# Identifying additional necessary changes from version control history

Scott Newson

CPSC 503—University of Calgary

**Abstract.** Adding new features to an existing system is a common scenario in software engineering. Often, a series of features will require a similar set of changes across several locations in the source code of the project. When the addition of these features is spread over time and the locations requiring changes are spread over several files in a large project, it is difficult to identify all locations requiring change. In ideal cases, documentation exists to guide the development of these new features, or an individual involved in a previous feature is available for consultation. In less ideal scenarios, such documentation or references are not available and the creator of new features must depend on alternative sources of information—one such source being the version control history of the project.

We propose the creation and study of tooling to help software developers identify additional locations that need changing for an arbitrary feature, based off of identified locations that were changed for a similar features and the project's version control history. Specifically, we assume the developer will have already identified a subset of the changes made for a similar previous feature, and aim to identify additional locations that need to be changed based on the version control history of the project.

## 1 Introduction

Software development involves the process of understanding existing software as well we writing new software. In addition to reading the current source code of a project, this process also includes understanding the history of the source-code—how it has changed over time. A specific case where this temporal aspect becomes relevant is when a software developer needs to identify the set of changes that introduced a new feature or fixed a bug.

Version control systems provide the history of how a software project has been developed, recording snapshots of the state of the project at points in time. These snapshots are often referred to as *commits* or *versions*. These commits usually include some additional metadata besides the changes to the code under management, such as: a free-form text field used for comments and messages, the date and time of the local system when the commit was created, and the commit author's username and email. The repository system tracks the series of commits created and orders the commits such that future users can explore the full version history of the project.

There are a variety of existing version control systems, including: SCCS (Source Code Control System) from 1972, RCS (Revision Control System) from 1982 [10], CVS (Concurrent Versioning System) from 1986, Subversion from 2000, BitKeeper also from 2000, Git from 2005, Mercurial also from 2005, Fossil from 2007, and many more [11]. These systems include both free/open-source and proprietary systems, and range in functionality: local only, client-server, and distributed [8]. Common operations with version control systems include the following: creating new versions (usually appended to the most-recent version), checking out a previous version to review the state of the files at that point in time, creating branches in the history so that two different paths of development can explored in parallel, and merging branches to incorporate changes from both histories into a single version.

We believe there is room for improvement in the tooling available for identifying the full set of changes involved in features that touch many files in a large project. Although in an ideal setting these features would be introduced in a single commit or branch, in practice they are often spread across many commits. Especially in an agile development team where feature development undergoes continuous integration leading to the commits for a large feature being interleaved with commits for unrelated features.

## 2 Previous Work

An existing workflow provided by Git as well as other version control systems is to annotate the lines of a source file with the commits that most recently changed each line [5]. This provides the user with a line-by-line breakdown of the provenance of the contents of a file at any given commit. There is a know problem with this approach where a single character change, sometimes including whitespace changes, counts equally to rewriting the entire line.

Previous work has identified automated ways of collecting version control history [7], compared different types of version control systems [4], and explored the metadata of changes preserved by version control systems [3].

Academic work has also identified that tasks get inappropriately partitioned over multiple commits [2], and that data-mining techniques on version control histories can identify files and functions that tend to change at the same time [12, 13]. These latter approaches focus on associating changes within commits, instead of finding additional commits.

## 3 Proposed Solution

To support the process of finding the full set of changes required to implement a large feature that touches many files, we propose building a tool that identifies a more complete set of changes from a known subset of the changes. We will approach this by building a tool that takes as input a version reference and a set of lines of source code files, and returns the set of commits that introduced all of those lines.

We choose to work with Git as it is a modern, open-source, version control system that is used in industry and open source software development [6]. In addition Git has been used outside of software development tasks in other branches of scientific work [1, 9]. Building on top of Git will be facilitated by its open source nature, allowing for the possible incorporation of a successful useful tool.

Once the feasibility of such a tool has been demonstrated with an initial implementation, we will explore possible improvements including:

1. identifying commits that happened in temporal proximity as those commits based on the commit date.
2. identifying commits that are proximal to those commits based on commit relationships. Possibly similar to, but not necessarily the same as 1.
3. identifying proximal commits (of either form 1 or 2) by other attributes such as commit message sub-strings or commit author.
4. identifying larger structures such as branches which contain all of the identified commits.

Finally, we can test the effectiveness of this tool on existing software repositories to demonstrate benefits and identify areas of future improvement. An important question to answer at this stage will be how best to present the output of the tool, possibly dependant on the total number and proximity of commits identified for output.

## 4 Timeline

- Oct 4 - Proposal completed and delivered
- Oct 18 - Initial version of tool built
- Nov 1 - Additional extensions to the tool explored
- Nov 15 - Tool used on existing code bases with example workflows
- Nov 29 - Initial draft of final paper and presentation complete
- Last week of classes - Final presentation
- Dec 7 (Last day of classes) - Final paper completed and delivered

## 5 Conclusion

Based on the existence of large, complex software projects and imperfect version control histories, we believe it will be possible to aid in the identification of necessary project touch-points for new features. We expect positive results from taking advantage of version control records in order to identify where known changes were introduced and to suggest likely locations for additional necessary changes. Challenges will include scoping the meaning of a identifying consistent patterns of changes that identify a single feature.

## References


1. A practical guide for improving transparency and reproducibility in neuroimaging research.
2. Ryo Arima, Yoshiki Higo, and Shinji Kusumoto. A study on inappropriately partitioned commits: how much and what kinds of ip commits in java projects? In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 336–340. ACM, 2018.
3. Thomas Ball, Jung-Min Kim, Adam A Porter, and Harvey P Siy. If your version control system could talk. In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, volume 11, 1997.
4. 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 *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 1–10. IEEE, 2009.
5. Git community. Git-blame — git documentation. `https://git-scm.com/docs/git-blame`, 2018. [Online; accessed 2-October-2018].
6. Linux contributors. Linux github repository. `https://github.com/torvalds/linux`, 2018. [Online; accessed 2-October-2018].
7. Audris Mockus. Amassing and indexing a large sample of version control systems: Towards the census of public source code history. 2009.
8. Stefan Otte. Version control systems. *Computer Systems and Telematics*, 2009.
9. Karthik Ram. Git can facilitate greater reproducibility and increased transparency in science. *Source code for biology and medicine*, 8(1):7, 2013.
10. Walter F Tichy. Rcsa system for version control. *Software: Practice and Experience*, 15(7):637–654, 1985.
11. Wikipedia contributors. Version control — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Version_control&oldid=861419886`, 2018. [Online; accessed 2-October-2018].
12. Annie TT Ying, Gail C Murphy, Raymond Ng, and Mark C Chu-Carroll. Predicting source code changes by mining change history. *IEEE transactions on Software Engineering*, 30(9):574–586, 2004.
13. Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.