Evaluating Software Merge Quality

Ahmed-Nacer Mehdi
Université de Lorraine
LORIA, INRIA
54506 Vandœuvre-lès-Nancy, France
mahmedna@loria.fr

Pascal Urso
Université de Lorraine
LORIA, INRIA
54506 Vandœuvre-lès-Nancy, France
urso@loria.fr

François Charoy
Université de Lorraine
LORIA, INRIA
54506 Vandœuvre-lès-Nancy, France
charoy@loria.fr

ABSTRACT

Software merging is a time-consuming and error-prone activity but essential to large-scale software development. Developers use software merging tools, but if such tools return results with too many conflicts and errors, this activity becomes even more difficult. To help developers, several algorithms have been proposed to improve the automation of merge tools. These algorithms aim at minimising conflict situations and therefore improving the productivity of the development team, however no general framework is proposed to evaluated and compare their result.

This paper proposes a methodology to measure the effort required to use the result of a given merge tool. We employ the large number of publicly available open-source development histories to automatically compute this measure and evaluate the quality of the merging tools results. We use the simple idea that these histories contains both the concurrent modifications and their merge results as approved by the developers. Through a study of six open-source repositories totalling more than 2.5 millions lines of code, we show meaningful comparison results between merge algorithms and how to use the results to improve them.

Categories and Subject Descriptors

[**Software and its engineering**]: Software notations and tools—*Software configuration management and version control systems*; [**Software and its engineering**]: Software creation and management—*Software version control*; [**Human-centered computing**]: Collaborative and social computing systems and tools—*Asynchronous editors*; [**Computing methodologies**]: Design and analysis of algorithms—*Concurrent algorithms*

General Terms

Methodology, Experimentation, Automatic evaluation.

Keywords

Distributed Systems, Collaborative Editing, Asynchronous System, Collaboration, Benchmark, Commutative Replicated Data Types, Algorithms, Experimentation, Merge quality, Conflicts.

EASE '14 London, UK

1. INTRODUCTION

Collaborative software development requires to manage effectively the contribution of different participants. This is the goal of *asynchronous* tool such as version control system[1]. Currently there are two main concurrent control policies in asynchronous collaborative systems that allow concurrent work: pessimistic or optimistic policies. Some version control systems such as RCS[30] and Visual SourceSafe[25], adopt a pessimistic policy. These systems prevent concurrent editing by locking. Consequently, they cannot ensure high responsiveness for collaboration because the initiator of an update should acquire an exclusive access and they do not scale in terms of software developers number [19]. Therefore, large-scale software development processes rely on version control systems which are *optimistic asynchronous* such as CVS [12] and Git[31].

Software source merging is an essential operation for optimistic version control systems [23] when several developers modify concurrently the same file. Each developer works on its own copy in isolation and synchronise after to establish a common view of the document. The system manages several lines of software development produced in parallel. However, the merging can create different kinds of conflict and consequently this can be very time consuming, when errors fail to be resolved automatically.

Some authors have proposed numerous solutions to merge separate lines of software development. Mens [19] classifies solutions into two dimensions. The first dimension differentiates textual, syntactic, semantic and structural merging. The textual approaches consider documents as a list of elements (characters or lines) and can be used in any context while the others uses grammatical, semantic or structural information specific to the type of document. The second dimension differentiates state-based and operation-based merging. State-based merging uses only the original and final versions of the document. Operation-based merging (aka change-based) uses information about each modification of the document. For instance, three-way-merge tool used in CVS, SVN or Git version control systems is textual and state-based merging [13].

In [19], Mens recognise that *"In general, we need more empirical and experimental research [...], not only regarding conflict detection, but also regarding the amount of time and effort required to resolve the conflict."*. A solution would be to conduct user studies to measure user satisfaction. However, such studies are time consuming and hardly reproducible. They are also difficult to evaluate automatically since one need to know both concurrent modifications and their correct merge result. In addition, the effort that the developer will make to produce the final code depends on the result of the merge. Automated diff and merge techniques [19] help in resolving direct conflicts [6], but often require manual intervention

[1]E.g. git, svn, cvs, darcs or mercurial

[6, 27, 14].

In this paper we propose a methodology to compute this effort to evaluate the quality of merge algorithms. We strive to optimise the developer's time and effort, minimise conflicts occurrence, and thus to improve the development team's productivity. To achieve this goal,

- we designed and implemented an open source tool[2] to measure the quality of any merging tool using large-scale software merges;
- we analyzed six open-source repositories and examined more than 2.5 million publicly available lines of code from repositories containing thousands of commits made by hundreds of developers. In these repositories, merge conflicts are solved manually by the developer;
- using the previous informations, we diagnosed important class of conflicts and offered a solution to improve the merge result.

However, to compute the effort made by users in case of conflict, we need to know what the users want as the final result before starting the collaboration. The history of Distributed Version Control Systems (DVCS) contains the results that the user corrected. Thus, in the histories, the merge result is correct. This paper, focuses on the history of Git system.

The basic idea is to develop a tool that replay the same editing sessions as in git DVCS history by using state-based and operation-based merging tools. Since git's histories are generated by state-based tools. We transform them into operations in order to simulate execution of operation-based tools. To evaluate the quality of the merge tools, we compute the difference between the result computed by the tools and the merged version available in the git history. We use the size and the composition of the difference as a metric of user effort using a given merge tool. The framework is built upon an open-source performance evaluation tool. Thus, software merging tool developers will be able to evaluate and select the best merge procedure for their needs. Also, merge tools designers will be able to evaluate and to improve their tools.

The remainder of this paper is structured as follows. First, we describe the content of git histories and the git merge procedure. The second section explains how our framework simulates state-based and operation-based merging and how it measures merge quality results. The third section shows and analyses a comparison obtained by our framework between representative algorithms of textual distributed merging tools. This comparison was conducted to evaluate our framework, and shows how it can exhibit meaningful differences within this delimited class and how we can analyse this difference in order to improve the existing merge algorithms. Then, we shortly survey the state of the art in merge analysis in large scale collaboration. The last section draws some conclusions.

2. AVAILABLE EXPERIMENTAL DATA

The massive number of git histories are publicly available and constitute a very valuable corpus of distributed asynchronous editing traces to understand how developers collaborate. For instance, thousands of programmers around the world develop the Linux Kernel using git [31]. Hundreds of developers modify the same project and the community succeeds in merging their documents manually.

[2]https://github.com/score-team/replication-benchmarker.git

Web-based hosting services for software development projects provide large Git histories : GitHub has 3.4M developers and 6.5M repositories[3], Assembla has 800,000 developers and more than 100,000 projects[4], or SourceForge with 3.4M developers and 324,000 projects[5]. In this section, we first describe how collaboration is possible using a distributed version system and then, we explain how we selected the data for our experiment.

2.1 Branching and Merging

In distributed collaborative software development [15], developers work on separate workspaces that they modify locally. They edit, compile and test their own version of the source code. Submission of developer modifications on their code is a "commit". A developer decides when to commit, when to publish their commits to others and when to incorporate other developer commits. Different developers submissions appear on different branches. When incorporating other developer commits, in the absence local commits, the action is made silently, and the git recorded history is linear. If the user had produced a commit, git uses *git merge* to produce a best effort merge of the concurrently modified files. When concurrent modifications occur at the same position in the document, *git merge* produces a conflict. The developers have to resolve these conflicts before committing the result of the merge [31]. To help them to resolve these conflicts, the git software can be configured to call an interactive visual *mergetool* such as emerge, gvimdiff or kdiff3.

All branching informations are stored in the history of git as a graph of histories. As presented in Figure 1, the developers work on different branches, afterward they merge their modifications on the same branch. Modifications made by two different developers can appear as a list if they are produced in order, or as branches if produced concurrently. A merge can be produced by one of the developers who produced the concurrent commits or by a third party. A merge can involve two or more branches.

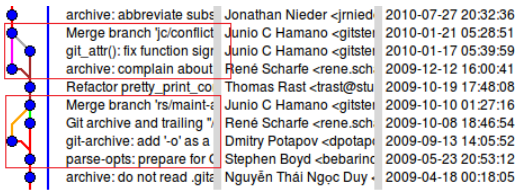

Figure 1: History of git corpus

Several branches can exist at the same time in git repository. Some of them represent old versions or development alternatives. Whatever the branch objective may be – collaborative editing, development alternative, bug fixing, etc. – when a branch is merged into an other, the merge procedure is the same.

2.2 Conflicts in Practice

Bird and colleagues [4], explored the "promise and peril of mining Git [histories]". They point out that git histories can be rewritten, and commits can be reordered, deleted or edited. For instance, the Git's "rebase" command reorder merged branches into a linear history. Moreover, we do not have access to each developer private repository that may contain a very rich and complex history. Thus, publicly available git histories do not represent the entire history of

[3]https://github.com/about/press
[4]https://www.assembla.com/about
[5]http://sourceforge.net/about

collaboration. Still, users are the authors of the conflict resolutions that are available in these histories.

We consider the merge state available in the git history as the user expected result. Of course, the merge may generate problems. For instance, the git merge command may commit source code which does not compile. Bruno et al. [6] investigated direct and indirect conflicts. They found in three open source project studied, 33% of the 399 automatic merges that the version control system reported as being a clean merge, actually were a build or test conflict. Moreover, 16% of merges resulted in a textual conflict, 1% of merges resulted in build failure and 6% of merges resulted in a test failure. Still, nearly all of the problems introduced by the merge are quickly detected since they produce compiling errors [17]. With git, the developer can manually revert the commit to correct the problem. Due to the graph structure of git histories, a reverted commit does not appear in the paths.

However, even if automatic clean merge can be problematic they may not be present in the studied master histories, since they may be reverted. To evaluate the number of problematic merge commit in histories, we studied the git software repository[6]. We measured the number of non-compiling merge commit, and the number of reverted merge.

- 30 of the $10,000$ most recent commits of the master branch in the repository don't compile. Of these $10,000$, $3,085$ are merge commits and only 1 merge commit does not compile.
- On the entire history, only 4 commits have the default message for reverting a merge – "`Revert "Merge ...\"`" – compared to $7,231$ commits with the default message for a merge.

These measures shows that the manual merge present in the histories are most of the time done very properly, at least in the repository of the git software.

3. FRAMEWORK

Our framework allows to replay the history of git repositories to compute each merge result produced by a given merge algorithm. It compares the merge result, without assumption on how it is produced, to the merge version produced by the actual developer. The difference between these two versions correspond to the effort that the same developer would have produced when using a DVCS based on the evaluated merge tool.

To replay state-based merges, we simply call the evaluated tool to replay every merge of a given git history. For instance, to evaluate git itself, we call the git merge command using the standard strategy for merging branch. For operation-based merge, the framework must simulate a collaboration composed of concurrent modifications and their merges. To simulate this collaboration, we adapted an open-sourced replication performance evaluation tool of distributed optimistic replication mechanisms [1].

3.1 Operation-based simulation

To replay git histories with any operation-based merge tool, we need to let the tool produce these operations. When modifications occur on different branches of the history, they affect different version of the document. We replay this branching history by simulating collaborative editing replicas.

However, the git software does not manage replica information in its data storage. It stores only the email of the user who produced a given commit. The user's email is not reliable since a same user may work on several replicas, or change his email while working on the same replica. To simulate replicas, the framework creates a graph of the merge/commit node based on history. Then, it parses the graph and assigns a replica identifier to each node. Since on merge, different parents are different replicas, the framework assigns different replica identifiers to the parent of the merge as in figure 2.

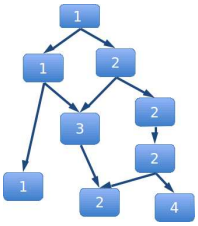

Figure 2: Each node is labeled by its replica identifier

For performance reasons, the framework heuristically minimises the number of replicas. Thus, the number of simulated replica is much lower than reality were each developer works on its own (or even several own) replica – e.g. 60 replicas simulated against 583 developers that participated to the history of the git software itself. The number of replicas does not affect the quality of the merge. The merge result only depends on the concurrent operations. Since replica identifiers are selected without *a priori*, we ensure that all concurrent modifications have the same impact on the merge result. Thus, the algorithms evaluated must be independent from the replicas priority to obtain good results. To ensure the same concurrency history as in the traces, the simulator assigns to each commit a vector clock [18] that represents the partial order of modifications given by the DVCS graph history.

States to operations.

Some operation-based merge tools are able to detect modifications by themselves. We provide them with the successive versions using the above concurrency information. The framework is in charge to distribute the operations produced to the other replicas.

The framework can also provide textual modifications to the operation-based merge tools. For each commit that has only one parent the framework computes the diff [21] between the two states. The diff result is a list of $insert$, $delete$ or $replace$ modifications concerning blocks of lines. The framework stores the state of the resulting merge commit. All commits (diffs and merge states) and their vector clocks, are stored in an Apache CouchDB database in order to be used by all the runs of different algorithms.

Example.

In figure 3, the git history contains a merge: a commit with two parents. The framework assigns two different identifiers to the parents: replica 1 and replica 2. Both replicas initially share the same document `"A"`, replica 1 commits `"AC"` and replica 2 commits `"AB"`. Based on the diff retrieved from the history, the framework asks to the replica 1 to handle insert("C",2) and asks to replica 2 to handle insert("B",2).[7] The framework obtains from the replicas the operations corresponding to these modifications. The content and the format of the operations depends on the merge tools evaluated. They can be textual, syntactic, semantic or structural operations [19].

[6] `https://github.com/git/git`, starting from commit `0da7a53a`

[7] The framework checks if the replica "obeys", i.e. if the replica presents the intended result.

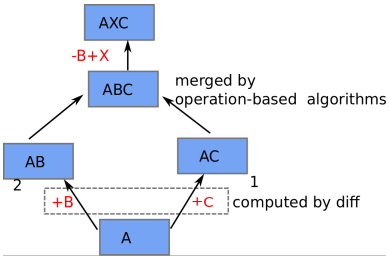

Figure 3: Imitate human merge correction

Depending on the replica's identifiers assignment, the merge can occur on any replica (1, 2 or a third one), with the same behaviour. The operations produced by replicas 1 and 2 are distributed by the framework to the replica where the merge occurs. In our example, the result obtained by the merge is `"ABC"`. However, in the Git history of Git, the merged version is `"AXC"`. To correct the document, the framework asks to the merging replica to handle `replace(2, "X")` to modify `"B"` in `"X"`. In other words, our framework imitate human merge checking and correction.

3.2 Merge Quality Metric

Our merge quality metrics represent the effort that a developer would make if he used a given merge tool to produce the manually merged version. They are computed using the difference between the automatically merged document – *"computed merge"* – and the version in the Git history – *"user merge"*. We use the Myers difference algorithm [21] to calculate this difference. The algorithm returns a list of modifications: insertions and deletions of text blocks. The framework calculates two metrics:

Merge blocks the number of modifications in the difference.

Merge lines the number of lines manipulated by these modifications.

These metrics are a classical Levenshtein distance between the computed merge result and the user merge with two levels of granularity, the line and the block of lines. The two levels are important : a developer spends more effort in identifying and updating, ten lines in ten separate blocks than ten lines in one contiguous block. To obtain valid measures, substitution edits are counted as one insertion plus one deletion. Elsewhere, for a given conflict, a merge that randomly presents one of the conflicting version will obtain better result that the same merge that presents the two versions. Assuming that the two versions are of same size n, and just one is correct, our measure for both merges will be n in average.

Whatever the nature (textual, syntactic, semantic, ...) or the mechanics (state-based or operation-based) of the evaluated merge tool, we compute these metrics in the same way. The framework can also manage merge tools that present to the developer conflicts when they are unable to merge some concurrent modifications, as well as fully automatic merge tools. We consider every difference between the computed merge and the user merge as requiring the users effort.

Since not all the studied approaches introduce conflict markers (lines beginning by "`>>>>>>>>>`", "`<<<<<<<<<`" and "`=========`") into the merge result, we remove them before measuring the metrics. For instance, in figure 4, the git merge tool produces a conflict between two modifications. In git merge result, the order of appearance of the conflict block depends on the replica that executes the merge. When we remove the conflict markers, the difference with the user merge is either one block and two lines or three blocks and four lines.

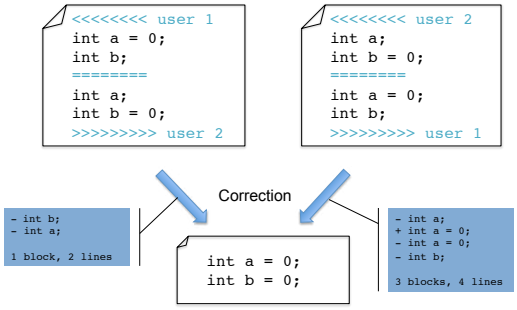

Figure 4: Computation of the metrics

The order in the presentation of conflict blocks impacts our metric. However, both orders are simulated with the same probability in our framework. Also, every evaluated merge tools will face the same issue when merging concurrent modifications, whether they mark the conflicts or silently merge both modifications.

4. EXPERIMENTAL EVALUATION

We have used our framework to evaluate several merge tools representative of a class of decentralised textual merge. We want to check if our framework and our metrics are able to detect meaningful differences between merge tools with similar purpose and if an analysis of these differences can help with improving the merge tools.

We choose the class of decentralised textual merge tool due to the number of available open-sourced implementations and since they are the only candidates to replace the default merge tool massively used DVCSs. Indeed, a DVCS is decentralised and requires at least a default merge tool that can handle every textual document without prior knowledge on the syntax, the semantic or the structure of the document. Within this class, we evaluated both state-based and operation-based tools.

In what follows, we first describe shortly the merge tools that we evaluate. Then, we show the obtained results, and we explain the difference between these results. Finally, we show how such difference detection can be used to improve merge tools.

4.1 The tools

The distributed textual merge algorithms that we evaluate are: the default git merge, one operational transformation approach, and two commutative replicated data types. All these approaches were originally proposed or adapted to manage asynchronous collaborative editing such as synchronizer or wikis [20, 33]. They all consider the text as a sequence of lines.

Git merge.

Git merge allows to join two or more development histories together. After merging, if one or more files conflict, the merge tool program such as diff3 [13] will be run to resolve differences on each file (skipping those without conflicts).

During a merge, the working tree files are updated to reflect the result of the merge. Among the changes made to the common ancestor's version, if both sides made changes to the same area, git cannot randomly pick one side over the other, and asks users to resolve it by leaving what both sides did to that area [31].

The basic idea, is to compare the versions that have diverged from the origin version (let be version A and version B) with the original version (let be O). First, git compares versions A and B with O to find the maximum matchings between O and A and between O and B. It then parses the results and identifies the region where the original version O differs from A and B. When the region are different the modifications are applied automatically, otherwise a conflict block containing both modifications is produced.

By default, git uses the same style as the one used by the "merge" program from the RCS[30] as presented in figure 5.

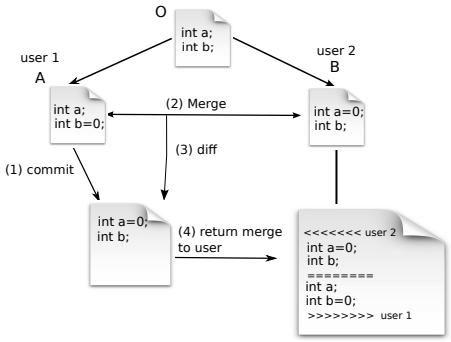

Figure 5: git merge conflict

Following git merge result, the modifications can be merged successfully or unsuccessfully and then create a conflict. However, a successful merge successfully does not mean that the document is correct. For example, if two users modify collaboratively a project, user 1 makes call to the method when another user changes concurrently the name of this method. Git merge does not detect a conflict since both users modify the document in different part of the document. This kind of problem is also called *false negatives merge* (or indirect conflict). When the users modify the same document concurrently at the same position as in figure 5, the merge conflicts but it could be avoided by the users, the conflict is called *false positives merge* (or direct conflict).

Operational Transformation.

The Operational Transformation (OT) approach [11, 29] has been introduced for synchronous collaborative editing systems. Asynchronous systems [7] apply it successfully. OT approaches can be decomposed into a concurrency control algorithm and transformation functions. A transformation function changes an operation to take into account the effects of the concurrent executions.

For a decentralised system as DVCS, OT concurrency control algorithms such as SOCT2 [32] or MOT2 [7], require the transformation functions to satisfy some conditions. These conditions are only met by the Tombstone Transformation Functions (TTF) [22]. TTF approach keeps all characters in the model of the document, i.e. deleted characters are replaced by tombstones.

Commutative Replicated Data Types.

Unlike OT algorithms, Commutative Replicated Data Types (CRDT) algorithms do not modify their operations. CRDT algorithms are designed for concurrent operations to be natively commutative. CRDT algorithms assign a unique identifier to each element. The identifiers are totally ordered and remain constant for the whole lifetime of the document. However, the algorithms apply different techniques to identify the operations.

RGA [26] specifies on the identifiers (aka $s4vector$) the last previous element visible during its generation. Thus, the tombstones also remain after the deletions. RGA algorithm is based on the hash table. During the integration of the remote operation, RGA iterates over the hash table and compares the s4vector until retrieved the correct target position.

Logoot [33] CRDT generates identifiers composed of a list of positions. The identifiers are ordered with a lexicographic order. A position is a 3-tuple containing a digit in a specific numeric base, a replica identifier and a clock value. Identifiers are unbounded to allow for arbitrary insertion between two consecutive elements. Unlike OT and RGA algorithms, Logoot does not need to store the tombstones since elements are not linked through insertions operations.[8]

4.2 Empirical Study

We analysed six open source projects (see Table 1). We chose these projects from GitHub and Gitorious web services, based on the following criteria: (1) popularity of the projects [9] from GitHub, (2) activity of the projects from Gitorious, (3) the project is developed by using git in Github, and not a mirror of another system repository such as SVN. We also selected the git repository of the git software itself, since it contains more commits and merges than these projects and since we think that it contains the best merge result since they are done by specialists of the tool.

In Table 1, we present the characteristics of the six selected projects. The head commit sha1 used to run our experiments is presented below the name of each repository. In addition, we give the main programming language and the number of lines edited for each repository. The characteristics are computed per file present in the repository. We computed the maximum, average and minimum number of commits and merge that affected the files. We also present the number of blocks and lines modified by the users and the number of simulated replicas. Obviously, we restrict our study only to files that are merged at least once.

4.3 Quantitative Results

Table 1 presents also the number of merge blocks and the number of merge lines produced by all algorithms including git merge. To compare the results between repositories, figures 6 and 7 present respectively the merge block and the merge lines metric on all repositories. Since *git merge* is the default algorithm used in git system to merge the modifications, we use its results as the reference (=100%). Figure 8 presents the sum of the metrics for all the projects.

Statistical analysis. We also perform a statistical test of significance. Since the dataset used is independent, a non-parametric analysis method would be the most suitable approach for analysis. Using Kruskal-Wallis test[10], we observe that all operation-based algorithms outperform git merge, and all results obtained are very significant (p-value<0.05).

For the block metric, the average gain is between 31% and 33% and p-value is 0.004 for all operation-based merge except Logoot. Even if the average gain in Logoot is 5%, the result remains significant (p-value=0,00019). In addition, the difference between all

[8]Since the Logoot algorithm generates its identifiers by using a random function and the order of these identifiers is not necessarily the same in two different executions, we conducted four executions and we computed the average metric.

[9]`https://github.com/popular/starred`, April 2013

[10]The Kruskal-Wallis test does NOT assume that the data are normally distributed.

PROJECT	git			bootstrap			node		
Head sha1	8c7a786b6c			37d0a30589			88333f7ace		
Main programing language	C			CSS			JavaScript		
Files with merge	557			69			44		
Total line edited	1091125			225596			1192244		
Characteristics	max	min	avg	max	min	avg	max	min	avg
Commits	1742	4	59,4	526	5	88,4	631	9	111,1
Merge	451	1	10,1	49	1	6,4	37	1	6,3
Nb. Block.	3887	4	190,3	1975	1	117,3	2390	21	496,9
Nb. Lines.	25305	24	1222,4	25216	66	3317,6	56076	1392	13782,1
Replica	59	2	3,5	10	2	3,7	4	2	2,1
Merge blocks in git merge	3184			1614			1138		
Merge lines in git merge	10159			14658			8159		

PROJECT	html5-boilerplate			d3			gitorious		
Main programing language	CSS			JavaScript			Ruby		
Head sha1	f27c2b7372			d1d71e16			c1105ebe86		
Files with merge	4			38			72		
Total line edited	4463			99093			66517		
Characteristics	max	min	avg	max	min	avg	max	min	avg
Commits	208	27	140,7	891	5	63,27	437	4	58,2
Merge	20	1	11,3	184	1	8,75	10	1	2,1
Nb. Block.	230	25	115,7	1348	1	70,13	771	2	58,7
Nb. Lines.	2753	142	1488,3	64691	30	2668,45	12902	54	936,9
Replica	6	2	4,0	30	3	4,16	5	3	3,1
Merge blocks in git merge	24			648			489		
Merge lines in git merge	87			4658			2303		

Table 1: Projects characteristics

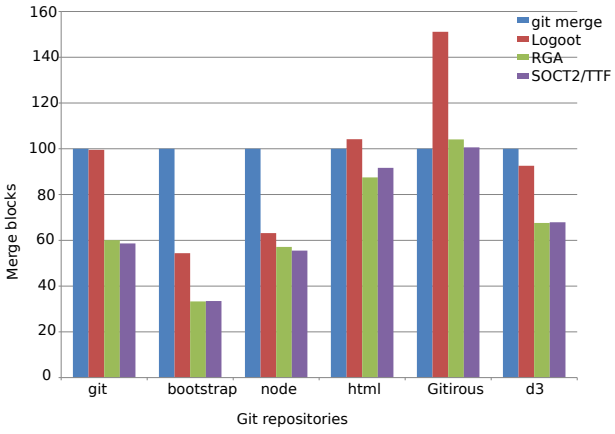

Figure 6: Relative merge block measures

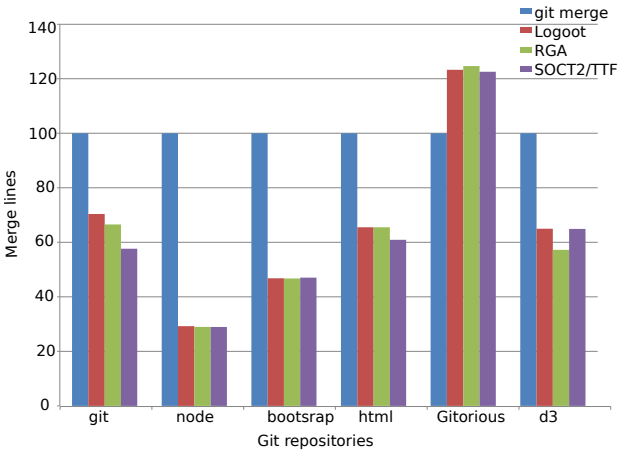

Figure 7: Relative merge lines measures

operation-based merge (including Logoot) is on average between 26% and 27% and very significant (p-value < 0.001).

For the line metric, the average gain is between 32% and 35% for all operation-based algorithms including Logoot. This difference is very significant since p-value = 0. The difference between the operation-based merge is below 3% but also significant (p-value = 0,00235).

We present here aggregated results, but our framework produces results detailed per file. Analysing these files with significantly different results, we were able to understand in which collaboration pattern difference occurs. In the following, we explain why such a difference exists between the algorithms and why on Logoot we obtain a different outcome.

4.4 Qualitative Discussion

Difference between git and operation-based merges.

The difference is due to a very common type of collaboration: concurrent edits done on two consecutive blocks. In figure 9, developer 2 and developer 3 share the same original piece of code `A` containing four lines. Afterwards, developer 2 updates two lines (let be line 1 and line 2) and produces the document `B`. Concurrently, developer 3 updates two lines (let be line 3 and line 4). Developer 3 commits his modifications and produces the document `C`. During the merge procedure, git merge creates a conflict on the whole piece of code, even if both edits occurs on different lines. Indeed, git merge algorithm is state-based, it manages the two whole versions as irreconcilable edits done on the four original lines.

The way that git merge tool presents the conflict depends on the developer usage of the git merge command. Usually, it returns the code `B'` to developer 2 and `C'` to developer 3. To obtain the final

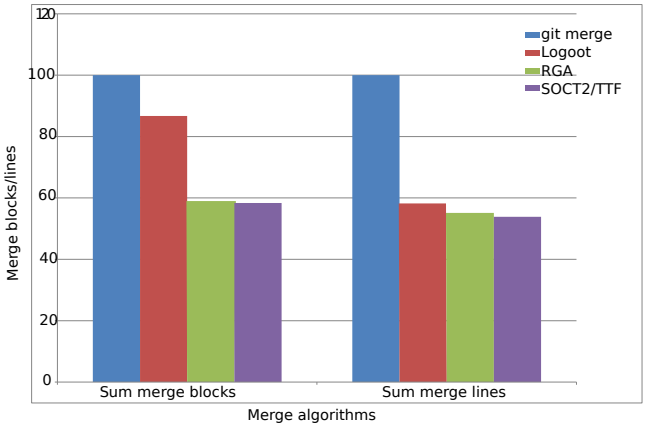

Figure 8: Total merge blocks/lines in all repositories

code, the developer that operates the merge has to edit at least 1 block and 4 lines.

Contrary to git merge, operation-based algorithms merge automatically the edits. This scenario is well known [19] and is one of the traditional arguments in favour of operation-based merge. We analyse the files history where git merge does not perform perfectly (existing differences between computed and user merges), and at least one operation-based algorithm performs perfectly. For 87% of such files – taken randomly[11] – this collaboration pattern occurs and the actual developer has to manually make the effort of combining the two edits, instead of choosing one complete version. This fine grain analysis and the overall results of operation-based algorithms demonstrate for the first time the actual benefits of such solutions.

Difference between operation-based merges.

We found that this difference occurs when concurrent edits affect the document in the same position. Logoot uses randomness to generates its identifier. So it will more frequently interleave the lines added concurrently. If he must keep only one of the edits, the developer has to remove each interleaving lines separately, instead of removing a single block. Thus, Logoot obtains a worse block metric than other operation-based approaches but a similar line metric. In the scenario illustrated in Figure 10, two developers insert concurrently at the same position two different blocks. Since Logoot identifies each line by a random – but successive – identifier, the different lines of blocks can be mixed. Consequently, the developer has to edit 4 blocks and 4 lines to correct their document. However, using other algorithms such as RGA or TTF, the order between lines inserted concurrently is determined first by the replica identifier and the two blocks are contiguous. The developer has to edit only 1 blocks and 4 lines.

We also noticed a slight difference between RGA and the other operation-based approaches. This is due to the scenario of figure 11. The reason we were able to identify is that these approaches set different identifiers for the lines `int x;` and `int a=0;`. The line `int a=0;` replaces `int a;` and usually have a similar identifier; while `int x;` is placed before `int a;` and has a lower identifier. In RGA, the identifier is built on the preceding element and not on both preceding and next element. Thus, `int x;` has a similar identifier than `int a=0;`, and may be placed after. Therefore, RGA obtains different results than TTF.

[11]The fifteen first such files in alphabetical order in the git repository.

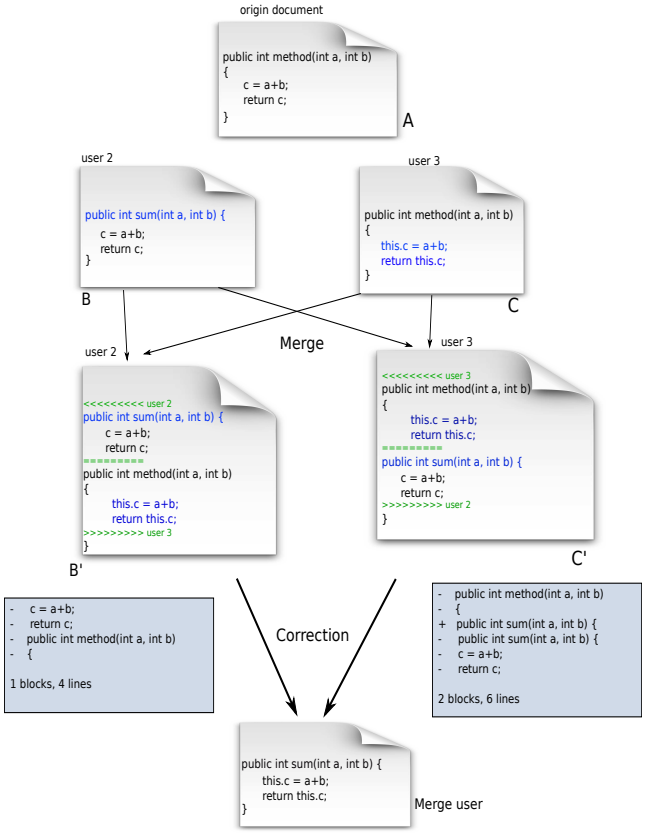

Figure 9: Edits on two consecutive blocks

Gitorious repository.

For both metrics – blocks and lines –, the behaviour of operation-based algorithms on Gitorious repository is different from other repositories. All operation-based algorithms are less efficient than git merge. Half of the block and line values are due to a specific collaboration pattern on one file "`diff_browser.js`". The collaboration on this file begins with the merge of two branches that have no common ancestor. However, these two branches contains code with many lines in common that git merge is able to merge. This pattern is known as a "accidental-clean-merge" by the VCS community, and is not well handled by existing operation-based merge.

We analysed, in the different projects, the history of the files where git merge outperforms the operation-based merges. We also noticed that the number of commits that Revert other commits can be high in the studied repositories (sometimes half the number of merge commit). This can lead to well-known undo puzzles [28]. For instance, a developer deletes the element `A` when concurrently another developer deletes the same element `A` and then undoes this deletion. Git merge manages well this case to obtain the document without `A`, while others reinsert the element. Several undo mechanisms for operational-based merge approaches exist [28, 34] and should be evaluated by our framework.

4.5 Improving merge: Update operation

Even if some merges perform better than other to order lines concurrently updated, we observed that none of them always order the lines correctly. In the example of figure 11, lines `int x;` and `int a = 0;` have a new identifier that the algorithm may be order wrongly. A solution is to reuse the identifier of `int a;`, i.e. to

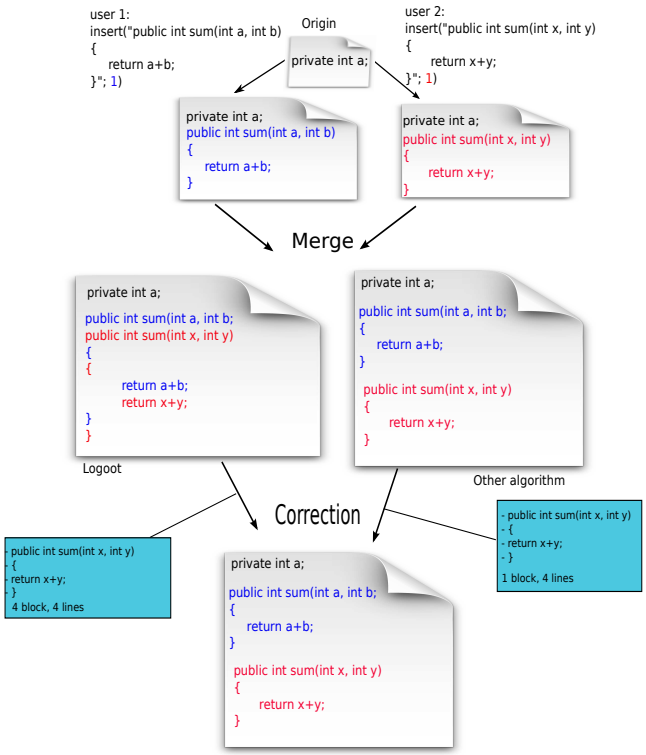

Figure 10: Different merge in operation-based approaches

update the line instead of deleting it and inserting a new one.

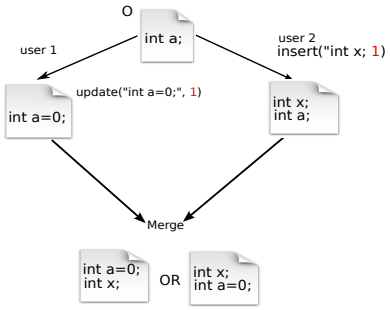

Figure 11: Merge order in operation-based approaches

In the following, we add an update operation for TTF. The update operation keeps the same position and changes the content. The delete operation is considered as an update operation to the *nil* value. To understand the behaviour of concurrent updates of the same element, we conducted two experiments on the git repository:

1. Update: concurrent updates – and deletions – are managed by using replica identifier priority.

2. delWin: we give the priority to delete in case of delete/update operations.

Figure 12 presents the block and line metric of these experiments and the original TTF in git repository. When a conflict occurs between delete and update, prioritising the delete operations does not improve the metric. Keeping the same identifier and thus the correct position for update operation has a notable effect. It saves 36% of merge blocks and 16% of merge lines[12].

Algorithm 1 Transform update with update

1: **function** TRANSFORM(upd(p_1,$Site_i$), upd(p_2, $Site_j$))
2: **if** ((p_1 <> p_2) or (p_1=p_2 and $Site_i$ > $Site_j$)) **then return** upd(p1,$Site_i$);
3: **else return** return noop(); // no effect

Algorithm 2 Transform update with update

1: **function** TRANSFORM(op1, op2)
2: Let t_1 and t_2 respectively the type of op1 and op2
3: Let p_1 and p_2 respectively the position of op1 and op2
4: **if** (t_1 = del and t_2 = upd) **then return** del(p1,$Site_i$);
5: **else if** (t_1 = upd and t_2 = del) **then**
6: **if** (p_1 <> p_2) **then return** upd(p_1,$Site_i$);
7: **else return** return noop();

Threats to validity.

To help users to detect and resolve conflicts, merge tools usually add awareness [10] mechanism such as git markers "»»»". We removed these markers to obtain comparison results. However, all other studied merge tools evaluated can integrate an awareness mechanism without modify their results as in [27]. For instance, the so6 tool [20], used in the web software forge Libresource, adds conflict markers on top of a traditional operational transformation merge mechanism.

Another threat is the metric used in this paper. We presented the number of lines that a user must modify but not differentiate the kind of modification (insertion or deletion). While, an effort made by users to delete a line can be considered as more disturbing than insertion of a new line. However, our framework allows to capture separately these kinds of modifications. In this paper, due to space limitation, we focused on presenting a first automatic measurement of the merge result quality.

The operation-based algorithms presented do not treat move operations. Therefore, move operations are treated as deletions following by insertions in another position. However, our framework is able to detect move operations. In [2], we explain how move

[12]In these experiments, we considered any line replacement as an update; the results may be improved if we consider only replacement that have a similar content.

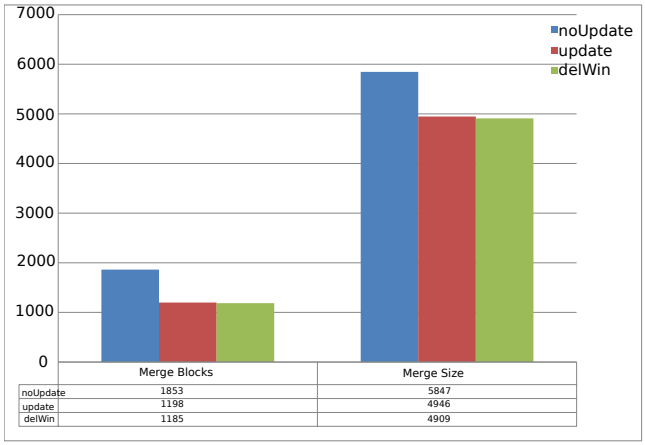

Figure 12: TTF algorithms on git repository

operations are detected and their affect on the automatic merge.

Moreover, depending on the historical data, an automatic merge procedure better than git's one may paradoxically be badly evaluated. The merge committed by a developer is influenced by the git merge procedure. With another procedure, he or she may produce a different result that is just as satisfactory to the developer. We consider that the closer result of merge procedure on average is the result of the merge produced manually after a conflict generated by merge tool, the better it is.

Finally, we limit our study to DVCS histories that contain source code. The merge decision may be different for other kind of documents. We have conducted the evaluation on several repositories that contain source code written in various programming languages to limit the impact of the language syntax on the evaluation. Some DVCS histories also contain more "human-oriented" documents such as html files, software documentation or wiki files.

5. RELATED WORK

As presented in this paper, measure the effort of users during a concurrent collaboration plays an important role in the software development process. It reduces the number of conflicts and improves the productivity of the development team. Here we discuss some related work on evaluation of merge results.

Some approaches, such as Orian presented in [24], study the concurrency control policies in version control systems and propose two metrics: concurrency level and merge effort. The concurrency level metric measures the portion of concurrent operations that affect the document. While, merge effort metric measures how difficult it is to merge two versions in order to solve the conflicts. However, this approach has a limitation. It requires some information to calculate the proposed metrics which are not available in some version control systems.

Other researchers have studied large scale collaboration through distributed version control systems. Perry and colleagues [23] early demonstrate the importance of parallel development, and thus the importance of merge in term of performance and quality, in large scale software engineering. They studied only one repository and they evaluate only the existing merge tools attributes, and not their actual result. More recently, other researchers studied the same corpus as us. Brun and colleagues [5] explored GitHub projects to understand how conflict could appear and conclude developers merge branches with less potential conflict between them. They studied only the result of the git automatic merge tool, without analysis the developer merge or other merge tools. De Souza and colleagues [8] proposed several metrics to estimate how difficult a merge would be, without evaluating merge tools themselves. Combined with our framework, such metrics would be useful to understand with which kind of modification a merge tool over- or under-performs.

Palantir [27], Crystal [6] and CollabVS [9] propose a solution to detect and resolve the conflicts earlier. They anticipate the actions a developer may wish to perform and execute them in the background. The conflict can only be detected after the conflict has already developed in background. Cassandra [16] proposes a novel conflict minimisation technique that evaluate task constraints in a project to recommend optimum task orders for each developer. However, any methodology has been proposed to measure the size of conflict and the quality of the merge.

In more general collaborative editing study field, Wikipedia – the online encyclopedia – has caught the attention of researchers and presents a huge amount of available collaboration traces. Researcher studied the behaviours of developers and the collaboration patterns they follow, e.g. [3]. Researchers have used the Wikipedia data to evaluate the performance, in term of algorithmic complexity, of collaborative editing algorithms [34]. However, in Wikipedia, no concurrency information is available and all edits are serialised. There is no way to capture the developer behaviour when they face concurrent editions. Due to the high frequency of modifications when important news occurs, such concurrent editions happen, and the Wikipedia would also benefit of adapted merge algorithms.

As far as we know, the framework presented in this paper is the first open-source automated tool that allows the comparison of different software merging algorithms.

6. CONCLUSION AND FUTURE WORK

In this paper, we proposed a methodology to measure the quality of the merge algorithms in asynchronous collaborative editing systems. In addition, we presented an open-source tool that allows the comparison of different merge algorithms. The tool computes the developer effort in terms of number of modifications that have to be made by users to correct the document. Using this, we allow to evaluate, compare and improve merge algorithms. This is the first time to our knowledge that this kind of systematic work is conducted. This is an important information in the software development field where the concurrent edition of document represents a large and fundamental part of the activity. Reducing the user's efforts, improves the quality of collaboration and encourage users to work collaboratively.

Our experiments demonstrate that our framework is able to detect meaningful differences between merge algorithms of the same class of distributed textual merge. Analysing the results obtained by our framework, we were able to explain the cause of these differences and to propose a solution to improve the result of the algorithms.

Using our tool and method, one can evaluate and compare more complex algorithms such as syntactical or semantics merge tools. The framework is open source and publicly available in order to let researchers evaluate their own algorithms and compare it with others.

Our plan now is to study existing generic undo/redo mechanisms or mixed granularity editing operations for textual merging and to extend our tool to capture file systems modifications that are also present in DVCS histories in order to allow the evaluation of tree merging algorithms.

7. ACKNOWLEDGMENTS

This work is partially supported by the ANR project Concordant ANR-10-BLAN 0208. The authors would like to thanks following people for their contributions to the algorithms implementation : G. Oster (SOCT2) and S. Martin (Git Walker).

8. REFERENCES

[1] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. Evaluating crdts for real-time document editing. In ACM, editor, *ACM Symposium on Document Engineering*, page 10 pages, San Francisco, CA, USA, september 2011.

[2] M. Ahmed-Nacer, P. Urso, V. Balegas, and N. Preguica. Concurrency control and awareness support for multi-synchronous collaborative editing. In *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pages 148–157, 2013.

[3] J. Antin, C. Cheshire, and O. Nov. Technology-mediated contributions: editing behaviors among new wikipedians. In *Proceedings of the ACM 2012 conference on Computer*

Supported Cooperative Work, CSCW '12, pages 373–382, New York, NY, USA, 2012. ACM.

[4] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu. The promises and perils of mining git. In *Mining Software Repositories, 2009. MSR '09. 6th IEEE International Working Conference on*, pages 1–10, 2009.

[5] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: exploring future development states of software. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 59–64, New York, NY, USA, 2010. ACM.

[6] 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*, ESEC/FSE '11, pages 168–178, New York, NY, USA, 2011. ACM.

[7] M. Cart and J. Ferrie. Asynchronous reconciliation based on operational transformation for P2P collaborative environments. In *Proceedings of the 2007 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 127–138. IEEE Computer Society, 2007.

[8] R. de Souza Santos and L. Murta. Evaluating the branch merging effort in version control systems. In *Software Engineering (SBES), 2012 26th Brazilian Symposium on*, pages 151–160, 2012.

[9] P. Dewan and R. Hegde. Semi-synchronous conflict detection and resolution in asynchronous software development. In *ECSCW*, pages 159–178, 2007.

[10] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[11] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *SIGMOD Conference*, pages 399–407. ACM Press, 1989.

[12] K. Fogel and M. Bar. *Open source development with CVS*. Coriolis Group Books, 2001.

[13] F. S. F. GNU. Diff3. Three way file comparison program, (Septembre 2005).

[14] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 342–352, Piscataway, NJ, USA, 2012. IEEE Press.

[15] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81. ACM, 2004.

[16] B. K. Kasi and A. Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 732–741, Piscataway, NJ, USA, 2013. IEEE Press.

[17] D. B. Leblang. The cm challenge: Configuration management that works. In *Configuration management*, pages 1–37. John Wiley & Sons, Inc., 1995.

[18] F. Mattern. Virtual time and global states of distributed systems. In M. C. et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Château de Bonas, France, October 1989. Elsevier Science Publishers.

[19] T. Mens. A state-of-the-art survey on software merging. *Software Engineering, IEEE Transactions on*, 28(5):449–462, 2002.

[20] P. Molli, G. Oster, H. Skaf-Molli, and A. Imine. Using the transformational approach to build a safe and generic data synchronizer. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work - GROUP 2003*, pages 212–220, Sanibel Island, Florida, USA, November 2003. ACM Press.

[21] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[22] G. Oster, P. Urso, P. Molli, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *CollaborateCom 2006*, Atlanta, Georgia, USA, November 2006. IEEE Press.

[23] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 10(3):308–337, 2001.

[24] J. G. Prudencio, L. Murta, C. Werner, and R. Cepeda. To lock, or not to lock: That is the question. *Journal of Systems and Software*, 85(2):277 – 289, 2012.

[25] T. Roche and L. Whipple. *Essential SourceSafe*. Hentzenwerke Series. Hentzenwerke Publishing, 2001.

[26] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011.

[27] A. Sarma, D. Redmiles, and A. van der Hoek. Palantir: Early detection of development conflicts arising from parallel code changes. *Software Engineering, IEEE Transactions on*, 38(4):889–908, 2012.

[28] C. Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(4):309–361, December 2002.

[29] C. Sun and C. A. Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work - CSCW'98*, pages 59–68, New York, New York, États-Unis, November 1998. ACM Press.

[30] W. F. Tichy. Rcs—a system for version control. *Softw. Pract. Exper.*, 15(7):637–654, July 1985.

[31] L. Torvalds. git, (April 2005). http://git-scm.com/.

[32] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.

[33] S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pages 404 –412, Montréal, Québec, Canada, jun. 2009.

[34] S. Weiss, P. Urso, and P. Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 21:1162–1174, 2010.