The Landscape of Concurrent Development

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 for large open-source projects GCC, JBOSS, JEDIT, and PYTHON to investigate parallel development: How frequently do developers update their workspaces? What is the degree of parallel development? How many workspaces do developers have? 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. Such data points out software evolution, guides developers, and identifies instabilities in source code.

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: How frequently do developers update their workspaces? What is the degree of parallel development? How many workspaces do developers have? How frequently do conflicts occur during updates and how are they resolved? How do we identify changes that contain integrations?

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

We introduce in Section 2 the CVS *history* command on which we base our case studies of four large open-source projects: the Compiler Collection GCC, the application server JBOSS, the editor JEDIT, and the PYTHON interpreter. In Sections 3, 4, and 4 we address the above questions. Section 6 visualizes the relationships that are created by parallel development between developers. In Section 7 we discuss the limitations of activity data; Section 8 presents related work and Section 9 concludes the paper with future work.

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 (O)[2] 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.

[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.

- 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.

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:

For rtag. A *tag* can be added (A), deleted (D), or modified (then the previous tag is given as *modified_tag*).
type date user module|file [tag:A|D|modified_tag] workspace?

For checkout, release, and export.
A developer optionally can provide a *tag* or a *date*.
type date user ([tag|date])? repository? =module= workspace

For update and commit.
type date user revision? file repository =module= workspace

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 following we will investigate the records 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).

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

3.1. Distribution of Commands

Table 2 shows the frequency of CVS commands. For all investigated projects the *update* command is the most frequent, followed by the *commit* command. For JBOSS the high number of *checkout* records sticks out because JBOSS is distributed across many modules in the CVS repository; for each module one checkout operation is performed. We also observe that the *release* command is rarely used. This suggests that most developers remove workspaces without the help of CVS. The low frequency of *export* and *rtag* is no surprise since these two commands are related to product releases.

3.2. Degree of Parallel Development

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

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

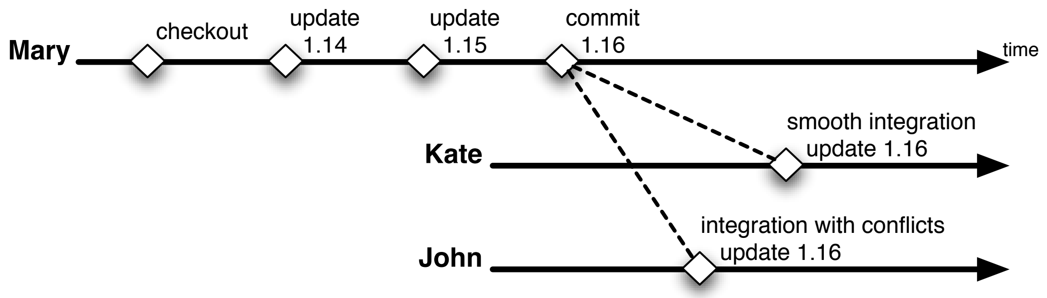
$$\textit{Conflict Rate} = \frac{C}{G+C} \tag{2}$$

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). 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).[5] The value is lowest for JBOSS; in

[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).

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



```
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*

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.

3.3. The Most Frequently Integrated Files

Table 3 shows for each project the files with the highest number of integrations. In all investigated projects, there are source files with many integrations, as well as build files like Makefile.in, build.xml, build.properties, and configure. The file axis-ws4ee.jar in JBOSS has only conflicts because CVS does not integrate binary files. However, it is interesting to observe that ChangeLog has four times more conflicts as smooth integrations.

In Table 4 we break down the number of integrations to the file extension. In all projects more than 50% of integrations happened for source files. For GCC, conflicts were more frequent for ChangeLog files than for source files. For JBOSS and PYTHON, conflicts were more frequent for source files than for XML or $\TeX$ files.

4. Concurrency and Single Developers

Concurrent development is not always a matter of multiple developers. Single developers also cause concurrency, e.g., by working at different places or on different branches. In this section, we search for traces of such concurrency.

4.1. Scope of Developers

We measured the number of *checkout* operations[6] per developer and the number of modules and branches a developer is working on (as indicated by a checkout of a module and branch respectively). In Tables 5, 6, and 7 we show the distribution of the results.

- For GCC, JEDIT, and PYTHON
  - more than 50% of all developers checkout the project less than six times (see Table 5).
  - most developers work only on few modules and on few branches (see Tables 6 and 7).
- JBOSS once again sticks out because of its high number of modules. This results in a high number of checkouts per user. However, most JBOSS developers work on few branches.
- Both GCC and JBOSS have users with several thousand checkouts (*gccadmin* and *jboss-build*). Such users have special tasks, for instance to create nightly builds of a project and frequently work with or even are automatic scripts.

The above numbers only state that developers work on several workspaces and branches, but the numbers do not consider if this happened simultaneously. We will address this in the next two sections.

4.2. Self-integrations and Self-conflicts

In order to get evidence for simultaneous workspaces, we measured how frequently CVS integrates local changes of a developer with a commit that has been made by the same developer. We refer to such updates as *self-integrations*

[6]We combined checkouts having the same timestamp into one single checkout operation.

	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 to Commands				
Checkout command (O)	16,104	849,054	693	302
Commit command (M+A+R)	86,223	54,976	4,412	9,378
Update command (W+U+P+G+C)	4,662,843	1,339,201	88,323	650,487
Release command (F)	2	1	10	0
Tag command (T)	207	43	10	0
Export command (E)	68	202	28	56
Breakdown of Commits (M+A+R)				
Modified file (M)	63,639	32,158	3,187	8,212
Added file (A)	18,172	15,252	812	680
Removed file (R)	4,412	7,566	413	486
Breakdown of Updates (W+U+P+G+C)				
File was removed (W)	299,935	129,142	3,073	36,131
File was created or replaced (U)	1,994,101	968,647	63,891	284,218
File was patched (P)	2,356,787	239,415	20,882	327,315
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				
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%
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%

Table 2. Breakdown of Commands

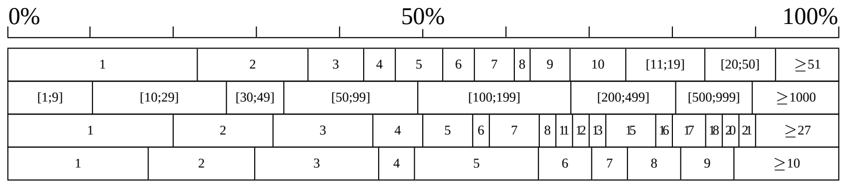
or in the presence of conflicts as *self-conflicts*. Such self-integrations are caused by multiple workspaces or hacking in the administrative CVS files. Tables 8 shows that self-integrations and self-conflicts occur in all investigated projects, especially in JBOSS and JEDIT. Self-integrations are a good indicator that developers had several workspaces of a module at the same time. However, they show only the presence not the frequency of simultaneous workspaces.

4.3. Simultaneous Workspaces

When developers work at several places, e.g., at home and at office, they maintain several workspaces of the same module and the same branch. Unfortunately, CVS records for client/server connections only the workspace relative to the repository and not its absolute path.[7] This means we cannot use this information to estimate the number of simultaneous workspaces per developer.

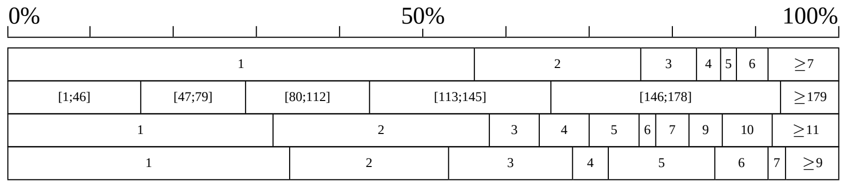
If developers maintain several workspaces and synchronize them regularly, they perform update operations for single revisions several times. Thus, we decided to measure for each developer the average number of *update* and *commit* records per revision. This average is an indicator of the number of workspaces. For instance, an average of two means that for each revision of a file the developer performed on average two commands, likely, because of two simultaneous workspaces. In Table 9 we show the distribution of these averages. Although most developers seem to have one workspaces, there is strong evidence that several developers have more than one workspaces for the same module and the same branch. However, we still do not take into account that the number of workspaces may fluctuate over time.

[7]In Figure 1, `<remote>` denotes the client workspace; for local connections the absolute path is recorded, e.g., `/home/kate/foo`.



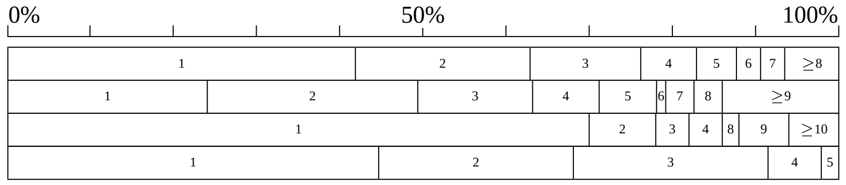
	Maximum	Average
GCC	3,780	61.5± 379.80
JBOSS	36,758	1076.9±5034.40
JEDIT	164	12.7± 25.96
PYTHON	33	6.2± 6.86

Table 5. Distribution of *Number of Checkouts per Developer*



	Maximum	Average
GCC	25	2.8± 3.91
JBOSS	442	112.8±66.03
JEDIT	65	5.0± 9.39
PYTHON	21	3.3± 3.40

Table 6. Distribution of *Number of Modules per Developer*



	Maximum	Average
GCC	24	2.9± 3.23
JBOSS	168	7.8±20.80
JEDIT	80	4.1±11.57
PYTHON	5	2.0± 1.07

Table 7. Distribution of *Number of Branches per Developer*

5. 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.

5.1. 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 11 shows the results for the following categories:

Changes were committed. 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. 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. Some projects stick out, for instance, in JBOSS almost 70% of all integrations with conflicts are discarded. For JBOSS and PYTHON smooth integrations are twice as frequent as conflicts. For GCC it is the other way round, likely because of the high number of conflicts for ChangeLog.

Changes were kept. 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 kept until the next update in which another integration took place. For JBOSS and PYTHON conflicts are more frequently discarded than smooth integrations.

Other. 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.

5.2. Identification of Integrated Revisions

In order to identify revisions that contain integrated changes, we refined the techniques we used in the previous section. To locate such revisions, 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

	Integrated without conflicts (G)						Integrated with conflicts (C)					
	committed	discarded		kept		other	committed	discarded		kept		other
Project	GM	GU	GP	GG	GC	G$	CM	CU	CP	CG	CC	C$
GCC	11.8%	9.4%	35.4%	21.0%	1.8%	20.6%	24.6%	18.8%	19.6%	6.8%	19.2%	11.1%
JBOSS	31.8%	23.6%	10.8%	10.0%	1.6%	22.2%	14.3%	61.9%	6.6%	1.1%	4.6%	11.6%
JEDIT	8.6%	21.9%	36.8%	25.5%	1.9%	5.3%	10.3%	50.0%	7.8%	5.2%	4.3%	22.4%
PYTHON	8.3%	9.1%	40.5%	29.0%	1.8%	11.2%	4.4%	17.3%	40.2%	2.9%	32.5%	2.7%

Table 11. Resolution of Conflicts

Filename	G	C
GCC:		
gcc/gcc/ChangeLog	138	669
gcc/gcc/config/rs6000/rs6000.c	302	37
gcc/gcc/Makefile.in	269	36
gcc/gcc/tree-cfg.c	176	30
gcc/gcc/tree.h	175	17
gcc/gcc/testsuite/ChangeLog	22	148
gcc/gcc/config/i386/i386.c	131	14
gcc/gcc/expr.c	118	16
JBOSS:		
jbosstest/build.xml	99	13
build/jboss/build.xml	37	4
thirdparty/ws4ee/ws4ee/lib/axis-ws4ee.jar	0	21
jbosstest/imports/test-jars.xml	20	1
jboss-cache/src/main/.../cache/TreeCache.java	7	7
contrib/jboss.net/.classpath	7	7
nukes-2/core/build.xml	11	3
JEDIT:		
jEdit/org/gjt/sp/jedit/jedit_gui.props	40	2
jEdit/org/gjt/sp/jedit/jEdit.java	33	1
jEdit/build.properties	28	1
jEdit/modes/catalog	19	4
jEdit/org/gjt/sp/.../textarea/JEditTextArea.java	19	1
jEdit/org/gjt/sp/...xtarea/TextAreaPainter.java	17	1
jEdit/org/gjt/sp/jedit/Buffer.java	17	1
PYTHON:		
python/dist/src/Python/ceval.c	77	21
python/dist/src/Misc/NEWS	39	20
python/dist/src/setup.py	44	7
python/dist/src/configure.in	43	2
python/dist/src/configure	26	17
python/dist/src/Doc/whatsnew/whatsnew24.tex	30	3
python/dist/src/Python/compile.c	18	15

Table 3. Most Frequently Integrated Files

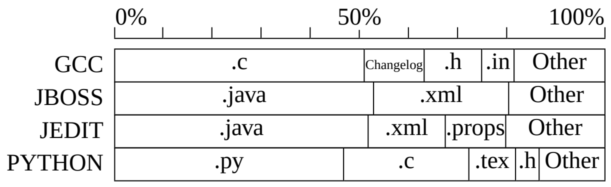


Table 4. Integrations per File Extension

	Self-integrations (G)		Self-conflicts (C)	
GCC	1,373	(15.4%)	307	(11.8%)
JBOSS	314	(29.7%)	396	(44.4%)
JEDIT	71	(20.1%)	41	(36.9%)
PYTHON	145	(8.6%)	56	(5.2%)

Table 8. Self-integrations and Self-conflicts

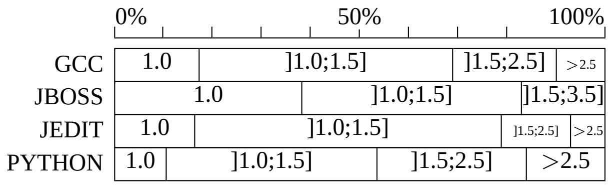


Table 9. Simultaneous Workspaces

	Without conflict		With conflict		Total
GCC	1,045	(61.4%)	656	(38.6%)	1,701
JBOSS	293	(76.5%)	90	(23.5%)	383
JEDIT	17	(60.7%)	11	(39.3%)	28
PYTHON	148	(75.9%)	47	(24.1%)	195

Table 10. Commits with Preceding Integration

a revision r. 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 Table 10 we show the number of revisions we have identified by this approach. In the future, we will use such revisions to identify the risk of integrations.

6. Visualization of Concurrency

As a final analysis, we decided to map the concurrency in development using graphs in which each developer is represented with a vertex. An edge indicates that between two developers conflicts occurred and its thickness expresses the frequency of the conflicts. We refer to such graphs as *conflict graphs*. Self-conflicts as described in Section 4.3 are represented with self-loops.

In order to get many conflicts, we used the complete CVS history for each project. Figure 2 shows the conflict graph for PYTHON. We can recognize two groups: an *inner circle* in which the developers have many conflicts and an *outer circle* in which the developers have conflicts with only few developers. It is likely that the inner circle corresponds to the *core* developer team of PYTHON. Figure 3 shows the conflict graph for JBOSS. In this graph we can identify two clusters that are rarely connected with each other. These clusters likely correspond to independent developer teams.

In Figure 4 we treated the conflict graph for JEDIT as a *social network*. The size of a vertex represents with how many other people a developer had conflicts. This value is highest for developer 1.[8] Furthermore, the vertices are ranked by the layers according to their *betweenness*. In social networks, the betweenness is used to identify actors that play a central role. Once again developer 1 is most important followed by developer 17. Social network has frequently been applied to developer networks [3, 4] We believe that social network analysis on conflict graphs is a powerful tool for managers to improve the communication and collaboration in projects.

7. 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

[8] We decided to anonymize the developer names.

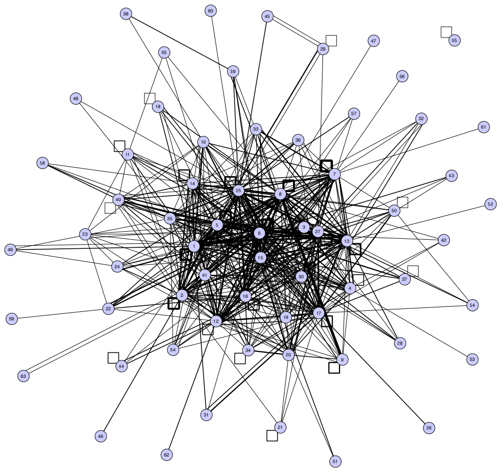


Figure 2. Concurrency in PYTHON

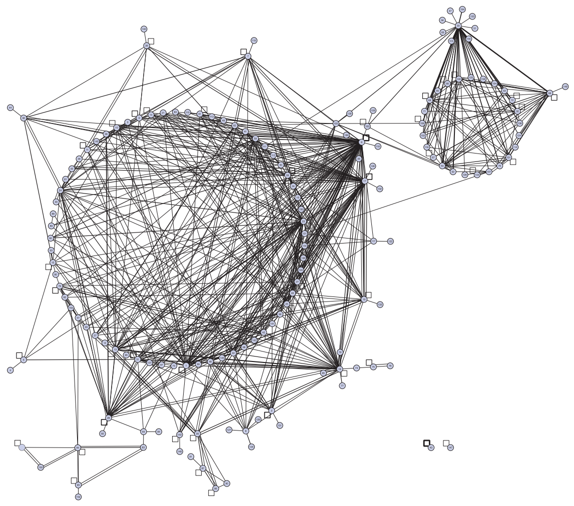


Figure 3. Concurrency in JBOSS

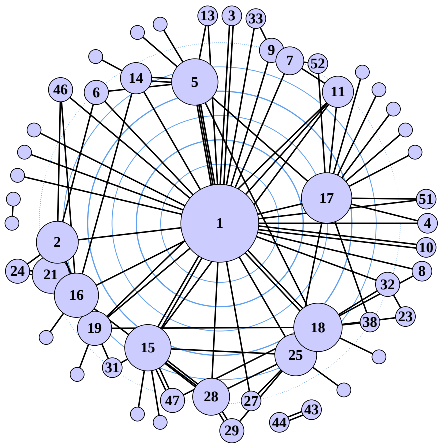


Figure 4. Concurrency in JEDIT

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, Pinzger, and Gall [2].
- 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. ECLIPSE and MOZILLA have disabled the history feature; for APACHE the history is only available for the anonymous CVS mirror. Fortunately for us, all projects that are hosted at SourceForge.net have CVS histories.

8. Related Work

To our knowledge this is the first work that analyzes CVS *activity data*. A similar case study based on *change data* was performed by Perry, Siy, and Votta [5]. They investigated parallel changes on different levels and observed a high degree of parallelism as well as a significant correlation between parallelism and the number of defects. We could not observe a high degree of parallelism on within single files.

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 [6]. Palantír continuously shares information about changes. This way it increases the awareness among developers and can reduce conflicts.

9. 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.
- The degree of parallel development was highest for PYTHON; every fifth commit caused a later integration.
- 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.
- Most developers work only on a single module and on a single branch.
- For JBOSS and PYTHON conflicts have been discarded more often than smooth integrations.
- 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, our plans for the future are as follows:

Deal with simultaneous workspaces. Right now, our approach cannot distinguish between different workspaces of the same developer. We plan to use *time windows* to combine several records into transactions. This should help to distinguish workspaces.

Analyze conflict graphs. The conflict graphs that we introduced in Section 6 represent relations between developers that *should* work very closely together. By applying graph and network analysis, we plan to identify virtual teams and important developers of a project. This can be used to improve the communication in a project and thus reduces the number of conflicts.

Assess the risk of commits. We can use the CVS history to categorize commits based on the outcome of the preceding update, i.e., without integration, with smooth integration, 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.

For ongoing information on this project, see

```
http://www.st.cs.uni-sb.de/softevo/
```

Acknowledgments.
Thanks to Holger Cleve, Christian Lindig, and Stephan Neuhaus for their valuable discussions and helpful suggestions on earlier revisions of this paper. Christian Lindig also provided the TEX macros used for Tables 4–7, and 9. Figure 2 and 3 were created with yEd [7] and Figure 4 was create with VISONE [1]. This research was funded by a scholarship from the Graduiertenkolleg "Leistungsgarantien für Rechnersysteme" sponsored by the Deutsche Forschungsgemeinschaft.

References

[1] U. Brandes and D. Wagner. Visone–analysis and visualization of social networks. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 321–340. Springer Verlag, 2003. Tool download http://www.visone.de/.

[2] 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.

[3] L. Lopez-Fernandez, G. Robles, and J. M. Gonzalez-Barahona. Applying social network analysis to the information in CVS repositories. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, pages 101–105, Edinburgh, Scotland, UK, May 2004.

[4] V. F. G. Madey and R. Tynan. The open source development phenomenon: An analysis based on social network theory. In *Americas Conference on Information Systems (AMCIS)*, pages 1806–1813, 2002.

[5] 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.

[6] 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.

[7] yWorks. *yEd–Java Graph Editor*, Apr. 2005. http://www.yworks.com/en/products$_yed_about.htm.$