Parallelism and conflicting changes in Git version control systems

Hoai Le Nguyen, Claudia-Lavinia Ignat

Parallelism and conflicting changes in Git version control systems

Hoai Le Nguyen
Université de Lorraine, F-54000
Inria, F-54600
CNRS, F-54500
hoai-le.nguyen@loria.fr

Claudia-Lavinia Ignat
Inria, F-54600
Université de Lorraine, F-54000
CNRS, F-54500
claudia.ignat@inria.fr

ABSTRACT

Version control systems such as Git support parallel collaborative work and became very widespread in the open-source community. While Git offers some very interesting features, resolving collaborative conflicts that arise during synchronization of parallel changes is a time-consuming task. In this paper we present an analysis of concurrency and conflicts in official Git repository of four projects: Rails, IkiWiki, Samba and Linux Kernel. We also analyze how often users decide to rollback to previous document version when the integration process results in conflict. Finally, we discuss the mechanism adopted by Git to consider changes made on two continuous lines as conflicting.

CCS CONCEPTS

•Human-centered computing →Empirical studies in collaborative and social computing; •Software and its engineering →Software configuration management and version control systems;

KEYWORDS

version control systems, parallel work, conflicts

1 INTRODUCTION

Computer supported collaboration is an increasingly common occurrence that becomes mandatory in academia and industry where the members of a team are distributed across organizations and work at different moments of time. Version control systems are popular tools that support parallel work over shared projects. These systems keep a history of versions of the documents published by users. These document versions have associated meta-data such as publication date, author and short description of the modifications which facilitates detection of a certain problem or recovering of deleted contributions.

As a version control system supports parallel work on the same documents, it has to offer support for synchronization of parallel changes on those documents. Version control systems can be classified into centralized version control systems (CVCSs) and decentralized version control systems (DVCSs).

CVCSs such as CVS[1] and Subversion[5] rely on a client-server architecture. The server keeps a complete history of versions while clients keep only a local copy of the shared documents. Users can modify in parallel their local copy of the shared documents and synchronize with the central server in order to publish their contributions and make them visible to the other collaborators. This work mode is called copy-modify-merge paradigm.

DVCSs such as Git[8], Mercurial[14] and Darcs[6] became popular around 2005. These systems rely on a peer-to-peer architecture where each client keeps the history of versions plus a local copy of the shared documents. Users can work in isolation on their local repositories. They can also synchronize their local repository with the ones of other collaborators. Actually, in practice, the core-development-team will organize at least one repository as the primary repository where the latest approved changes can be found. Contributors can clone from this official repository. However, only the core-development-team has the write-access to commit directly. Other contributors need to use the pull-based development model in which a contributor creates a pull-request for his changes. A core-team's member then inspects the changes and decides to pull and merge contributor's changes to the main repository or not. And in some cases, contributors are requested to update or add more changes before their pull-request is accepted. Nowadays, the pull-based model is naturally supported by web-based hosting services such as GitHub [9] and Bitbucket [3].

Even if CVCSs are largely used in academia and industry they feature some limitations due to their centralized architecture: they offer a limited scalability and limited fault tolerance, administration costs are not shared and data centralization in the hands of a single vendor is an inherent

IWCES'17, Portland, Oregon, USA

threat to privacy. On the other side, DVCSs overcome these limitations of centralized architectures. Their peer-to-peer architecture allows them supporting a large number of users and tolerating faults. Each user keeps a copy of the history and decides with whom to share it without storing it on a central server. These features made DVCSs widely used in the domain of open software development where projects have often a large number of contributors. Parallelism of activities is very important for supporting collaboration, but merging different parallel contributions might require a significant period of time which can alter the productivity gain.

Our objective is to study the parallelism of activities in a DVCS such as Git[8]. In Git, users can synchronize their changes with other users working in parallel with them. In this process, a merge is performed between local changes and remote changes. If concurrent changes refer to the same file, we say that these changes are conflicting. Conflicts on a file can be automatically resolved or they need user intervention for their resolution. We call the former category automatically resolved conflicts and the latter category unresolved conflicts. If conflicting changes refer to to different non-adjacent lines of the file, the conflict is automatically resolved by the system. If, on the contrary, conflicting changes refer to the same or adjacent lines of the file, the conflict cannot be automatically resolved and the user has to manually resolve it. Unresolved conflicts also occur if a file is renamed and modified/deleted concurrently, if it is modified and deleted concurrently, if it is renamed concurrently by two users, if a user renames a file with the same name as another user gives to a concurrently created file and if two users concurrently add two files with the same name.

Understanding how often conflicts happen and how users resolve conflicts can help proposing a merging mechanism that minimizes conflicts that users have to manually resolve.

The rest of the paper is organized as follows. In Section 2 we explain briefly related works. Section 3 presents our measurements of conflicts during merge process. Section 4 discusses about some limitations of analyzing Git repositories. Section 5 gives some concluding remarks.

2 RELATED WORK

In this section we present some studies on parallelism of changes performed in version control systems.

The user study presented in [16] reports on conflict resolution experiences with the optimistic file system Ficus. Conflicts were classified into *update/update*, *remove/update* and *naming* conflicts. *Update/update* conflicts appear when two concurrent updates are performed on the same file. *Remove/update* conflicts appear when an update of a file and the removing of that file were performed concurrently. A *naming* conflict occurs when two files are independently created with the same name. The study found out that only about 0.0035% of all updates made to non-directory files resulted in conflicts and among them less than one third could not be resolved automatically. Authors mentioned that conflicts that cannot be resolved automatically are any *update/update* concurrent changes on source code or text files as they have arbitrary semantics and therefore require user intervention. Note that in contrast to the definition of conflicts used in [16], in the terminology of version control systems two updates done on the same file (source code or textual) lead to non-automatically resolved conflicts only if the updates refer to the same line in the file. All *update/remove* conflicts required human intervention and about 0.018% of all *naming* conflicts led to name conflicts which have to be resolved by humans.

In [15], the authors presented a study about parallel changes in the context of a large software development organization and project. The study analyses the complete change and quality history of a subsystem of the Lucent Technologies' 5ESS over a period of 12 years. Each set of change requests representing all or part of a solution to a problem was recorded by the system. When a change from this set was made on a file, the system kept track of the lines added, edited or deleted. This set of changes composes a *delta*. It was found that 12.5% of all *deltas* were made to the same file by different developers within a day. 3% of all these *deltas* made within a day by different developers physically overlap. However, interference of these *deltas* is analyzed over a quite large period of time (1 day) and not all these *deltas* are performed concurrently.

In [19] authors investigated four large open-source projects (GCC, JBOSS, JEDIT and PYTHON) and found that in CVS the *integration rate* is very low (between 0.26% and 0.54%), and the *conflict rate* is between 23% and 47%. They indicated that the parallel changes within single files (integration) are rare and have small impact to the development process. However, they frequently affect the same location and can not be integrated automatically by CVS.

Only few studies analysed parallel changes and conflicts for projects developed using DVCSs. In [4], authors studied the *conflict rate* of textual conflicts in terms of change-sets for nine open-source projects using Git repositories. They found that the average conflict rate was 16%. Note that change-sets in Git are commits that refer to the whole project whereas change-sets in CVS are per file.

However, no study analysed in detail textual conflicts in DVCSs and how people resolve these conflicts.

3 MEASUREMENTS

In order to measure the level of parallelism and the proportion of conflicting modifications in DVCSs, we adopted an

experimental methodology where we analyzed the corpus of four large open-source projects developed using Git:

- **Ruby on Rails** [17] is a web framework, with integrated support for unit, functional, and integration testing. We analysed version 5.0.0.alpha of the Rails.
- **IkiWiki** [11] is a wiki software system that compiles wiki pages into HTML pages for publication. We analysed IkiWiki version 3.0.
- **Samba** [18] is an implementation of networking protocols to share files and printers between Unix computers and Windows computers. We analysed Samba 3.0.x.
- **Linux Kernel** [13] is an implementation of a Unix-like computer operating system kernel. We analysed version 4.x of the Linux kernel.

Beside the large size and the popularity of these projects, they are representative for the different software development pull-based [10] models that they adopt. Rails projects uses pull-request model which is naturally supported by GitHub [9]. Contributors will fork (clone) from the official GitHub repository and contribute via GitHub's pull-requests. IkiWiki looks like a *private repository* where contributors send their *patches* to Josey Hess, the main developer of the project. Samba uses a *shared repository* among registered contributors. It uses an auto-build system for code-review process. Contributors need to join a technical mailing-list before contributing. Linux Kernel uses a pull-based model via mailing-list. Contributors need to send their *patches* to the appropriate subsystem maintainer's mailing-list in charge of the different parts of the project.

Table 1 presents some details about these projects: the period of their development (until 05-October-2015), the number of commits, the number of contributors (authors), the number of created files during the lifetime of the project and the number of existing files on 05-October-2015. Note that if a file is moved during the lifetime of the project from a place to another, we counted it as a new created file.

Project name	*Period (days)*	*No. of commits*	*No. of authors*	*No. of created files*	*No. of existing files*
Rails	3,967	19,375	3,422	10,272	2,984
IkiWiki	3,496	53,625	982	4,610	3,362
Samba	7,094	100,301	386	33,626	7,582
Kernel	5,132	547,515	14,395	90,173	51,567

Table 1: Open source projects developed using Git

In contrast to CVCSs, Git does not support the centralized logging feature of all user activities. The best overview of user activities is provided by the commit history (including merges) from the primary repository. To identify concurrences and conflicts in each project, we created a *shadow* repository and recursively re-integrated developer's changes into this repository. In other words, by means of Python scripts, we re-played all merges that were performed during the development period of each project.

Concurrency and conflicts on files

We first determined the files modified concurrently and then the ones that had unresolved conflicts. Similar to [19] we computed the *integration rate* and *conflict rate* as provided in Table 2. *Integration rate* represents the proportion of concurrently modified files over all modified files. *Conflict rate* is calculated by the proportion of files which have unresolved conflicts over concurrently modified files.

Project name	*Modified files*	*Integration rate*	*Unresolved conflict rate*
Rails	117,960	4.04%	16.26%
IkiWiki	37,327	1.08%	50.50%
Samba	306,182	0.68%	87.84%
Kernel	1,278,247	10.99%	4.86%

Table 2: Concurrency and conflicts on files

We can notice that Kernel and Rails projects have larger integration rate than IkiWiki and Samba. For instance, integration rate in Kernel project is 10 times larger than IkiWiki and 16 times larger than Samba. This can be explained by the large size of Kernel project. In contrast with the *integration rate*, Rails and Kernel have smaller *conflict rates* than IkiWiki (50.50%) and Samba (87.84%). We do know that Rails is a large project using advantages of GitHub, which supports pull-based model naturally. GitHub interface allows not only the author of a pull-request and the reviewer but also other contributors and core-team members to discuss about that pull-request and its issues. It brings a big advantage of sharing collaborators knowledge to solve problems during integration. In case of Linux Kernel, it uses pull-based model via mailing list with a list of subsystem maintainers. It also has a list of delegated servers, such as *linux-next*, where commits are tested before they are pushed to primary repository [7]. On the other side, Samba uses shared repositories among contributors and IkiWiki is maintained as a private repository by Josey Hess. Nowadays, all of them provide a list of Todo tasks and a list of Bugs where contributors can focus their work to avoid conflicting integration.

The lack of a central server that holds a reference copy of the project introduces more parallelism between user versions allowing them to diverge more in DVCSs than in CVCSs. For instance, Kernel, Rails, IkiWiki and Samba projects developed in Git have significantly higher *integration rate* (22,

8, 2 and 1.5 times respectively) than projects in CVS analysed in [19]. However, the higher *integration rate* does not result into higher *conflict rate*. For instance, Kernel and Rails have 5 and 1.5 times respectively lower *conflict rate* than projects in CVS whereas Samba and IkiWiki have almost 2 times higher *conflict rate*. The *conflict rate* in Git's projects depends on collaboration process management.

Conflict resolution

Several files can be in conflict during a merge. We counted the total number of merges performed during the lifetime of the project and the number of merges that leaded to unresolved conflicts. Table 3 gathers these results for all the projects analyzed. When a merge is not resolved automatically by Git, users need to resolve it manually. A user can decide to rollback to a previous version. We provide also the rollback rate, i.e the number of times user uses rollback action over all the merges that contain unresolved conflicts. The rollback rate is lower in Kernel and Rails compared to IkiWiki and Samba.

Project name	*No. of merges*	*Unresolved conflict rate*	*Rollback rate*
Rails	9,728	4.34%	1.66%
IkiWiki	1,037	7.52%	5.13%
Samba	1,281	10.07%	6.98%
Kernel	38,961	9.11%	0.70%

Table 3: Frequencies of conflicting merges

In practice, after a successful merge, users build and test the merging results. A successful merge can result in build-failed or test-failed. In order to be considered as a *successful integration*, a successful textual merge needs to be built and tested successfully also. Otherwise, it is considered as *higher-order* conflicts [4]. To detect these conflicts, we have to build using provided project build-scripts and test using provided test-sets every successful merge. We did not measure these types of conflicts as not all projects provided test-sets.

We provided in Table 4 a more detailed analysis of the rollback action in which we find rollback actions in all merges, not only the ones that contain textual-conflicts. *Level* represents the number of following commits after a merge when the rollback actions happen. Figure 1 illustrates *levels* of rollback in which branch *B2* is merged to branch *B1*. The merge result is *M0* and *C1, C2, C3* are the following commits after the merge.

Normally, if a user decides to rollback, he can use the *'git revert'* command to rollback the merging. The *'git revert SHA-1-B1'* usually creates a new commit after the merging commit (i.e commit *C1* has the same *SHA-1 hash* with *B1*).

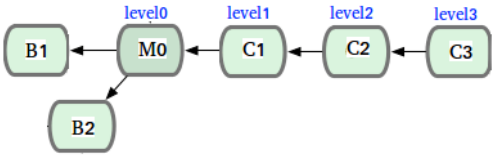

Figure 1: Levels of rollback action

Project name	*Level 0*	*Level 1*	*Level 2*	*Level 3*	*Level 4*
Rails	3,217	36	2	0	0
IkiWiki	39	13	1	1	0
Samba	260	2	0	0	0
Kernel	2,016	1	0	0	0

Table 4: Rollback action after merging

We assigned *level=1* to this case. However, in the most of the cases, users don't want to create new commits. Users can use a set of commands to rollback without creating a new commit such as *'git checkout SHA-1-B1; git reset –soft HEAD; git commit -amend'* that will rollback *M0* to *B1* version (i.e commit *M0* has the same *SHA-1 hash* with *B1*). We assigned *level=0* to this case. Note that, in practice, when merging a contributor's work to the main repository, the core-team members can use *'git merge -s ours'* to chose the main-line version as the default result when conflicts happen or use *'git merge –no-commit'* to test the merging results and manually fix the conflicts before committing them. Furthermore, the rollback actions can happen in the next one, two or three commits. We assigned *level=1, level=2, level=3* to these cases. In our measurements, we limit *level* to four. In fact, we could not find any rollback actions after three commits from the merging result.

Table 4 shows that Rails (33.46%) and Samba (20.45%) have much more rollback actions than IkiWiki(5.21%) and Kernel(5.18%).

Adjacent-line conflicts

Git considers that concurrent modifications of two continuous lines as being in conflict (called adjacent-line conflicts). Figure 2 illustrates an example of adjacent line conflict. User at site-1 makes changes on line 1 and user at site-2 makes changes on line 2. They then merge their work and Git generates a *CONFLICT(content)*. In our hypothesis, this is not a content conflict because these changes are made on two different lines. Git should merge them successfully by applying changes from both sites. To test our hypothesis, we analysed all content conflicts in the four projects to detect all adjacent-line conflicts. We then analysed the adjacent-line

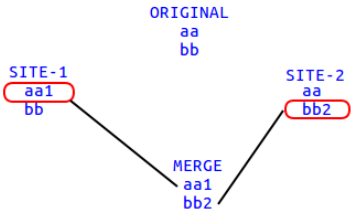

Figure 2: Adjacent-line conflict

conflicts's resolutions that had been manually fixed by the authors to see if they use both changes solution or not. The algorithm for checking adjacent-line is described briefly as below:

```
for each merge in [list_of_merges]:
  if is_content_conflict(merge):
    list_diff_files = git_diff3_tool(merge)
    for each diff_file in list_diff_files:
        process(diff_file,merge)
```

In our algorithm, we used *git diff* (in *diff3* format) to generate the merge-matrix in which each column presents the difference between one site and the original. Figure 3 illustrates how the merge-matrix is built by marking changed line as 'X' and unchanged line as 'Y'. After building the merge-matrix for each *content conflict*, we used adjacent-line patterns in figure 4 to detect adjacent-line conflicts. Each pattern is compared with two continuous lines. If the first line is changed by only one site and the second line is changed by only another side, it's adjacent-line case.

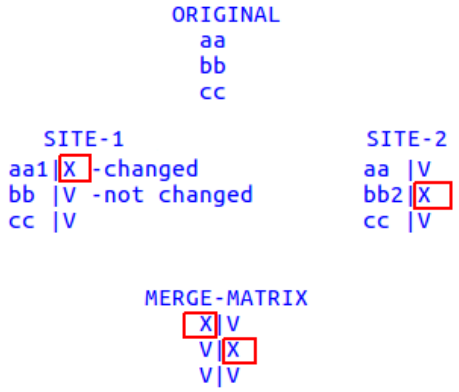

Figure 3: Merge matrix

Noted that in figure 4, only patterns in group (1) are considered as adjacent-line patterns. Group (2) presents four patterns which include adjacent-line conflict and also normal content conflict. In the first two patterns of this group, when analyzing the first line, we get a normal content conflict. In the last two patterns, when analyzing the second line, we get a both normal conflict. In our analysis, we considered only in adjacent-line conflicts which do not include normal content conflict.

Project name	*Adjacent-line conflicts*	*Applying both changes*	*Applying one change*	*Other cases*
Rails	80	46	28	6
IkiWiki	4	3	1	0
Samba	41	10	23	8
Kernel	367	312	51	4

Table 5: Adjacent line conflicts and resolutions

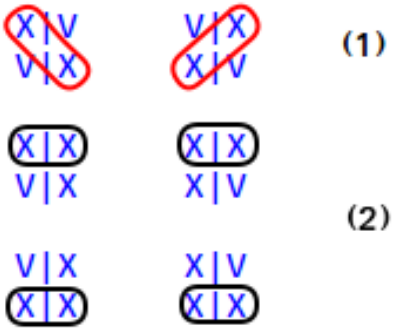

Figure 4: Adjacent-line patterns

The results are presented in Table 5 with number of adjacent-line conflicts and their resolutions. There are three resolutions that we found from the merging results: applying both changes, applying change from one site only (either from site-1 or site-2) and other cases in which no changes are applied. The proportions of applying both sites are 24.39% in Samba , 57.5% in Rails, 75% in IkiWiki and 85.01% in Linux Kernel. More details in how users resolve content conflicts, we do the same analysis with normal content conflict. Table 6 presents the results of normal-content conflicts in comparison to adjacent-line conflicts. As Svn (CVCS) and Darc (DVCS) can merge changes in adjacent lines successfully, it's significant to propose that Git should not consider change in adjacent lines as conflict.

4 DISCUSSION

There's very little data on conflicts in the domain of version control systems. Zimmermann [19] (Cvs), Brun [4] (Git) and Kasi [12] (GitHub) are all we could find that analyze about this work.

Beside,DVCSs do not have centralized logging servers. The history maybe lost or modified by the repository owner

	Adjacent-line conflicts		Normal content conflicts	
Project name	*No. of conflicts*	*Applying both*	*No. of conflicts*	*Applying both*
Rails	80	57.50%	317	5.67%
IkiWiki	4	75.00%	22	9.09%
Samba	41	24.39%	1149	14.19%
Kernel	367	85.01%	1326	13.38%

Table 6: Adjacent-line and normal content conflicts

[2]. A branch where changes are not accepted can be deleted, a sequence of commits can be altered their order, flattened from multi-branches in to single branch via *rebase*. In [7] the authors presented a new method called *continuous mining*. Instead of mining only the primary repository (called *blessed*), this method continuously observes all known repositories of a software project to uncover the more complete history of the project's development. Their empirical study focuses on Linux Kernel [13] which has 479 repositories (from 2012). Among these repositories, 22% did not contribute a single commit to the *blessed*. Nevertheless, *continuous mining* is still a re-active logging mechanism, it is not considered as a permanent solution to the need of centralized logging features of Git.

With analyzing only the history from official server of four projects, we missing data about how users communicate during collaborative time. The short description of commits does not include this data. Email threads in project's mailing-list and GitHub conversations (comments of a pull-request or an issues) can be promise for extracting this data.

5 CONCLUSIONS

The goal of this work is to analyze concurrencies and conflicts in Git, a decentralize version control system. We analyzed the corpus of four large projects in Git, ours findings are as follows:

The higher *integration rate* of a project does not mean the higher *unresolved conflict rate*. It depends on how the integration process is managed. Linux Kernel is a large-scale project with a list of subsystem maintainers which keep the lowest *conflict rate* for this project although its *integration rate* is much more higher than others. Except Linux Kernel, Rails which developed using GitHub pull-based model has 5 times lower *conflict rate* than Samba(shared repository) and 3 times lower than IkiWiki(private repository).

The rollback action is used more often in case of higher-order conflicts than in case of textual-conflicts. Most of rollback actions do not create new commits (*level=0*). We can say that user has tried to test the merge result before deciding to rollback or keep it.

Git, in different from Darcs and Svn, considers changes made in two adjacent lines as content conflict. From ours analysis, it is significant to suggest that Git should merge two adjacent-line instead of considering them as a conflict.

REFERENCES

[1] Brian Berliner. 1990. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter Technical Conference*. Washington, D. C., USA, 341–352.

[2] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. German, and Prem Devanbu. 2009. The Promises and Perils of Mining Git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories (MSR '09)*. IEEE Computer Society, Washington, DC, USA, 1–10.

[3] Bitbucket 2008. Bitbucket. *Code, Manage, Collaborate.* (2008). https://bitbucket.org/.

[4] Yuriy Brun, Reid Holmes, Michael D. Ernst, and David Notkin. 2011. 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)*. ACM, New York, NY, USA, 168–178.

[5] Ben Collins-Sussman, Brian W. Fitzpatrick, and C. Michael Pilato. 2004. *Version control with Subversion.* O'Reilly & Associates, Inc.

[6] Darcs 2003. Darcs. *Distributed. Interactive. Smart.* (2003). http://darcs.net/.

[7] Daniel M. German, Bram Adams, and Ahmed E. Hassan. 2015. Continuously mining distributed version control systems: an empirical study of how Linux uses Git. *Empirical Software Engineering* 21, 1 (2015), 260–299.

[8] Git 2005. Git. *Fast version control system.* (2005). http://git.or.cz/.

[9] GitHub 2008. GitHub. *Web-based Git repository hosting service.* (2008). https://github.com/.

[10] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-based Software Development Model. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 345–355.

[11] IkiWiki 2015. IkiWiki. (2015). http://ikiwiki.info/.

[12] Bakhtiar Khan Kasi and Anita Sarma. 2013. Cassandra: Proactive Conflict Minimization Through Optimized Task Scheduling. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 732–741.

[13] LinuxKernel 2015. Linux Kernel). (2015). http://kernel.org/.

[14] Mercurial 2005. Mercurial. *A fast and lightweight Source Control Management system.* (2005). http://www.selenic.com/mercurial/.

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

[16] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. 1994. Resolving file conflicts in the ficus file system. In *In Proceedings of the Summer Usenix Conference (USTC'94)*. 183–195.

[17] Ruby 2015. Ruby on Rails (The popular MVC framework for Ruby). (2015). http://rubyonrails.org/.

[18] Samba 2015. Samba - Opening Windows to a Wider World). (2015). http://www.samba.org/.

[19] Thomas Zimmermann. 2007. Mining Workspace Updates in CVS. In *Proceedings of the Fourth International Workshop on Mining Software Repositories (MSR '07)*. IEEE Computer Society, Washington, DC, USA, 11–.