History Slicing

Francisco Servant
Department of Informatics
University of California, Irvine
Irvine, CA, U.S.A. 92697-3440
fservant@ics.uci.edu

James A. Jones
Department of Informatics
University of California, Irvine
Irvine, CA, U.S.A. 92697-3440
jajones@ics.uci.edu

***Abstract*—To perform a number of tasks such as inferring design rationale from past code changes or assessing developer expertise for a software feature or bug, the evolution of a set of lines of code can be assessed by mining software histories. However, determining the evolution of a set of lines of code is a manual and time consuming process. This paper presents a model of this process and an approach for automating it. We call this process *History Slicing*. We describe the process and options for generating a graph that links every line of code with its corresponding previous revision through the history of the software project. We then explain the method and options for utilizing this graph to determine the exact revisions that contain changes for the lines of interest and their exact position in each revision. Finally, we present some preliminary results which show initial evidence that our automated technique can be several orders of magnitude faster than the manual approach and require that developers examine up to two orders of magnitude less code in extracting such histories.**

I. Introduction

Information overload is a problem with which most software developers have to cope today. Therefore, many techniques have been developed in order to reduce the amount of information that developers need to process in order to carry out software engineering tasks. One example is program slicing [15], which reduces the search space of the lines of code that developers need to inspect when debugging to only those that are included inside the program slice. A program slice [16] represents the mental abstractions that developers create of interrelated lines of code, according to the flow of data, during the debugging process.

A different kind of task is that in which a developer is interested in exploring the history of a set of lines of code in order to understand why a code feature has evolved in a certain way. Even though Software Configuration Management (SCM) systems use a high diversity of version models [5], [10] and tools to operate with them [12], they still do not provide an automated way to perform this operation. Developers have to go through a tedious process of inspecting every past revision of each file that contains modifications to the lines of interest. Also, they need to determine, for each past revision, which lines of interest changed and, if so, identify which lines correspond to their previous contents.

We model in *history slicing* the process through which developers select the subset of the history of the software project which is relevant for the lines of interest. The *history slice* for a set of lines of code of interest (i.e., slicing criterion) contains all their equivalent lines of code in all the past revisions of the software project in which they were modified.

The goal of a history slice is to provide a reduced amount of information about the history of a set of lines of code. In the same way that program slicing selects the interesting areas of the code in the dimension of space, history slicing selects the interesting revisions of the code in the dimension of time, as well as selecting the appropriate lines of code in each of those revisions in the space dimension.

In this paper, we define the concept of a history slice, and provide an approach for its automatic computation, which not only reduces the information overload to relevant revisions (time) and lines of code (space), but also drastically reduces the time it takes for computation.

II. Applications

We envision history slicing being used for a high diversity of applications. First, we focus on history slicing being applied to program understanding. Some of the questions asked by novices might be answered by looking at how code was implemented in the past. Also, knowing exactly how many times some lines of code were modified and in which way should help novices understand how the code evolved into the current implementation.

History slicing allows slicing criteria to be composed of sparse lines of code that encompass multiple files, which also enables new applications of looking at the history of lines of code, which probably were not previously pursued manually because it was too time consuming. For example, when a bug may be fixed by modifying one of multiple methods, having the history slice for all of them should help to decide which method to modify. The assumption being that it is probably riskier to modify older code.

History slices may also be used as the input to be consumed by another algorithm, therefore allowing more complex analyses over the history of a set of lines of code. For example, we could gather statistics about how often a set of lines of code are changed, and by who. We could also find who has made the most modifications to a set of lines in order to identify that person as the *expert* in that block of code, which can inform who should be consulted when questions arise regarding that code.

Studying the evolution of lines which we know were fixed in the past will probably help us in the study of three things: how bugs are inserted, how bugs are fixed, and how bugs regress. This might help us identify patterns of changes that have a high potential to evolve into bugs. Also, we could gather patterns of how complex bugs have been fixed in the past.

Finally, history slicing can also enable an automated technique to compute a list of all the people who have made changes to sensitive areas of a project for legal matters.

III. Background

This section describes a series of background techniques upon which we rely for the construction of our approach. All of these techniques can be applied in different ways to building an abstract representation of the history of lines of code.

In order to track the history of individual lines of code, Zimmerman et al., proposed *annotation graphs* [19]. Annotation graphs are multipartite graphs in which each node represents a line of code. Edges are drawn between nodes to represent their evolution. Zimmerman et al. used the *annotate* feature of an SCM and *diff* to create the annotation graphs. However, *diff* often does not provide a precise or accurate line-to-line match between two revisions.

Diff performs a textual analysis and returns regions of difference between the two revisions of the file. These regions of difference are called *hunks* [13]. Zimmerman et al. link all lines in a modification hunk between the two revisions, but a finer grained analysis would be needed to identify specifically which lines in the older revision correspond to which lines in the newer revision. This problem has been previously referred to as *line mapping* by Williams and Spacco [17], and different authors have provided different solutions for it.

Canfora et al. [4] propose a two step approach. In the first step, they compare all ranges of code that exist only in the previous revision to all ranges of code that exist only in the newer revision to detect which ones should be mapped to each other. The second step involves computing the Levenshtein distance [9] between pairs of lines inside the ranges of code classified as changed. This way, they iteratively classify each pair of lines as changed or deleted/added, depending on whether their Levenshtein distance is less than or greater than a threshold, respectively.

A similar approach is followed by Williams and Spacco [18]. First, they use DiffJ [2] to obtain the differences between revisions. Second, they solve the line mapping problem by using the Kuhn-Munkres algorithm [8] to find the mapping with the best similarity for all lines. Finally, they mark as deleted/added those pairs which have a Levenshtein distance lower than a threshold.

Reiss presented a comparison of multiple methods for line mapping [14], which he defines as *source tracking*. While all these approaches to line mapping use purely syntactical comparisons of the source code, other researchers propose utilizing models of the program [3], [11] for performing the differencing of individual lines of code. More sophisticated techniques even allow the detection of moved code [7], while others are designed for specific languages or domains [6].

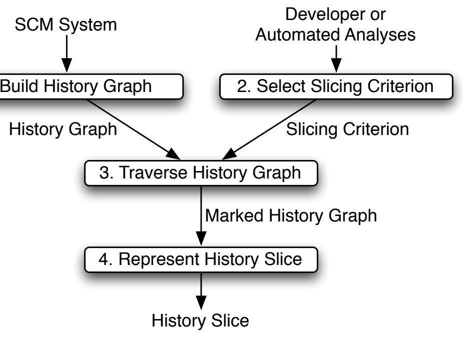

SCM System
Developer or Automated Analyses
1. Build History Graph
2. Select Slicing Criterion
History Graph
Slicing Criterion
3. Traverse History Graph
Marked History Graph
4. Represent History Slice
History Slice

Fig. 1: Approach to automatically compute a history slice. Each step can be customized in a variety of ways.

IV. Approach

Utilizing the techniques and models presented in Section III, we describe the approach to automatically computing history slices. The process for its automatic computation leverages the insights gained from the conceptualization of the annotation graph as well as the techniques for computing line mapping.

The process, which is parameterizable, is depicted in Figure 1. We describe each step of the process in turn.

A. Build History Graph

The first step of our approach involves building a *history graph* to represent the complete history of every line of code. Like annotation graphs, a history graph is a multipartite graph in which each part represents a revision of a file in the source code. Each node in a part represents a line of code in that specific revision of the file.

Unlike annotation graphs, in history graphs each node is linked to only one node in the previous part and/or only one node in the next part. Linkings of one node to many are not permitted. Figure 2a shows a simple example of a history graph. We represent modified lines with black nodes and unmodified lines with white nodes.

In order to assign the edges between nodes of two consecutive revisions, we need to use a line mapping technique. The decision about which line mapping technique to use will depend on the application for which we are building the history graph. For instance, the technique proposed by Williams and Spacco [17] ignores non-executable lines in the history, while the technique proposed by Canfora et al. [4] allows the tracking of annotations, comments and other non-executable code. Also, the technique by Apiwattanapong et al. [3] detects movements of methods between files.

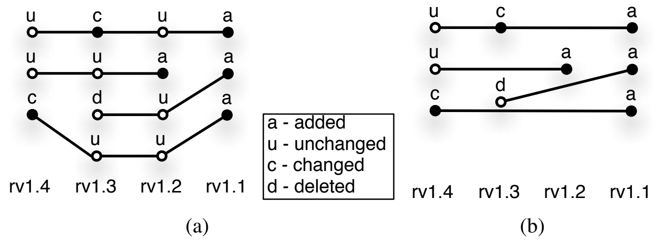
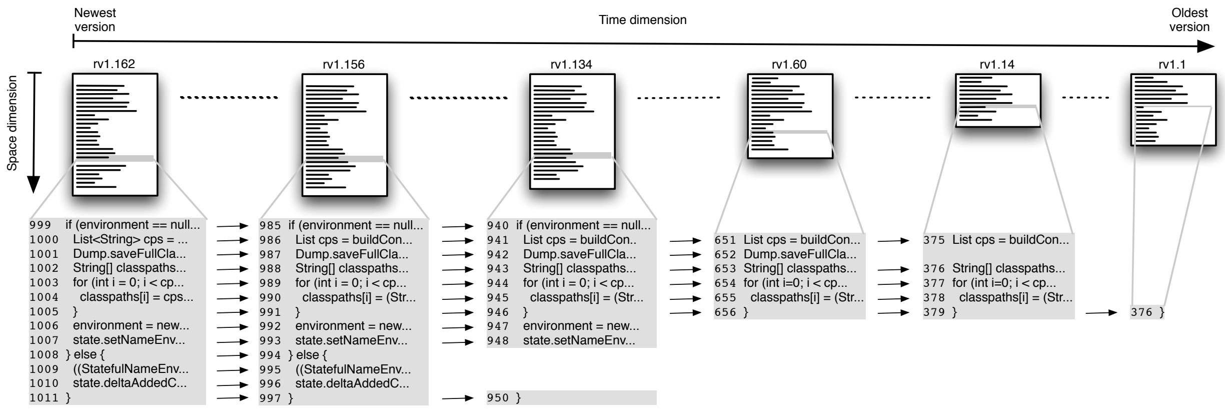


Fig. 2: History Graph (a). Collapsed History Graph (b).



```
999  if (environment == null...
1000   List<String> cps = ...
1001   Dump.saveFullCla...
1002   String[] classpaths...
1003   for (int i = 0; i < cp...
1004     classpaths[i] = cps...
1005   }
1006   environment = new...
1007   state.setNameEnv...
1008 } else {
1009   ((StatefulNameEnv...
1010   state.deltaAddedC...
1011 }
```

```
985 if (environment == null...
986   List cps = buildCon...
987   Dump.saveFullCla...
988   String[] classpaths...
989   for (int i = 0; i < cp...
990     classpaths[i] = (Str...
991   }
992   environment = new...
993   state.setNameEnv...
994 } else {
995   ((StatefulNameEnv...
996   state.deltaAddedC...
997 }
```

```
940 if (environment == null...
941   List cps = buildCon..
942   Dump.saveFullCla...
943   String[] classpaths...
944   for (int i = 0; i < cp...
945     classpaths[i] = (Str...
946   }
947   environment = new...
948   state.setNameEnv...

950 }
```

```
651 List cps = buildCon...
652 Dump.saveFullCla...
653 String[] classpaths...
654 for (int i=0; i < cp...
655   classpaths[i] = (Str...
656 }
```

```
375 List cps = buildCon...

376 String[] classpaths...
377 for (int i=0; i < cp...
378   classpaths[i] = (Str...
379 }
```

```
376 }
```

Fig. 3: History of lines 999–1011 from file *AjBuildManager.java*.

During the process of line mapping, each node is also labeled according to the action that produced it for its particular revision, as can be seen in Figure 2a. These labels are the mechanism that enables the detection of which revisions should be part of the history slice when traversing the history graph. However, other labels are also allowed in both nodes and edges for different applications.

Finally, history graphs may also be *collapsed*, by joining only changed nodes, i.e., skipping all unchanged nodes. Figure 2b depicts the collapsed version of Figure 2a. A collapsed history graph improves the efficiency of the computation of history slices. However, it also discards some information, like the line number for unchanged lines. This choice must be made with regard to the application and its need for such potentially discarded information.

B. Select Slicing Criterion

The next step for computing a history slice is selecting the slicing criterion, which represents the set of lines of interest for which we intend to obtain the history. The slicing criterion may be selected manually by a developer, or automatically by a client analysis.

For example, the slicing criterion may be chosen as the lines that were executed by a failing test case in order to understand the failure, as the most suspicious lines according to a semi-automated statistical fault-localization technique, or all lines committed by a particular developer within a particular window of time for potentially performing forensics.

C. Traverse History Graph

Once the slicing criterion is selected, we follow the *history path* of every line in it towards their older revisions. Each set of interconnected nodes in the history graph is a *history path*, and it represents the history of a line of code. The history of a line of code starts when it is added and finishes when it is deleted or it belongs to the newest revision of the file. We can compute two kinds of slices, depending on which nodes we consider part of the history slice:

minimal: visited modified nodes.
extended: visited modified and unmodified nodes, for all revisions in which there is at least one modified node.

Computing *minimal* history slices will be more efficient when traversing a collapsed history graph. If the history graph is not collapsed, we still visit unchanged nodes even though we do not add them to the history slice. Note that, computing *extended* history slices, where we add all nodes (modified or unmodified) within a modified revision, is only possible with uncollapsed history graphs.

In the process of traversing the history graph, nodes are marked for inclusion into the slice according to the choice of a minimal or extended history slice. Marked nodes will be considered as part of the final history slice.

D. Represent History Slice

The final step of our approach is representing the history slice in a suitable format for its purpose. Different applications of the history slice will require different representations. For example, a developer who is directly inspecting a history slice to understand the evolution of certain features will require a visual interface that displays whole files for each revision included in the history slice, along with highlights and mappings in the changed lines in each revision. Alternatively, if the developer's goal is merely to gather statistics of the number of changes made on a set of lines, an abstract data structure of the history slice is appropriate.

In summary, each of these four steps for computing a history slice can be customized in a variety of ways. Each choice in the approach may be influenced by the intended utility of the history slice, and may affect the computed slice.

V. Motivating Example

This section introduces a motivating example which demonstrates the application of history slicing. In this example scenario, a developer is interested in understanding the set of changes that led to a particular set of lines of code. Concretely, she is examining lines 999–1011 in revision 1.162 of file *AjBuildManager.java* in the AspectJ [1] open source project.

For this goal, the developer needs to find all the different modifications that have been performed over those lines of code through the history of the software project. This process involves four key steps:

TABLE I: Preliminary Results

Slicing Criterion Size	Approach	Revisions	Total Lines	Seconds
10	Manual	5	272	1,919.00
10	Automatic	5	50	0.06
20	Manual	2	1,150	623.00
20	Automatic	2	40	0.17
50	Manual	4	365	1,637.00
50	Automatic	4	200	0.44
18 (non-contiguous)	Manual	2	388	1,020.00
18 (non-contiguous)	Automatic	2	36	0.04

1) Retrieve the previous revision *r* of the file.
2) Find inside revision *r* the lines corresponding to the lines of interest.
3) Check the content of those lines and see if they were modified.
4) If they were modified, save them. Loop back to Step 1.

If the developer is familiar with SCM functionality, she will use capabilities provided by many of them, such as *annotate*.[1] *Annotate* displays, for each line of a file, the revision in which it obtained its current contents.

In Step 1, she could run *annotate*, then manually find in its output the lines of interest, and note down the highest revision r in which any of them was modified. Then, in Step 2, she would have to retrieve $r-1$ and manually find in it the previous states of the lines of interest. A shortcut for this step would be to run *diff* over r and $r-1$, although one would still have to manually inspect the contents of *diff*. In Step 3, she would manually check whether each of them was modified or not, and record that information. Finally, our developer would run *annotate* over $r-1$ in order to obtain the next revision over which to iterate, going back to Step 1.

Despite this partial automation, the four steps still involve a high amount of interaction overhead with multiple tools, making much of the effort quite tedious and manual.

In contrast, an implementation of history slicing will provide our developer with the contents of all the lines of interest in every revision in which they were modified in less than one second and by running just one command. This result is pictured in Figure 3 as an *extended* history slice.

VI. Preliminary Results and Future Work

We implemented a prototype tool to perform history slicing and ran a pilot experiment to estimate the savings provided by it. We computed the history slice both manually and automatically for some sample slicing criteria in the AspectJ [1] open source project and measured the number of lines of code that had to be inspected along with the time that was required to compute them.

For the manual approach, we used `cvs annotate` and `cvs diff`. We also took note of roughly how many lines of code we needed to inspect and measured the time that we invested in the process. Table I shows the results of this experiment for slicing criteria of 10, 20 and 50 contiguous lines, and a slicing criterion of 18 non-contiguous lines.

[1]Example *annotate* commands include `cvs annotate`, `git annotate`, and `svn blame`.

These results show evidence that the automation of history slicing can provide an improvement over a manual approach up to two orders of magnitude in the lines needed to be processed by the developer. Additionally, the automation requires several orders of magnitude less time to compute.

In the future, we intend conduct an experiment with a higher number of samples and with different implementations to better evaluate the savings provided by history slicing, as well as exploring how useful it could be when applied to other applications.

VII. Acknowledgments

We thank the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under award CCF-1116943, and by a Google Faculty Research Award.

References

[1] Aspectj. http://www.eclipse.org/aspectj/.
[2] Diffj. http://www.incava.org/projects/java/diffj/.
[3] T. Apiwattanapong, A. Orso, and M. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *Automated Software Engineering*, 14:3–36, 2007.
[4] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, pages 14–21, Washington, DC, USA, 2007. IEEE Computer Society.
[5] R. Conradi and B. Westfechtel. Version models for software configuration management. *ACM Computing Surveys*, 30:232–282, June 1998.
[6] A. Duley, C. Spandikow, and M. Kim. A program differencing algorithm for verilog HDL. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 477–486, New York, NY, USA, 2010. ACM.
[7] J. Hunt and W. Tichy. Extensible language-aware merging. *IEEE International Conference on Software Maintenance*, 0:511–520, 2002.
[8] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.
[9] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, pages 707–710, Feb. 1966.
[10] B. Magnusson, U. Asklund, and S. Minör. Fine-grained revision control for collaborative software development. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '93, pages 33–41, New York, NY, USA, 1993. ACM.
[11] J. I. Maletic and M. L. Collard. Supporting source code difference analysis. *IEEE International Conference on Software Maintenance*, 0:210–219, 2004.
[12] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28:449–462, 2002.
[13] W. Miller and E. W. Myers. A file comparison program. *Software: Practice and Experience*, 15(11):1025–1040, 1985.
[14] S. P. Reiss. Tracking source locations. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 11–20, New York, NY, USA, 2008. ACM.
[15] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
[16] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25:446–452, July 1982.
[17] C. Williams and J. Spacco. Szz revisited: verifying when changes induce fixes. In *Proceedings of the 2008 Workshop on Defects in large software systems*, DEFECTS '08, pages 32–36, New York, NY, USA, 2008. ACM.
[18] C. C. Williams and J. W. Spacco. Branching and merging in the repository. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 19–22, New York, NY, USA, 2008. ACM.
[19] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr. Mining version archives for co-changed lines. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 72–75, New York, NY, USA, 2006. ACM.