2015 30th IEEE/ACM International Conference on Automated Software Engineering

Development History Granularity Transformations

Kıvanç Muşlu[†] Luke Swart[†] Yuriy Brun[☇] Michael D. Ernst[†]
[†]University of Washington
Seattle, WA 98105
[☇]University of Massachusetts
Amherst, MA 01003
{kivanc, lukeswart, mernst}@cs.washington.edu, brun@cs.umass.edu

***Abstract*—Development histories can simplify some software engineering tasks, but different tasks require different history granularities. For example, a history that includes every edit that resulted in compiling code is needed when searching for the cause of a regression, whereas a history that contains only changes relevant to a feature is needed for understanding the evolution of the feature. Unfortunately, today, both manual and automated history generation result in a single-granularity history. This paper introduces the concept of *multi-grained* development history views and the architecture of Codebase Manipulation, a tool that automatically records a fine-grained history and manages its granularity by applying granularity transformations.**

I. Introduction

Most software development uses version control to enable collaboration and to create a development history. The version control history is useful for many tasks, such as localizing changes that caused regression failures, identifying developers responsible for specific code, and manually examining recent changes. However, each of these tasks is best performed at a different granularity of history. For example, finding the cause of a regression failure is best performed on a history of all points during development at which the code compiled, studying fine-grained change patterns [34] or backtracking [46] requires the finest possible granularity, and understanding how a bug was fixed requires seeing one snapshot before the bug repair began and one snapshot after the repair completed.

Unfortunately, today's approaches generate inflexible histories, each of which works well for only a subset of software engineering tasks. Manually-managed histories tend to be too coarse-grained, while automatically-recorded histories are too fine-grained. Specifically, manually-managed histories' coarse granularity causes them to omit many points during development at which the code compiled, necessary for localizing the cause of a regression. It is virtually impossible to manually create the fine-grained history necessary for studying change patterns [34] and backtracking [46]. And a manually-managed history is unlikely to contain the snapshots right before and after a repair. Manually-managed histories are suboptimal for development tasks not just because they fail to include important moments during development, but also because manual version control checkpoints often tangle changes made for multiple development tasks, such as fixing a bug and refactoring code. By contrast, automatically-managed approaches record all developer actions [24], [35], [45] and lead to fine-grained histories that are well-suited to studying developer behavior [34], [42] but poorly-suited to manual examination and for analyses that rely on semantic checkpoints.

Even if a history can be created at an ideal granularity for a specific task, no single way of recording the history can satisfy all tasks, because different development tasks require different granularities. Additionally, development tasks sometimes require the development history to be restructured. For example, one way to understand a bug fix is to reordering the history to move away any unrelated changes that were performed during the fix, so the developer can focus on the relevant changes.

We argue that since different development tasks require accessing the development history at different granularities, and that the histories produced using today's methods are inflexible and offer no tools to change history granularity [24], [35], [45], a new approach is needed. We posit that the development history should not be restricted to a single granularity. Instead, the history should be recorded automatically in a way that allows its granularity to be transformed into the one best suited for the particular development task at hand. To that end, we designed Codebase Manipulation to mitigate the *inflexibility* of current development histories by (1) automatically recording a fine-grained development history and (2) providing the developer with tools to manipulate the granularity and the order of the history. Codebase Manipulation allows the developer to change the history granularity repeatedly, and all its history manipulations are reversible. This supports development tasks that require the developer to view the history at *multiple* granularities.

This paper presents a set of primitive manipulation transformations that can be combined to manage history granularity. We demonstrate powerful granularity transformations that can be composed of these primitives and design an architecture for a tool that automatically records and manages development history granularity.

The three history-transforming primitives from which all necessary transformations can be composed are COLLAPSE for combining several edits into a single edit, EXPAND for splitting a previously collapsed edit into its parts, and MOVE for reordering edits. These primitives are sufficient, for example, to transform a fine-grained development history into granularities such as all file-level changes, all compilable code, and all collocated edits. In turn, this supports activities such as finding the cause of a regression and separating distinct development tasks into separate revisions.

The rest of the paper is organized as follows. Section II formally defines Codebase Manipulation concepts and primitive transformations. Section III shows that powerful granularity transformations can be composed of these primitives. Section IV proposes an architecture for a Codebase Manipulation implementation. Finally, Section V places our work in the context of related research, and Section VI summarizes our contributions.

II. Definitions

Our goal is to improve the usability of development histories by automatically recording a fine-grained version control history and by providing automated granularity transformations to make

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.53

the history available at multiple granularities. To that end, we design Codebase Manipulation. To aid in understanding Codebase Manipulation's high-level granularity transformations, we first explain how Codebase Manipulation represents the development history and how Codebase Manipulation's primitives operate on that history.

This section defines the representation and primitives, and Section III specifies the high-level granularity transformation algorithms. For brevity, the definitions ignore file creation and deletion; they can be extended to handle these actions.

Definition 1 (Snapshot)**.** A snapshot s is a single developer's view of a program at a point in time, including the current contents of unsaved editor buffers. Unsaved editor buffers have priority: if a file on disk differs from the editor buffer for that file, the snapshot contains the contents of the editor buffer.

An edit can either be atomic or compound. An *atomic edit* encodes replacement of one chunk of text in a file by another chunk; either the original or the final chunk of text may be empty. A *compound edit* is a sequence of edits, each of which is either atomic or compound. A *development history* is an edit that can be applied to the empty snapshot, $\emptyset$. Two development histories are views of each other if when applied to $\emptyset$, they produce the same snapshot.

Definition 2 (Edit)**.** An edit may be *atomic* or *compound*.
(Atomic edit)**.** Let S be the set of all snapshots. An *atomic edit* is a 4-tuple $r = \langle \mathit{filepath}, \mathit{offset}, \mathit{length}, \mathit{text} \rangle$. We treat r as a function: $r\colon S \to S$. $r(s)$ is the same as s except that in $r(s)$, the *length* characters in s in the file *filepath* starting at position *offset* are replaced by *text*.[1]

(Compound edit)**.** Let S be the set of all snapshots. For all $n \geq 0$, a *compound edit* is a sequence of edits $e = \langle e_1, e_2, \ldots, e_n \rangle$. We treat e as a function $e\colon S \to S$ such that $e(s) = e_n(e_{n-1}(\ldots(e_2(e_1(s)))))$.

For example, the atomic edit $e_1 = \langle \text{foo.txt}, 0, 0, \text{“public”} \rangle$ adds the word "public" at the beginning of `foo.txt`. After that, the atomic edit $e_2 = \langle \text{foo.txt}, 1, 5, \text{“rivate”} \rangle$ replaces "ublic" with "rivate", constructing the word "private"; and after that, the atomic edit $e_3 = \langle \text{foo.txt}, 0, 7, \text{“”} \rangle$ deletes the word "private". Example compound edits are $\langle e_1, e_2, e_3 \rangle$ and $\langle e_1, \langle e_2, e_3 \rangle \rangle$.

Definition 3 (Applicability)**.** Let S be the set of all snapshots. An atomic edit $r = \langle \mathit{filepath}, \mathit{offset}, \mathit{length}, \mathit{text} \rangle$ is applicable to a snapshot $s \in S$ if the file *filepath* has at least $\mathit{offset} + \mathit{length}$ characters. A compound edit $e = \langle e_1, e_2, \ldots, e_n \rangle$ is applicable to a snapshot $s \in S$ if $e_1, e_2, \ldots, e_n$ can be applied in sequence to s. More formally, e is applicable to s iff e_1 is applicable to s, e_2 is applicable to $e_1(s)$, …, and e_n is applicable to $e_{n-1}(e_{n-2}(\ldots(e_2(e_1(s)))))$.

If an edit e is not applicable to a snapshot s, $e(s)$ is undefined.

Definition 4 (Development history)**.** A development history is a compound edit that is applicable to the empty snapshot, $\emptyset$.

Definition 5 (Development history view)**.** Let h, h' be two development histories. We call h' a view of h (and h a view of h') iff $h(\emptyset) = h'(\emptyset)$.

[1] Other definitions for the atomic edit are possible. For example, instead of using character offsets to indicate where to change the text, an atomic edit could specify the surrounding text.

There are three history manipulation primitives: COLLAPSE, EXPAND, and MOVE. Collapse replaces a sequence of edits by a compound edit that consists of that sequence. Expand is the reverse of collapse; it replaces a non-top-level compound edit by the sequence of its component parts. Move moves the location of an edit within the history. These three primitives are sufficient to express all of Codebase Manipulation's high-level granularity transformations.

Definition 6 (COLLAPSE)**.** For all compound edits $e = \langle e_0, \ldots, e_{i-1}, e_i, \ldots, e_j, e_{j+1}, \ldots \rangle$, $\mathrm{COLLAPSE}(e, i, j)$ returns $\langle e_0, \ldots, e_{i-1}, \langle e_i, \ldots, e_j \rangle, e_{j+1}, \ldots \rangle$.

For example, $\mathit{collapse}(\langle e_0, e_1, e_2 \rangle, 0, 1) = \langle \langle e_0, e_1 \rangle, e_2 \rangle$ and $\mathit{collapse}(\langle e_0, e_1, e_2, e_3, e_4, e_5 \rangle, \{\langle 0,1 \rangle, \langle 3,5 \rangle\}) = \langle \langle e_0, e_1 \rangle, e_2, \langle e_3, e_4, e_5 \rangle \rangle$.

Definition 7 (EXPAND)**.** For all compound edits $e = \langle e_0, \ldots, e_{i-1}, e_i, e_{i+1}, \ldots \rangle$, where e_i is a compound edit $\langle e_{i_1}, e_{i_2}, e_{i_3}, \ldots, e_{i_{last}} \rangle$, $\mathrm{EXPAND}(e, i)$ returns $\langle e_0, \ldots, e_{i-1}, e_{i_1}, e_{i_2}, e_{i_3}, \ldots, e_{i_{last}}, e_{i+1}, \ldots \rangle$.

Definition 8 (MOVE)**.** For all development histories $h = \langle e_0, e_1, \ldots, e_{i-1}, e_i, \ldots, e_j, \ldots \rangle$, $\mathit{move}(h, i, j)$ returns $\langle e_0, e_1, \ldots, e_{i-1}, e'_{i+1}, \ldots, e'_{j-1}, e'_i, e_j \ldots \rangle$ if the resulting sequence of edits is applicable to an empty snapshot $\emptyset$; otherwise, it returns h unmodified. The edits between the reordered edits $(e_i, \ldots, e'_{j-1})$ might need to be modified to ensure that the resulting history reaches the same snapshot. The operational transform [41] defines how two adjacent edits should modified to have their positions swapped.

Definition 9 (Granularity transformation)**.** Let H be the set of all development histories. A granularity transformation is a function $g\colon H \to H$ that applies a series of collapse, expand, and move transformations. In other words, g is a sequence of history manipulation primitives. Granularity transformations may be parameterized. That is, their domain may be $H \times P$, where P is a set of parameters, such as starting and ending edits in the history to which the transformation should apply.

For simplicity of exposition, this paper gives algorithms for development histories with a single linear branch of development. Our work generalizes to multiple developers working concurrently and to using branches.

III. DEVELOPMENT HISTORY GRANULARITY TRANSFORMATIONS

This section describes powerful granularity transformations that can be composed of the three primitive transformations, COLLAPSE, EXPAND, and MOVE. Using the algorithms described in this section, a developer who wishes to find the cause of a regression failure can convert an automatically-recorded history into one consisting of every compilable edit then use history bisection on that ideal-granularity history. To manually inspect how a code element has evolved (e.g., which developer added a class and which other developers helped repair bugs related to the class), the developer can convert the history into one that groups together changes based on the files they affect. Finally, to better understand a set of changes made over time to a part of a class, the developer can group all collocated edits together.

We first define two fundamental granularity transformations, COLLAPSEBYGROUP and REORDERBYGROUP, and then show how these two transformations can be serve as a basis for other transformations. Both COLLAPSEBYGROUP and REORDERBYGROUP are composed entirely of the primitives defined in Section II. These transformations allow regrouping and reordering edits. To direct these transformations, the the GROUPNAME interface, specifies relationships between edits. An implementation of this interface map each edit in a history to a name string; edits that are related map to the same name. For example, an implementation of GROUPNAME can return a single name for: all edits related to a feature, all edits to the same file, or all edits by the same developer. If an edit is compound and composed of edits with different names, GROUPNAME throws the `multiplegroups` exception, which could prompt the algorithm using GROUPNAME to, for example, consider these edits individually, fail, or use an alternate method to classify the compound edit. Some GROUPNAME implementations may be project-specific (e.g., the same feature example), while others are general (e.g., the same file or same developer examples).

GROUPNAME:
Input: history h and edit e in h **Output:** The name of the group to which the edit belongs Throws a `multiplegroups` exception if e is compound and the edits making up e belong to more than one group.

In all algorithms that follow that use GROUPNAME, an implicit preprocessing step is to recursively EXPAND edits for which GROUPNAME throws the `multiplegroups` exception.

The COLLAPSEBYGROUP algorithm COLLAPSEs consecutive edits with the same name without reordering the history. An implementation of the GROUPNAME interface specifies which consecutive edits should be COLLAPSEd.

COLLAPSEBYGROUP:
Input: history h, two edit indices *start* and *end* in h, and an implementation of GROUPNAME **Output:** A view of h consisting of $\langle e_0,\ldots,\ e_{start-1},\ e_\alpha,\ e_\beta,\ e_\gamma,\ldots,e_\omega,e_{end+1},\ldots\rangle$, where: • $e_\alpha = \langle e_{start},e_{start+1},\ldots,e_a\rangle$, $e_\beta = \langle e_{a+1},\ldots,e_b\rangle$, $e_\gamma = \langle e_{b+1},\ldots,e_c\rangle$, $\ldots$, $e_\omega = \langle e_{z+1},\ldots,e_{end}\rangle$, • for all $\hat{e} \in \{e_\alpha,e_\beta,e_\gamma,\ldots,e_\omega\}$, for all $e,e' \in \hat{e}$, GROUPNAME(e) = GROUPNAME(e'), and • GROUPNAME(e_a) $\neq$ GROUPNAME(e_{a+1}), GROUPNAME(e_b) $\neq$ GROUPNAME(e_{b+1}), $\ldots$, GROUPNAME(e_z) $\neq$ GROUPNAME(e_{z+1}).

The REORDERBYGROUP algorithm enables history reordering. An implementation of the GROUPNAME interface specifies which edits should be MOVEd to be together.

REORDERBYGROUP:
Input: history h, two edit indices *start* and *end* in h, and an implementation of GROUPNAME **Output:** A view of h produced only by MOVEing edits in h, such that for all $start \le i,j \le end$, GROUPNAME(e_i) = GROUPNAME(e_j) $\iff$ for all $i<k<j$, GROUPNAME(e_k) = GROUPNAME(e_i).

REORDERBYGROUP and COLLAPSEBYGROUP are powerful and enable expressing interesting history transformations, including producing the following histories:

Compilable code. A compilable code history consists only of edits that produce compiling snapshots. This history view is useful for analyses, such as history bisection of test failures, that only apply to compilable code and benefit from having access to every compilable snapshot that occurred during development. GROUPCOMPILABLE COLLAPSEs consecutive edits of a history into a compilable code history. By default, GROUPCOMPILABLE has a preprocessing step of recursively EXPANDing all edits, but this step is optional. Without preprocessing, GROUPCOMPILABLE can preserve a custom history granularity and select only the edits in the history's current granularity that produce compiling snapshots.

GROUPCOMPILABLE:
Input: history h, two edit indices *start* and *end* in h, and a procedure COMPILE whose input is a snapshot and output is true if that snapshot compiles, and false otherwise **Output:** A view of h produced only by COLLAPSEing consecutive edits in h, such that the view consists of $\langle e_0,\ldots,e_{start-1},e_\alpha,e_\beta,e_\gamma,\ldots,e_\omega,e_{end+1},\ldots\rangle$, where: • Snapshots $\langle e_0,\ldots,e_{start-1},e_\alpha\rangle(\emptyset)$, $\langle e_0,\ldots,e_{start-1},e_\alpha,e_\beta\rangle(\emptyset)$, $\ldots$, $\langle e_0,\ldots,e_{start-1},e_\alpha,e_\beta,e_\gamma,\ldots,e_\omega\rangle(\emptyset)$ all COMPILE, and • For all $\hat{e} \in \{e_\alpha,e_\beta,e_\gamma,\ldots,e_\omega\}$, there does not exist an edit $e \in \hat{e}$ such that the snapshot $\langle e_0,\ldots,e\rangle(\emptyset)$ COMPILEs.

File-level. A file-level change history reorders all of each file's edits to be adjacent in the history, and keeps the edits to different files separate. This history view is useful for manual inspection and analyses that are limited to individual files. Many version control systems already provide `diff` commands that allow developers to view all the changes made to a single file, and other commands to view the history of a single file, e.g., `git log filename`. GROUPFILES rewrites a history into a file-level change history.

GROUPFILES:
Input: history h, and two edit indices *start* and *end* in h **Output:** A view of h, transformed by REORDERBYGROUP where the implementation of GROUPNAME returns the file(s) in which the edit was made.

Collocated edit. A sequence of consecutive edits in a history is collocated if each edit in the sequence touches at least one character that is either touched by or is adjacent to a character touched by a previous edit in the sequence (see Definition 10). Such a sequence represents a series of edits in the same place in the codebase. For example, if a developer types a line of text at the start in a file, edits parts of that line, types another line right after the first, makes more edits to the first line, and then moves on either to a distant part of the file or to another file, the creation of and edits to the first two lines would all be considered collocated. If the developer later returned to edit the first two lines after making the changes elsewhere, these new edits would not be collocated with the original ones. GROUPCOLLOCATED rewrites a history into a collocated edit change history, which COLLAPSEs together maximal sequences of collocated edits (Definition 11). The preprocessing step of recursively EXPANDing all edits is optional. This history view is useful when a developer wants to manually examine a set of changes related to a particular piece of code, or partially rollback some changes to a piece of code.

Definition 10 (First maximal sequence of collocated edits)**.** For all sequences of edits $e_0,e_1,e_2,\ldots,e_z$, the *first maximal sequence of collocated edits* is $e_0,\ldots,e_k$, such that k is the largest value such that either $k=0$ or for all $0 \le i \le k$, there exists $0 \le j < i$ such that e_i touches at least one character touched by e_j.

Definition 11 (Grouping of maximal collocated edits)**.** For all sequences of edits $e_0, e_1, e_2, \ldots, e_z$, the *grouping of maximal collocated edits* is $e_\alpha, e_\beta, e_\gamma, \ldots, e_\omega = \langle e_0, \ldots, e_a \rangle, \langle e_{a+1}, \ldots, e_b \rangle, \ldots, \langle e_{y+1}, \ldots, e_z \rangle$, such that:

- $\langle e_0, \ldots, e_a \rangle$ is the first maximal sequence of collocated edits of $e_0, e_1, e_2, \ldots, e_z$,
- $\langle e_{a+1}, \ldots, e_b \rangle$ is the first maximal sequence of collocated edits of $e_{a+1}, e_{a+2}, \ldots, e_z$,
- $\ldots$, and
- $\langle e_{y+1}, \ldots, e_z \rangle$ is the first sequence of collocated edits of $e_{y+1}, e_{y+2}, \ldots, e_z$.

GROUPCOLLOCATED:
Input: history h, and two edit indices *start* and *end* in h
Output: A view of h consisting of $\langle e_0, \ldots, e_{start-1}, e_\alpha, e_\beta, e_\gamma, \ldots e_\omega, e_{end+1}, \ldots \rangle$, where $e_\alpha, e_\beta, e_\gamma, \ldots e_\omega$ is the grouping of maximal collocated edits of $e_{start}, e_{start+1}, \ldots, e_{end}$.

IV. Codebase Manipulation Architecture

This section describes an architecture for a Codebase Manipulation implementation for the Eclipse IDE. Codebase Manipulation automatically records a fine-grained development history and enables the developer to modify the granularity of that history and access the resulting history as a typical version control repository. Codebase Manipulation removes the burden of manual development history creation, improves existing historical analyses, and simplifies the implementation of new historical analyses. Our Codebase Manipulation design aims to satisfy the following requirements (although evaluating that our design meets the requirements is outside of the scope of this paper):

Complete history: Codebase Manipulation records every developer action (including ones that the developer undoes) and the resultant code changes.

Easy-to-use history: Codebase Manipulation's history views are easy to use by the developer, and by automated analysis tools.

Unobtrusive recording: Codebase Manipulation does not interfere with existing development tools. It neither slows down the developer's IDE nor affects manually-managed version control histories.

Codebase Manipulation automatically records the fine-grained history into a Git repository. Each developer action, even ones that do not alter the source code, results in a commit, with the log message storing information on the action itself. Do do this, Codebase Manipulation is built on top of Solstice [31], an Eclipse plug-in that enables Codebase Replication [29], [30] and facilitates IDE interactions (Figure 1). Solstice maintains a copy of the developer's code in parallel to the developer's work, detects all code changes, and provides Codebase Manipulation with observer patterns for the changes.

Codebase Manipulation satisfies the complete-history requirement by detecting every developer action within Eclipse via the Eclipse's API, and recording all such actions and every textual change to the source code.

Codebase Manipulation satisfies the easy-to-use requirement by providing a history manipulation framework to automatically transform the recorded development history into coarser granularities. The converted histories are themselves Git repositories, which can be inspected manually and interface with automated tools. Future work will evaluate how well Codebase Manipulation satisfies this requirement.

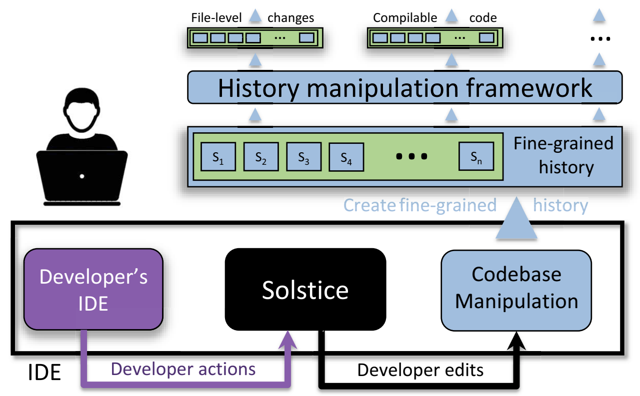

Fig. 1: Codebase Manipulation architecture. Codebase Manipulation (blue) extends Solstice (black) to automatically maintain the fine-grained development history, which the manipulation framework transforms into views of other granularities.

Finally, Codebase Manipulation satisfies the unobtrusive-recording requirement by storing its fine-grained Git repository in a unique folder on the filesystem. The developer may continue to use any version control system, including Git, to create a manual history in parallel, and tools can access both the codebase and the manual history. We believe Git to be fast enough for Codebase Manipulation's overhead to be negligible. Future work will evaluate how well Codebase Manipulation satisfies this requirement.

Codebase Manipulation architecture limitations. Codebase Manipulation is susceptible to Solstice's design limitations. Solstice detects source code changes through the IDE API; if the source code is changed outside the IDE, Codebase Manipulation will not record these changes immediately. Developers rarely edit outside of their preferred IDE, but to mitigate this limitation, each time the IDE is opened, Codebase Manipulation checks for any changes to the source code that may have taken place and creates an edit containing these external changes. Codebase Manipulation could avoid this limitation by using OS-level file-system listeners to detect changes to the source code. However, this approach would prevent Codebase Manipulation from detecting changes that are not written to the file system, such as unsaved changes in editor buffers. Future work will investigate how these external edits affect information retrieval. Additionally, Solstice detects some developer actions initiated via tools as typing actions, and therefore Codebase Manipulation records them as such. For example, Codebase Manipulation records Eclipse refactorings as a series of text replace operations to the source code. Thus, Codebase Manipulation is complete in its recording, but inherits Solstice's limitations in recognizing how some actions are initiated. Improvements to Solstice would be immediately reflected in Codebase Manipulation.

V. Related Work

The typical way to create development histories is by using version control systems (VCSs), such as Subversion [7], Mercurial [26], and Git [10]. Unlike Codebase Manipulation, these systems are manual and the history they provide has a fixed, typically coarse granularity. Developers may change the filesystem state to earlier snapshots in the history, and may compare the differences between two snapshots, but cannot easily alter the history to suit particular development tasks.

VCSs require the developer to manually create each snapshot. Developers frequently forget to create snapshots, or simply do not know the best time to to so. As a result, the development history is often coarse-grained or incomplete. For example, a single edit may include changes relevant to multiple development tasks, and changes developers make but overwrite before creating a snapshot are lost. This makes VCS histories suboptimal for many analyses or manual inspection. Codebase Manipulation addresses these limitations by automatically recording the history of *all* edits and providing the framework for rewriting this history into custom granularities better suited for development tasks.

Some VCSs allow limited history rewriting [14], [27]. For example, `git rebase` can collapse, expand, move, and remove edits [15]. However, these tools are complex, prevent collaboration because rewriting a shared history prevents subsequent sharing, and are irreversible and lead to further history information loss. By contrast, Codebase Manipulation history transformations are high-level, which hides all internal complexity, reversible, and keep intact the recorded history's integrity to enable collaboration.

Fine-grained version control can simplify merging and improve collaboration [23], [36]. Development histories can also be created automatically by recording developer actions. Fluorite [45] stores fine-grained edits to visualize, replay, and query the development history, and implements fine-grained selective undo [5]. Built on Fluorite, Azurite studies developers' backtracking patterns [46] and also enables selective undo [47]. Azurite also introduces change summarization with *collapse levels* [44]: changes can be displayed at the raw (fine-grained) level, parsable by the compiler level, method level, and type level. Users reported wanting to see changes at higher-levels than the fine granularity, e.g., at the level of the method [44], so these collapse levels, similar to views presented in this paper, are likely to be useful in practice. Changing between these levels is similar to change summarization [19], [38], and tools that summarize changes, or select which changes belong to the same summary (e.g., semantic version history slicing [21]) are complementary to Codebase Manipulation, which enacts collapsing, expanding, or moving changes. Additionally, choice calculus can be used to map features to implementation elements [43], which, again, can select which changes Codebase Manipulation should collapse. CodingSpectator [35] and CodingTracker record and use the fine-grained development history to study refactoring practices [42], development practices [35], and fine-grained change patterns [34]. Storyteller VCS uses the fine-grained history to transfer knowledge from an experienced developer to an inexperienced one [24]. IDE++ [16], [17] maintains a fine-grained development history to improve development by analyzing fine-grained code changes. Each of these tools focuses on particular development tasks or research goals. As a result, these automatically-recorded fine-grained histories are inflexible and only suitable for the tasks that require their particular granularity. By contrast, Codebase Manipulation is applicable to many tasks because it records a flexible history whose granularity can be transformed to match each particular task.

To aid understanding how a history should be rewritten, heuristics can detect related changes to help identify which changes in a large edit may need to be untangled. These heuristics include historical code change patterns [20] and change couplings, data dependencies, and code metrics[18]. These approaches focus on detangling large edits, which is a problem of manually-recorded histories. Meanwhile change distilling can difference changes made in parallel on projects sharing code [8], which can suggest edit patterns. Codebase Manipulation provides access to overwritten changes, potentially improving the effectiveness of these tools.

Visualization is also an important part of history understanding and many repository hosting services (e.g., GitHub and Bitbucket) include visualization tools. Azurite visualizes edits on a timeline at different collapse levels [44] and research has argued that visualizations of changes relevant to bug fixes are useful for understanding the state of development [9].

Development histories simplify some software engineering tasks. For example, git's `annotate` [11] and `blame` [13] commands can help understand the context of an earlier change, and test bisection [12] and delta debugging [48], [49] can help find the cause of a regression failure. However, the history's granularity affects the effectiveness of these tools. Codebase Manipulation is complementary to these tools and can improve their effectiveness by transforming the granularity into one most suitable for the task. Further, because Codebase Manipulation automatically records every developer edit, it can create richer history views of more granularities than is possible with manually-created histories, further improving tool effectiveness.

Mining software repositories research uses development histories to understand development practices [3], [4], [50], to localize bugs [33], [22], [32], [37], [28], [25], and to help collaborative teams work together [1]. However, performing analyses on manually-recorded histories may lead to incorrect conclusions [2]. A history created by recording the edits at each save operation can be used to visualize the development and create development summaries [6] and to study the evolution of students' projects [39]. These repositories are finer-grained and more complete than manually-created ones and research on such repositories has, for example, identified a correlation between static analysis warnings and test failures [40]. The histories created by Codebase Manipulation are finer-grained, richer in terms of containing information about developer actions, and more complete, as they include edits a developer may overwrite before saving a file. This potentially creates better data sets for mining software repositories research.

VI. CONTRIBUTIONS

Development histories are necessary for software engineering tasks, but their inflexible granularity hinders their utility. We have presented Codebase Manipulation to automatically record a fine-grained history of all developer actions and to provide high-level history transformations to rewrite the history's granularity to make it more suitable for specific tasks. We have identified COLLAPSE, EXPAND, and MOVE as three primitive transformations that can be combined to construct powerful high-level history transformations and shown how two such transformations, COLLAPSEBYGROUP and REORDERBYGROUP, can be used to create histories of many useful granularities. Finally, we have designed a Codebase Manipulation architecture that enable it to record a complete history of development, produce easy-to-use history views at multiple granularities, and function unobtrusively, without affective the developer's workflow. Overall, Codebase Manipulation shows promise for automating version control and improving the utility of development histories.

ACKNOWLEDGMENTS

This material is based upon work supported by the United States Air Force under Contract No. FA8750-12-C-0174 and by the National Science Foundation under grants CCF-0963757 and CCF-1453474.

REFERENCES

[1] Christian Bird, Nachiappan Nagappan, Premkumar Devanbu, Harald Gall, and Brendan Murphy. Does distributed development affect software quality? An empirical case study of Windows Vista. In *ICSE*, pages 518–528, 2009.

[2] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. The promises and perils of mining Git. In *MSR*, pages 1–10, 2009.

[3] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Proactive detection of collaboration conflicts. In *ESEC/FSE*, pages 168âĂŞ–178, 2011.

[4] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. Early detection of collaboration conflicts and risks. *IEEE TSE*, 39(10):1358–1375, 2013.

[5] Aaron G. Cass and Chris S. T. Fernandes. Modeling dependencies for cascading selective undo. In *INTERACT*, 2005.

[6] Jacky Chan, Alan Chu, and Elisa Baniassad. Supporting empirical studies by non-intrusive collection and visualization of fine-grained revision history. In *eTX*, pages 60–64, 2007.

[7] Ben Collins-Sussman. The Subversion project: Building a better CVS. *Linux Journal*, 2002(94):3, Feb. 2002.

[8] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE TSE*, 33(11):725–743, 2007.

[9] Daniel M. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[10] Git. http://www.git-scm.com/. Accessed on September 21, 2014.

[11] Git annotate. https://www.kernel.org/pub/software/scm/git/docs/git-annotate.html. Accessed on September 21, 2014.

[12] Git bisect. https://www.kernel.org/pub/software/scm/git/docs/git-bisect.html. Accessed on September 21, 2014.

[13] Git blame. https://www.kernel.org/pub/software/scm/git/docs/git-blame.html. Accessed on September 21, 2014.

[14] Git: History rewriting. http://git-scm.com/book/en/Git-Tools-Rewriting-History. Accessed on September 21, 2014.

[15] Git: Rebase. http://www.git-scm.com/book/en/Git-Branching-Rebasing. Accessed on September 21, 2014.

[16] Zhongxian Gu. Capturing and exploiting fine-grained IDE interactions. In *ICSE*, pages 1630–1631, 2012.

[17] Zhongxian Gu. *Toward Effective Debugging by Capturing and Reusing Knowledge*. PhD thesis, University of California, Davis, CA, USA, 2013.

[18] Kim Herzig and Andreas Zeller. The impact of tangled code changes. In *MSR*, pages 121–130, 2013.

[19] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *ICSE*, pages 309–319, 2009.

[20] Hiroyuki Kirinuki, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. Hey! Are you committing tangled changes? In *ICPC*, pages 262–265, 2014.

[21] Yi Li, Julia Rubin, and Marsha Chechik. Semantic slicing of software version histories. In *ASE*, 2015.

[22] Benjamin Livshits and Thomas Zimmermann. DynaMine: Finding common error patterns by mining software revision histories. In *ESEC/FSE*, pages 296–305, 2005.

[23] Boris Magnusson, Ulf Asklund, and Sten Minör. Fine-grained revision control for collaborative software development. In *FSE*, pages 33–41, 1993.

[24] Mark Mahoney. The Storyteller version control system: Tackling version control, code comments, and team learning. In *SPLASH demonstrations track*, pages 17–18, 2012.

[25] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: Current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[26] Mercurial. http://mercurial.selenic.com/. Accessed on September 21, 2014.

[27] Mercurial: Editing history. http://mercurial.selenic.com/wiki/EditingHistory. Accessed on September 21, 2014.

[28] Ayse Tosun Misirli, Ayse Basar Bener, and Resat Kale. AI-based software defect predictors: Applications and benefits in a case study. *AI Magazine*, 32(2):57–68, 2011.

[29] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. Making offline analyses continuous. In *ESEC/FSE*, pages 323–333, 2013.

[30] Kıvanç Muşlu, Yuriy Brun, Michael D. Ernst, and David Notkin. Reducing feedback delay of software development tools via continuous analyses. *IEEE TSE*, 2015.

[31] Kıvanç Muşlu, Luke Swart, Alain Orbino, Yuriy Brun, Michael D. Ernst, and David Notkin. Solstice. https://bitbucket.org/kivancmuslu/solstice/. Accessed on September 21, 2014.

[32] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE*, pages 452–461, 2006.

[33] Nachiappan Nagappan, Andreas Zeller, Thomas Zimmermann, Kim Herzig, and Brendan Murphy. Change bursts as defect predictors. In *ISSRE*, pages 309–318, 2010.

[34] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E. Johnson. Mining fine-grained code changes to detect unknown change patterns. In *ICSE*, pages 803–813, 2014.

[35] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is it dangerous to use version control histories to study source code evolution? In *ECOOP*, pages 79–103, 2012.

[36] Dirk Ohst and Udo Kelter. A fine-grained version and configuration model in analysis and design. In *ICSM*, pages 521–527, 2002.

[37] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *ISSTA*, pages 86–96, 2004.

[38] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOSPLA*, pages 432–448, 2004.

[39] Jaime Spacco, David Hovemeyer, and William Pugh. An Eclipse-based course project snapshot and submission system. In *eTX*, pages 52–56, 2004.

[40] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh. Software repository mining with Marmoset: An automated programming project snapshot and testing system. In *MSR*, pages 1–5, 2005.

[41] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. A consistency model and supporting schemes for real-time cooperative editing systems. In *Australian Computer Science Conference*, pages 582–591, 1996.

[42] Mohsen Vakilian, Nicholas Chen, Stas Negara, Balaji Ambresh Rajkumar, Brian P. Bailey, and Ralph E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE*, pages 233–243, 2012.

[43] Eric Walkingshaw and Martin Erwig. A calculus for modeling and implementing variation. In *GPCE*, pages 132–140, 2012.

[44] YoungSeok Yoon. *Backtracking Support in Code Editing*. PhD thesis, Carnegie Mellon University, Pittsburg, PA, USA, 2015.

[45] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. In *PLATEAU*, pages 25–30, 2011.

[46] YoungSeok Yoon and Brad A. Myers. A longitudinal study of programmers' backtracking. In *VL/HCC*, pages 101–108, 2014.

[47] YoungSeok Yoon and Brad A. Myers. Supporting selective undo in a code editor. In *ICSE*, 2015.

[48] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *ESEC/FSE*, pages 253–267, 1999.

[49] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE TSE*, 28(2):183–200, 2002.

[50] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *ICSE*, pages 563–572, 2004.