques that allow a line-of-code granularity require the visualization of the history of whole files. My proposed research will provide the first technique that allows to obtain the minimum complete history of a set of lines of code.

III. Work to Date

I have already performed some work to provide automated support for bug investigation. First, I designed a technique to perform fine-grained analysis of source-code history. Then, I started exploring how fine-grained history analysis and its combination with program analysis can support developers in answering the proposed questions for bug investigation.

Novel History Analysis. In order to allow an analysis of source-code history at the fine-grained level of lines of code, I presented History Slicing [16]. History Slicing is the first technique that allows to obtain the minimum complete history of a set of lines of code. History Slicing automatically tracks the whole history of each line of code in a revision-control system by building a *History Graph*. A History Graph is a bi-partite graph, which is built by mapping the corresponding lines of code of the program between consecutive revisions. History Slicing uses the traversal of the History Graph to allow an efficient analysis of the history of any set of lines of code by obtaining *History Slices*. The History Slice for a set of lines of code of interest (i.e., slicing criterion) contains all their corresponding lines of code in all past revisions of the software project in which they were modified.

History slicing addresses the limitations of traditional revision-control systems by providing the ability to (1) query source-code history at the line level, for any arbitrary set of lines, across any set of files, and (2) obtain the minimum complete evolution of those lines, along with their tracing among revisions. In the same way that program slicing selects the relevant areas of the code in the dimension of the program, history slicing selects both the relevant revisions of the code in the dimension of history and the appropriate lines of code in each of those revisions in the dimension of the program.

In my experiments, History Slicing showed drastic improvements over conventional revision-control systems. History Slicing reduced the amount of information that needed to be processed in order to obtain the history of a set of lines of code by up to three orders of magnitude.

Who is the most suitable developer to fix the bug? In order to provide an automated technique to identify the most suitable developer to fix a bug, I proposed WhoseFault [18]. WhoseFault was the first developer-to-bug assignment technique to not require a description of the bug written in natural language.

WhoseFault first uses a statistical coverage-based fault-localization technique to identify the lines of code that are correlated with the bug and to provide a score of correlation for each one of them. Then, WhoseFault obtains the History Slice for each line of code involved in the bug, which includes, for each change, its type, its time, and who made it. In its final step, WhoseFault uses an expertise-finding analysis that processes the whole history of every line of code that was correlated with the bug. Through the expertise-finding step, every developer that ever modified any one of such lines of code receives an expertise score that depends, for each line of code, on: (1) its score of correlation with the bug, (2) the number of changes made to it by each developer, and (3) the recency of such changes. After the expertise-finding step, WhoseFault returns a ranked list of developers in terms of how much expertise they have over the faulty lines of code.

My experiments showed that WhoseFault chooses the correct developer in the first ranked position for 35% of the bugs studied, for 50% when considering the top two positions, and for 81% when considering the top three positions. I also compared WhoseFault against an existing expertise-assessment technique and found that WhoseFault provided greater accuracy, by up to 37%.

Where is the bug located? My research uses fault-localization techniques to identify the areas of the code that are correlated with the bug. I have performed experiments to understand how the use of a simple fault-localization technique compares to the use of a statistical coverage-based fault-localization technique within WhoseFault's algorithm to find the most suitable developer to fix a bug. I observed no change in WhoseFault's accuracy when using a simple technique that assigns the same score of correlation with the bug to all the lines of code that were executed by a failing execution. Since WhoseFault performs a fault-localization step as part of its approach, it is the first developer-to-bug assignment technique to provide a recommendation of the location of the bug as well as the most suitable developers to fix it.

When was the bug inserted? As a first step for supporting the identification of the moment when a bug was inserted, I have studied the utility of History Slicing in helping developers to find the origin of a code snippet. I performed such study as part of a set of experiments to assess the utility of History Slicing in supporting developers for answering multiple questions about the history of source code [17]. I studied developers using conventional revision control techniques and developers using an automated implementation of History Slicing to perform software maintenance tasks that involved understanding the history of source code. Developers using History Slicing: (1) correctly performed the tasks in more than double the cases, and (2) used almost half the time to complete the tasks.

Why was the bug inserted? My approach to supporting developers in finding out the reason why a bug was inserted uses a software visualization. I have developed a visualization for source-code history to support developers in understanding the rationale behind a set of lines of code. Such visualization is implemented in a tool called CHRONOS [19]. CHRONOS is a plug-in for Eclipse that displays the History Slice for any set of lines of code. For every change to a set of lines of code, CHRONOS shows the actual contents of that change and the commit message for it.

IV. FUTURE WORK

In the future, I plan to extend my research in bug analysis for all of the four proposed questions for bug investigation.

Who is the most suitable developer to fix the bug? In my WhoseFault [18] project, I showed that the combination of program analysis and History Slicing was highly successful at identifying the most suitable developer to fix a bug. Next, I intend to extend WhoseFault by improving its expertise-finding algorithm, by comparing it with other state-of-the-art developer-to-bug assignment techniques, and by creating a visualization to provide a better understanding of why each developer was recommended.

WhoseFault's expertise-finding algorithm may be improved by modifying each of its components. I intend to improve WhoseFault's expertise-finding algorithm by experimenting with multiple fault-localization techniques, different line-mapping techniques for building the History Graph, and variations of WhoseFault's expertise formula. For example, the results provided by WhoseFault may improve by using the more recent Ochiai [1] fault-localization technique instead of Tarantula [11], or by penalizing the age of code changes in an exponential manner instead of linearly. Additionally, WhoseFault may also achieve an acceptable effectiveness by reducing its requirements. I plan to experiment with decreasing amounts of test suite execution and history analysis to study how much we can reduce the complexity of WhoseFault's algorithm without impacting its effectiveness.

Once I have found an optimal configuration for Whose-Fault's algorithm, I also plan to compare its effectiveness with other state-of-the-art developer-to-bug assignment techniques. Other authors have proposed techniques to recommend the most suited developers to fix a bug, given its description in a bug report, e.g. [3], [5], [15]. By comparing the performance of WhoseFault and these techniques in terms of both effectiveness and efficiency, I plan to find out the trade-offs of their use when both a failing execution and a bug report are available.

Finally, I also plan to create a visualization to provide more detailed information about why each developer was recommended for a bug. Currently, WhoseFault returns its developer recommendation for a bug in the form of a ranked list of developers. However, it does not provide much information about why each developer is recommended on each rank. By visualizing more detailed information about the recommendation, such as what areas of the code influenced the expertise of each developer or how recent their expertise with such areas of the code is, I plan to provide developer teams with a better understanding of the expertise that each developer provides for a bug.

Where is the bug located? As I mentioned in Section III, my research builds on top of fault-localization techniques. Therefore, I plan to study how different approaches to fault-localization adapt to answering the questions of *Who is the most suitable developer to fix a bug?*, and *When and why was a bug inserted?* I explain how I intend to perform such study in the sections corresponding to each one of these questions.

When was the bug inserted? Another next step for my research is to study how the combination of program and history analysis can help to identify when a bug was inserted. In previous research [17], I showed that history slices were highly successful at helping developers to find the origin of a code snippet. However, identifying when a bug was inserted presents a new challenge, since the area of the code that contains the bug is not normally well defined.

I plan to explore how different fault-localization techniques can be combined with History Slicing to support the task of finding the point in time at which a bug was inserted. I expect that showing developers the history slice for every line of code that is correlated with the bug will result in too much information for them to process. One possible alternative would be to show developers the history slice only for the lines of code that have the highest correlation with the bug. An additional product of this study might be a heuristic to fully automate the process of answering when a bug was inserted.

Why was the bug inserted? I also plan to study how my approach can support the process of understanding why a bug was inserted. I intend to perform this experiment in two parts. In the first part, I will study how effectively the visualization of history slices can support developers in understanding the rationale behind a code snippet that contains a bug. In this study, I intend to observe how effectively this approach can represent why the bug was inserted when the developer has perfect knowledge of the location of the bug. In the second part, I will study fault-localization techniques to learn how they can be combined with history slicing to support the process of answering why the bug was inserted when the location of the bug is unknown.

Improvements to the Visualization of History Slices. Finally, I plan to make general improvements to the visualization of History Slices. In past work [17], I presented a visualization for History Slices that allows infinite panning and zooming of a canvas that contains the snapshots of code within a history slice, with some of its meta-data. I plan to improve this visualization by making it more interactive. I believe that this visualization would help developers in their bug investigation if they could hide or show portions of a history slice at will. Another possible feature would be to provide the ability to select what meta-data will be shown, as well as to allow the display of additional meta-data, such as the set of files — or snapshots within them — that were committed together with a snapshot that belongs to the history slice. Finally, I also think that developers would benefit from the ability to interactively re-define the lines of code of interest for which to visualize their history slice. I will study the need for these improvements by comparing how efficiently developers can perform bug investigations when using the new visualization and when using the old visualization.

V. Publications

I am currently the first author of a workshop paper [20], a short paper [16], two full conference papers [17], [18], and a tool demo paper [19].

VI. Contributions

When my dissertation work is finished, it will provide multiple contributions. First, it will provide a fine-grained approach to analyze the history of source code, enabling the analysis of the history of any set of lines of code, contiguous or fragmented, from any number of files. This code-history analysis would be the first technique that allows to obtain the minimum complete history of a set of lines of code. Second, it will provide a visualization of the history of a set of lines of code that enables an efficient inspection of it by humans. Third, it will provide the first automated technique to recommend the most suitable developers to fix a bug without the need of a natural-language description of the bug. Additionally, this would be the first technique to provide a recommendation of where the bug is located as well as the recommendation of developers to fix it. Fourth, it will provide a comparative study of the accuracy of many different developer-to-bug assignment techniques. Fifth, it will provide a visualization that describes why each developer is recommended for fixing a bug. Sixth, it will provide a technique to support developers in finding out when a bug was inserted. Seventh, it will provide a technique to support developers in finding out why a bug was inserted.

Acknowledgements

This material is based upon work supported by the National Science Foundation under award CCF-1116943.

References

[1] R. Abreu, P. Zoeteweij, and A. van Gemund. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques*, pages 89 –98, 2007.

[2] J. Anvik, L. Hiew, and G. C. Murphy. Who should fix this bug? In *International Conference on Software Engineering*, pages 361–370, 2006.

[3] J. Anvik and G. C. Murphy. Reducing the Effort of Bug Report Triage: Recommenders for Development-Oriented Decisions. *ACM Transactions on Software Engineering and Methodology*, 20(3):10:1—10:35, 2011.

[4] T. Ball and S. G. Eick. Software visualization in the large. *Computer*, 29(4):33–43, 1996.

[5] D. Cubranic. Automatic Bug Triage using Text Categorization. In *International Conference on Software Engineering & Knowledge Engineering*, pages 92–97, 2004.

[6] L. Erlikh. Leveraging Legacy System Dollars for E-Business. *IT professional*, 2(3):17–23, 2000.

[7] D. J. Gilmore. Models of Debugging. *Acta Psychologica*, 78(1):151–172, 1991.

[8] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How Developers Drive Software Evolution. In *Workshop on Principles of Software Evolution*, 2005.

[9] L. Hattori, M. Lungu, and M. Lanza. Replaying past changes in multi-developer projects. In *Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, pages 13–22, 2010.

[10] R. Holmes and A. Begel. Deep Intellisense: a Tool for Rehydrating Evaporated Information. In *International Working Conference on Mining Software Repositories*, pages 23–26, 2008.

[11] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *International Conference on Software Engineering*, pages 467–477, 2002.

[12] A. J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *International Conference on Software Engineering*, pages 344–353, 2007.

[13] T. D. LaToza and B. A. Myers. Hard-to-Answer Questions about Code. In *Evaluation and Usability of Programming Languages and Tools*, pages 8:1–8:6, 2010.

[14] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable Statistical Bug Isolation. In *Programming Language Design and Implementation*, PLDI '05, pages 15–26.

[15] D. Matter, A. Kuhn, and O. Nierstrasz. Assigning Bug Reports using a Vocabulary-Based Expertise Model of Developers. *Mining of Software Repositories*, 2009.

[16] F. Servant and J. A. Jones. History Slicing. In *International Conference on Automated Software Engineering*, pages 452–455, 2011.

[17] F. Servant and J. A. Jones. History Slicing : Assisting Code-Evolution Tasks. In *Foundations of Software Engineering*, pages 43:1–43:11, 2012.

[18] F. Servant and J. A. Jones. WhoseFault: Automatic Developer-to-Fault Assignment through Fault Localization. In *International Conference on Software Engineering*, 2012.

[19] F. Servant and J. A. Jones. Chronos: Visualizing Slices of Source-Code History. In *IEEE Working Conference on Software Visualization*, 2013.

[20] F. Servant, J. A. Jones, and A. Van Der Hoek. CASI: Preventing Indirect Conflicts through a Live Visualization. In *Workshop on Cooperative and Human Aspects of Software Engineering*, pages 39–46, 2010.

[21] R. Shokripour, J. Anvik, Z. M. Kasirun, and S. Zamani. Why so Complicated? Simple Term Filtering and Weighting for Location-based Bug Report Assignment Recommendation. In *Mining of Software Repositories*, pages 2–11, 2013.

[22] I. Vessey. Expertise in Debugging Computer Programs: A Process Analysis. *International Journal of Man-Machine Studies*, 23(5):459–494, 1985.

[23] L. Voinea, A. Telea, and J. J. van Wijk. Cvsscan: Visualization of code evolution. In *ACM Symposium on Software visualization*, pages 47–56, 2005.

[24] A. von Mayrhauser and A. M. Vans. Program Understanding Behavior during Debugging of Large Scale Software. In *Workshop on Empirical Studies of Programmers*, pages 157–179, 1997.

[25] R. Wettel and M. Lanza. Visualizing software systems as cities. In *International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, 2007.