History Slicing: Assisting Code-Evolution Tasks

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

Many software-engineering tasks require developers to understand the history and evolution of source code. However, today's software-development techniques and tools are not well suited for the easy and efficient procurement of such information. In this paper, we present an approach called *history slicing* that can automatically identify a minimal number of code modifications, across any number of revisions, for any arbitrary segment of source code at fine granularity. We also present our implementation of history slicing, CHRONOS, that includes a novel visualization of the entire evolution for the code of interest. We provide two experiments: one experiment automatically computes 16,000 history slices to determine the benefit brought by various levels of automation, and another experiment that assesses the practical implications of history slicing for actual developers using the technique for actual software-maintenance tasks that involve code evolution. The experiments show that history slicing offered drastic improvements over the conventional techniques in three ways: (1) the amount of information needed to be examined and traced by developers was reduced by up to three orders of magnitude; (2) the correctness of developers attempting to solve software-maintenance tasks was more than doubled; and (3) the time to completion of these software-maintenance tasks was almost halved.

Categories and Subject Descriptors. D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement — *version control*; D.2.9 [Software Engineering]: Management — *software configuration management*

Keywords. Mining Software Repositories, Program Comprehension, Software Evolution, Software Visualization

1. INTRODUCTION

Software developers frequently face hard-to-answer questions about source code. A study by LaToza et al. [22] found that answering questions about rationale is a serious problem for developers. Additionally, Ko et al. [18] found that questions about why code had been implemented in a certain way were among the most time consuming to answer. A different study by LaToza and Myers [21] found that some of the hard-to-answer questions seek knowledge about the history of code — like when, how, by whom, and why some code was changed or inserted. Answering questions about the history of source code also serves as a strategy for understanding rationale because it potentially provides context and motivation. Given the frequency and difficulty of such tasks, mechanisms that efficiently support them can significantly and positively affect software-developer performance.

Existing software-configuration management (SCM) systems, such as CVS [10], Subversion [1] and Git [31], allow users to track and query revisions and investigate differences *per file*. However, studies have found that developer tasks require information about the history at the level of a block of code and that developers require information about a series of changes [15, 21].

To address this need, existing SCM systems provide features, such as "`annotate`," that allow a developer to view, for each line of a file, the last revision in which it was modified. Such features address the need for line-level querying, but do not enable for efficient inspection of a sequence of changes for an arbitrary block of code. While such querying can be performed with these tools, they require significant manual effort by the developer to track a line of interest, track and query each of its past revisions, and then to repeat and synthesize the results for each line of interest.

To better support such querying and exploration of the evolution of source code, we created a technique called *history slicing* [30] that automatically tracks the lineage of each line of code in an SCM system and efficiently enables developer querying and exploration of such evolution. 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. 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 relevant areas of the code in the dimension of space, history slicing selects both the relevant revisions of the code in the dimension of time and the appropriate lines of code in each of those revisions in the dimension of space.

History slicing addresses the limitations of traditional SCM systems by providing the ability to (1) query at the line level, any arbitrary set of lines, across any set of files, and (2) view the minimum complete evolution of those lines, along

SIGSOFT'12/FSE-20, November 11–16, 2012, Cary, North Carolina, USA.

with their tracing among revisions. A third, emergent, benefit of such a technique is that the results reveal interesting patterns that indicate characteristics of the code evolution that were previously obscured (such as co-evolution of two method bodies).

In this paper, we provide three main contributions:

- A detailed description of the concept of *history slicing*, which enables developers to generate a *history slice* for any arbitrary set of lines of code (i.e., contiguous or fragmented, and in a single file or across any number of files). We provide a definition of history slicing, a detailed description of how history slicing is performed with today's SCM systems, and summarize our approach to automating it.
- A description of CHRONOS, our implementation of history slicing, which includes the model generation, analysis, and a novel user interface that provides a visualization of any history slice that may include many files and revisions. The visualization also supports developers to visually recognize patterns in the evolution of the code.
- An extensive evaluation consisting of two experiments. The first experiment involves the computation of 16,000 history-slicing tasks to evaluate the degree to which our technique may alleviate the information overload that would be present with existing tools. The second experiment is a user study of actual developers and includes 48 history-slicing tasks to evaluate the real-world impact of our technique.

2. MOTIVATION

A study by LaToza et al. [21] observed developers at their workplace and found that they frequently ask questions about the history of source code. They also found that developers wanted to know both the latest changes and the entire history at the level of a code snippet. In addition, two different studies by LaToza et al. [22] and Ko et al. [18] found that developers often asked questions about why a snippet of source code had been implemented in a specific way. They also found that resolving such questions about the rationale of source code was highly time-consuming. Methods that can provide the code's creation and evolutionary history may help developers to understand such design-rationale motivations and the context in which they were made.

The exploration of the entire history of a set of lines of code can be applied in a wide variety of scenarios. Consider an example scenario in which a developer wants to find an earlier implementation of a specific piece of code, for example, a loop body. In this example situation, our developer implemented multiple versions of the functionality performed by that loop throughout the history of the project. At some point, she realizes that a specific earlier implementation better suits the needs of the program, but she can't remember which revision contains the desired implementation. In this situation, finding which revisions contain changes for that specific loop body and the corresponding lines to that piece of code in each of those revisions can be a tedious task if performed with today's SCM tools, unless log messages were quite explicit and detailed (and, even still, there would likely be an overabundance of log messages to read to find the specific revision).

In another example scenario, a project manager may need to know all of the developers who ever modified a specific segment of the source code, for example, two methods that interact to form a specific functionality. LaToza et al. found that developers asked questions about the authorship of code and their teammates [21], and a number of researchers proposed approaches for mapping "expert" developers to components of source code [*e.g.,* 11, 13, 17, 25, 26, 29]. However, if a developer wanted to perform a detailed exploration through the history of when, how and why each developer made changes to that segment of the source code, she would again need to perform a tedious task with today's SCM tools.

In a final example scenario, developers may want to explore the parallel history of multiple segments of source code in order to find out whether and when they were modified together. Zimmermann et al. found that multiple software artifacts being committed together to the source code repository is a signal of these entities being dependent on each other. This dependency is the definition for *evolutionary coupling* [34]. Another motivation for tracking commits across multiple arbitrary segments of code is to help assess code clone risks. Bakota [3] found that segments of code that are identified as code clones often were modified together, and that violations of this pattern can indicate possible problems. In each of these cases, extracting and presenting a detailed exploration of the history of two pieces of code (for example, two small methods in two large files) would be a highly repetitive and time-consuming task if performed with today's SCM tools.

A common characteristic of each of these three scenarios is that they are difficult to answer with today's SCM tools, for three reasons: (1) they require deep exploration of the history (i.e., not simply a pairwise *diff*); (2) they require the exploration of multiple sets of lines of code, potentially across multiple files; and (3) they require the recognition of patterns and characteristics across potentially long epochs of the projects' life. We posit that a technique that can automatically address and assist with these three code-evolution task characteristics can benefit developers attempting to solve such tasks.

We conjecture that for all three of these scenarios, developers often give up on answering such code-evolution questions. That is, because these tasks are exorbitantly time consuming to perform with today's SCM tools, the possibility for successfully and accurately answering them is considered infeasible. In this paper, we first present history slicing as a means to assist such code-evolution tasks, and secondly, study whether and how efficiently history slicing allows their performance.

3. SLICING OF HISTORY

History slicing models the process that developers have to follow in order to obtain the whole history of a set of lines of code. To display a real-world example, Figure 1 represents the whole history of lines 999–1011 up to revision 1.162 of file *AjBuildManager.java* of the AspectJ [9] open-source project.

Source code files are normally committed multiple times, generating multiple revisions. However, not all revisions contain changes to all lines of code. In our example, out of a total of 162 revisions for file *AjBuildManager.java*, only revisions 1.156, 1.134, 1.60, 1.14 and 1.1 contain changes to the lines of interest.

We define *lines of interest* as the lines of code whose history a developer wants to explore. Each line of interest will

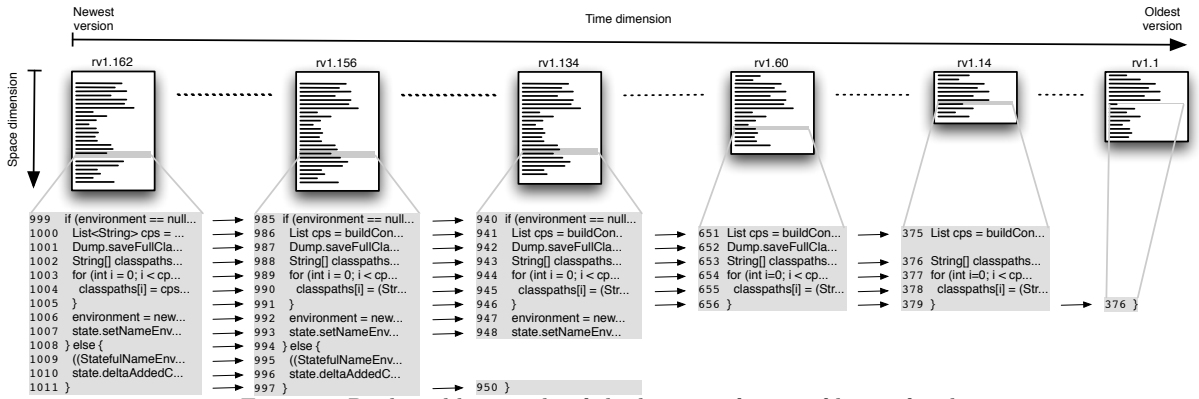

Figure 1: Real world example of the history of a set of lines of code.

also be characterized by the *revision of interest* of the *file of interest* inside which it is contained. These lines of interest may be few (e.g., a single line or a basic block of code) or numerous (e.g., a method or class); contiguous (e.g., a method) or fragmented (e.g., a dynamic slice); and within a single file (e.g., a method) or across multiple files (e.g., a test case's statement-coverage set). Such sets of lines of interest are akin to a "slicing criterion" for program slicing.

We also define *snapshot* as the set of lines of code in a particular version that correlate, either directly or transitively, to the original lines of interest; i.e., a snapshot represents a previous state of the lines of interest. Thus, the snapshot of lines of interest 999–1011 for revision 1.134 is represented by lines 940–948 and 950. The line numbers of a snapshot are different in each revision, since an undetermined number of lines will be added and/or deleted before and after it.

The product of history slicing is a *history slice*. The *history slice* for a set of lines of interest contains all their snapshots in all the past revisions in which they were modified.

Slicing history with today's SCM systems is not trivial because they do not provide an automated way of obtaining the whole history of an arbitrary set of lines of code. Instead, four subtasks are required:

1. Retrieve the previous revision r of a file.
2. Find inside revision r which lines correspond to the lines of interest.
3. Check the contents of those lines and identify whether they were modified.
4. If they were modified, save them. Return to Step 1 until all history is explored.

Depending on their level of expertise with today's SCM systems, developers may perform these steps by using a fully manual approach or a more advanced conventionally assisted approach, as well as a spectrum of approaches in between.

3.1 Manual, Naive Slicing of History

We present this approach, (1) because it is the most straightforward solution and one that is likely to be employed by more novice developers who are unaware of advanced features of SCM systems and (2) because it clearly demonstrates the challenges brought by each of the four subtasks.

To determine the previous revision to the revision of interest (Step 1), a developer could manually retrieve every individual revision of the file(s) of interest from the SCM system. Of course, this requires an exorbitant amount of unnecessary work because only a subset of those revisions contain changes for the lines of interest. We can say that this inefficiency affects the search in the dimension of *time* (as indicated by the "Time dimension" arrow in Figure 1).

To determine the snapshot in a prior revision (Step 2), the developer would, in the worst case, manually inspect the full contents of the files in order to find the position of the snapshot. If the files are large, this step also involves a high amount of unnecessary work. We can say that this inefficiency affects the search in the dimension of *space* (as indicated by the "Space dimension" arrow in Figure 1).

To determine whether there are changes in the snapshot (Step 3), the developer would compare each pair of snapshots. This comparison may not be straightforward when the snapshots are large, disjoint, or contain subtle and hard-to-notice changes.

Finally, to keep track of the entire history of the lines of interest (Step 4), the developer would need to keep a log of all the snapshots for all the files. This log could be kept in a text editor, requiring application and context switching, and thereby imposing an additional overhead on the process.

For all of these reasons, following such a naive approach can be extremely time consuming. We want to point out that while this manual approach may often be an unrealistic scenario, developers are likely to follow it in cases when they don't have much expertise with SCM systems or in cases where the task only involves a few changes in a few lines in a single file. Nonetheless, we present the manual approach mostly for illustrative purposes.

3.2 Conventionally Assisted Slicing of History

Developers may use capabilities of SCM systems to support the previously mentioned four steps for slicing history.

In order to determine the previous, modified revision (Step 1), the developer can utilize the *annotate/blame* feature on the revision of interest. *Annotate* will return, for every line in the file, the latest revision to modify it. Then, the developer would need to manually find the snapshot inside *annotate*'s output. Since *annotate* will return different *last revision* for different lines, the developer would select the most recent (highest) revision r in which any of the lines in the snapshot was modified. This automation would resolve the inefficiency in the dimension of *time* of the manual approach.

To find the snapshot inside revision r (Step 2), the developer may use the *Ctrl+F* functionality in a text editor with the contents of the revision in order to navigate to the area of the file that contains the snapshot. However, this option might still involve multiple attempts. Since developers would be searching in an older revision, they might be searching for words that were different then. Another short-

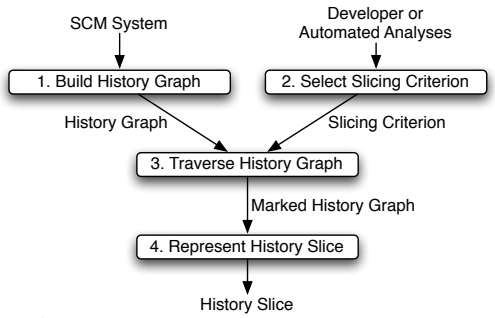

Figure 2: Approach to automatically compute history slices.

cut for this step would be to run *diff* over r and $r-1$, but the developer would still have to manually inspect the contents of its output. Additionally, for both *Ctrl+F* and *diff*, in the case of fragmented snapshots, each line of the snapshot would have to be found individually and manually. While this automation mitigates the inefficiency in the dimension of *space* of the manual approach, it is hard to predict by how much. In a worst case scenario, the developer would still need to inspect the full contents of the file.

Determining whether the snapshots are different (Step 3), will be straight-forward, since *annotate* will directly point developers to only those revisions which do contain changes to the snapshot. However, some degree of inefficiency would be introduced in this step if the specific implementation of *annotate* detects false modifications, such as modifications in white space. In such case, developers would have to inspect some unnecessary revisions.

Keeping track of the history slice (Step 4), is still performed exactly as in the manual approach.

In summary, this approach assists the process of finding the history of a set of lines of code. However, despite some automation, it is still a highly manual process of performing multiple commands and correlating and interpreting their output. In the extreme case where the lines of interest are fragmented and scattered among multiple files, this process can be extremely time consuming, even with such assistance.

4. AUTOMATION OF HISTORY SLICING

Our approach to automating history slicing involves multiple steps, each of which can be parameterized in a number of ways. The overall process is depicted in Figure 2.

Step 1: Build History Graph. In the first step, we create a *history graph*, which contains the history of each line of code. A history graph is a multipartite graph where each part represents a revision of a file. Inside a part, each node represents a line of code in that revision. Each node is linked to only one node in the previous part and/or only one node in the following part. Additionally, each node can be labeled to store additional metadata, such as authorship, time stamps, and log messages. Figure 3 shows a simple example of a history graph. In this figure, each node contains a label, which describes the operation that produced each line in each revision. In general, history graphs are similar to annotation graphs [33]. Unlike the "modification hunk"-granularity of annotation graphs, history graphs require a one-to-one node mapping between revisions.

The links between nodes are assigned by applying a line-mapping (i.e., fine-grained program-differencing) technique. Because existing line-mapping techniques vary in their power and flexibility, our framework allows the choice of line mapping to be customized according to the tasks at hand. For example, a simple plain-text differencing technique may be desirable if following the history of non-executable code, such as comments or XML metadata. Alternatively, a differencing technique based on the abstract syntax tree may be desirable to track changes that change the structure of the executable code.

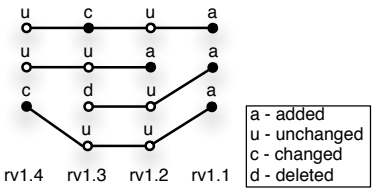

Figure 3: History Graph.

Once the history graph is built, it can be used for the computation of any history slice. Building the history graph is a one-time expense and, as the code history grows, it can be extended with minimal effort.

Step 2: Select Slicing Criterion. In the second step of our approach, we need to select a slicing criterion, which contains a set of lines of interest from a specific revision of a program. The slicing criterion may contain any set of lines of code, contiguous or fragmented, from any file and revision or combination of files and revisions. This freedom is provided as a result of the fine grain in which the history graph stores the evolution of the source code.

Step 3: Traverse History Graph. In the third step, the approach traverses the history graph, starting from the nodes that represent the slicing criterion. In this traversal, the approach visits all the nodes included in the *history path* of all the lines of code in the slicing criterion. A *history path* for a line of code includes all the changes to it, from the moment that it was initially conceived, until it is deleted or is part of the newest revision of the file. In the process of traversing a history path, nodes are marked for inclusion into the slice according to the choice of a *minimal* history slice (visited modified nodes) or *extended* history slice (visited modified and unmodified nodes, for all revisions in which there is at least one modified node). Marked nodes will be considered as part of the final history slice.

Step 4: Represent History Slice. The fourth and final step of our approach is the visualization of the history slice. This step is also performed differently, depending on the task being solved. Some scenarios will require minimal information, such as statistics on the number of changes made on a set of lines, while other scenarios will require more context and therefore will need to display the history slice together with the lines that surround the snapshots.

In summary, all four steps can be performed in multiple ways. The decision about how to implement each of them will be influenced by the intended use of the history slice. This decision will also affect the contents of the computed history slice. In any case, a history slice will contain a minimal, but complete amount of information about the history of a set of lines of code.

5. CHRONOS

In order to facilitate experimentation, we instantiated the automated history-slicing framework that we described in Section 4 in a tool, CHRONOS. For each step of the framework, we describe the choices made in its implementation.

Step 1: Build History Graph. We implemented a component of CHRONOS that queries an SCM system and builds a history graph. This component first obtains the list of all files stored in the repository. Next, for each file, it queries its list of revisions. Then, for each revision, it retrieves its contents as well as its *diff* with the previous revision. For each line of code for a particular revision, it creates a node.

The final step for the graph builder is to create the correlation edges between consecutive revisions. This step requires a line-mapping technique [*e.g.,* 2, 5, 6, 7, 8, 16, 24, 27, 32, 33]. We incorporated many of the lessons and methods of such techniques to form our own line-mapping technique. In the same way as Chen et al. [6], our technique performs a first phase that utilizes the SCM system's *diff* functionality to determine added, deleted, and changed individual lines (and labels their nodes as such), and for those lines that *diff* marks as *blocks* of changes (i.e., modification hunks), a second phase is utilized to provide a line-to-line mapping. Much like Williams and Spacco [32], we used the Kuhn-Munkres combinatorial optimization algorithm [19], coupled with a computed Levenshtein distance [23], to compute an optimal mapping among lines in such change blocks.

The final result of graph-building component is a history graph that contains the complete history of any line of the project. The graph is stored in a relational database that can be queried both directly and through a supporting API.

Step 2: Select Slicing Criterion. We implemented the slicing-criterion selector as an Eclipse plugin. A developer can open any revision of any file, select any set of lines, and right-click to reveal a contextual menu that includes an option to "Add to History Slicing Criterion." As such, any set of lines, contiguous or fragmented, across any number of files or revisions, can be added to the slicing criterion.

Step 3: Traverse History Graph. Once the history graph is built (Step 1) and the slicing criterion has been specified (Step 2), the history slice can be computed. The history-slicer component is initiated through the Eclipse plugin (or, alternatively through the API). The history slicer interacts with the relational database that contains the history graph through SQL queries. The history slicer traverses the history graph from the most recent revision of each line in the slicing criterion, and traces their evolution going backward in time. Each revision that contains changes is recorded and the snapshot at that revision captured.

Step 4: Represent History Slice. With the history slice computed, it can be presented to the developers for them to explore and interpret. For this component, we implemented an interactive visualization of the history slice. The visualization is a zoom-able canvas that depicts all snapshots for all lines in the slicing criterion, with mappings between them. In addition, timelines are presented to show proportionally, in time, when changes were made.

Figure 4 is a screenshot of the history-slice visualization in CHRONOS. The history slice that is being visualized was computed from a slicing criterion that spanned two files. The top two gray bars (labeled as (1) in the figure) represent a global display of the timelines for these two files. Just above them, a gray ruler marks the months that the timelines encompass. Inside each of these timelines, each blue mark represents a revision of that file that includes changes relevant to the original slicing criterion, at the position in the timeline that represents the time of the revision.

The top two timelines are placed adjacent to allow the user to view correlations between changes and their times, and thus potentially recognize patterns. In contrast, the timelines are repeated below (labeled (2) and (3)), with call-out lines for each modification revision that lead to the full source code of the snapshot.

Each snapshot is colored to indicate which lines in the snapshot were modified in that revision: blue lines were changed in that revision and gray lines stayed unchanged. In addition, each snapshot is annotated with metadata, such as the revision number, date, and author, colored in green.

The entire visualization is zoom-able and pan-able allowing the user to both (1) see a high-level view of all changes and revisions, and to potentially recognize patterns in the changes, and (2) to inspect and explore the fine details of what was changed, and when. We depict a zoomed-in screenshot in the overlaid call-out (labeled (4)).

6. EVALUATION

Utilizing CHRONOS, we conducted two experiments to determine the merits of automating the slicing of history. The first experiment involves the construction of 16,000 history slices using various levels of automation. The goal of the first experiment is to determine the degree to which extraneous information can be minimized by the use of history-slicing automation. The second experiment involves the study of 24 developers that use tools that provide different levels of automation assistance for history slicing. The goal of the second experiment is to determine the practical benefits brought to actual developers for performing actual code-evolution tasks.

To evaluate the effectiveness of history slicing, we define the following research questions:

- **RQ1**: How much does the automation of history slicing reduce the problem space in terms of the total number of lines of code needing to be examined?
- **RQ2**: How much does the size of the slicing criterion affect the problem space reduction (in both the time and space dimensions)?
- **RQ3**: What is its practical benefit to developers performing code-evolution tasks?

In both experiments, we use the AspectJ open-source project [9] as the code base on which we perform history slicing. AspectJ is an aspect-oriented extension to the Java programming language. It consists of over 75,000 lines of code and has been in active development for more than eight years.

6.1 Experiment 1

To answer research questions RQ1 and RQ2, we randomly produced 1,000 slicing criteria at each of four sizes and computed their history slices using four parameterizations of the history-slicing approach (discussed in Sections 3 and 4), for a total of 4,000 different slicing criteria, and 16,000 computed history slices. The goal of computing history slices with four parameterizations of the history-slicing approach is to approximate the benefits brought by the range of conventional tools and our automated history slicing approach (Research Question RQ1). The goal of computing history slices with four, differently sized slicing criteria was to determine how the size of the slicing criteria affects the degree of benefit from automation (Research Question RQ2).

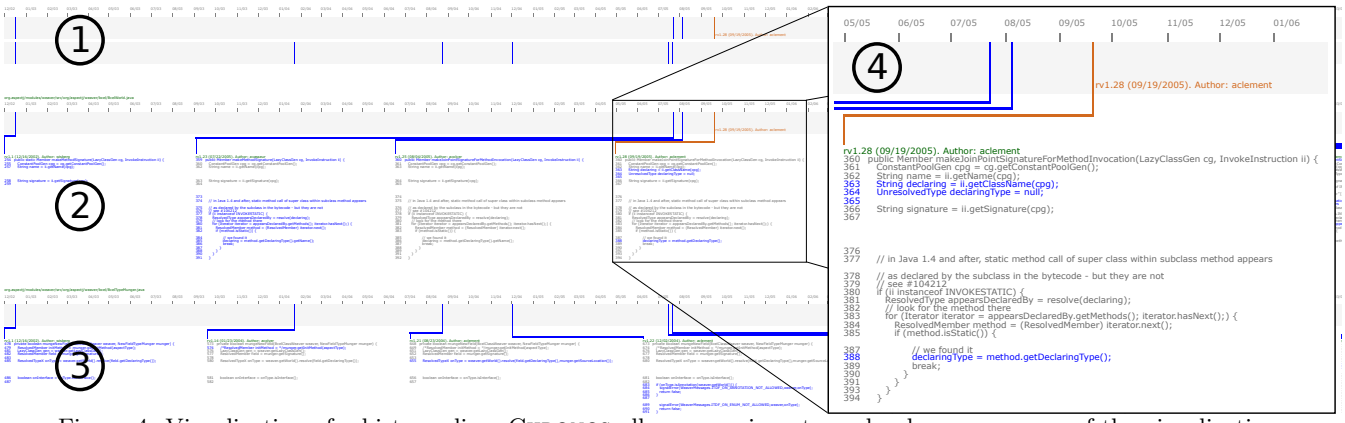

Figure 4: Visualization of a history slice. CHRONOS allows zooming at any level over any area of the visualization.

6.1.1 Experimental Variables

We experimented using two *independent variables*, technique treatments and slicing-criteria size, and evaluated their benefit using three *dependent variables*, number of revisions, average number of lines, and total number of lines. Each variable will be defined and motivated in turn.

Independent Variable 1: Technique Treatment. In order to determine the degree to which the automation of history slicing can reduce the problem space for code-evolution tasks (RQ1), we parameterized its computation in the following four ways:

Treatment 1: Naive. Every revision is examined to determine where and if any changes were made that are relevant to the slicing criterion. This parameterization approximates the effort that would be necessary without any automation at all, and is presented as a baseline.

Treatment 2: Conventionally Assisted. Only the revisions required by the *annotate* tool are examined, and the snapshots are found without assistance. For fragmented slicing criteria, such as a dynamic program slice, each line would be found individually and manually. Then, we consider that all lines in each identified revision are manually inspected.

Treatment 3: History Slicing with Context. Only the revisions identified by the automated history slicing approach are examined, and only the lines inside the snapshots are examined. In each such revision, at least one line changed in the history of the slicing criterion. In addition, all lines that have an evolutionary relationship with the slicing criterion are included (as "context") to provide a full view of each snapshot at each relevant revision.

Treatment 4: History Slicing without Context. Only the revisions identified by the history slicing approach are examined, and only the lines that changed in the snapshot are examined. For this treatment, the unchanged-yet-correlated code is omitted. Such a technique may be useful for subsequent automated analyses that utilize the output of history slicing.

Independent Variable 2: Slicing Criterion Size. In order to determine the degree to which the initial slicing criterion size affects the problem space reduction, we varied the sizes of our randomly generated slicing criteria as such:

Size 10. A set of 10 randomly generated contiguous lines of code. This size was chosen to approximate the size of a block of code (e.g., an `if` block).

Size 20. A set of 20 randomly generated contiguous lines of code. This size was chosen to approximate the size of a small method or function.

Size 50. A set of 50 randomly generated contiguous lines of code. This size was chosen to approximate the size of a large method or a small class.

Fragmented. A potentially fragmented set of varying numbers of lines of code. In order to select a fragmented (non-contiguous) set of lines of code, we selected the lines executed by a test case inside a file. We randomly chose a test case, generated its statement coverage, randomly chose one of the files that it executed, and selected the lines executed in it as the fragmented slicing criterion.

Dependent Variables. To assess the problem space size for a developer (or an automated client analysis) in interpreting a history slice, we define the following three dependent variables:

Number of Revisions. The number of revisions in the history of the slicing criterion that need to be examined.

Average Number of Lines for a Revision. The average number of lines of code that a developer would need to inspect in each revision. This is computed as the average number of lines for any given revision inside the history slice, aggregated across all relevant revisions, according to the treatment technique.

Total Number of Lines for a Task. The total number of lines of code that a developer would need to examine across all relevant revisions. This is computed as the sum of all examined lines across all relevant revisions, according to the treatment technique. This serves as a proxy measure of the total amount of work that a developer would need to expend to fully explore and process the history slice.

6.1.2 Experiment Design

For each slicing criterion size (i.e., Independent Variable 2), we randomly generated 1,000 slicing criteria. Although history slicing is generalizable to criteria that span multiple files, for this experiment each individual criterion is contained in a single file for the continuity requirement of criterion-size 10–50, and to limit the size of the fragmented criterion. Also, we discarded and replaced randomly generated criteria that resulted in no history: that is, the code in the slicing criteria had no previous revisions. In such cases, we viewed the prospect of exploring history unnecessary and fruitless. Then, for each slicing criteria and treatment technique, we used CHRONOS to generate the resulting history slice. The resulting history slice was used to compute the resulting dependent-variable metrics to determine the problem-space costs.

6.1.3 Results

After computing the values of the dependent variables for each combination of slicing criterion size and treatment technique, we averaged them and obtained the results displayed in Table 1.

Table 1: Results for Experiment 1

Slicing crit. size	Approach	Avg. # revisions	Avg. # lines	Avg. total lines
10	Naive	29.16	1,177.44	34,334.29
10	Conventional	3.17	783.04	2,482.24
10	H.S. w/ Context	3.17	10.01	31.74
10	H.S. w/o Context	3.17	3.54	11.24
20	Naive	30.58	1,204.98	36,848.30
20	Conventional	4.03	867.58	3,496.36
20	H.S. w/ Context	4.03	19.98	80.52
20	H.S. w/o Context	4.03	5.58	22.51
50	Naive	34.62	1,350.64	46,759.26
50	Conventional	5.92	978.00	5,789.75
50	H.S. w/ Context	5.92	49.96	295.75
50	H.S. w/o Context	5.92	9.54	56.47
Fragm.	Naive	23.95	840.03	20,118.83
Fragm.	Conventional	4.41	567.61	2,503.18
Fragm.	H.S. w/ Context	4.41	60.65	267.46
Fragm.	H.S. w/o Context	4.41	6.40	28.24

From Table 1, we focus on the value of the "total lines" variable as this is a proxy measure of the amount of work required by the developer in processing the history slice, and plot it in Figure 5. The value of this variable represents the size of the problem space when a developer searches for the history of each slicing criterion by following each approach.

For example, when a developer tries to find the history of 50 contiguous lines of code in a file in the AspectJ project by manually inspecting each revision of the file, she will have to search inside an average problem space of 46,759 lines of code. Such scenarios are likely unrealistic, as any developer would likely give up very quickly after starting the *Naive* approach. We do not expect developers to actually follow this approach — we present it to demonstrate the size of the original problem space.

The current practice is probably to use tool-supported approaches, like *Conventionally Assisted*, which reduces the problem space by one order of magnitude. For a slicing criterion of 50 lines, developers face an average problem space of 5,790 lines of code. The reduction of the problem space provided by the *Conventionally Assisted* approach result from the savings in the dimension of time (i.e., fewer revisions). However, developers still may need to inspect many lines of each revision (i.e., the space dimension) in order to correlate each line of each snapshot with their corresponding earlier and later versions.

The *History Slicing with Context* approach also provides another reduction of one order of magnitude in the problem space over the *Conventionally Assisted* approach. For a slicing criterion of 50 lines, the average problem space is reduced to 296 lines of code. This approach provides savings in both the dimensions of time and space, because it reduces the number of revisions that need to be inspected and the number of lines that need to be inspected inside each revision.

Finally, the *History Slicing without Context* approach also provides yet another reduction of one order of magnitude over the *History Slicing with Context* approach. For a slicing criterion of 50 lines, the average problem space gets reduced to 56 lines of code. In this case, the full problem space represents the solution that developers are looking for if they are interested in a minimal history slice — i.e., only the lines of the snapshots that actually changed. This approach provides further savings, because it further reduces the number of lines that need to be inspected inside each revision.

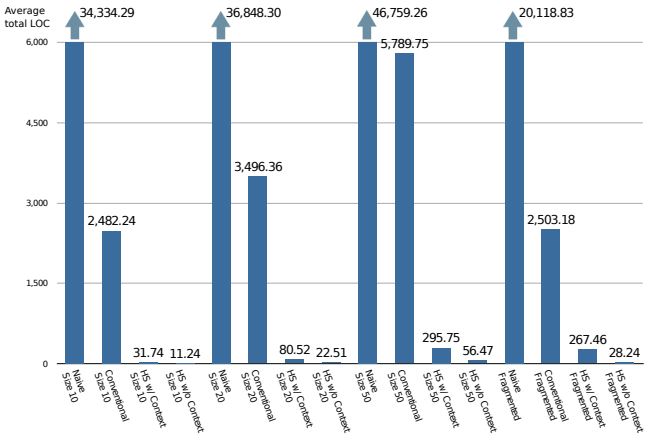

Figure 5: Average total number of lines to be examined by a developer using each treatment technique. CHRONOS' automated History Slice (HS) drastically reduces the expense for a developer. Note that the columns representing the Naive approach far exceed the bounds of the chart.

In summary, from Experiment 1, we observed a savings of needed developer effort of an order of magnitude (as measured by the cumulative number of lines of code needed to be examined and processed) for *each* additional level of automation. In all, the full *History Slicing without Context* approach reduced the problem space by *three* orders of magnitude over the *Naive* approach.

6.2 Experiment 2

Whereas Experiment 1 demonstrates the quantitative benefits of automating history slicing through the computation of a large number of slices, Experiment 2 is designed to assess how these benefits translate to actual use by real developers seeking to solve real code-evolution tasks (i.e., Research Question RQ3). As such, Experiment 2 is a comparative study of users using a Conventionally Assisted approach and CHRONOS.

6.2.1 Experimental Variables

We experimented using two *independent variables* — *treatment technique* and *code-evolution task* — and evaluated their benefit using two *dependent variables* — *time to task completion* and *task success*. In addition, we controlled for two variables — *user subject skill* and *task order*. Each variable will be defined and motivated in turn.

Independent Variable 1: Technique Treatment. We varied the technique that the subjects used to solve a code-evolution task. The two techniques were:

Treatment 1: Conventionally Assisted. We selected this technique as it is the one currently employed by developers. For the conventionally assisted approach, we used Eclipse's default CVS plugin, which is a widely used graphical user interface for CVS, an industry-standard revision control system. The plugin enables an easy and automated use of the *annotate* feature: the user can select a file, and through a context menu, choose to see the last revision in which each of its lines was changed. It also provides implementations of the *Ctrl+F* and *diff* features.

Table 2: Number of subjects with n years of experience with IDEs and revision control systems.

Tool	<1	1–2	3–4	5–6	6–7	8–9	>10
IDEs	0	6	8	5	1	1	3
Revision control	3	8	8	1	2	1	1

Treatment 2: History Slicing with Context. Our automated implementation of History Slicing, CHRONOS, implemented as an Eclipse plugin, along with its visualization (presented in Section 5). We chose to represent history slices with context, given that they were going to be consumed by humans, as opposed to history slices without context, which we envision being used as input for other analysis techniques.

Independent Variable 2: Code-evolution Task. We varied the code-evolution task that the subjects were asked to solve. The three tasks were:

Task 1: Authorship. Our user subjects were asked to identify the complete set of developers who had ever contributed changes to a segment of code. This task reflects the real task demand of determining authorship and expertise (discussed in Section 2).

Task 2: Original Revision. Our user subjects were asked to identify the original revisions in which a segment of code was originally created. This task reflects the real task demand of determining an earlier implementation of a given functionality (discussed in Section 2).

Task 3: Co-evolution. Our user subjects were asked to identify the revisions in which two segments of code in two different files were changed within a day of each other. This task reflects the real task demand of determining "evolutionary coupling" for identifying related code or code-clone risk (discussed in Section 2).

Dependent Variables. To assess the benefits of automated history slicing to actual developers (RQ3), we define two dependent variables:

Time to Task Completion. The time that the subject needed to perform a task. The time is determined by when the subject decides that they are confident and satisfied with their result.

Task Success. The correctness of the answer given by the subject. We precomputed the correct answer for each task in advance, and were able to determine whether the answer supplied by the subject matched it.

Controlled Variables. To control for the effects of outside influences, we vary the following two variables:

Subject Skill. We studied 24 subjects of varying levels of experience with programming in integrated development environments (IDEs) and revision control systems. The number of years of experience with each of these is presented in Table 2

Task Order. We varied the order in which the subjects used each technique. Half of the subjects used Treatment 1 first and Treatment 2 second (and vice versa).

6.2.2 Experiment Design

We recruited 24 subjects four our user study. We required subjects to have knowledge about both Java and revision control systems. To ensure such knowledge, we asked subjects to fill in an online questionnaire in order to sign up for our study. We used the information in this questionnaire to screen subjects according to our requirements and to capture the demographics of our population. Twenty two of our subjects were male and two were female. Twenty three subjects fell in the age group 18–29 and one of them fell in the age group 30–39. All our subjects were graduate students at UCI, two of them were also professional software developers and one was also a professional software tester. Thirteen subjects were majoring in Informatics, ten were majoring in Computer Science and one was majoring in Computer Engineering. Table 2 contains the years of experience that subjects reported to have with IDEs and revision control systems.

The experiment was structured as such for each subject:

1. We trained the subject on how to use the chosen treatment technique.
2. We presented the subject with the randomly chosen task and the source on which it will be performed.
3. We asked the subject to answer the chosen task.
4. We repeated Steps 1–3 for the other treatment technique and another randomly chosen task.

Each subject was asked to perform two tasks with two techniques. Twelve of the 24 subjects used the *Conventionally Assisted* approach first and the *History Slicing with Context* approach second, and other twelve used the tools in the opposite order. Each of the three tasks was assigned to an equal number (16) of subjects, and each technique was used on each task in equal numbers — i.e., each task was answered with each tool eight times. In other words, every combination of treatment techniques, task, and task order was performed in equal numbers.

To encourage subject participation, we offered a small base monetary compensation. Additionally, we offered a small additional compensation to encourage speed and correctness. The subjects were given a maximum of ten minutes to answer each task that they were given, regardless of the treatment technique. A correct answer was rewarded with a small additional compensation, and the earlier that the answer was given determined the size of that small additional compensation. For example, a correct answer given at minute 5 was rewarded more greatly than a correct answer given at minute 10. This compensation structure was implemented equally for both techniques to encourage the subjects to answer quickly and accurately.

Table 3: Time to task completion and success rate for each technique and task.

Technique	Task	Avg. time	% Success
Conventional	Task 1	6:04	37.5%
History Slicing	Task 1	3:21	100%
Conventional	Task 2	7:34	37.5%
History Slicing	Task 2	3:15	100%
Conventional	Task 3	9:57	0%
History Slicing	Task 3	5:19	62.5%

6.2.3 Results

For each technique-task combination, we averaged the time that all subjects took to perform the task, and we calculated the percentage of subjects whose answer was correct. Table 3 contains the results that we obtained for all technique-task combinations.

As we can see in this table, the subjects who used the *History Slicing with Context* technique could perform any of the tasks in around half the time that was needed by the subjects who used the *Conventionally Assisted* technique. We performed a paired t-test with 7 degrees of freedom over the times that we captured for subjects performing the tasks. The difference in time needed to perform the tasks for the two different techniques was statistically significant, with

p values of 0.02, 0.01, and 0.00006 for Tasks 1, 2, and 3, respectively.

In the specific case of Task 3, the differences in time are so high because most subjects using the *Conventionally Assisted* technique for Task 3 were unable to answer it in the maximum ten minutes allotted. Only two subjects provided an incomplete, incorrect answer when they had 15 seconds and 5 seconds left, respectively. In a post-experiment discussion, some of the subjects mentioned that, in real life, they would have just given up on performing Task 3 before spending ten minutes on it by using the *Conventionally Assisted* technique.

Regarding success rate, the subjects who used the *History Slicing with Context* technique were in general 62.5% more likely to provide correct solutions to any of the tasks than those subjects who used the *Conventionally Assisted* technique. We also performed a paired t-test with 7 degrees of freedom over the correctness of the answers provided by our subjects. The difference in correctness of the answers for the two different techniques was also statistically significant, with a p value of 0.01 for each task.

We observed multiple reasons why our subjects did not provide a correct answer for each question when using the *Conventionally Assisted* technique. For Task 1, some subjects only reported information about the last change to each line of interest, instead of their whole history; some other subjects only reported information about the header of the method of interest, instead of all its lines; and another subject checked all revisions of the file one by one, manually checking which revisions affected the lines of interest, and ending up providing a partial answer. For Task 2, some subjects interpreted changes in lines as additions; and other subjects ran out of time. For Task 3, all subjects ran out of time long before even composing a small portion of the correct answer. We also observed a common phenomenon to all tasks when using the *Conventionally Assisted* technique: subjects very frequently lost track of where in the file they were, and they needed to backstep to remind and re-focus their search.

When subjects used the *History Slicing with Context* technique, they only provided incorrect answers for Task 3 — that is, for Tasks 1 and 2, *all* of our subjects provided the correct answer, and a majority of the subjects provided correct answers to Task 3 (despite zero success for the other technique). For the 37.5% that incorrectly answered Task 3 with history slicing, we observed the following reasons for their failure: One subject did not explore the beginning of the timeline, missing the first revision. Another subject rushed and explored only the latter half of the timeline. Another subject only explored a small subset of the timeline because she thought that her partial view was complete — the zooming function did not refresh the image until the button of the mouse was released, causing her to think that her zoomed-in view contained the whole timeline.

6.3 Summary of Results

Considering the results from both Experiment 1 and 2, we now answer our research questions specified in Section 6.

To research question RQ1, we assess that automating the computation of history slices greatly reduced the problem space needed to be examined by the developer. Experiment 1 confirmed this and found multiple orders of magnitude in savings by automating the history slice computation.

To research question RQ2, we assess that the size of the slicing criterion did, indeed, affect these reductions: the larger the slicing criterion, the larger the cost in its computation. However, the cost increases are minimal, especially when considering the automated approaches. Also, the fragmentation of the slicing criterion does not affect the cost.

To research questions RQ3, we assess that the practical benefits of the automated history slicing had profound influence on both the speed and correctness of the users attempting to solve the code-evolution tasks. The users offered their answers significantly more quickly with Chronos, and even so, their answers were correct significantly more often.

Overall, we find these results as strong evidence that the task of computing history slices is a non-trivial task for developers to compute on their own with current toolkits.

7. THREATS TO VALIDITY

An external threat to the validity of our evaluation is that our results may not hold for other codebases or users. Despite the fact that we studied only one codebase, AspectJ, we do not consider this a significant threat. The main factors that affect history slicing are the codebase's age and the size of the slicing criterion, regardless of the nature or functionality of the codebase. We experimented with the size of the slicing criterion and found an effect that easily scales with criteria size. And, given the random selection of 4,000 slicing criteria, history slices were computed on files that were brand new, as well as other files that were nearly a decade old. Regardless of the age of the code sliced, Chronos computed quickly (the maximum was 30 seconds in our unoptimized tool). Moreover, older codebases and larger criteria will likely affect the conventional SCM approaches much more than the automated history-slicing technique.

Another external threat to validity is that our user study in Experiment 2 does not study people who are already familiar with the code base, and thus, its results may not generalize to such more experienced developers. However, we believe this factor to be insignificant because the same experience level was brought to both approaches. We expect that greater experience would assist the more automated approach because its results require more interpretation and application of experience, whereas the manual approach requires more tedium. Nevertheless, in the future, we plan to evaluate with in-vivo field studies.

An internal threat to validity is that Chronos, our implementation of history slicing, uses a textual line mapping technique, which may not always identify the correct line equivalence in between revisions, e.g. when a method is moved to another file. However, this limitation affects both Chronos, and the manual approaches to history slicing, since traditional SCM tools do not provide any mechanism for tracking movements of code in between files. As a consequence, we believe that this limitation does not affect the results of our experiments. Moreover, history slicing may address this limitation because its approach allows other line mapping techniques [*e.g.,* 7, 16].

A threat to construct validity is that in Experiment 1, we measured developer expense in terms of the number of lines that need to be examined. While we use the metric of the number of lines to be examined as a proxy measure for developer effort, we posit that it is actually indicative of the amount of effort that would be needed by a developer — more lines will require the developer to inspect those

lines, leading to greater effort. Regardless, while Experiment 1 allowed us to compute a great number of history slices (16,000), we intentionally conducted Experiment 2 to allow us to determine if the proxy measure of effort manifests for real developers on real code-evolution tasks.

8. RELATED WORK

Existing research into analyzing and presenting code history follows two general directions that we will describe here: (1) analyses to trace history through revisions, and (2) user interfaces to access these histories.

Girba et al. [12] presented an evolution meta-model to represent the history of source code artifacts at multiple levels of granularity (e.g. class, method, attribute). To model the history of source code at the level of individual lines of code, Zimmermann et al. [33] proposed annotation graphs. While similar to the history graphs defined herein, blocks of changed code (i.e., *modification hunks*) are undifferentiated in their historical evolution due to the limited granularity offered by *diff*, on which they are built. To address the problem of modification hunks, multiple researchers proposed line-mapping techniques. Canfora et al. [5] first perform an inexact difference (like the annotation graph approach) and then further refine blocks of code using a distancing algorithm among the constituent lines. Chen et al. [6] and Williams and Spacco [32] propose a similar approach using a combinatorial optimization algorithm to refine blocks. Reiss [27] proposed a set of line mapping techniques and performed an empirical comparison of them. Other researchers proposed more sophisticated algorithms by performing the mapping over models of the program [*e.g.,* 2, 24] allowing the detection of moved code [*e.g.,* 7, 16] or providing techniques for specific domains [*e.g.,* 8]. In this work, we leverage their results to implement our own line-mapping technique that incorporates aspects drawn from each. Moreover, the concept of history slicing applies to any line-mapping technique.

To provide interfaces to code history, some authors presented visualizations of the evolution of code at the system level [*e.g.,* 13, 20, 28]. At the file level, industry-standard SCM systems provide functionalities that allow for a user to access course-grained whole history information (such as `cvs log`) or fine-grained history information that is limited to a pair of revisions (such as `svn diff`). In addition, they allow for a functionality (e.g., *annotate*) that presents a single revision number for each line of code to represent its most recent change. Such limited functionality leaves much of a traversal-of-the-history effort to the labor of the developer. Bradley and Murphy [4] augment the information presented by such conventional SCM tools by making information, such as the date and author, available for each line of an *annotate* query. Hassan and Holt [14] and Holmes and Begel [15] propose techniques that provide whole-history information, but at the level of granularity of a single program method, without details for the lines therein. History slicing, in contrast, targets a different problem: allowing a developer to view deep history evolution across any number of revisions for any block of code in any number of files, and to be able to draw inferences from the patterns seen.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we present an approach, *history slicing* for assisting code-evolution tasks that involve the exploration of source-code history. This approach enables the extraction of all revisions and lines of code that are related to a set of lines of interest. We define the concept of a *history slice* as this resulting set of lines across multiple revisions, and their mapping between revisions. History slices can be used to efficiently and accurately solve software-maintenance tasks that require knowledge of the code evolution.

History slicing has four key benefits over conventional history-exploration tools that are included with SCM systems and traditional add-on user interfaces to them: (1) History slicing can drastically reduce the amount of information that a developer would need to examine in order to trace the history of a set of lines of code; (2) History slicing enables the computation and representation of any number of revisions, whereas conventional tools primarily perform pairwise differences; (3) History slicing directly supports tracing of the history for any arbitrary set of lines of code, regardless of whether they are contiguous or fragmented, in a single file or in multiple, and in a single revision or across multiple revisions; and (4) The presentation of the history slice can enable users to recognize patterns in the evolution and changes to the constituent lines and files.

We present the history-slicing approach as a framework of multiple constituent steps, each of which are configurable with multiple possible techniques. We also present our instantiation of this framework to implement our technique in a prototype history-slicing tool, Chronos.

We performed two experiments, using Chronos, to evaluate the merits of history slicing. The first experiment involved the automatic computation 16,000 history slices with various levels of automation and various sizes of slicing criteria. The results of this experiment show that history slicing can drastically reduce the amount of information that a developer would need to examine to trace the history of a set of lines of code. We found savings of three orders of magnitude in the lines to be examined by the developer for the fully automated history slicing over the baseline.

The second experiment involved actual users performing code-evolution tasks to determine the practical benefits of history slicing. The results of this study show that the reduction of the information that needs to be examined translated to actual benefit to users. In addition, the experiment demonstrated the practical benefits of the ability to represent several revisions in a single view, as well as viewing change patterns among code in multiple files. The users could solve their tasks correctly more than twice as often with history slicing over conventional tools, and moreover, could give those correct answers in about half the time.

While the results of our experiment are quite positive, more experiments are necessary. We will conduct more experiments using other codebases and user subjects to ensure generalizability. We will also explore how different line mapping techniques affect our results. In addition, we will explore the tasks that can be assisted by coupling history analyses with traditional program analyses — for example, viewing the recent history of lines that are identified as faulty according to a fault-localization technique.

10. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under award CCF-1116943, and by a Google Research Award.

References

[1] Apache Software Foundation. Apache Subversion. http://www.eclipse.org/aspectj/, 2000.

[2] T. Apiwattanapong, A. Orso, and M. J. Harrold. JDiff: A Differencing Technique and Tool for Object-Oriented Programs. *Automated Software Engineering*, 14:3–36, 2007.

[3] T. Bakota. Tracking the Evolution of Code Clones. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 86–98, 2011.

[4] A. W. J. Bradley and G. C. Murphy. Supporting Software History Exploration. In *International Working Conference on Mining Software Repositories*, pages 193–202, 2011.

[5] G. Canfora, L. Cerulo, and M. D. Penta. Identifying Changed Source Code Lines from Version Repositories. In *International Workshop on Mining Software Repositories*, pages 14–21, 2007.

[6] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through Source Code using CVS Comments. In *International Conference on Software Maintenance*, pages 364–373, 2001.

[7] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-Aware Configuration Management for Object-Oriented Programs. In *International Conference on Software Engineering*, pages 427–436, 2007.

[8] A. Duley, C. Spandikow, and M. Kim. A Program Differencing Algorithm for Verilog HDL. In *International Conference on Automated Software Engineering*, pages 477–486, 2010.

[9] Eclipse Foundation. AspectJ. http://www.eclipse.org/aspectj/, 2001.

[10] Free Software Foundation. CVS - Concurrent Versions System. http://www.nongnu.org/cvs/, 1990.

[11] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A Degree-of-Knowledge Model to Capture Source Code Familiarity. In *International Conference on Software Engineering*, pages 385–394, 2010.

[12] T. Gîrba and S. Ducasse. Modeling History to Analyze Software Evolution. *Journal of Software Maintenance*, 18(3):207–236, 2006.

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

[14] A. E. Hassan and R. C. Holt. Using Development History Sticky Notes to Understand Software Architecture. In *International Workshop on Program Comprehension*, pages 183–192, 2004.

[15] 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.

[16] J. J. Hunt and W. F. Tichy. Extensible Language-Aware Merging. In *International Conference on Software Maintenance*, pages 511–520, 2002.

[17] H. Kagdi, M. Hammad, and J. Maletic. Who Can Help Me with this Source Code Change? In *International Conference on Software Maintenance*, pages 157–166, 2008.

[18] 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.

[19] H. W. Kuhn. The Hungarian Method for the Assignment Problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

[20] M. Lanza and S. Ducasse. Understanding Software Evolution using a Combination of Software Visualization and Software Metrics. In *Langages et Modèles à Objets*, pages 135–149, 2002.

[21] 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.

[22] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining Mental Models: a Study of Developer Work Habits. In *International Conference on Software Engineering*, pages 492–501, 2006.

[23] V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[24] J. I. Maletic and M. L. Collard. Supporting Source Code Difference Analysis. pages 210–219, 2004.

[25] D. W. McDonald and M. S. Ackerman. Expertise Recommender: a Flexible Recommendation System and Architecture. In *Computer Supported Cooperative Work*, pages 231–240, 2000.

[26] A. Mockus and J. D. Herbsleb. Expertise Browser: a Quantitative Approach to Identifying Expertise. In *International Conference on Software Engineering*, pages 503–512, 2002.

[27] S. P. Reiss. Tracking Source Locations. In *International Conference on Software Engineering*, pages 11–20, 2008.

[28] R. M. Ripley, A. Sarma, and A. van der Hoek. A Visualization for Software Project Awareness and Evolution. In *International Workshop on Visualizing Software for Understanding and Analysis*, pages 137–144, 2007.

[29] D. Schuler and T. Zimmermann. Mining Usage Expertise from Version Archives. In *International Working Conference on Mining Software Repositories*, pages 121–124, 2008.

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

[31] Software Freedom Conservancy. Git: the Fast Version Control System. http://git-scm.com/, 2005.

[32] C. C. Williams and J. W. Spacco. Branching and Merging in the Repository. In *International Working Conference on Mining Software Repositories*, pages 19–22, 2008.

[33] T. Zimmermann, S. Kim, A. Zeller, and E. J. Whitehead, Jr. Mining Version Archives for Co-changed Lines. In *International Workshop on Mining Software Repositories*, pages 72–75, 2006.

[34] T. Zimmermann, P. Weibgerber, S. Diehl, and A. Zeller. Mining Version Histories to Guide Software Changes. In *International Conference on Software Engineering*, pages 563–572, 2004.