Mining Workspace Updates in CVS

Thomas Zimmermann
tz@acm.org

Department of Computer Science, Saarland University, Saarbrücken, Germany

Abstract

The version control archive CVS records not only all changes in a project but also activity data such as when developers create or update their workspaces. Furthermore, CVS records when it has to integrate changes because of parallel development. In this paper, we analyze the CVS activity data of four large open-source projects GCC, JBOSS, JEDIT, and PYTHON to investigate parallel development: What is the degree of parallel development? How frequently do conflicts occur during updates and how are they resolved? How do we identify changes that contain integrations?

1. Introduction

The version control system CVS allows concurrent development and is widely adopted in the open-source community, especially for large projects like ECLIPSE, GCC, or MOZILLA. Therefore, recent research used CVS to investigate *change data*, that is, who changed what, why, when, and how [8].

Beside change data, CVS also records *activity data* that contains additional events: When did developers update their workspaces and did this update happen smoothly without any incidents? In particular, has another developer meanwhile changed the same file? And if so, could CVS integrate[1] the changes automatically or did the developer have to resolve the conflicts manually? Such events are interesting as they point out parallel development: What is the degree of parallel development? How frequently do conflicts occur during updates and how are they resolved? How do we identify changes that contain integrations?

We introduce in Section 2 the CVS *history* command on which we base our case studies of four large open-source projects: the GNU Compiler Collection GCC, the application server JBOSS, the editor JEDIT, and the PYTHON interpreter. In Sections 3 and 4 we address the above questions. In Section 5 we discuss the limitations of activity data; Section 6 presents related work and Section 7 concludes the paper with future work.

[1]We prefer the term *integrate* over the CVS terminology *merge* to avoid confusion with the merge of branches.

2. CVS History in a Nutshell

In addition to change data, CVS records *activity data* that is when did developers use which commands with what parameters. Currently, CVS tracks the activities of the following commands in a special file, called the *history* file:

- The *checkout* command[2] (O) creates a workspace in which developers can make their changes to a module.
- The *release* command (F) removes a workspace and issues a warning in case a change is not yet committed. Note that it is possible to remove workspaces without CVS interaction.
- The *update* command synchronizes a workspace. It retrieves all changes since the last checkout or update and creates new files (U), replaces outdated files (both U and P)[3], and removes files that have been deleted in the repository (W). If a file has been changed in both the workspace and the repository, CVS tries to integrate the changes automatically (G for smooth integration, C for integration with conflicts).
- The *commit* command submits changes made by a developer to the repository. Changes can modify (M), add (A), or remove (R) files.
- The *rtag* command (T) assigns symbolic names, called tags, to revisions in the repository. The *tag* command that works on the revisions in the workspace rather than on the repository, is *not* tracked in the history.
- The *export* command (E) creates a copy of a workspace without the administrative CVS files. This is useful for preparing releases.

[2]The history of CVS distinguishes the commands with a single capital letter. For convenience we will reuse them throughout the paper.

[3]The U update transfers the complete new revision; in contrast, the P update only transfers the differences to the new revision, i.e., a *patch* that is applied to the revision in the workspace. CVS chooses automatically between U and P updates based on the size of files and differences.

ourth International Workshop on Mining Software Repositories (MSR'07)
7695-2950-X/07 $20.00 © 2007 IEEE

```
O  2004-06-13 05:45 +0000  mary  foo  =foo=  <remote>/*
U  2004-06-15 06:56 +0000  mary  1.14  Bar.java  foo  ==  <remote>
P  2004-06-17 07:22 +0000  mary  1.15  Bar.java  foo  ==  <remote>
M  2004-06-19 07:50 +0000  mary  1.16  Bar.java  foo  ==  <remote>
C  2004-06-21 07:48 +0000  john  1.16  Bar.java  foo  ==  <remote>
G  2004-06-22 08:48 +0000  kate  1.16  Bar.java  foo  ==  /home/kate/foo
```

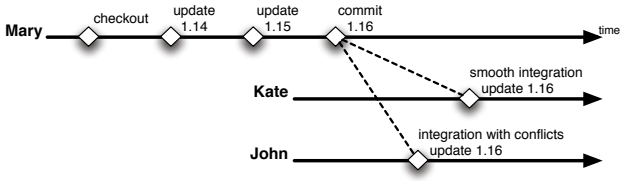

Figure 1. A sample output for CVS *history*

While the first four commands are used by all developers, the last two commands, *rtag* and *export*, are used mainly by developers to prepare releases.

We access the history file with the CVS *history* command. Figure 1 shows a sample output. For each record CVS returns a line that tells us that the *developer* called at *timestamp* the command that is indicated by the single capital letter *type*. Additionally, we get the location of the developer's *workspace* and the affected *module*, *file*, and *repository*. The specific syntax depends on the commands and further information may be included [9].

In Figure 1, the history snippet tells us that Mary first created a workspace (O), then synchronized Bar.java two times (U and P), and finally submitted changes on Bar.java to the repository (M). Meanwhile, Kate and John also have changed Bar.java; thus, during their next update CVS integrated their changes with the changes of Mary. For Kate's changes, the automatic integration worked fine (G), but the changes of John interfered with the changes of Mary and resulted in conflicts (C).

In the remainder of this paper, we will analyze entires for *commit* and *update* to measure the degree of concurrency.

3. A First Investigation of Concurrency

We investigated the CVS histories for four large open source projects: GCC, JBOSS, JEDIT, and PYTHON. Unfortunately, the implementation of CVS did not record updates correctly until version 1.11.7 which has been released on September 29, 2003.[4] For this reason, we started our investigation for a project with the first recorded update (see Table 1).

3.1. Degree of Parallel Development

Table 2 contains a breakdown of the *update* commands. We use them to measure the parallel development within files:

$$\textit{Integration Rate} = \frac{G+C}{W+U+P+G+C} \qquad (1)$$

[4]The release 1.11.7 of CVS fixed "a long-standing bug that prevented most client/server updates from being logged in the history file"; it also introduced the logging of updates that are done via a patch (P).

Project	Recorded since	Investigated Period
GCC	2004-09-16	2004-09-16 to 2005-02-02
JEDIT	2000-01-13	2004-01-12 to 2005-02-03
JBOSS	1999-10-13	2004-01-12 to 2005-02-09
PYTHON	2000-05-12	2004-01-12 to 2005-02-05

Table 1. Investigated Projects

$$\textit{Conflict Rate} = \frac{C}{G+C} \qquad (2)$$

The *integration rate* measures the percentage of updates which were integrated with local user changes. It is very low for all projects (between 0.15% and 0.54%, see Table 2(a)). This indicates that parallel changes within single files are rare and have only little impact on the development process. However, the *conflict rate* that measures the frequency of conflicts is between 22.75% (for GCC) and 46.62% (for JBOSS). These rather high values indicate that parallel changes frequently affect the same locations within a file or cannot be integrated by CVS.

Additionally, we measured how many commits led to an integration (see Table 2(b)).[5] The value is lowest for JBOSS; in GCC and JEDIT approximately every 11th commit led to an integration, for PYTHON even every 5th commit. If we focus on conflicts, the order of projects remains unchanged. This suggests that the degree of parallel development is highest in PYTHON.

3.2. Self-integrations and Self-conflicts

Integrations are not always caused by other developers. Many developers work at different places (home, office) or on different branches and use CVS to synchronize their changes. An interesting phenomenon are *self-integrations* (or in the presence of conflicts *self-conflicts*), that are updates where CVS integrates local changes of a developer with a commit that has been made by the same developer. Table 2(c) shows that self-integrations and self-conflicts occurred in all investigated projects. They are a good indicator that developers have several workspaces at the same time. However, they show only the presence not the frequency of simultaneous workspaces.

4. How Concurrency is Resolved

After CVS integrates changes, developers can decide whether to commit or discard the integrated file. In this section, we will address how integrations are resolved and how to identify revisions that include integrated changes.

[5]This number is smaller than the sum of G and C because one commit can lead to several integrations.

	GCC	JBOSS	JEDIT	PYTHON
General statistics				
Number of developers	166	91	56	57
Number of recorded events	7,776,010	2,326,323	95,800	662,002
– ignored *(anonymous)*	3,010,563	82,846	2,324	1,779
– investigated	4,765,447	2,243,477	93,476	660,223
Breakdown of updates (W+U+P+G+C)	4,662,843	1,339,201	88,323	650,487
File was integrated without conflicts (G)	9,285	1,066	361	1,743
File was integrated with conflicts (C)	2,735	931	116	1,080
Concurrency				
(a) Integration rate (G+C)/(W+U+P+G+C)	0.26%	0.15%	0.54%	0.43%
Conflict rate C/(G+C)	22.75%	46.62%	24.32%	38.26%
(b) Commits (only M and A) that led to integrations (G or C)	9.06%	3.89%	9.03%	20.20%
Commits (only M and A) that led to conflicts (C)	2.84%	1.86%	2.58%	7.82%
(c) Self-integrations (G caused by a commit of the same developer)	1,373	314	71	145
Self-conflicts (C caused by a commit of the same developer)	307	396	41	56

Table 2. Breakdown of commands. For a detailed breakdown, we refer to our technical report [9].

4.1. Resolution of Integrations

If a developer has made local changes to a file which has meanwhile changed in the repository, CVS integrates these changes with the other changes during the next update. We investigated what developers do with such integrated files: Do they commit their changes to the repository? Or, do they discard their changes by deleting the file and performing a second update? To answer these questions, we looked at the record that succeeded an integration. For instance the sequence GM means that a smooth integration (G) was followed by a commit (M). Table 3 shows the results for the following categories:

Changes were committed (M). The sequences GM or CM indicate that the integrated changes were committed to the repository. Between 8.3% and 31.8% of all integrations without conflicts are committed to the repository; for integration with conflicts these values are slightly lower between 4.4% and 24.6%.

Changes were discarded (UP). The sequences GU or CU indicate that the developer discarded the integrated changes and replaced the file with a fresh version from the repository; the sequences GP or CP indicate that the local changes were discarded manually without deleting the file. In every investigated project more than 30% of all integrations are discarded; this percentage is higher when conflicts occurred.

Changes were kept (CG). The sequences GG, GC, CG, and CC indicate that the local changes were neither discarded nor directly committed to the repository, i.e., they were carried over to the next update in which another integration took place.

	Without conflicts (G)				With conflicts (C)			
Project	(M)	(UP)	(GC)	($)	(M)	(UP)	(GC)	($)
GCC	11.8	44.8	22.8	20.6	24.6	38.4	26.0	11.1
JBOSS	31.8	34.4	11.6	22.2	14.3	68.5	5.7	11.6
JEDIT	8.6	58.7	27.4	5.3	10.3	57.8	9.5	22.4
PYTHON	8.3	49.6	30.8	11.2	4.4	57.5	35.4	2.7

Table 3. How integrations are resolved (in %).

Others ($). The sequences G$ and C$ are integrations where we could not identify a next record, i.e., the integration was the last record for this file by the developer.

4.2. Identification of Integrated Revisions

In order to locate revisions that contain integrated changes, we searched for activity patterns of the form [GC]+M, i.e., a sequence of integrations [GC]+ that is followed by a commit operation M for a revision r, all by the same developer. Furthermore, we disallowed any other operation between the integrations and the commit because then it would unlikely that r contains any integrated changes. If the sequence of integrations [GC]+ contains a conflict (the position of C does not matter) we say that the revision r contains integrated changes *with a conflict*, otherwise we say *without conflict*. In total, we located 2,307 revisions with integrated changes [9].

5. Limitations

- The CVS history has only limited functionality if it is used on a CVS server (most distributed projects use it this way). For instance, until version 1.11.7 the record types U and P were not logged. Furthermore, the workspace is logged relative to the repository. This makes it almost impossible to distinguish between different workspaces of one developer.
- Only a subset of the commands is recorded in the CVS history. For instance, the *import* and *join* commands are not yet recorded. The latter would be valuable to precisely detect the merge of branches without heuristics like the one proposed by Fischer et al. [1].
- Until version 1.11.7 of CVS, developers could suppress the logging of commands with the -l option. This means that the data in the history may be incomplete for older entries.
- The history data of CVS is only available for a few projects. We had difficulties finding projects for our case studies.

6. Related Work

To our knowledge this is the first work that analyzes CVS *activity data* as obtained from CVS *history*. A similar case study on *change data* was performed by Perry et al. [4] who investigated parallel changes on different levels. In contrast to their work, we could not observe a high degree of parallelism on within single files. Voinea and Telea studied how developers interact with each other via changing similar files in the context of CVS and Subversion [7, 6].

The high percentage of integrations with conflicts reflects a shortcoming of CVS and underpins the need for tools like Palantír which was developed by Sarma, Noroozi, and van der Hoek [5]. Palantír continuously shares information about changes. This way it increases the awareness among developers and can reduce conflicts.

Research has been aware of the problem of conflicts for a long time and several solutions were proposed to handle source code merging in a more syntax and semantic-aware way [3]. Unfortunately, only few of them are used in today's version control systems.

7. Conclusions and Consequences

We investigated CVS activity data of four large open source projects. Our results are as follows:

- We observed that parallel development within the same file has only little impact on other developers (between 0.26% and 0.54% of all updates).
- CVS can integrate many changes but not all; in our case studies between 22.75% and 46.62% of all integrations resulted in a conflict.
- Developers work with different workspaces, e.g., at work and at home. Between 7.3% and 26.4% of all integrations are caused by this circumstance.
- We can identify revisions that contain integrated changes by analyzing the sequence of updates and commits.

CVS activity data is a valuable supplement to other project data. In future work we plan to assess the risk of integrations by correlating with bug-introducing changes [2]. One can use the CVS history to distinguish commits without integration, with smooth integration, and with conflicts. One would guess that commits that succeed a conflict are more risky, but we expect the opposite because conflicts are (hopefully) inspected by developers.

Acknowledgments. Thanks to Holger Cleve, Christian Lindig, Stephan Neuhaus, and the anonymous reviewers for their helpful suggestions on earlier revisions of this paper. This research was funded by a fellowship from the Graduiertenkolleg "Leistungsgarantien für Rechnersysteme" sponsored by the Deutsche Forschungsgemeinschaft.

References

[1] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.

[2] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr. Automatic identification of bug-introducing changes. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 81–90, Sept. 2006.

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

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

[5] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising awareness among configuration management workspaces . In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 444–454, Portland, Oregon, May 2003.

[6] L. Voinea and A. Telea. Cvsgrab: Mining the history of large software projects. In *EUROVIS - Eurographics/IEEE VGTC Symposium on Visualization*, pages 187–194, 2006.

[7] L. Voinea and A. Telea. Multiscale and multivariate visualizations of software evolution. In *Proc. of ACM Symposium on Software Visualization (SoftVis)*, pages 115–124, 2006.

[8] T. Xie. Bibliography on mining software engineering data. http://ase.csc.ncsu.edu/dmse/. Retrieved in Feb 2007.

[9] T. Zimmermann. The landscape of concurrent development. Technical report, Universität des Saarlandes, Saarbrücken, Germany, August 2006.