2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Scalable Software Merging Studies with MERGANSER

Moein Owhadi-Kareshk
University of Alberta
Edmonton, AB, Canada
owhadika@ualberta.ca

Sarah Nadi
University of Alberta
Edmonton, AB, Canada
nadi@ualberta.ca

***Abstract*—Software merging researchers constantly need empirical data of real-world merge scenarios to analyze. Such data is currently extracted through individual and isolated efforts, often with non-systematically designed scripts that may not easily scale to large studies. This hinders replication and proper comparison of results. In this paper, we introduce MERGANSER, a scalable and easy-to-use tool for extracting and analyzing merge scenarios in Git repositories. In addition to extracting basic information about merge scenarios from Git history, our tool also replays each merge to detect conflicts and stores the corresponding information of conflicting files and regions. We design a normalized and extensible SQL data schema to store the information of the analyzed repositories, merge scenarios and involved commits, and merge replays and conflicts. By running only one command, our proposed tool clones the target repositories, detects their merge scenarios, and stores their information in a SQL database. MERGANSER is written in Python and released under the MIT license. In this tool paper, we describe MERGANSER's architecture and provide guidance for its usage in practice.**

***Index Terms*—Collaborative Software Development, MERGANSER, Scalable Tool, Software Merging.**

I. INTRODUCTION

Today's software industry relies on collaborative development. Tools such as Git and GitHub help developers to store the development history, share software artifacts, and collaborate with each other [1]–[3]. Using these tools, developers can work on the same codebase simultaneously. Branching is an important feature of Git which allows developers to have independent versions of the code, other than the main branch (usually *master*). This way, at least one code branch remains stable and developers add new features and fix bugs on the other ones. Once a branch is ready, developers merge it into the master branch. This type of merging is called *direct merging*. Another type of merging can happen through *pull requests* (PRs) on GitHub, or other social coding platforms, where developers explicitly indicate that they want to merge their changes from a given fork or branch into a target repository.

We use the term *merge scenario* to describe both of the above cases. We focus only on merge scenarios that integrate two branches. Such merge scenarios are represented by a *merge commit* that has two *merge parents* in the Git history. Additionally, there is a *common ancestor* that marks the point at which the history diverged. When the same line of code is changed simultaneously by the two branches, Git cannot decide which change to use and reports a *merge conflict*. Developers typically resolve these conflicts manually, which is an error-prone and time-consuming task [4], [5].

Researchers study merge scenarios in order to design better development tools or to introduce better practices to reduce the number of conflicts. Empirical studies on software merging can also shed light on the characteristics of current software development practices and ways of improving them. There is a lot of previous work that analyzes the performance and functionality of different merging techniques [6], [7], predicts merge conflicts [8], [9], detects conflicts early [10]–[12], analyzes the merging status of PRs [13], [14], or studies the code review process associated with PRs [15]–[17].

To conduct any of the above work, researchers need a reliable and scalable tool to extract relevant information from merge scenarios. To the best of our knowledge, there is no existing scalable and extensible tool for extracting relevant information from merge scenarios to facilitate the study of collaborative software development. The above previous studies usually developed their own tools for extracting merge scenarios, along with their different characteristics, from version control history. Such individual efforts are typically customized for the goal of the study and thus are not easily reusable or may not scale well for large studies. Additionally, since such a mining tool is a means to an end, rather than the main contribution of the work, the tool may not be well-tested or designed for others to use it. Moreover, each study may have its own predefined assumptions about the criteria for choosing repositories to study and may differ in the technical details and assumptions used to extract the data. All of the above means that every researcher that starts working on the software merging problem needs to re-invent the wheel every time, and worse, the results of the work in the literature may not be easily comparable and reproducible.

To address the above challenges, we propose MERGANSER, a scalable tool that extracts merge scenarios and merge conflicts data from Git repositories, and stores this data in a normalized SQL database. Given a list of repositories, MERGANSER (1) clones the repositories, (2) extracts repository meta-data such as the number of stars and forks, (3) detects merge scenarios, (4) extracts the information about the merge scenarios such as the number of developers involved or development duration, (5) replays the merge scenarios to

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00084

detect conflicting files and regions, and (6) stores all the above information in a SQL database. MERGANSER can analyze multiple repositories in parallel, using as many CPU cores as the user specifies. We designed the data schema to be easily extensible, such that extracting any new features in the future is easy. MERGANSER is fully documented and open-sourced under an `MIT` license[1].

II. Related Tools

Over the last couple of decades, several tools were proposed for mining software repositories, especially for Git and GitHub repositories. However, to the best of our knowledge, there is no off-the-shelf tool that focuses on software merging. Boa [18] is a tool, with an accompanying language, for running large-scale queries on data from GitHub and SourceForge. However, it queries snapshots of these websites, rather than real-time data. GHTorrent [19] is an offline mirror of GitHub that allows users to either download the data as a SQL or MongoDB database, or run their queries online. PyDriller [20] is a recent tool for analyzing Git history to extract data such as commits, developers, source code, etc. GitMiner [21] is an open-source tool that stores data extracted from Git and GitHub in a database. Although both PyDriller and GitMiner can be used to detect merge commits by selecting commits with more than one parent, neither provide any additional option to analyze merge scenarios or merge conflicts. GrimoireLab [22] is an industrial-level tool that is capable of gathering data from version control systems, issue trackers, mailing lists, wikis, but it does not contain any tooling for analyzing merge scenarios. Some papers that study software merging (e.g., [7]) release the tool used for mining the merge data. However, the tool is specific to the study and cannot be directly employed for general software merging research due to lack of scalability and covering only a limited number of merge-scenario features extracted for the specific study.

III. MERGANSER Overview

The goal of MERGANSER is to extract all relevant information about merge scenarios found in a repository's Git history, such that it can later be used for various studies on software merging and collaborative software development. Given a set of GitHub repositories, MERGANSER extracts all the relevant information of the merge scenarios from their histories and stores it according to the data schema shown in Figure 1. We selected the list of the features to extract from a merge scenario (i.e., those stored in the schema in Figure 1) based on analyzing the kind of information previous studies on software merging typically need [9], [10], [13], [23]. Note that MERGANSER users have the option of extracting only a subset of tables or fields to avoid unnecessary computation by focusing only on the fields they are interested in.

The first step of the mining process with MERGANSER is to determine the list of repositories to be analyzed. Our tool supports two ways of doing this. First, if the user already has a list of repositories in mind, she can simply feed that list to MERGANSER. However, sometimes researchers want to analyze repositories that match specific characteristics, e.g., repositories with more than 100 stars, and do not have an explicit list of repositories in mind. Our tool can (1) receive a list of criteria, (2) search GitHub for repositories that satisfy these criteria using GitHub's search API, and (3) store the list of repositories.

Second, given a list of repositories, MERGANSER extracts the relevant information to store in the database. In Table I, we summarize the description of each table in our schema and the tool(s) we use to extract the corresponding fields in the table. The exact details of the tools and commands we use are available in MERGANSER's online documentation.

In general, given the list of repositories, MERGANSER extracts the information of each repository such as their popularity metrics (i.e. the number of stars, forks, watches) and their description using the GitHub API. Then, to gather the necessary data about merge scenarios, MERGANSER first clones the repositories locally and then detects merge scenarios using Git commands. We consider any commit with two parents as a merge scenario[2]. MERGANSER then replays each merge commit to detect the conflicting files and regions. Finally, it stores all the extracted information in a SQL database, according to the schema in Figure 1. We provide all information the user needs to run MERGANSER in its online documentation.

One of the advantages of MERGANSER is that it is easily configurable. A user can choose the exact information they wish to extract, using the various flags that control the tool. Table II describes all the currently supported flags. For example, if the user is interested in checking whether the resolution Git created compiles successfully, they would specify the `-c` flag. If studying such *syntactic conflicts* [24] is not a goal of the researcher's study, then MERGANSER will simply skip this step and save the execution time and resources needed to compile the merge resolution.

Note that, as shown in Table I, some of the extracted features are language-independent while others are currently limited to certain programming languages. For example, for checking the compilation status of the resolution, we currently support only Java repositories that use Maven. There are no conceptual limitations for adding other build systems for Java or other programming languages; it simply requires additional engineering effort that we plan to add in the future.

IV. Practical Usage of MERGANSER

We already used MERGANSER to collect data from $267,657$ merge scenarios from 744 open-source GitHub repositories in seven programming languages. For these merge

[1]https://github.com/ualberta-smr/merganser

[2]While merge commits can have more than two parents, the ones with two parents represent the more typical scenario and the focus of software merging studies, which is why we focus on them in MERGANSER. However, supporting n-way, or *octopus*, merges can easily be supported in the future. Our detection strategy also means we miss rebased merge scenarios since rebasing creates a linear history. However, there is currently no accurate technique for detecting merge scenarios that have been rebased.

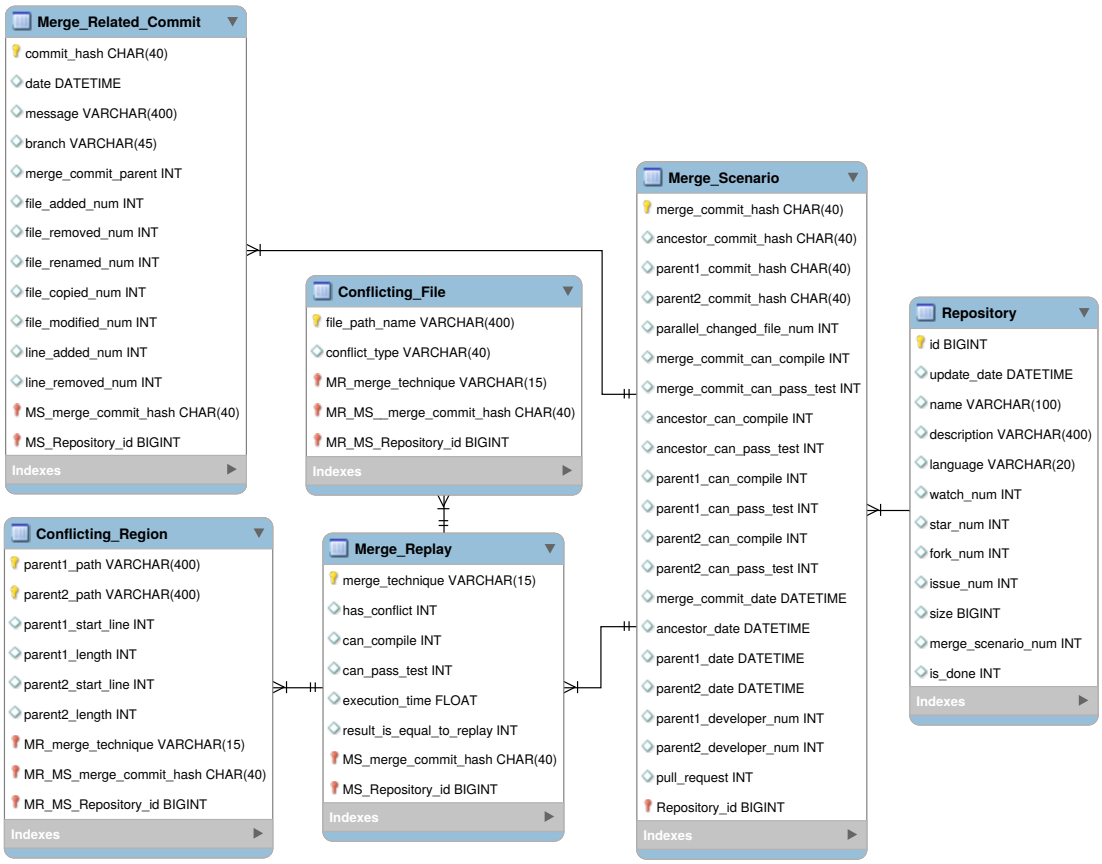

Fig. 1. The MERGANSER Data Schema

TABLE I
THE LIST OF TABLES IN THE DATA SCHEMA OF MERGANSER

No.	Table Name	Description	Extraction Method
1	`Repository`	Data of analyzed repositories including the analysis date, name, description, programming language, popularity measures such as the number of stars, size, the total number of merge scenarios in this repository, and whether the analysis is done yet.	GitHub API Git commands
2	`Merge_Scenario`	Data of merge scenarios including the SHA-1 and the date the ancestor, parents, and the merge commit, whether these commits can compile and pass the tests, the number of simultaneously changed files in two parents, the number of active developers in each parent, and whether the merge is a pull-request or a normal merge by using the commit message. To check whether the code can compile or pass the tests, we use Maven and therefore, these two specific fields can be extracted for only Java repositories at the moment.	Git commands Maven
3	`Merge_Related_Commit`	Data of all commits that are involved in the merge scenario, including the SHA-1, date, commit message, branch name, the parent (the first one or the second one), and the number of edited files and lines.	Git commands
4	`Merge_Replay`	MERGANSER replays merge scenarios to detect conflicts and stores their characteristics. This table stores whether the merge scenario has any conflict, whether the involved commits (merge commit, the ancestor, and the parents) compile and pass the test suite, the execution time of replaying, and whether the replayed version for automated resolutions is equal to the committed resolution.	Git commands Maven
5	`Conflicting_File`	Data of all files that have conflicts, including their relative path and the type of conflict reported by Git (e.g., content conflict vs. delete/modify).	Git commands
6	`Conflicting_Region`	Data of conflicting regions, including the paths of the two parent files and the location of the conflict region, represented as the start line and length of the region.	Git commands

scenarios, we populated the tables shown in Figure I, which would have not been possible without the scalability of MERGANSER. In this section, we provide example SQL queries to demonstrate how a user can make use of the data collected

TABLE II
THE LIST OF PARAMETERS FOR USING MERGANSER

No.	Parameter	Description	Affected Tables
1	`-r`	The list name of GitHub repositories, located in `./working_dir/repository_list`	All Tables
2	`-c`	If set, the repository will be compiled after a successful merge to check if the merge introduced a compilation problem	`Merge_Replay`
3	`-t`	If set, the repository test suite will be run after a successful merge to check if the merge introduced a semantic problem	`Merge_Replay`
4	`-cf`	If set, the information of each conflicting file is stored	`Conflicting_File`
5	`-cr`	If set, the information of each conflicting region is stored	`Conflicting_Region`
6	`-rc`	If set, the replays and merge commits are compared to check if the developer changed the Git's resolution before the merge commit	`Merge_Replay`
7	`-cd`	If set, the information of all involved commits in merge scenarios is stored	`Merge_Related_Commit`
8	`-cores`	The number of threads that MERGANSER uses (the default is using all available CPU cores)	None
9	`-sd`	Specify the time window of merge scenarios to analyze	All

after running MERGANSER.

a) Simultaneously Changed Files: To extract the number of files that are edited in two branches in parallel, the user can run the following query:

```
SELECT merge_commit_hash, parallel_changed_file_num
FROM Merge_Data.Merge_Scenario
```

b) The number of commits: The following query extracts the number of involved commits in merge scenarios in Java:

```
SELECT merge_scenario.merge_commit_hash, COUNT(
    commits.commit_hash)
FROM Merge_Data.Repository as repository
JOIN Merge_Data.Merge_Scenario as merge_scenario
    ON repository.id = merge_scenario.Repository_id
JOIN Merge_Data.Merge_Related_Commit as commits
    ON merge_scenario.merge_commit_hash = commits.
    MS_merge_commit_hash
WHERE repository.language = 'java'
GROUP BY merge_scenario.merge_commit_hash
```

c) The number of added/removed lines: To extract the difference between the number of lines that are added and deleted in two branches:

```
SELECT merge_scenario.merge_commit_hash,
        SUM(commits.line_added_num * IF(commits.
    merge_commit_parent=2,1,-1)) AS 'line_added',
        SUM(commits.line_removed_num * IF(commits.
    merge_commit_parent=2,1,-1)) AS 'line_removed'
FROM Merge_Data.Merge_Scenario as merge_scenario
JOIN Merge_Data.Merge_Related_Commit as commits
    ON merge_scenario.merge_commit_hash = commits.
    MS_merge_commit_hash
GROUP BY merge_scenario.merge_commit_hash
```

V. CONCLUSION

In this paper, we introduced MERGANSER, an easy-to-use tool for large-scale software merging studies. Our proposed tool receives a list of repositories as input and stores the information of their merge scenarios into a normalized and extensible SQL database using only a one line command. The tool is written in Python, released under the `MIT` license, and can analyze Git repositories in a distributed manner. MERGANSER is configurable to cater to different user needs, such as providing them the ability to collect data only for specific database tables to avoid unnecessary computations. To demonstrate how data collected by MERGANSER can be used, we showed three sample SQL queries to extract relevant data about merge scenarios.

REFERENCES

[1] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *2009 6th IEEE International Working Conference on Mining Software Repositories*, pp. 1–10, IEEE, 2009.

[2] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "The promises and perils of mining github," in *Proceedings of the 11th working conference on mining software repositories*, pp. 92–101, ACM, 2014.

[3] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian, "An in-depth study of the promises and perils of mining github," *Empirical Software Engineering*, vol. 21, no. 5, pp. 2035–2071, 2016.

[4] A. Sarma, D. F. Redmiles, and A. Van Der Hoek, "Palantir: Early detection of development conflicts arising from parallel code changes," *IEEE Transactions on Software Engineering*, vol. 38, no. 4, pp. 889–908, 2012.

[5] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, p. 45, ACM, 2012.

[6] G. Cavalcanti, P. Borba, and P. Accioly, "Evaluating and improving semistructured merge," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 59, 2017.

[7] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source java projects," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2051–2085, 2018.

[8] P. Accioly, P. Borba, L. Silva, and G. Cavalcanti, "Analyzing conflict predictors in open-source java projects," in *Proceedings of the 15th International Conference on Mining Software Repositories*, pp. 576–586, ACM, 2018.

[9] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen, "Indicators for merge conflicts in the wild: survey and empirical study," *Automated Software Engineering*, vol. 25, no. 2, pp. 279–313, 2018.

[10] Y. Fan, X. Xia, D. Lo, and S. Li, "Early prediction of merged code changes to prioritize reviewing tasks," *Empirical Software Engineering*, pp. 1–48, 2018.

[11] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Early detection of collaboration conflicts and risks," *IEEE Transactions on Software Engineering*, vol. 39, no. 10, pp. 1358–1375, 2013.

[12] M. L. Guimarães and A. R. Silva, "Improving early detection of software merge conflicts," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 342–352, IEEE Press, 2012.

[13] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. de Water, "Studying pull request merges: a case study of shopify's active merchant," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pp. 124–133, ACM, 2018.

[14] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in github: What can we learn from code review and bug assignment?," *Information and Software Technology*, vol. 74, pp. 204–218, 2016.

[15] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proceedings of the 2006 symposium on Eye tracking research & applications*, pp. 133–140, ACM, 2006.

[16] A. D. Da Cunha and D. Greathead, "Does personality matter?: an analysis of code-review ability," *Communications of the ACM*, vol. 50, no. 5, pp. 109–112, 2007.

[17] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *Proceedings of the 2013 international conference on software engineering*, pp. 712–721, IEEE Press, 2013.

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

[19] G. Gousios, "The ghtorent dataset and tool suite," in *Proceedings of the 10th working conference on mining software repositories*, pp. 233–236, IEEE Press, 2013.

[20] D. Spadini, M. Aniche, and A. Bacchelli, "Pydriller: Python framework for mining software repositories," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 908–911, ACM, 2018.

[21] https://github.com/Prickett/gitminer.

[22] https://chaoss.github.io/grimoirelab.

[23] H. C. Estler, M. Nordio, C. A. Furia, and B. Meyer, "Awareness and merge conflicts in distributed software development," in *Global Software Engineering (ICGSE), 2014 IEEE 9th International Conference on*, pp. 26–35, IEEE, 2014.

[24] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pp. 168–178, ACM, 2011.