Integrating Source Code Search into Git Client for Effective Retrieving of Change History

Miwa Sasaki, Shinsuke Matsumoto, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
{m-sasaki, shinsuke, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—In order to achieve effective development management, it is important to manipulate and understand the change histories of source code in a repository. Although general version control systems provide change history manipulation, these systems are restricted to line-based and textual operations such as grep and diff. As such, these systems cannot follow the syntax/semantics of the source code. While various studies have examined querying and searching source codes, these methods cannot follow historical changes. The key concept of this paper is the integration of a source code search technique into Git commands that manipulate historical data in a repository. This paper presents MJgit, a prototype tool for achieving the above goal. In order to evaluate the proposed tool, we conducted a performance experiment using actual software repositories.**

***Index Terms*—Code change history, source code search, MJgit, Git, abstract syntax tree**

I. Introduction

In previous decades, version control systems, such as Git and SVN, have been widely used for most software development projects [1]. In order to achieve effective development management, it is important to manipulate and understand the change histories of source code in a repository [2]. The change history helps to answer *why the code was changed* [3], *when the bug was introduced* [4], and *who should be assigned to the bug* [5].

Although current version control systems generally support history manipulation, these systems are restricted to *line-based* and *textual* operations, such as grep and diff [6]. These operations have been designed as general-purpose utilities, which provide powerful pattern matching based on regular expressions [7]. However, because of their general versatility, the operations are not specialized to follow syntax/semantics of source code.

In many programming languages, a single source code file is composed of not only execution statements, but also various types of accompanying information (e.g., comments, annotations, and copyrights). Some of this information is described by natural language. Imagine that a developer looks for `if` statements using line-based operations to check for bug propagation [8] caused by code clones [9]. Then, the developer may encounter several unnecessary lines in which "`if`" occurs within comments.

In order to solve this problem, various studies related to searching and querying source code have been conducted [10]–[18]. These approaches can handle a given query by traversing an abstract syntax tree and/or a control flow graph. Thus, these approaches are advantageous for finding parts of source code (e.g., methods and statements) using syntax/semantic information beyond textual representation.

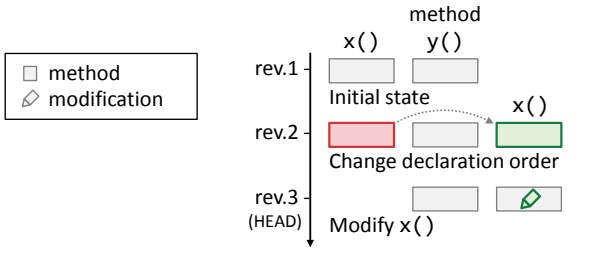
However, the advantage is available only for source code at a specific revision. In other words, these approaches cannot track historical changes. Few studies have considered fine-grained source code search from software repositories [19], [20]. The focus of these studies has been scalability for analyzing numerous software repositories. Although Historage [21] provides fine-grained change analysis of source code, it requires significant re-construction of an existing repository. In contrast, our focus is *retrieving a change history from his/her original repository by querying a specific syntax/semantic information beyond textual representation*.

The key concept of this paper is the integration of a source code search capability into several Git commands that manipulate historical data (e.g., diff, log, and blame). We believe that the integration may be highly valuable because code searching provides fine-grained filtering capabilities, and the Git command complements filtering by meta-information (e.g., who, when, and why). The concrete example of the combination is follows.

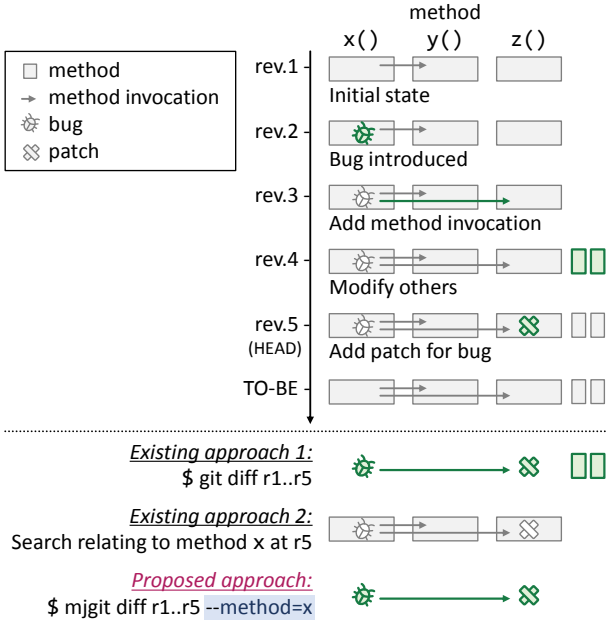
```
$ git log  --method=x --author=miwa
$ git log  --method=x --since=2018-01-01
$ git log  --method=x --grep='fix for'
$ git diff --method=x revA..revB
```

The `method` option is a proposed (and simplified) query. Git supports `author`, `since`, and `grep` queries by default. Each query is aimed at *who*, *when* and *why*. In all cases, we can confirm code evolution together with a focus on method `x` while specifying meta-information.

The goal of this study is to provide effective retrieving of change history from a software repository by realizing the above concept. In this paper, we present MJgit, a prototype tool for achieving the above goal. The proposed tool can narrow the change history of Java files based on a specified method name or variable name. MJgit is backward compatible with the standard Git command because MJgit behaves the same when extended query is not specified. In order to evaluate MJgit, we conducted a performance experiment using actual software repositories.



(a) Attempt to understand the change history of method x



(b) Attempt to fix the bug and remove the temporary patch

Fig. 1: Motivating examples

II. Motivating Example

First, we introduce motivating examples of our work. Two examples are illustrated in Figure 1, in which the change histories of two or three methods are shown vertically.

A. Understanding the Change History of Method x

Figure 1(a) shows the histories of two methods, `x` and `y`. At revision 2, a declaration of method `x` was moved to the bottom of the file (i.e., after method `y`). In addition, method `x` was modified at revision 3. Here, a developer is trying to understand the history of method `x`.

Existing approach: As a simple method by which to confirm a history, the git-diff command can be used without any additional installations, except for the Git client itself. However, these text-based commands cannot consider source code structure or flow. Thus, these commands deliver many unnecessary results when non-functional and extensive modifications such as code formatting and method sorting are made. In this case, method `x` appears to have been removed and added, although it was only moved. The essential modification, shown as a pen, is barely noticeable.

Proposed approach: The key idea is to combine Git commands and code searching for efficient retrieval of the code change history. This enables the developer to specify syntax/semantic information to a repository. The bottom example of the figure shows the usage and results of the proposed tool MJgit. In this case, superficial and non-essential changes can be filtered by specifying the method name, `x`.

B. Fixing the Bug and Removing the Temporary Patch

Figure 1(b) illustrates a more practical and complicated situation. At revision 2, a bug was introduced into method `x`. The buggy method additionally invoked method `z` at revision 3. After some modifications were made at revision 4, a patch for the bug was applied to method `z` at revision 5. This patch is a temporary solution, such as null checking for a parameter passed from method `x`. Here, a developer found the bug and is trying to fix it. In addition, the temporary patch must also be removed.

Existing approach 1: Using the git-diff command, *textual differences between revisions 1 and 4* can be easily gathered. Similar to the first example, this command faces an obstacle caused by unrelated modifications by revision 4. Although it is generally recommended to keep commits small and atomic [22], we encounter numerous tangled changes [23]. The developer may need to find the required information from several such results. In consideration of the source code structure, we need a more specialized search operation.

Existing approach 2: In such a situation, code search [12]–[18] and change impact analysis [24]–[26] are well known as effective approaches. The impact analysis identifies potential effects of changes by tracing dependencies based on a syntax tree and/or flow graph. The developer can find *methods related only to method `x`* by applying the methods to revision 5. However, these techniques have a limitation in that they do not support historical changes. As such, the result includes method `y`, which has not been changed in the timeline. This information may become "noise" if the method is sufficiently mature (i.e., receives no attention from the developer).

Proposed approach: Using the proposed tool, the developer can grasp *textual differences related only to method `x` between revisions 1 and 5*. The bug and its patch will be easily identified and maintained.

III. Proposed Tool: MJgit

A. Overview

The purpose of MJgit is to achieve effective retrieval of code change history by integrating code search capability into the Git command. An overview of MJgit is shown in Figure 2. This figure illustrates a case in which the diff command and a structure-based (i.e., abstract-syntax-tree based) code search are used. The detailed process flow is as follows:

1. First, a developer executes a diff command in order to reveal the difference between revisions A and B. The

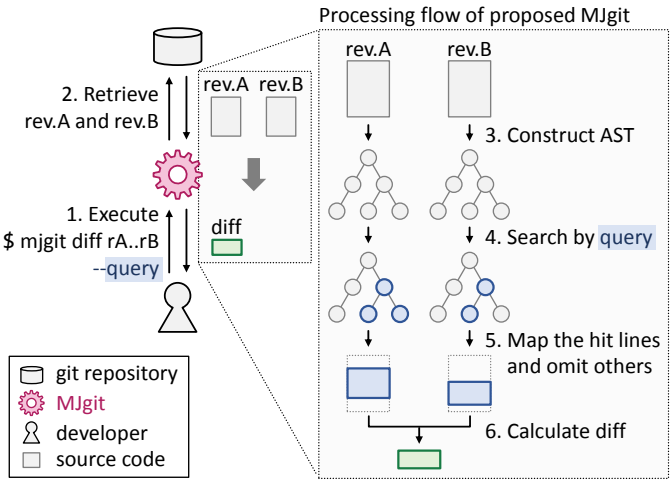


Fig. 2: Overview of MJgit

developer also specifies a code search query that is available only on MJgit.

2. MJgit attempts to retrieve both source code revisions. Then, in the case of the standard diff command, MJgit calculates textual diff using the longest common subsequence algorithm [7]. MJgit behaves the same whether or not the code query is specified. Thus, MJgit is backward compatible with the standard Git command. When a code query is given, MJgit performs the following additional processing.
3. The tool constructs ASTs from both revisions. Note that the AST construction is performed only for Java files which were modified at revision B.
4. The tool searches by the specified query for the constructed AST.
5. Statements, hit by the query, are mapped to raw source code lines, and the other lines are omitted. This mapping allows the use of a powerful and familiar diff command for difference calculation between two revisions. Furthermore, although textual representations, including coding styles, can be included in the result, they are omitted in AST.
6. Finally, the standard diff algorithm is applied to the masked code.

B. Code Search Technique

There are several techniques to retrieve a certain program snippets or statements from a single source code [10], [11], [26]. In Figure 2, AST-based program slicing is illustrated as a demonstration. Similarly, many approaches have been proposed to search a certain source code file from a database that stores numerous source code files [19], [20]. Both approaches are similar, but have slightly different purpose and characteristics. In the proposed MJgit, both approaches can be applied as a code search component.

C. Extended Git Command

Git supports several methods and commands to manage a Git repository. The proposed concept can be applied to *retrospective* and *retrieval* commands. Concrete extended commands are listed in Table I. Each of these commands retrieves logs or source code files from one or more revisions, which means that the commands support several queries with revision information, such as `since` and `author`. These commands have good compatibility with code searching. By focusing only on *retrieval* commands, the proposed tool has no side effects on the repository itself.

D. Prototype Implementation

MJgit is implemented as a prototype by extending a Java-based open-source Git client, jgit[1]. The tool currently supports a simple search capability that can specify a method and a variable name. The code search is enabled by specifying `method` and `variable` parameters. For example, when `--method=x` is given, the tool searches `x`'s declaration statement and invocation statements. The other statements are omitted. Furthermore, MJgit supports `exec-statement`, `comment`, `annotation`, and `javadoc` parameters, which can be used to retrieve only the specified type of program statements. For example, we can reveal the differences of only execution statements using the `exec-statement` parameter. When the extended query is not given, the tool has the same behavior as a standard Git client. If AST cannot be constructed (i.e., compile errors exist), the specified query is ignored.

The extended query is available only for git-diff and git-show commands. Further command extension is an important subject for future work.

IV. Performance Evaluation

A. Purpose and Experimental Settings

The purpose of the experiment is to show whether the execution performance of MJgit is practical. The execution time of the git-diff command with extended query using MJgit was compared with that of the original jgit. The diff command was performed for all pairs of adjacent revisions of JUnit4 and Log4j.

The target projects are summarized in Table II. Both projects include over 1,000 revisions and continue over 15 years. We filtered revisions in which no Java files were changed because the proposed tool does not affect such situations. Similarly, if a revision includes a compile error, the revision was filtered.

[1]https://eclipse.org/jgit/

TABLE I: Extended git commands

Name	Summary	# revs processed
git-diff	shows changes between revisions	2
git-show	shows detail of revision	1
git-log	shows revision logs	$\geq$1
git-blame	shows who last modified	$\geq$1

MJgit supports various combinations of code search queries. In this experiment, three commands shown in Table III were compared. They are labeled as *jgit*, mj_{method}, and $mj_{exec\text{-}stmt}$, respectively. The mj_{method} shows differences of method `main` only, and the $mj_{exec\text{-}stmt}$ shows differences of execution statements only (i.e., comments, annotations, and JavaDoc are omitted).

B. Results

The experimental results are shown in Figure 3. The x-axis indicates the execution time, and the three boxplots correspond to the compared diff commands. A small percent of outliers, which exceed five seconds, are omitted from the figure.

For the original jgit, all diff commands were accomplished within two seconds. The median execution time was approximately 0.8 seconds. This result can be regarded as a baseline. In both cases in which MJgit were applied to the diff command (mj_{method} and $mj_{exec\text{-}stmt}$), the execution time increased significantly to 1 to 5 seconds. The average increase of execution time was 3.1. There is no difference between mj_{method} and $mj_{exec\text{-}stmt}$. In addition, both projects have approximately the same tendency.

In order to determine the reason for the performance degradation, we examined the relationship between the rate of increase of the execution time and the number of Java files. The results are shown in Figure 4. Each plot in the figure represents a single revision. The x-axis shows the number of java files committed in the revision, and the y-axis indicates the performance reduction rate.

The correlation coefficients were 0.57 and 0.77 for mj_{method} and $mj_{exec\text{-}stmt}$, which are significant. In other words, the execution performance is reduced by increasing the number of Java files.

C. Discussion

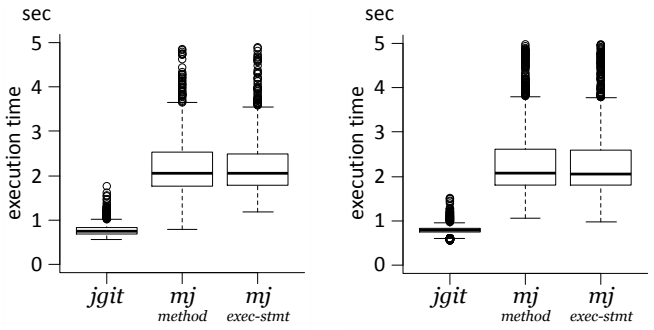
The performance of MJgit is considered to be practical because, in most cases, the git-diff command was executed within 4 seconds. However, performance improvement is an important challenge for MJgit. Although the git-diff command considers only two revisions, the git-log command considers one or more revisions. The performance may be drastically decreased for the git-log command, which is commonly used and convenient to grasp code evolution.

TABLE II: Summary of target projects

	JUnit4	Log4j
Since	Dec. 2000	Nov. 2000
# total revisions	1,801	3,274
# compared pairs of revisions	900	1,891
# Java files at latest revision	443	309

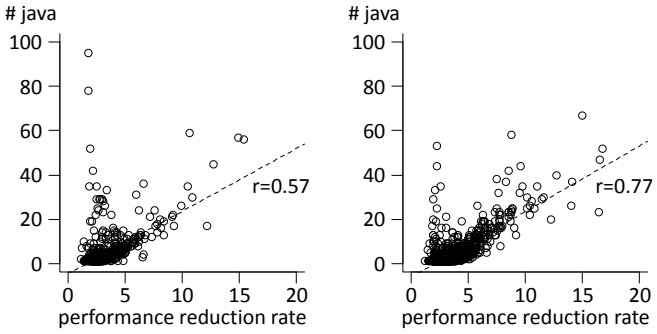
TABLE III: Compared three commands

Label	Actual executed command
jgit	`$ jgit diff A..B`
mj_{method}	`$ mjgit diff A..B --method=main`
$mj_{exec\text{-}stmt}$	`$ mjgit diff A..B --exec-statement`



(a) JUnit4 (b) Log4j

Fig. 3: Execution times for compared three commands



(a) JUnit4 (b) Log4j

Fig. 4: Relationship between performance reduction rate and number of java files

Since there is no difference between the mj_{method} and $mj_{exec\text{-}stmt}$, a dominant factor of the performance reduction is not code searching, but rather AST construction. In order to achieve the improvement, the concept of *tree versioning* [27]–[29] can be applied. Tree-structured representations (e.g., AST) can be stored in a version control repository. By using these technique, the performance reduction caused by AST construction may be solved.

V. Conclusion

In this paper, we proposed MJgit to achieve effective retrieval of code change history by integrating code search into the Git command. Based on the experimental results, the execution performance of MJgit is practical in most cases involving small commits.

The current MJgit has a limited capability for code search. In the future, we intend to introduce advanced search queries which allows efficient retrieval of code evolution. Extending other Git commands, especially the Git log command, is also an important challenge. Furthermore, we need to evaluate the effectiveness of change history retrieval by conducting semi-structured interview or controlled experiment with practitioners.

Acknowledgment

This work was supported by JSPS/MEXT KAKENHI Grant Number JP25220003 and JP26730155.

REFERENCES

[1] B. D. Alwis and J. Sillito, "Why are software projects moving from centralized to decentralized version control systems?" in *Proc. Workshop on Cooperative and Human Aspects on Software Engineering*, 2009, pp. 36–39.

[2] X. Sun, B. Li, H. Leung, B. Li, and Y. Li, "MSR4SM: Using topic models to effectively mining software repositories for software maintenance tasks," *Information and Software Technology*, vol. 66, pp. 1–12, 2015.

[3] S. Rastkar and G. C. Murphy, "Why did this code change?" in *Proc. International Conference on Software Engineering*, 2013, pp. 1193–1196.

[4] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proc. International Conference on Automated Software Engineering*, 2006, pp. 81–90.

[5] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proc. International Conference on Software Engineering*, 2006, pp. 361–370.

[6] B. W. Kernighan and J. R. Mashey, "The unix programming environment," *Software: Practice and Experience*, vol. 9, no. 1, pp. 1–15, 1979.

[7] J. W. Hunt and T. G. Szymanski, "A fast algorithm for computing longest common subsequences," *Communicaitons of the ACM*, vol. 20, no. 5, pp. 350–353, 1977.

[8] M. Mondal, C. K. Roy, and K. A. Schneider, "Bug propagation through code cloning: An empirical study," in *Proc. International Conference on Software Maintenance and Evolution*, 2017, pp. 227–237.

[9] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[10] M. Weiser, "Program slicing," in *Proc. International Conference on Software Engineering*, 1981, pp. 439–449.

[11] D. Binkley and M. Harman, "A large-scale empirical study of forward and backward static slice size and context sensitivity," in *Proc. International Conference on Software Maintenance*, 2003, pp. 44–53.

[12] S. Paul and A. Prakash, "A framework for source code search using program patterns," *Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, 1994.

[13] R. G. Urma and A. Mycroft, "Source-code queries with graph databases – with application to programming language usage and evolution," *Science of Computer Programming*, vol. 97, no. Part 1, pp. 127–134, 2015.

[14] C. D. Roover, C. Noguera, A. Kellens, and V. Jonckers, "The SOUL tool suite for querying programs in symbiosis with eclipse," in *Proc. International Conference on Principles and Practice of Programming in Java*, 2011, pp. 71–80.

[15] O. de Moor, M. Verbaere, E. Hajiyev, P. Avgustinov, T. Ekman, N. Ongkingco, D. Sereni, and J. Tibble, "Keynote address: .QL for source code analysis," in *Proc. International Working Conference on Source Code Analysis and Manipulation*, 2007, pp. 3–16.

[16] T. Cohen, J. Gil, and I. Maman, "JTL: The java tools language," in *Proc. International Conference on Object-oriented Programming Systems, Languages, and Applications*, 2006, pp. 89–108.

[17] K. D. Volder, "Jquery: A generic code browser with a declarative configuration language," in *Proc. International Symposium on Practical Aspects of Declarative Languages*, 2006, pp. 88–102.

[18] M. Kimmig, M. Monperrus, and M. Mezini, "Querying source code with natural language," in *Proc. International Conference on Automated Software Engineering*, 2011, pp. 376–379.

[19] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An infrastructure for large-scale collection and analysis of open-source code," *Science of Computer Programming*, vol. 79, pp. 241–259, 2014.

[20] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proc. International Conference on Software Engineering*, 2013, pp. 422–431.

[21] H. Hata, O. Mizuno, and T. Kikuno, "Historage: Fine-grained version control system for java," in *Proc. International Workshop on Principles of Software Evolution and the annual ERCIM Workshop on Software Evolution*, 2011, pp. 96–100.

[22] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.

[23] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Proc. Working Conference on Mining Software Repositories*, 2013, pp. 121–130.

[24] R. S. Arnold, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[25] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley, "Chianti: A tool for change impact analysis of java programs," in *Proc. International Conference on Object-oriented Programming, Systems, Languages, and Applications*, 2004, pp. 432–448.

[26] M. Acharya and B. Robinson, "Practical change impact analysis based on static program slicing for industrial software systems," in *Proc. International Conference on Software Engineering*, 2011, pp. 746–755.

[27] T. N. Nguyen, E. V. Munson, and J. T. Boyland, "The molhado hypertext versioning system," in *Proc. Conference on Hypertext and Hypermedia*, 2004, pp. 185–194.

[28] T. T. Nguyen, H. A. Nguyen, N. H. Pham, and T. N. Nguyen, "Operation-based, fine-grained version control model for tree-based representation," in *Proc. International Conference on Fundamental Approaches to Software Engineering*, 2010, pp. 74–90.

[29] D. Asenov, B. Guenat, P. Müller, and M. Otth, "Precise version control of trees with line-based version control systems," in *Proc. International Conference on Fundamental Approaches to Software Engineering*, 2017, pp. 152–169.