EAI Endorsed Transactions
on Collaborative Computing

Research Article **ICST.ORG**

Merging By Decentralized Eventual Consistency Algorithms

Ahmed-Nacer Mehdi[1,*], Pascal Urso[1], François Charoy[1]

[1]Université de Lorraine
INRIA, LORIA.

Abstract

Merging mechanism is an essential operation for version control systems. When each member of collaborative development works on an individual copy of the project, software merging allows to reconcile modifications made concurrently as well as managing software change through branching. The collaborative system is in charge to propose a merge result that includes user's modifications. The users now have to check and a
this result. The adaptation should be as effort-less as possible, otherwise, the users may get frustrated and will quit the collaboration.

This paper aims to reduce the conflicts during the collaboration and improve the productivity. I
objectives: study the users' behavior during the collaboration, evaluate the quality of textual merging results produced by specific algorithms and propose a solution to improve the result quality produced by the defa
merge tool of distributed version control systems.

Through a study of eight open-source repositories totaling more than 3 million lines of code, we observe the behavior of the concurrent modifications during the merge procedure. We identified when the existing merg
techniques under-perform, and we propose solutions to improve the quality of the merge. We finally compare with the traditional merge tool through a large corpus of collaborative editing.

Received on 27 February 2015 accepted on 19 January 2015; published on 21 December 2015

Keywords: Operational Transformation, Commutative Replicated Data Types, collaborative editing, merging interfering, merge procedure, conflicts.

doi:10.4108/eai.21-12-2015.150817

1. Introduction

Nowadays, many collaborative editing systems are developed and available to users online. Such systems allow users to edit shared documents as easily as one edits a single author document. To achieve high responsiveness and to support disconnected collaboration in such systems, data are optimistically replicated [16, 44]; i.e. each user has a local copy of the document that can be modified independently of the other replicas. In addition, to achieve high availability, locking mechanism to handle concurrent operations is prohibited. In peer to peer collaborative editing, the systems allow replicas to diverge temporarily, but must eventually reach the same value if no more mutations occur. This consistency model is called Eventual Consistency (EC) [54].

Usually, the collaborative editing systems integrate a *synchronizer* tool that is in charge to propagate and merge the changes between different copies of the data. In order to provide a comfortable environment for collaboration, the synchronizer tool must merge correctly the modifications. Merging totally concurrent modifications on large scale collaboration is impossible. However, the system must reduce the human effort to obtain a correct merge. In the other case, the users correct by themselves the conflicts. If there is too much correction, the users may get frustrated and will quit the collaboration.

The synchronization algorithms are classified into state-based and operation-based synchronizers. State-based synchronizer sees only the current versions of the replicas to be reconciled, together with an archive of the last state they had in common. While, operation-based synchronizers work by keeping track of the complete sequences of operations that have been applied to each replica and, during reconciliation, attempting to synthesize a single unified view of the data structure's edit history [22].

[*] Email: mehdi.ahmed-nacer@loria.fr

In asynchronous collaboration mode, e.g Distributed Version Control System (DVCS) softwares, users modify their document in isolation and synchronize after to establish a common view of the document. Usually, these kind of systems manages the modifications as a set of state (state-based approach) as on git system [52] or So6 [32]. When a replica receives a remote state, it computes the difference between the local state of the document and the received one before merging the modifications. If there are modifications in the same part of the document in both versions, the system can return a conflict information to the user and let him resolve them. The conflict is generated when the system cannot merge the concurrent modifications.

Many solutions have been proposed to improve automatic merge. A distinction can be made between textual [37], syntactic [19] and semantic [13] 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. DVCS as git system supports any type of collaboration. The users can collaborate to produce XML files or a simple text document or software source code. For this purpose, in this paper we focused only of on textual merging.

Git system uses *state − based* approach to manage the concurrent modifications. During the merge procedure, *git merge* compares the local state with the remote one. If the document is modified at the same position, *git merge* produces a conflict. On the other hand, many *operation − based* approaches were suggested to solve concurrency control in collaborative editing [1, 12, 53]. Unlike *git merge*, these approaches represent the modifications as a sequence of operations that are integrated automatically in the document. Both approach kinds are designed to reduce the effort of users during the collaboration. However, study what degree their result satisfy the users on real collaboration is never established.

In [30], Mens recognize 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 [30] help in resolving direct conflicts [6], but often require manual intervention [6, 14, 45].

In addition, to reduce the conflicts during the collaboration and improve the productivity, study only how merge tools manage the modifications is not sufficient. Indeed, study the users' behavior during the collaboration is necessary. Until now, there is no tool that allows us to track a pattern of users' collaboration.

In this regard, this paper proposes a methodology to study the users' behavior during the collaboration, to compute the effort made by developers to correct their document when conflicts occur and evaluate the quality of merge produced by specific algorithms. We strive to understand the behavior of users during the collaboration, optimize the developer's time and effort, minimize conflicts occurrence, and thus improve the development team's productivity. To achieve this goal:

- we designed and implemented an open source tool[1] to measure the quality of any merging tool using large-scale software merges;
- we analyzed the users behavior through eight open-source repositories and more than 3 million 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 information, we diagnosed important class of conflicts and offered a solution to improve the merge result.

However, to track the pattern of users' collaboration and 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 System (DVCS) contains the results that the user corrected. Thus, in the histories, the merge result is the result intended by the users. This paper, focuses on the history of Git system.

The basic idea is to develop a tool that replays 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. This difference is the effort made by users in case where Git system uses op-based approaches. To reduce the users effort and improve the development team's productivity, we observe the common cases that create a conflict during the merge procedure to understand how the users collaborate. Then, we adapt a solution to solve them by using operation-based approach.

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

We validate our contribution by several experiments on large scale histories and perform a statistical test of significance. The experiments simulate traditional tool used for merging and the solution proposed. We measure the effort made by users in the document when a conflict is generated. Afterward, we compare our approach with traditional tool used for merging.

This paper is organized into eight sections. Section 2 describes the merge management by using existing approaches. Then, in Section 3 we describe our methodology and tool which allowed us to observe the different patterns of collaboration, to detect the different conflicts and to compute the effort made by users during merging procedure. Section 5 studies how users manage the important class of concurrent behavior which are considered as conflict by existing merge tools. Then in Section 6, we propose a solution to correct these specific conflicts and present the results of the experimental evaluation of our approach. Finally, we cite the related work and we finish with a conclusion.

2. Merge Management

The merge result depends strongly on the type of algorithms used. In state-based systems as in git [52], the modifications are executed by states, while using operation-based algorithms the modifications are executed by operations. In the following, we describe how the modifications are managed in both approaches state-based and operation-based.

2.1. State-Based Algorithm: Git Merge Tool

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 of the 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 [52]. 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.

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 in that area.

The basic idea is presented in Figure 1. *Git merge* compares 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 [51] as presented in Figure 1. These markers called awareness [10] are useful, especially if the size of the document is large. It specifies exactly the position of conflict, in addition to other information, like the modifications made by other users and the original document. The users are invited to make corrections on their document to solve the conflict and add modifications if necessary.

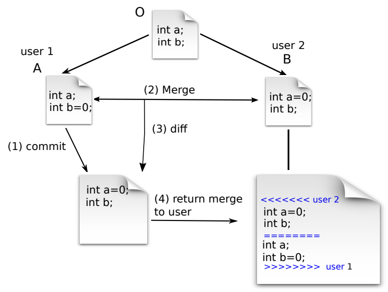


Figure 1. Conflict in state-based systems for collaborative editing

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 1, the merge conflicts but it could be avoided by the users, the conflict is called *false positives merge* (or direct conflict).

2.2. Operation-Based Algorithms: OT & CRDTs

Many operation-based algorithms are proposed and claim to integrate correctly the operations in the document. These algorithms respect the Eventual

Consistency (EC) model; i.e, the systems allow replicas to diverge temporarily, but must eventually reach the same value if no more mutation occurs. In this paper, we assume that a granularity of operations is a line. So, the modifications are executed per lines.

Operational Transformation (OT). [12, 32, 42] algorithms are operation-based are designed for collaborative editing context. They have been proposed to maintain the consistency of the shared document. For textual collaborative editing, they usually apply the *insert* and *delete* operations, and sometimes *update* operations.
To apply the operations at the correct position and to preserve the user's intention, OT algorithms transform the operation received before its execution with the concurrent one, to take into account the changes made on the document by other executed operations. In Figure 2, two users shared the same document initially "sstems" and work together to produce the document "system". User 0 inserts "y" at position 2 which intends to produce the document "systems", when concurrently, user 1 deletes the character at position 6 which intends to produce the document "sstem". When user 0 receives op2, it is transformed to take into account the effect of the concurrent operation op1, then op2 is transformed to $del(7)$ instead of $del(6)$ since the position of the concurrent operation (op1) is before the position of op2. While, on site 1 the operation op1 has not been transformed since the position of the concurrent operation (op2) is after the position of op1. Finally, both users produce the same document "system".

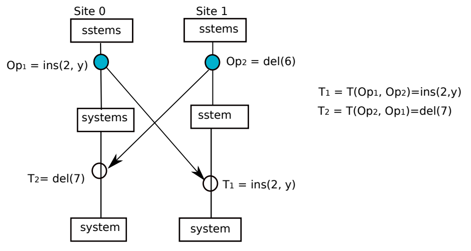
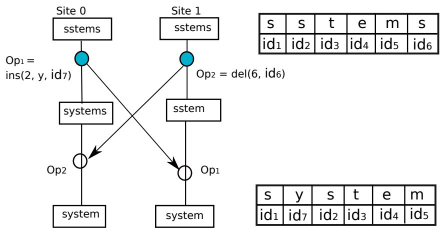


Figure 2. Integrate operation in OT algorithms

Although, OT algorithms allow to order the operations, problems can happen when two users modify concurrently the text at *the same position* since there is no order between the operations.

However, deploying such algorithm on real system should not merge automatically every concurrent operation silently. It is more appropriate to inform the users and let him check the result. For example, So6 [32] that is similar to *git merge* upon on OT algorithm, cannot merge silently the modifications when two concurrent operations are generated. the result is returned to the user and let him to solve the conflicts.

Commutative Replicated Data Types (CRDT). [34, 39, 43, 56] ensures consistency of highly dynamic contents on peer-to-peer networks emerged. Unlike OT algorithms, CRDTs require no history of operations, and no detection of concurrency in order to ensure consistency. Instead, they are designed for concurrent operations to be natively commutative by actively using the characteristics of abstract data types.



s	s	t	e	m	s
id₁	id₂	id₃	id₄	id₅	id₆

s	y	s	t	e	m
id₁	id₇	id₂	id₃	id₄	id₅

Figure 3. Integrate operation in CRDT algorithms

Instead of the previous example 2, in Figure 3 each character is identified by a unique identifier. The identifiers are ordered for instance following the lexicographic order. During the insertion of Op_1, the user generates a unique identifier id_7 for this operation. When user at site 1 receives this operation, it inserts the content and the identifier at the correct position. However, when a user in site 1 deletes at position 6, it deletes the content of the identifier id_6. Then, when user in site 0 receives this operation, it deletes also the content of id_6. Finally, both users converge to the same document.

3. Methodology

In order to detect the merge behaviors that create conflicts, we deploy a framework[2] which allows us to observe the merge procedure and locate easily the conflicts. In addition, this tool replays the collaboration as on DVCS histories and computes the effort made by users in the conflicting document by using the traditional algorithms *git merge* – state-based – and by using other operation-based algorithms. The difference is the gain in a user's effort.

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.

Assume that the modifications made by users to correct their document, when the conflict occur is the ground truth for merging procedure. The methodology consists of reproducing the same collaboration as on the histories of DVCS and observe through a tool the effort

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

made by users when conflicts occur. Firstly, by using the traditional algorithms (*git merge*), afterward by using our operation-based algorithms. Then, we compute the number of corrections performed to reproduce the same document as generated manually by the users.

In textual merging [31], the most common approach is to use *line-based merging*. Thus, operation-based algorithms evaluated in this paper manage the modifications per lines. They create a new operation for each line modified.

3.1. Corpus available

A large number of available Distributed Version Control Systems (DVCS) history publicly available constitutes a very interesting corpus of distributed asynchronous editing traces. DVCS are widely used to manage large scale asynchronous collaborative editing. For instance, the Linux kernel is developed by thousands of programmers around the world using Git [52]. Several web-based hosting services for software development projects provide large DVCS history such as GitHub (3.4M developers and 6.5M repositories)[3], Assembla (800,000 developers and more than 100,000 projects)[4], or SourceForge(3.4M developers and 324,000 projects)[5]. In this paper, we selected traces from the most used system: Git.

3.2. Conflicts in Practice

Bird and colleagues [5], 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 collaboration. Still, users are the authors of the conflict resolutions that are available in these histories.

We consider the merge state available in git history as the user expected result. Of course, the merge may generate problems. For instance, 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 [23]. 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.

3.3. Framework

To replay the same collaboration as in the history of git by using operation-based algorithms, we need to transform the states of the document extracted from the Git history to the whole of operations ready to be used by operation-based algorithms. Thus, we provide a framework which is the base of our experiment. The framework implements also the operation-based algorithms and computes the size of modifications made by users to correct their document. The framework is open source and publicly available in order to let researchers evaluate their own algorithms. It is developed in Java, and reveals the source on GitHub platform[7] under the terms of the GPL license.

After retrieving the traces and implementing the framework, we replay the collaboration using operation-based algorithms and we compute the user's effort.

3.4. 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

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

[6]https://github.com/git/git, starting from commit 0da7a53a
[7]http://github.com/PascalUrso/ReplicationBenchmark

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 4.

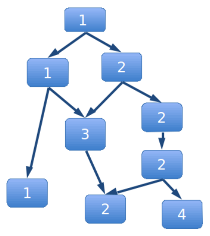


Figure 4. Each node is labeled by its replica identifier

For performance reasons, the framework heuristically minimizes 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 [29] 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 [33] 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 5, 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).[8] 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 [30].

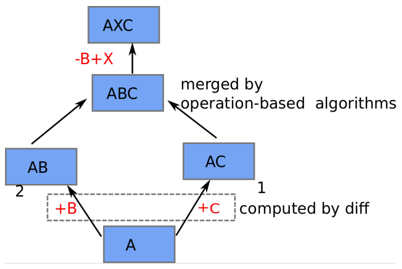


Figure 5. Imitate human merge correction

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.

3.5. Merge computation

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 [33] to calculate this difference. The algorithm returns a list of modifications: insertions and deletions of text blocks. The framework calculates two metrics:

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

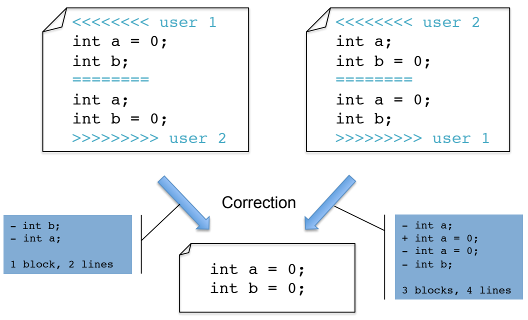
Merge blocks the number of modifications in the difference.

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 6, 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.



```
<<<<<<<< user 1
int a = 0;
int b;
========
int a;
int b = 0;
>>>>>>>>> user 2
```

```
<<<<<<<< user 2
int a;
int b = 0;
========
int a = 0;
int b;
>>>>>>>>> user 1
```

```
- int b;
- int a;

1 block, 2 lines
```

```
int a = 0;
int b = 0;
```

```
- int a;
+ int a = 0;
- int a = 0;
- int b;

3 blocks, 4 lines
```

Figure 6. 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

In order to improve the textual merge result, we evaluate and compare the different operation-based algorithms and *git merge* in different DVCS repositories, and we observe their behaviors. In the following, we present the algorithms evaluated, the experiments performed and after we present the results.

4.1. Algorithms Evaluated

Since the git system is based on peer to peer architecture, the algorithms evaluated have the particularity that support peer to peer collaboration. We evaluated in this experiment:

Git Merge Tool. the default *git merge* algorithm (described in Section 2.1) used by git system.

We evaluated also the usual textual algorithm used for collaborative editing: Operational Transformation (OT) algorithms and Commutative Replicated Data Type (CRDT) algorithms [35, 39, 43, 56].
The most OT algorithms that exist, use a central component. Some others do not require a central server such as SOCT2[47], MOT2[7] and Goto [50]. However, these algorithms require some property that only TTF [36] approach ensures. In addition, the impact of these algorithms on merge result is same since they apply the same transformation functions. For this reason, we evaluated only SOCT2 among OT algorithms.

SOCT2/TTF. SOCT2 [47] algorithm is a representative Operational Transformation (OT) algorithm that do not make any assumption on using a central server for a total order of operations. The principle of this algorithm is illustrated in Figure 7. When a causally ready operation is integrated on a site, the whole log of operations is traversed and reordered. After reordering, causally preceding operations come before concurrent ones in the history buffer. Finally, the remote operation has to be transformed according to the sequence of all concurrent operations.

Unfortunately, many proposed transformation functions fail to satisfy the concurrency control, as shown in [18]. To our best knowledge, the only existing transformation functions for collaborative editing that satisfy the concurrency control are the Tombstone Transformation Functions (TTF) [36]. To overcome problems, TTF approach keeps all characters in the model of the document, i.e. deleted characters are replaced by tombstones.

WOOT. WOOT [35] is the first CRDT algorithm which was proposed. In WOOT algorithm, the elements are

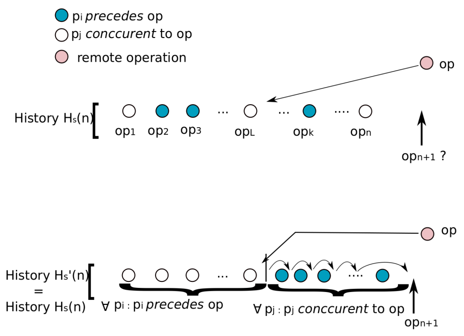


Figure 7. Integrate a remote operation in SOCT2

uniquely identified. An insertion is defined by specifying the new element identifier, the element content and the identifiers of the preceding and following elements. Concurrent operations determine partial orders between elements. The merging mechanism can be seen as a linearisation of the partial order to obtain a total order. In Figure 8, two users shared the same document initially ABC. User 1 inserts X between A and B to produce AXBC, when concurrently user 2 deletes B and produces AC. The element deleted is just marked as invisible to users. When user 2 receives the operation from user 1, it is executed in a correct order. Since, each element has a unique identifier, when user 1 receives the operation from user 2, the correct element is deleted. However, if two concurrent insertions are generated at the same position, the merged operations can generate a conflict document.

WOOTH [1] is a new version of WOOT that improves its performance by using a hash table.

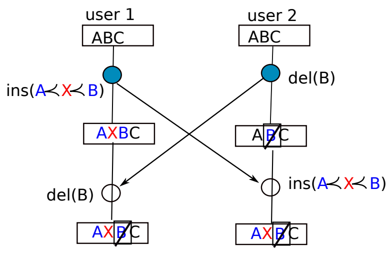


Figure 8. Integration in WOOT

Logoot. Logoot [56] is another CRDT approach that ensures consistency of textual documents. Logoot associates to the list of elements of the structure, an ordered list of identifiers. Identifiers are composed of a list of positions. Positions are 3-tuples formed with a digit in specific numeric base, a unique site identifier and a clock value. When inserting an element, Logoot generates a new identifier. Identifiers have unbounded lengths and are totally ordered by a lexicographic order. So a new identifier can always be generated between two consecutive elements. Different strategies can be adopted to produce the new identifier [57], all of them using randomness to prevent different replicas to produce concurrently close identifiers.

Treedoc. [39] is a CRDT algorithm that represents the document by a binary tree structure. The element identifier is the path to the element in the tree. If two users insert concurrently at the same position, Treedoc creates a major-node that contains the two elements.

4.2. Empirical Study

We analyzed eight 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/Gitorious, and not a mirror of another system repository such as SVN. We also selected the git repository of the git software it-self, 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.

Since there is no collaboration when the files are not merged, the framework replays only histories of files that are merged at least one.

In Table 1, we present the characteristics of eight projects. The head commit sha1 used to run our experiments is presented above the name of each repository. The characteristics are computed per file. Based on these files we compute the total number of commits and merges that affected the files, the number of operations and the maximum users that collaborate on each file of the project.

During the simulation of the collaboration, the framework computes number of corrections (Merge blocks and merge lines). Depending on the algorithms used and how an operation is generated, the order of blocks and lines in the document will be different. Thus, the number of correction changes from one algorithm to another.

4.3. Results

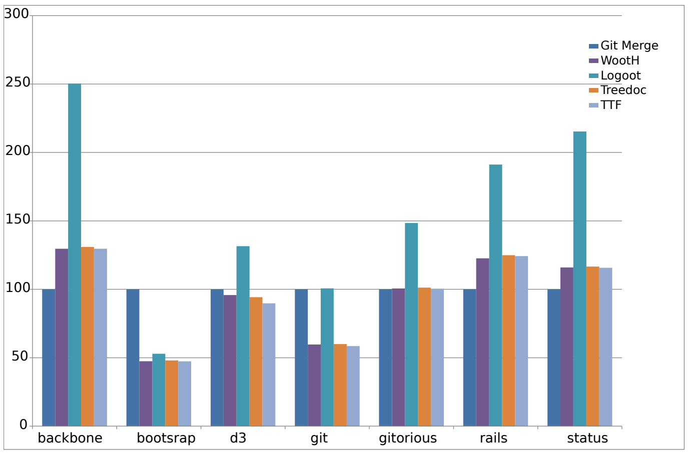
Figure 9a and 9b present respectively the percentage of merge blocks and merge lines for *git merge*, TTF, WOOT, Logoot and TTF. Since *git merge* is the default algorithm used in git system to merge the modifications, we use its results as the reference (=100%).

Difference between state-based and operation-based merges During the merge procedure, *git merge* detects which part of the document is changed. It analyzes also the modifications made by each user. If two

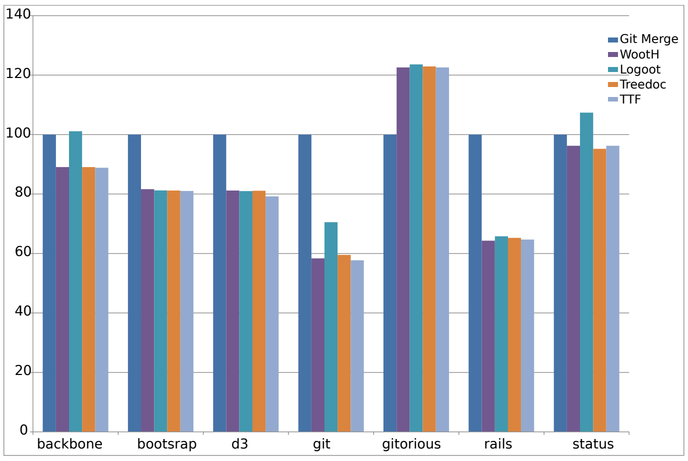
[9] `https://github.com/popular/starred`, April 2013

Table 1. Projects characteristics

Project	cloud/backbone	twitter/bootstrap	mbostock/d3	git/git	gitorious/mainline	rails/rails	statusnet/mainline
Head sha1	6ac7704c	37d0a30	d1d71e1	8c7a786b	c1105eb	36f7732	d7880c1
Files with merge	11	69	38	558	72	352	213
Commits	2293	6009	2192	32958	4136	28895	12057
Merge	274	434	282	5646	151	1153	1218
Num.Operation	2605	7626	2352	33084	3915	26899	11953
Max. Replica	13	10	30	59	5	6	11
Merge block by *git merge*	155	1614	648	3184	489	442	1159
Merge line by *git merge*	895	14658	4658	10159	2303	3899	4783



(**a**) Merge blocks on git projects



(**b**) Merge lines on git projects

Figure 9. Merge blocks and merge lines

users insert concurrently the same content at the same position for instance, *git merge* merges correctly the updates. In addition, *git merge* tool asks users to correct their document each time the merged document conflict. While, operation-based algorithms merged automatically the modifications. When two users insert concurrently the same content at the same position, a unique identifier is generated for each line by using CRDTs, and two duplicated operations are generated by using TTF algorithm. Then, all operation-based algorithms generate a duplicated text and needs an additional effort for users to correct their document. For this reason, *git merge* outperforms operation-based algorithms in some repositories.

Another difference is due to a very common type of collaboration: concurrent edits done on two consecutive blocks. Indeed, *git merge* requires more correction in this case than operation-based algorithms. For instance, when two users modify concurrently two consecutive blocks, *git merge* detects that both users modify the same part of the document. Then, it returns a conflict even if is not. This case is managed well by operation-based algorithms. The edits are merged automatically, and no conflict detected.

To understand more the difference between state-based and operation-based merges, we study in the next section the different collaboration patterns by using *git merge* and operation-based algorithms.

Difference between operation-based merges Even if OT and CRDT algorithms have a completely different behavior to merge the operations, the result of TTF and WOOT are almost the same in merge block and merge line. So, change the manner of operations' generation is not sufficient to improve the quality of the merge and reduce the users effort.

The main difference occurs when concurrent edits affect the document at the same position. Logoot uses randomness to generate its identifier. So it will more frequently interleave the lines added concurrently. If the developer must keep only one of the edits, he 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 WOOTH, Treedoc 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 1 block and 4 lines.

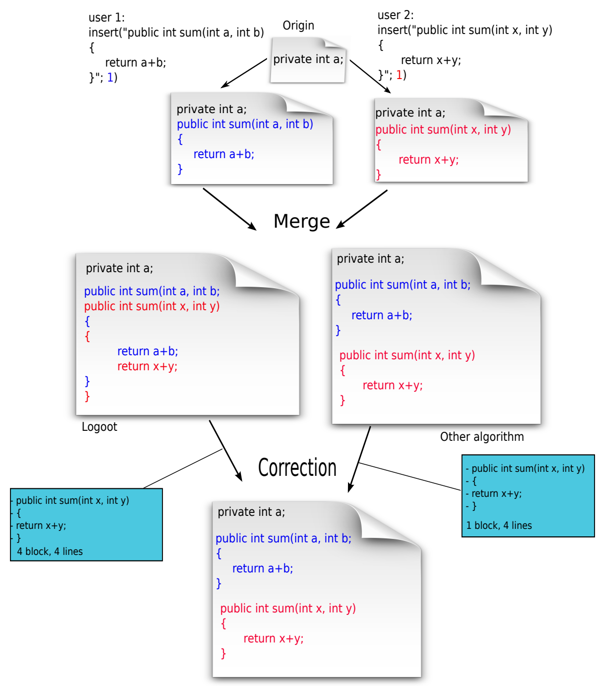

user 1:
insert("public int sum(int a, int b)
{
return a+b;
}"; 1)
Origin
private int a;
user 2:
insert("public int sum(int x, int y)
{
return x+y;
}"; 1)
private int a;
public int sum(int a, int b)
{
return a+b;
}
private int a;
public int sum(int x, int y)
{
return x+y;
}
Merge
private int a;
public int sum(int a, int b;
public int sum(int x, int y)
{
{
return a+b;
return x+y;
}
}
Logoot
private int a;
public int sum(int a, int b;
{
return a+b;
}
public int sum(int x, int y)
{
return x+y;
}
Other algorithm
Correction
- public int sum(int x, int y)
- {
- return x+y;
- }
4 block, 4 lines
- public int sum(int x, int y)
- {
- return x+y;
- }
1 block, 4 lines
private int a;
public int sum(int a, int b;
{
return a+b;
}
public int sum(int x, int y)
{
return x+y;
}

Figure 10. Different merge in operation-based approaches

Gitorious repository For both metrics – blocks and lines –, the behavior 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 analyzed, 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 [49]. 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.

5. Conflicts in Practice

To improve the merge procedure in asynchronous systems and understand more the results obtained , we launch the experiment and we observe through the framework which part of the document conflict, and detect where the user's effort is the most important. This allowed us to understand the conflicts and propose solutions to solve them automatically.

The behavior of users during the collaboration is different from one project to another. Several factors can influence the collaboration such as, number of users, type of project, proximity between users, latency in networks ...etc. For this reason, it is difficult to detect and know what are the most common cases that create conflicts during the collaboration. The framework helps us to extract these scenarios.

Addition at the same position. This kind of conflict happens when two users modify concurrently the text at the same position (not necessarily the same content) since there is no order between the operations.

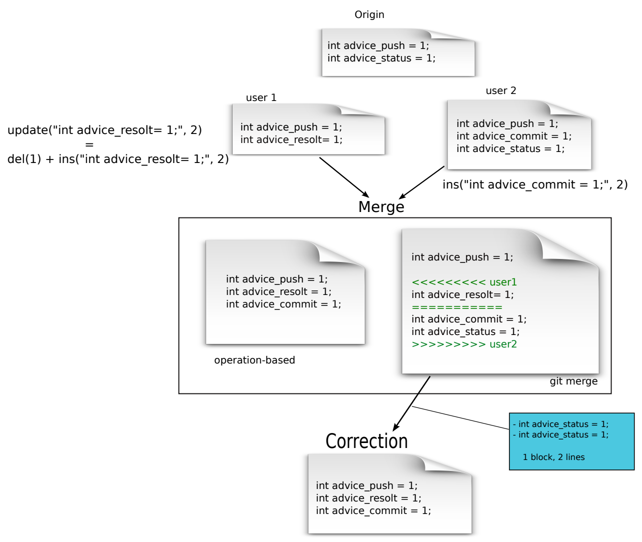


Figure 11. Addition at the same position

In this kind of concurrency and on textual merge, operation-based algorithms can outperform state-based approach when an update operation falls in concurrency with insert operation as shown in Figure 11. Initially, both users shared the same document. afterward, user 1 updates the first line by `"int advice_resolt=1;"` when concurrently user 2 inserts `"int advice_commit=1;"` at the same position. Git system cannot merge the documents since both users make modifications at the same position. On the contrary, operation-based approach merges the document correctly. Indeed, it transforms the update operation to a delete followed by an insertion. When user 2 receives the delete, it deletes `"int advice_status=1;"` and after it inserts `"int advice_resolt=1;"`, while user 1 executes the received insertion `"int advice_commit=1;"` in the correct position. Since there is no order between the concurrent operation , operation-based algorithm may also integrate the operations in the wrong order and force users to make corrections. However, in the worst case, operation-based algorithm requires 2 modifications (one delete and one insert). Depending how *git merge* presents the conflict result for users (user1/user2 or user2/user1), the users require at least one modification, and in the worst case three modifications to produce the correct document. In this example, the difference between the approaches is not very large, but this difference is larger in real collaboration since users produce many copy/paste operations.

We notice also that this case of conflict is very common on real collaboration.

Concurrent consecutive modifications. This conflict happens when two users modify concurrently the documents in two consecutive position. Using state-based approach, *git merge* considers that both users modify concurrently the same area of text and then produce a conflict. The users verify the merged documents and choose one of both version proposed by *git merge*. However, using operation-based approach, this case is not considered as a conflict and the modifications are merged correctly.

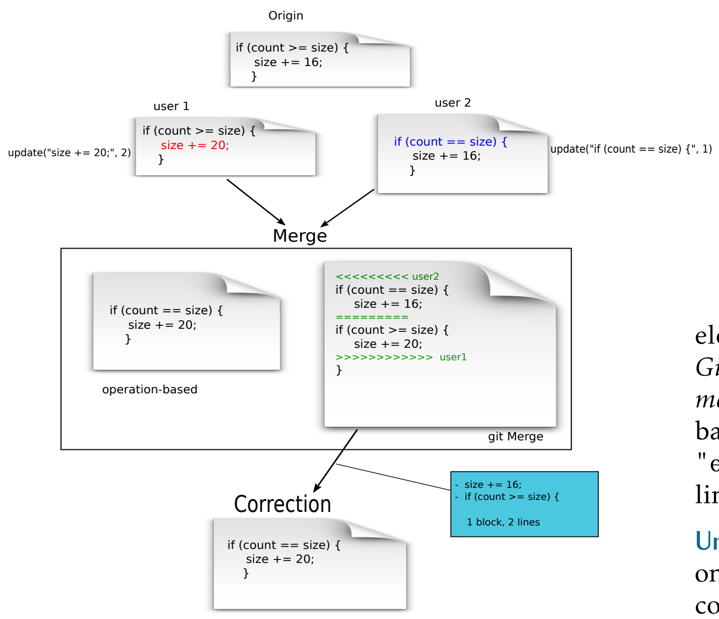


Figure 12. Concurrent consecutive modifications

In Figure 12, user 2 updates the first line by `"if (count == size)"`, while concurrently user 1 updates the second line by `"size+=20"`. Even if both users modify the document in different position, *git merge* detects that the modifications are made in the same area. Then, it returns a conflict result to users. In contrast, operation-based algorithms merge automatically and correctly the modifications.

Accidental Clean Merge (ACM). When users insert the same content at the same position, this is called accidental clean merge. *Git merge* manages well this kind of conflict as presented in Figure 13. Using operation-based algorithms that consider the modifications per line, a new operation is generated for each line, thus a duplicated line is inserted in the document and users must to correct line per line.

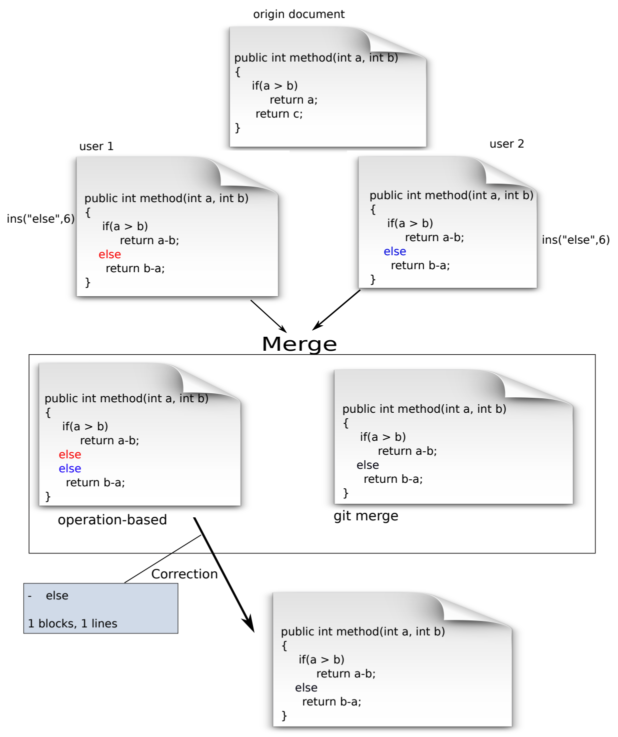


Figure 13. Accidental clean merge

In Figure 13, both users insert concurrently the same element at the same position, `"else"` at position 6. *Git merge* detects that two lines are identical. Thus, *git merge* merges correctly the document. While operation-based algorithm produces duplicated lines `"else;"` `"else;"` since it generates a different operation for each line.

Undo/Redo. The undo/redo operations are very useful on collaborative editing systems. They allow any user to correct any edit operation at any time. On git system the undo/redo operations are generated when users revert their modifications to one of the previous states[10].

[10] Command "git revert"

However, using the operation-based algorithms can produce a conflict document.

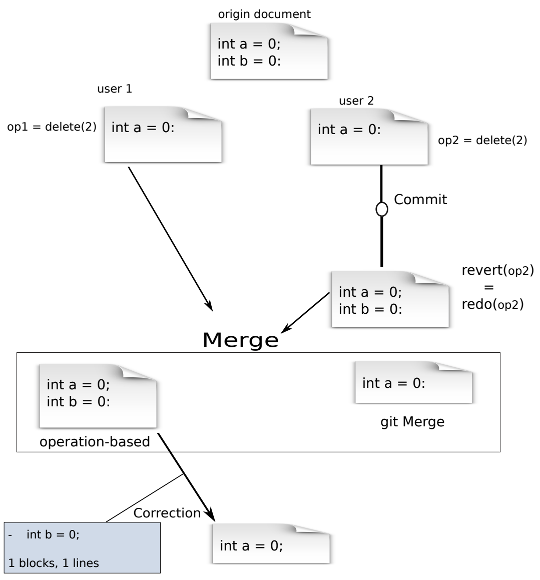


Figure 14. undo/redo operation

Figure 14 illustrates an example where state-based approaches manage well undo operations while operation-based algorithm creates a conflict. Initially sites 1 and 2 shared the same document "int a=0;""int b=0;". Site 1 deletes line 2 which intends to produce the document "int a=0;", while concurrently, site 2 deletes the same line and undo its operation. Then, site 2 does not change the initial document. During the merge operation, *git merge* merges both states and produces a correct document "int a=0;". While, operation-based algorithm creates a conflict in the user's documents. When site 1 receives the operations from site 2, it has reinserts "int b=0;" since site 2 cancels its deletion. Thus both users produce "int a=0;" "int b=0;" document. To have the same document as in the history of git, both users must delete "int b=0;" from their document.

6. Improve Textual Merge

To improve the performance of operation-based algorithms in asynchronous systems, we propose some improvement to avoid the "most" common cases that create conflicts: accidental clean merge and undo/redo conflicts. We adapt the Tombstone Transformation Functions (TTF) approach [36] to avoid these kind of conflicts. Before explaining our method we describe TTF algorithm.

6.1. TTF algorithm

TTF approach [36] was proposed to solve the problems occurred on Operational Transformation (OT) algorithms (described in section 2.1). OT approaches are based on the transformation property C_1 and C_2 [42] and some transformation functions. C_1 ensure that the execution of any pair of concurrent operations obtains the same result on all replicas. Using a central server, C_1 is sufficient. However, on peer to peer collaboration the system require C_2 [55]. These functions change the index of the operation to take into account the effects of the concurrent operations. Imine et al. [18] have shown that few operational transformation algorithms proposed fail to satisfy C_1 and C_2 conditions. In this context, Tombstone Transformation Functions (TTF) approach was introduced [36]. It overcomes the problems by keeping all characters in the model of the document. When user deletes an element, it is not physically removed from the document, but just marked as invisible to users, i.e. deleted elements are replaced by tombstones. However, TTF approach does not solve the conflict described previously.

6.2. Clean Merge Undo Algorithm (CMUndo)

To improve TTF approach on asynchronous systems we add some transformation functions to take into account the case of undo/redo and accidental clean merge.

- **undo/redo**:
  Undo/redo operations in collaborative editing are very useful but considered as difficult problem [8, 26, 28, 55, 57]. They allow users to correct any edit operation at any time. In git system, the only information that can be useful to detect a real undo/redo operation, is the message introduced by users when they revert their modifications. Unfortunately, not all users specify on their messages that is a revert operation. For this reason, it is difficult to manage this kind of conflict by a revert mechanism. To simplify the operation, we assume that all *delete* operations are considered as *undo* of insert operation. Moreover, before inserting an element in the model we test if this operation is a *redo* or a simple insert as shown in algorithm 1. The algorithm receives two arguments: position of insertion and the content of insertion. it returns the operation to be applied in the document and to be sent to other replicas. In line 1, the algorithm tests if it can find the element as a tombstone (invisible to users) at the same position. In this case this operation is considered as redo, in the other case it is considered as a simple insertion. [11]

[11] The user can delete an element, and after reinserts the same element in the same position without an explicit redo operation. During the collaboration there is a little chance to have this case.

Algorithm 1: LocalInsertion(pos, content)

```
Input: The content and the position on the
       document
Output: operation
if ((getDoc(pos).visibility = false) and
(getDoc(pos) == content) then
    return redo(position, content);
else
    return insert(position, content);
```

However, to manage the undo/redo operations, the algorithm uses the computation of *line visibility degree* [57]. When a line is created, it has a visibility of 1. Each time the line is deleted, the algorithm decreases its visibility degree. When a delete is undone or an insert is redone, the algorithm increases its line visibility degree. The line is visible only if its visibility degree is greater than 0.

- **Accidental Clean Merge (ACM):**
  ACM [27] happens when users insert concurrently the same content at the same position. During the merge procedure, the merged document may contain a duplicated element. OT algorithms (described in section 2.2) can be used to avoid these conflicts. They detect during the transformation phases the ACM cases and might transform them to *noop* operations (*nil* value).
  To ensure consistency of the document when two concurrent operations made in the same position, TTF and other OT algorithms use site id as a priority [41, 48]. Using this solution with ACM transformation may create a divergence as presented in Figure 15. Three sites shared the same document initially "ABC". Site 0 and site 2 inserts concurrently the same element "X" at position 1 and produce "AXBC" document. While, site 1 inserts concurrently "Y" at position 2 and produces "AYBC" document. To avoid the ACM conflict, when site 0 receives the operation from site 2, it does not execute the operation since both users insert the same content at the same position. However, when site 0 receives the operation from site 1, it detects that both operations have the same position. Since OT algorithms give the priority to replica number, op2 is transformed with op1. It is transformed to insert at position 2 instead of position 1. Finally, site 0 produces "AXYBC" document. On the other hand, when site 1 receives op1 from site 0, it is not transformed, since the priority is given to site 0. Thus, site 1 produces "AXYBC" document. Afterward, when site 1 receives the operation from site 2, it transforms it to insert "X" at position 3 and produces "AXYXBC" document. On site 1, ACM is not detected and replicas diverge.

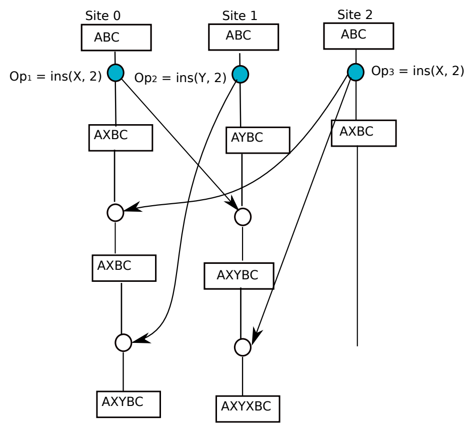


Figure 15. ACM divergence by using traditional OT

  For this purpose, we propose a solution to use the element of operation as a priority. As an example, in this paper we chose the content's hash code. During the transformation, we add a new test to detect the accidental clean merge cases. Indeed, the algorithm 2 tests in lines 4 and 5 if there are two concurrent insertions at the same position with the same content. In this case, it returns a *noop* operation, in the other case it makes a traditional transformation by comparing the position and the content's code. Applying algorithm 2 in Figure 15, the problem is resolved. Indeed, when site 1 receives an operation from site 2, the insertion of "X" is transformed into position 2 instead of position 3 since the hash code of "X" is less than hash code of "Y". Thus, the algorithm detects that two "X" are inserted at the same position. Site 1 detects ACM and does not execute op3. Finally, all replicas converge and produce "AXYBC" document.

In the following, we provide an experiment to compare our solution with *git merge* and the existing operation-based approaches.

6.3. Adapted merge evaluation

To observe the merge results improvement produced by our solution, we replay the same experiment made above (Section 4.2), only this time by using our solution.

During the experiment, the framework computes the number of accidental clean merge and undo/redo cases. Table 2 presents the number of accidental clean merge and the number of undo/redo operations produced in git repositories.

Algorithm 2: Transform(op1, op2)

```
Input: operations to transform : op1 and p2
Output: operation applied on the document : op
1  Let $c_1$ and $c_2$ respectively the content of op1 and 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 ($t_1$ = insert) and ($t_2$ = insert) then
5      if (c1=c2) and (p1=p2) then
6          return noop();/* An operation that returns null value */
7      else
8          if (p1 > p2) or (p1=p2 and
9          HashCode(c1) > HashCode(c2)) then
10             return insert(c1, p1+1,$Site_i$);
11         else
12             return insert(c1, p1,$Site_i$);
```

Table 2. ACM and Undo/Redo in git repositories

Project \ Features	Accidental Clean Merge	undo	redo
backbone	271	1357	1137
bootstrap	563	7210	3957
d3	7	19877	218
Git	1272	42734	1614
Gitorious	750	932	513
rails	426	5329	16172
status	2297	9060	6352

6.4. Results

Figure 16a and 16b represent respectively the percentage of merge blocks and merge lines for TTF, *git merge* algorithms and CMUndo. To observe how undo/redo and accidental clean merge operations impact on merge results, we present also Clean Merge (CM) algorithm that detects only accidental clean merge cases (without undo/redo operations). We consider the merge blocks and merge lines produced by *git merge* presented in Table 1 as the reference (=100%).

The number of merge blocks and merge lines correlates well with the number of accidental clean merge and undo/redo operations represented in Table 2. Indeed, more accidental clean merge and undo/redo operations detected in repositories and more the difference between our solution and other algorithms grows. For example, in git repository we detected a large accidental clean merge and undo operations, so the gain of user's effort obtained by our solution is around 54% in git repository. In Gitorious repository, *git merge* is more efficient than all algorithms in merge block, This is due to a specific collaboration pattern in the file "`diff_browser.js`". The users collaborate independently and each one produces almost the same document. During the merge procedure *git merge* manages well this kind of collaboration.

CMUndo algorithm implements more functions to detect the accidental clean merge and undo/redo operations. It reduces in all cases the effort of users, except in Gitorious repository.

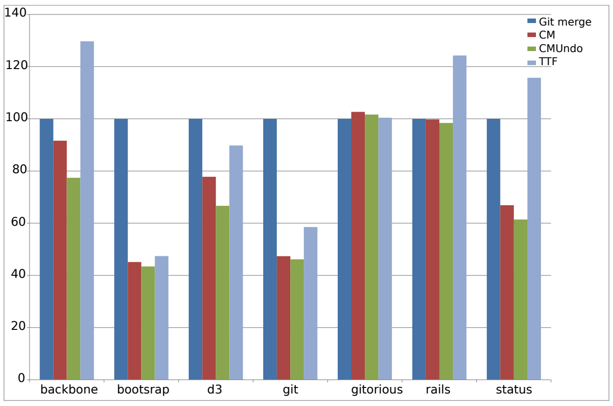
In Figure 16a and on repositories that contain much accidental clean merge and undo/redo operations, *git merge* algorithm outperforms TTF algorithm but remains worse than CMUndo algorithm. Indeed, TTF algorithm does not manage the accidental clean merge operations (see Figure 13), while *git merge* algorithm can merge them correctly and can retrieve some identical lines when two concurrent blocks are inserted. CMUndo is more efficient than all other algorithms except on Gitorious repository. Indeed, CMUndo takes the advantage of *git merge* since it detects accidental clean merge operations and takes the advantage of operation-based algorithms since it manages well the concurrent addition at the same position. Except for Gitorious repository, CMUndo algorithm is the best.

In Gitorious repository *git merge* is more efficient than all algorithms on merge block, This is due to a specific collaboration pattern on the file "`diff_browser.js`". The collaboration in this file begins with the merge of two branches that have no ancestor in common. However, these two branches contain states with common lines that the *git merge* tool is able to merge.

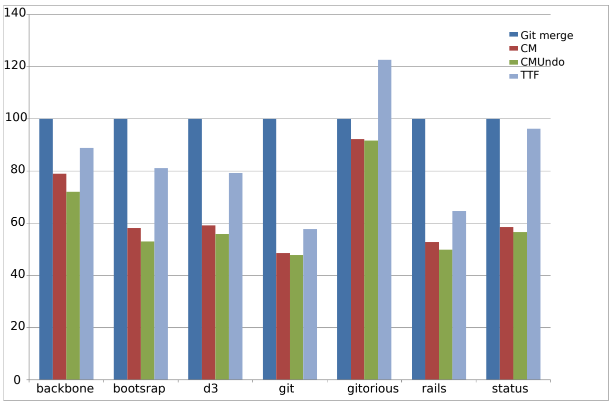
However, the impact of accidental clean merge operations on merge result is greater than undo/redo operations. Indeed, CM algorithm that manages only accidental clean merge cases and CMUndo algorithm that manages accidental clean merge and undo/redo cases improve almost the same merge result. The difference is only 3%.

Using *git merge* algorithm in asynchronous system creates more conflicts that CMUndo algorithm, Consequently, the document cannot be merged and users make more correction on their document. Comparing *git merge* and CMUndo algorithms, the later gain 54% on git repository and 59% on bootstrap repository. However, it loses just 1% on Gitorious repository.

In Figure 16b, it is clearly that CMUndo algorithm is the best. It outperforms widely all other algorithms and especially *git merge* algorithm. More algorithm generates merge blocks and more the document require corrections. In Figure 16a we found that CMUndo generate less blocks than *git merge* algorithm, for this reason the users introduce many lines by using *git merge* algorithm than CMUndo algorithm.
In addition, when a conflict occurs there is a high probability to generate a large block in state-based than



(**a**) Merge blocks on git projects



(**b**) Merge lines on git projects

Figure 16. Merge blocks and merge lines

operation-based approaches. Indeed, using operation-based approach, some operations can be inserted correctly, while on state-based approaches, the merge procedure depends on blocks. Then, when states are mixed the users require much correction. For this reason, TTF algorithms outperform *git merge* algorithm on merge lines. In Gitorious repository and precisely in "`diff_browser.js`' file, two users insert concurrently a large block with a content almost the same. During the merge procedure, *git merge* can merge correctly the identical lines while operation-based approaches do not. for this reason, *git merge* outperform TTF algorithms. We notice that, this kind of collaboration is specific and rarely comes.
Using CMUndo algorithm on asynchronous system, the users require few corrections, while *git merge* algorithm creates more conflict and require more corrections. Comparing *git merge* and CMUndo algorithms, the later gain 52% of lines on git repository and 57% on bootstrap repository.

To summarize the experiment, we compute the total merge blocks and merge lines on all repositories. We found that for 1335 files, we compute 5799 accidental clean merge, 118409 undo/redo operations, *a gain of 3583 blocks and 21675 lines by using CMUndo algorithm*. Figure 17 presents the total merge blocks and merge lines. In addition, we separate both algorithms (accidental clean merge –CM– and undo/redo) from our approach to observe the effect of each one on the result. TTF gains 26% in blocks and 23% on lines, while our solution gains 43% blocks and 50% lines. Moreover, accidental operations have the greatest effect on the document with a gain of 40% in blocks and 45% on lines. While undo/redo operations represent a gain of only 5% on blocks and lines.

Statistical analysis. We also perform a statistical test of significance. Since the dataset used is independent,

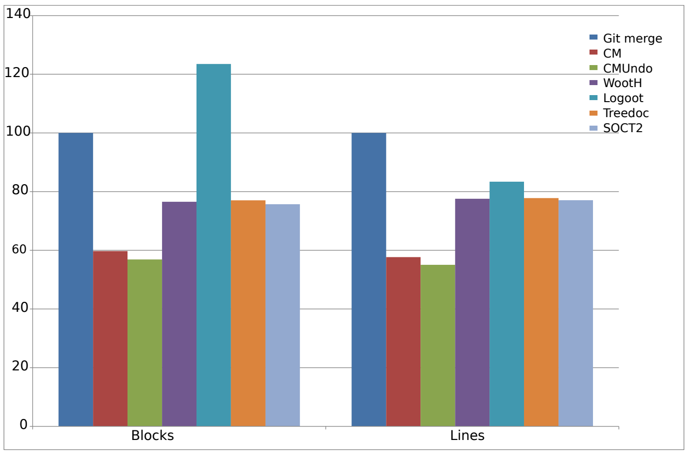


Figure 17. Total merge block and merge line

a non-parametric analysis method would be the most adapted approach for analysis. Using Kruskal-Wallis test[12], 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 operation-based merge (including Logoot) is in 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).

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

We present here aggregated results, but our framework produce results detailed per file. Analyzing these 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 Gitorious repository we obtain a different outcome.

7. Related Work

As presented in this paper, evaluate software merging by using operation-based algorithm during a concurrent collaboration plays an important role in the software development process. Indeed, understand the conflicts and resolve them, improve the productivity of the development team. Here we discuss some related work on evaluation of merge results.

Textual, syntactic and semantic merging is widely studied in [13, 19, 21, 31, 37, 40]. However, the git system deploys a generic model to allow any collaboration. The users can collaborate to produce XML files or a simple textual collaboration such as software source code. For this purpose, it is difficult to implement semantic and syntactic algorithms on git system. Thus, in this paper we focused only on textual merging.

In [27], many policies are proposed to solve conflict in structured documents such as XML files or file systems. These policies can be applied with our methodology to manage the files of git system. In this manuscript, we focused only in a simple linear text such as software development.

Palantir [45], 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 [20] proposes a novel conflict minimization technique that evaluate task constraints in a project to recommend optimum task orders for each developer. However, all methodology has been proposed just to measure the size of the conflict and the quality of the merges. Among them, no paper published to understand the conflicts and using operation-based merge.

Bayou [38] proposed a technique to maintain the consistency of the shared document. It used an epidemic algorithm to propagate modifications between weakly consistent replicas. If the merge procedure cannot find a solution, conflict resolution is delegated to the user. However, the authors do not compute the conflicts and the efforts made by users. D.Perry et al. [37] studied the various aspects of parallel development in the context of a large scale software development. They observed a large collaboration and studied some interfering changes. However, they do not offer a solution to merge correctly the modifications. In [24, 37] the authors specify that 90% of the modifications can be merged without detection conflict and only 10% cannot be merged automatically, since the tool does not consider any syntactic or semantic information. The authors do not study the effort made by users to correct the conflicts.

In [32], the authors propose an operational transformation algorithm that realizes a file system synchronization. However, The only operational transformation designed for collaborative editing and respect the transformation property C_1 and C_2 [42] is TTF approach evaluated in this paper.

On the other hand, operation-based algorithms designed for concurrency control such as Operation Transformation (OT) algorithms are widely studied on [1, 12, 53]. All the studies are focused on synchronous systems and they are focused on execution time or memory occupation. Recently new approaches called Commutative Replicated Data Type (CRDT) are proposed [35, 39, 43, 56] to be a substitution of OT algorithms. As OT algorithms, these approaches are evaluated only on execution time and memory occupation in [1] and [2]. *Git merge* algorithm that is widely considered as the gold standard for merging document on asynchronous systems. It is widely studied and presented by many researchers in [11, 22, 25, 46]. However, study the merge result to reduce the user's effort in asynchronous system by using operation-based approaches are never studied.

An awareness mechanism can be independently added upon the same kind of merge algorithm without affecting their result [4, 17]. So, an awareness mechanism can be added in system upon CMUndo algorithm. If a conflict occurs, the system proposes to users an automatic merge and they can accept it without efforts. It is possible also to add modifications in the automatic merge if necessary.

In this regard, this paper studies for the first time a decentralized solution that can offer a better merge than usual tool.

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 [45]. For instance, the so6 tool [32], 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 [3], we explain how move 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.

8. Conclusion

This paper presents an evaluation of eventual consistency algorithms in asynchronous systems, designed for collaborative editing. Firstly, we proposed a methodology to measure the quality of the merge algorithms in asynchronous collaborative editing systems. Then, we observe the users' collaboration to understand the common conflicts. Finally, we presented a solution to overcome the most cases of conflicts that can be occur during the collaboration by using a decentralized eventual consistency algorithms. Our contributions are made through an open-source framework which allow us to observe the collaboration and detect the real conflicts. The tool simulates a real collaboration as on the history of git repositories by using state-based and operation-based approaches. It computes the number of conflicts and the number of corrections requires by users to merge correctly their document.

Merging automatically the modifications can help users during the collaborations. When concurrent modifications occur, the merge tool can create conflicts. The users make an effort to correct their document. Reducing the user's effort improve the quality of collaboration and encourage users to work collaboratively.

In this paper, we observed the collaboration and studied the case where concurrent modifications interfere. We evaluated operation-based algorithms on asynchronous corpus. We found that, the existing operation-based algorithms perform well in asynchronous systems, but they do not manage any specific conflicts such as accidental clean merge and undo/redo operations. While, *git merge* algorithm handles these cases without problem.

For this purpose, we defined a new solution to avoid these kinds of conflicts and generate an operation-based algorithm that can be used correctly in asynchronous systems, reduce the conflicts and human interactions. It also outperforms the existing tool used in asynchronous systems: *git merge*.

Our experiments demonstrate in which cases operation-based algorithms are suitable for asynchronous systems and outperform the *git merge* tool, the default merge tool used in git systems. We investigate first on the collaboration to detect the problems of merging procedure. Thus, we give guidelines to improve such OT algorithms to take into account the most common cases that create conflicts when accidental clean merge and undo/redo operations are generated. Finally, we proposed a solution to handle these kinds of conflicts, make an experiment on asynchronous corpus, improve the quality of the merge and reduce the user's effort.

Acknowledgment

The authors would like to thanks following people for their contributions to the algorithms implementation : G. Oster (SOCT2), M. Zawirski (TreeDoc) and S. Martin (Git Walker).

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, C.-L. Ignat, G. Oster, and P. Urso. 8émes journées francophones mobilité et ubiquité. In ACM, editor, *ACM Symposium on Document Engineering*, page 12 pages, IUT de Bayonne âĂŞ Pays Basque, FR, jun 2012.

[3] 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.

[4] S. Alshattnawi, G. Canals, and P. Molli. Concurrency awareness in a p2p wiki system. In *Collaborative Technologies and Systems, 2008. CTS 2008. International Symposium on*, pages 285–294, 2008.

[5] 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.

[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. Choudhary and P. Dewan. A general multi-user undo/redo model. In *ECSCW'95: Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, pages 231–246, Norwell, MA, USA, 1995. Kluwer Academic Publishers.

[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] D. M. P. Eggert and R. Stallman. *Comparing and Merging Files with GNU diff and patch*. Network Theory Ltd, January 2003.

[12] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. *SIGMOD Record : Proceedings of the ACM SIGMOD Conference on the Management of Data - SIGMOD '89*, 18(2):399–407, May 1989.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code*. 1999.

[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] O. A.-H. Hassan and L. Ramaswamy. Message replication in unstructured peer-to-peer network. In *CollaborateCom*, pages 337–344, 2007.

[17] C.-L. Ignat, S. Papadopoulou, G. Oster, and M. C. Norrie. Providing awareness in multi-synchronous collaboration without compromising privacy. In *Proceedings of the 2008 ACM conference on Computer supported cooperative work*, pages 659–668. ACM, 2008.

[18] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, pages 277–293, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[19] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pages 243–252, Washington, DC, USA, 1994. IEEE Computer Society.

[20] 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.

[21] A.-M. Kermarrec, A. I. T. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing - PODC'01*, pages 210–218. ACM Press, 2001.

[22] S. Khanna, K. Kunal, and B. C. Pierce. A formal investigation of diff3.

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

[24] D. B. Leblang. Configuration management. chapter The CM challenge: configuration management that works, pages 1–37. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[25] T. Lindholm. A three-way merge for xml documents. In *Proceedings of the 2004 ACM symposium on Document engineering*, DocEng '04, pages 1–10, New York, NY, USA, 2004. ACM.

[26] J. Maeda. *The laws of simplicity*. MIT Press, 2006.

[27] S. Martin, M. Ahmed-Nacer, and P. Urso. Controlled conflict resolution for replicated document. In *CollaborateCom*, pages 471–480, 2012.

[28] S. Martin, P. Urso, and S. Weiss. Scalable xml collaborative editing with undo. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2010*, volume 6426 of *Lecture Notes in Computer Science*, pages 507–514. Springer, 2010.

[29] 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.

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

[31] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, May 2002.

[32] 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.

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

[34] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 259–267, Banff, Alberta, Canada, nov 2006. ACM Press.

[35] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 259–267, Banff, AB, Canada, November 2006. ACM Press.

[36] G. Oster, P. Urso, P. Molli, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *The Second International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006)*, Atlanta, Georgia, USA, November 2006. IEEE Press.

[37] D. E. Perry, H. P. Siy, and L. G. Votta. Parallel changes in large-scale software development: an observational case study. *ACM Trans. Softw. Eng. Methodol.*, 10(3):308–337, July 2001.

[38] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the sixteenth ACM symposium on Operating systems principles - SOSP'97*, pages 288–301. ACM Press, 1997.

[39] N. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. A Commutative Replicated Data Type for Cooperative Editing. In *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*, pages 395–403, Montreal, QC, Canada, June 2009. IEEE Computer Society.

[40] N. M. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE - OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003*, volume 2888 of *Lecture Notes in Computer Science*, pages 38–55. Springer, November 2003.

[41] M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 131–139, New York, NY, USA, 1999. ACM.

[42] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW*, pages 288–297, 1996.

[43] 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.

[44] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[45] 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.

[46] R. Smith. distributed with gnu diffutils package, GNU diff3 (1988) Version 2.8.1, April 2002.

[47] M. Suleiman, M. Cart, and J. Ferrié. Serialization of Concurrent Operations in a Distributed Collaborative Environment. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work - GROUP '97*, pages 435–445, Phoenix, AZ, USA, November 1997. ACM Press.

[48] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work: the integration challenge*, GROUP '97, pages 435–445, New York, NY, USA, 1997. ACM.

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

[50] 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.

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

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

[53] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.

[54] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.

[55] S. Weiss, P. Urso, and P. Molli. An Undo Framework for P2P Collaborative Editing . In *CollaborateCom*, pages 529–544, Orlando, USA, November 2008.

[56] 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. IEEE Computer Society.

[57] 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.