Studying Production Phase SourceForge Projects: An Exploratory Analysis Using cvs2mysql and SFRA+

Daniel P. Delorey
Brigham Young University
TMCB 2230
Provo, UT 84601
pierce@cs.byu.edu

Charles D. Knutson
Brigham Young University
TMCB 2214
Provo, UT 84601
knutson@cs.byu.edu

Alex MacLean
Brigham Young University
TMCB 2236
Provo, UT 84601
amaclean@nm.byu.edu

ABSTRACT

A wealth of data can be extracted from the natural by-products of software development processes and used in empirical studies of software engineering. However, the size and accuracy of such studies depend in large part on the availability of tools that facilitate the collection of data from individual projects and the combination of data from multiple projects. To demonstrate this point, we present our experience gathering and analyzing data from nearly 10,000 open source projects hosted on SourceForge. We describe the tools we developed to collect the data and the ways in which these tools and data may be used by other researchers. We also provide examples of statistics that we have calculated from these data to describe interesting author- and project-level behaviors of the SourceForge community.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*Performance Measures, Process Metrics, Product Metrics*

Keywords

Software Engineering, Metrics, Data Mining, Software Repositories, CVS, `cvs2mysql`, SFRA$^+$

1. INTRODUCTION

In order for an empirical study of software engineering to have generalizable descriptive power, it must be based on a substantial amount of data from multiple distinct software projects. The nature of software development dictates that practitioners rarely, if ever, undertake an identical project twice. Thus, research that identifies the factors that resulted in the success or failure of a single project, while anecdotally interesting, cannot provide the general purpose description necessary to understand the nuances of the impacts of those factors in disparate contexts.

Collecting enough data from software projects to allow descriptive empirical studies can be problematic, however. Two of the most troublesome concerns when collecting data for empirical studies of software engineering are the cost of the data collection and the impact of the data collection on the process being measured.

The costs associated with collecting data from software engineering projects can range from fairly inexpensive (such as electronically distributing surveys to a group of developers or paying college students a nominal sum to participate in a brief experiment) to extremely expensive (such as funding an experimental software development organization consisting of multiple developers, managers, and support staff). Unfortunately, there tends to be an inverse relationship between the cost of the data collection process and the quality, reliability, and general applicability of the data collected.

The impacts of data collection on the software development process can include both changes to the process itself and changes in the behavior of the developers. These issues are analogous to those observed by Jain [4] when monitoring computer hardware performance. Adding measurements that require the production of previously nonexistent artifacts changes the development process in significant ways with potentially unintended and unrecognized consequences. In addition, observation of human subjects often changes their behavior in unanticipated ways as demonstrated by the Hawthorne effect and the placebo effect. As with the cost of data collection, there is a tradeoff to be made between the effects of the changes to the development process and the usefulness of the data collected.

In addition to the data quality tradeoffs for both data collection costs and process change requirements, there tends to be a direct relationship between the amount of process control required and the cost of data collection. Observational case studies, especially those using historical data, tend to reduce the data collection costs while limiting the amount of process control available. Controlled experiments increase the level of control while raising the costs to sometimes prohibitive levels.

One way to balance the tradeoffs between data quality, data volume, and data cost is to use the natural by-products of actual software projects as source data for empirical studies. Every successful software engineering process produces measurable data of varying utility. Ideally, various documents and metrics are created or gathered, lending insight into the development history. At a minimum, source code is devel-

oped and can be examined for meaningful clues. We refer to these as *software artifacts*. Of course, there are advantages and disadvantages to using these software artifacts in empirical studies. The advantages are that they are plentiful, they are relatively inexpensive to collect, and their production did not impose artificial modifications on the process that created them. The disadvantages are that were created by a process the researchers did not control, they often provide only indirect metrics for the question of interest, and, as with all observational data, they can not necessarily be used to infer cause and effect.

Many software artifacts are stored in version control repositories and project management systems. These artifacts may include source code, documentation, developer tasks, bug reports, and feature requests. Useful data for empirical studies can be extracted from all of these.

In this paper we describe the tools we developed to collect data from the software artifacts stored in CVS repositories and in the SourceForge project management system. We have used these tools to collect data from nearly 10,000 open source projects hosted on SourceForge and we give brief examples of the descriptive information that can be extracted from these data.

We chose open source software development as our target domain for a number of reasons. The most obvious, of course, is the availability of the artifacts. Considerably more effort would have been required to collect artifacts for 10,000 commercial or governmental software development projects. Beyond the availability, however, open source development offers interesting opportunities for descriptive empirical studies because of its emergent nature.

2. RELATED WORK

Many of the benefits of using existing artifacts in empirical software engineering studies were identified by Cook et al. [1]. The authors emphasize the expense and intrusion imposed by traditional empirical methodologies. The authors make the additional point that such changes to the development process in existing companies are often rejected by the engineers thus dooming the experiment from the beginning. These concerns are all the more critical in the open source development environment where the researchers do not even have the modicum of control they may have in an industrial setting. Also, the authors point out that traditional methods often ignore the past history of a project, focusing instead on the post-data-collection time period exclusively, despite the fact that most existing organizations have at least some form of historical data which may be mined for information.

Koch et al. [5] demonstrated many descriptive statistics that can be calculated from the data in CVS repositories and public discussion groups in their study of the GNOME project. Among the metrics they present are the number of lines of code added and removed per developer, the number of commits per developer, and the number of weeks contributed to the project per developer along with correlations between these values. In addition, the authors graph the growth of the various modules of the project over time in terms of lines of code. The information in this paper provides an empirical basis for understanding how the GNOME project is organized and how that organization has evolved over time—a necessary first step in determining why this project has succeeded while others have failed.

German et al. created softChange [2], a tool that extracts what authors call *software trails* from CVS repositories, BugZilla repositories, and mailing list archives and converts them into *facts*. The authors report having used soft Change to process the data of five large open source projects—GNOME, Mozilla, Evolution, PostgreSQL, and GNU gcc. The information gathered from these projects is used to record and compare *modification request* which the authors identify as a set of files changed and committed together to fix a bug or add a feature.

Robles et al. [7] developed CVSAnalY which gathers data from CVS log files, inserts them into a MySQL database, performs a set of analyses, and produces summary statistics and graphics. In June, 2006, CVSAnalY was run on the entire set of then-active SourceForge projects with publicly available CVS repositories. The collected data set is available as the CVSAnalY-SF dataset.

In addition to the tools that have been created for gathering data from publicly available sources, various archives of these data are being kept. The FLOSSMole project [3] regularly crawls SourceForge, FreshMeat, and RubyForge. The data are available for download in their raw form or in a relational database which is made publicly available through a web query form. Madey et al. [6] have partnered with Open Source Technology Group (OSTG) to create the SourceForge Research Archive (SFRA) which makes monthly dumps of the back-end database of SourceForge available to researchers. Researchers wishing to access the SFRA must sign a licensing agreement. These data are also accessed through a web query form.

3. DATA COLLECTION TOOLS

The tools we present in this paper collect data from two sources—CVS repositories and the SourceForge Research Archive (SFRA) hosted at the University of Notre Dame.

We chose to focus on CVS as a source of version control data for two reasons. First, a majority of SourceForge projects use CVS as their version control solution. Projects hosted on SourceForge have three version control options: 1) CVS, hosted by SourceForge; 2) Subversion, hosted by SourceForge; 3) Version control systems privately hosted by individual projects. At the time of our data collection in August, 2006, 92.5% (155,293 projects) of the projects hosted on SourceForge were using SourceForge hosted CVS repositories compared to only 4.4% (7432 projects) using SourceForge hosted Subversion repositories. Second, using data gathered from CVS as a proof of concept is appropriate because of minimality of the data CVS collects. The data collected by CVS are the file name, path and an optional free-form description for each file, and the revision number, revision date, author, file state, a count of lines added and removed, and a free-form message for each revision. Revisions are tracked on a per file basis and commits are non-atomic. Clearly, any information that can be gleaned from these meager data can also be calculated from the data of

more robust version control systems, providing a confidence in the extensibility of these results to other version control systems.

We chose to use the SFRA as our source of project management data for two reasons as well. First, the data was well-structured for the types of queries we had planned. Second, since the data are a direct dump of the SourceForge database, we had less concern that errors may have been introduced during the data collection process. We still expect that there are errors in the SFRA data, but we feel more comfortable assuming that they are randomly distributed errors caused by the SourceForge users and not systematic errors caused by flaws in the data collection tools.

In order to exploit the relationships between the data in the CVS repositories and the data in the SFRA, it was necessary to combine the data into a single relational database. To accomplish this, we developed two tools, `cvs2mysql` and SFRA$^+$, that collect the data from the two systems and write them into SQL scripts that import the data into a MySQL 5.0 database. Throughout the development of these tools, our overarching goal was to keep the coupling between the various steps as low as possible so that the tools would not impose our data collection process on future researchers, but would instead be useful in many analyses.

3.1 CVS Data Collection

We first considered using softChange or CVSAnalY in our CVS data collection, but found that neither suited our needs. Both were robust tools that included the entire tool chain the authors used in their own analysis. For example, softChange is designed for an analysis that considers data from a single CVS repository, a Bugzilla defect tracking system, and mail archives with change logs. CVSAnalY is designed to gather data from a single CVS repository and, as part of the data gathering process, produces a number of tables with derived statistics and graphical displays required by the authors for their own research. Also, both these systems produce data files for individual projects that are not designed to be combined with the data files produced for other projects.

Our tool, which we have named *cvs2mysql*, is developed in Python. It is cross platform compatible and has been validated extensively on Windows XP, Cygwin, Red Hat Enterprise Linux 4, and Mac OSX. In order to use `cvs2mysql`, Python 2.4 or higher and a CVS client are required. The source code is publicly available.

3.1.1 *The* `cvs2mysql` *Tool*

The `cvs2mysql` scripts can process any CVS repository when given a valid CVS root; however, due to the nature of our project, we extended the script to also allow processing of SourceForge CVS repositories using either a single SourceForge project's unix group name or a text file containing multiple projects' unix group names each on a separate line. We found this approach to be the most appealing because it allows `cvs2mysql` to be used for any project that needs to extract data from a CVS repository while still streamlining our own processing.

Standard `cvs2mysql` processing follows four steps: 1) checkout a sandbox from the project repository, 2) retrieve a log file for the repository, 3) parse the log file and create a MySQL import script, 4) remove the sandbox and the log file. The execution of these standard steps can be modified, however, using command line flags. So, for example, the user may choose to keep the sandbox and the log file by skipping the last step, thus allowing further processing of the source files. This modified processing can also be used to forego the checkout and logging steps when processing an existing sandbox.

The CVS checkout command used to retrieve a sandbox is run with the -r 1.1 option so that the initial revision of most files, including those that were removed from the repository at some point, is retrieved. However, it is possible in CVS to manually set the initial revision number for a file to something other than 1.1. `cvs2mysql` detects these cases when it finds a record of a file in the log whose earliest revision is not revision 1.1. In these cases, `cvs2mysql` will execute an additional checkout operation for the file to retrieve the earliest revision. These earliest revisions are used by `cvs2mysql` to determine the initial file size, a value that is not stored by CVS. We find this method of calculating initial file size better than using a function of the current file size and the lines added and removed for each revision for two reasons. First, it allows us to calculate initial file size even for files that are currently or were at one time removed from the repository (moved to the Attic in CVS terminology). Second, it removes the complexity of attempting to sum the number of lines added and removed for files that have been branched.

A single log file for the entire repository is retrieved both to simplify log file processing and to reduce the amount of network traffic. However, for larger projects, the CVS server may fail to return a log file for the entire repository. In these cases, `cvs2mysql` recovers from the error by attempting to recursively log parts of the repository individually. Logging begins in the top level directory of the repository. If the initial log command fails, `cvs2mysql` attempts to retrieve logs for the subdirectories and files of the failed directory individually. This behavior continues until either a log has been retrieved for the entire repository or logging fails for an individual file.

CVS uses the RCS log file format; however, as noted by German et al. [2], there is no publicly available grammar documenting the structure of RCS log files. We perfected our log file parsing through manual inspection of many RCS logs and various script revisions while running `cvs2mysql` on over 16,000 SourceForge CVS repositories.

In order to make `cvs2mysql` as widely applicable as possible, the data for each project is dumped to a separate MySQL import script and the post processing functions are separated from the data gathering functions. The import scripts are structured so that multiple scripts can be imported into the same database without modification thus simplifying the process of comparing individual projects or pooling the data from multiple projects for use in a single analysis. Also, we have kept the imported data as pure as possible by putting only the raw data gathered from the CVS repositories into the import scripts. All subsequent processing is handled by additional SQL scripts which store their results in tables

separate from the CVS data.

The schema for the data produced by `cvs2mysql` consists of two tables, cvs_file and cvs_revision, indexed by the project name and the author name respectively. For SourceForge projects, the project name is the project's unix group as listed on SourceForge and the author name is the author's SourceForge user name. For projects processed using a CVS root, the project name is set using an optional command line parameter; the author name is the CVS user name for the repository being processed. Indexing by the project's unix group name and the author's SourceForge user name for SourceForge projects rather than by some arbitrary numerical identifier allows the data collected using `cvs2mysql` to be easily joined to other sources of SourceForge data such as the SFRA and the FLOSSMole data which both have tables that can be joined using these values.

3.1.2 The SourceForge CVS Data

To validate the functionality of `cvs2mysql`, we gathered data from the projects in the SFRA August 2006 dump that meet the following criteria: 1) the project's development stage is set as Production/Stable or Maintenance, 2) the project is active, 3) the project uses CVS, 4) the project is open source. We chose the first two criteria as baseline indicators of project success. There is a significant number of projects created on SourceForge that never get beyond the inception phase. These projects represent a significantly different population than the one we wish to study. The third and fourth criteria indicate those projects we are able to study using the `cvs2mysql` tool. Again, `cvs2mysql` can only process CVS repositories and requires the original source code in order to determine the initial file sizes.

There are 16,580 projects in the August, 2006 SFRA schema that meet our criteria. We used `cvs2mysql` to process the CVS repositories of all these projects. However, during our processing, we found that approximately 40% of the projects did not have useable CVS repositories either because the CVS repository had never been used by the developers, the CVS repository was not publicly available using anonymous pserver access, or the repository had become corrupted. Excluding these projects, there were 9,999 projects with usable CVS repositories.

We collected data between September 8, 2006 and September 21, 2006. We produced 9,999 individual MySQL import scripts which we imported into a MySQL 5.0 database. These import scripts are currently available upon request.

3.2 SFRA Data Collection

The SourceForge Research Archive (SFRA) is a joint project between the University of Notre Dame and the Open Source Technology Group (OSTG) to make monthly dumps of the SourceForge back-end database available to the research community. The dumps are stored in a PostgreSQL database. However, the data is licensed under a strict agreement which limits the ways in which it can be distributed, so the database may only be accessed through a restrictive web form.

To increase the efficiency of our data gathering and overcome certain shortcomings of the existing interface of the SFRA we created a Windows-based desktop application, which we named *SourceForge Research Archive Plus* (SFRA$^+$+). This application is available along with its source code is publicly available. However, a user name and password are required to access the SFRA and these must be obtained from the University of Notre Dame group [6].

3.2.1 The SourceForge Research Archive Plus

One problem we encountered with the existing SFRA interface is that, although the database schema changes with almost every monthly dump, only a single ER diagram is provided and it is only partially accurate for the first of the available schemas. To overcome the lack of information about the structure of the database, SFRA$^+$+ is able to reverse engineer the structure for each of the schemas from the database itself, save that structure to a local XML file, and produce ER diagrams. These automatically generated ER diagrams are far from perfect. For example, we use a heuristic to determine foreign key relationships because this information is not stored in the database. However, these ER diagrams do allow us to see the main relationships between the tables which is critical in determining what questions can be answered with the data.

Another problem with the existing SFRA interface is that, instead of directly querying the database using rich PostgreSQL select statements, queries must be run through an arbitrarily restricted web form with only three text boxes—one for a select clause, one for a from clause, and one for a where clause—each of which automatically prefixes its contents with the corresponding keyword. To solve this problem, SFRA$^+$+ is able to accept any valid PostgreSQL select statement, translate it into a form that can be run using the web interface, and submit it to the web form.

A third problem with the existing SFRA interface is that results are returned in comma, colon, or pound sign delimited text files or in an XML formatted file, but no effort is made to replace the delimiters or existing XML formatted text in the result set and no header row is included to identify the resultant fields. To overcome the impossibility of interpreting delimited result files that have no header information and may contain delimiters in the result fields, SFRA$^+$+ automatically adds SQL commands to replace delimiters in all character based result fields and also automatically reinserts the delimiters into the result fields before presenting the results to the user in an editable grid.

SFRA$^+$+ automates many of the common tasks associated with using the SFRA. It has a rich SQL editor with syntax highlighting. It automatically submits formatted queries to the web interface, retrieves the results, and displays them in a grid for easier manual browsing. In addition, we have included functions to export result sets to both Excel files and MySQL import scripts so that the data can be combined with data from other sources and analyzed more fully.

3.2.2 The SourceForge Research Archive Data

There are 130 tables in the August 2006 schema of the SFRA storing all the data available on the SourceForge website as well as administrative data. The tables that hold data on SourceForge users and projects were of particular interest to us. We used the SFRA query tool to retrieve the contents of 12 of these tables—those linking users to projects and those

Table 1: Projects Registered Per Year Compared to Projects Included in Our Study

Year	Registered	Production by August, 2006	Percentage
1999	892	217	24.32%
2000	13374	1380	10.31%
2001	23740	1859	7.83%
2002	26766	1830	6.83%
2003	27649	1732	6.26%
2004	28909	1395	4.82%
2005	28599	998	3.48%
2006	17902	381	2.12%

with the tracker and forum data—and exported the contents into MySQL import scripts. We imported these tables into the same MySQL 5.0 database with the `cvs2mysql` data. Due to the licensing agreement of the SFRA, we are unable to release these import scripts; however, they can be easily recreated using SFRA$^+$+ by anyone who has access to the SFRA.

3.3 Data Collection Summary

In all, 9,999 import scripts for individual projects were generated using `cvs2mysql` and imported into a MySQL 5.0 database with a grand total of 7,244,201 cvs_file records and 26,559,193 cvs_revision records. In addition, 12 complete tables—including the tables that store project data, user data, and all the tracker data—were extracted from the SFRA and imported into the same MySQL 5.0 database. These two data sets can be combined by joining the cvs_file table to the groups table using the projects' unix group names and by joining tables with user names to the cvs_revision table using the author field.

4. EXAMPLE APPLICATIONS OF SOFTWARE ARTIFACT DATA

The data we have collected provide a historical view of the evolution of the SourceForge community. By combining the SFRA data with the CVS data of a large number of individual projects, we can get a better understanding of the typical lifecycles of SourceForge projects as well as typical and atypical relationships between authors and projects. In this section we provide examples of ways in which these data may be used to describe the SourceForge community.

4.1 The SourceForge Community

The earliest project registration date on SourceForge is October, 1999. However, the CVS logs for the projects we studied go as far back as December, 1983. In fact, 290 of the 9,999 projects have CVS logs prior to October, 1999 indicating that these projects were migrated to SourceForge some time after their inception. A comparison of the number of projects registered on SourceForge versus the number of projects that reached the Production/Stable or Maintenance phases by August, 2006 is shown in Table 1. The smaller number of projects registered in 2006 is a result of our collecting the data in September, 2006.

In addition to describing projects, our data gives insight into the behavior of a large number of open source developers.

Table 2: SourceForge Users Registered Per Year Compared to Authors Included in Our Study

Year	Registered	Included in Our Study	Percentage
1999	2810	441	15.69%
2000	98656	2158	2.18%
2001	221504	3494	1.57%
2002	214596	3989	1.85%
2003	226145	4320	1.91%
2004	220864	4026	1.82%
2005	228429	3389	1.48%
2006	160674	1588	0.98%

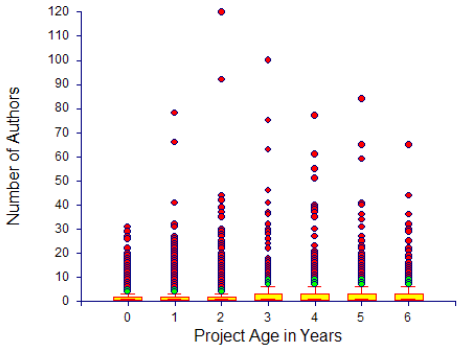

Figure 1: Distributions of authors per project by project age in years

In total, 23,838 distinct user names were recorded in the CVS data. Of these, 618 were not user names that were ever registered on SourceForge. Some of these user names were created automatically by the SourceForge system to facilitate anonymous commits for those projects that allow it. The rest were user names that were used in the repositories of migrated projects prior to their migration to SourceForge. Table 2 gives a comparison between the number of authors joining one or more of the projects in our study per year and the number of user names registered per year on SourceForge. To put these percentages into context, it is important to note that only 14.2% of the users registered on SourceForge have been granted CVS write access on at least one project, a necessary prerequisite for contribution to any project that does not allow anonymous CVS write access.

4.2 Analysis of Authors Per Project

The number of authors contributing to the CVS repository of an open source project implies something about the popularity and level of interest in the success of the project. Figure 1 shows a side-by-side box plots of the distributions of the number of authors contributing to a project for each year, measured from the date of the first commit to the project's CVS repository. The boxes are somewhat difficult to see due to the range of extreme values observed. However, the boxes, representing 75% of the observations range from one to two authors with the tails extending to three.

Of the projects studied, 83% have never had more than three authors in their life time and almost 91% have never had more than three in a given month. This raises interesting questions for future research. Is there something fundamental about open source development that favors smaller de-

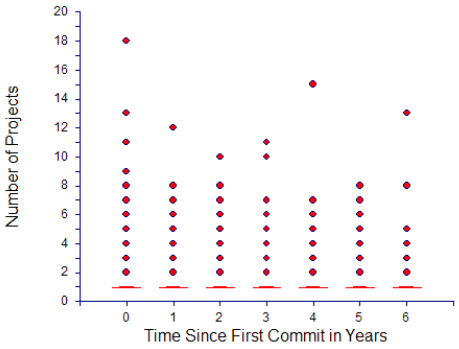

Figure 2: Distributions of projects per author by time in years since the author's first commit

velopment groups? If so, what is different about the organization of the other 10–20% of projects that allows them to have up to 120 active contributors?

4.3 Analysis of Projects Per Author

The number of projects to which an author contributes suggests something about the author's level of commitment to open source development as well as the author's availability and ability to multi-task. Figure 2 shows a side-by-side box plot of the distributions of the number of projects to which an author contributes for each year measured from the date of the first commit made by the author to the CVS repository of any of the projects studied.

For the projects we studied, the vast majority of authors devoted themselves exclusively to a single project at a time and a large portion of them have only ever contributed to one of the projects studied. These numbers also suggest potential avenues for future research. For the authors who never contribute to more than one project, how do they select the project to which they contribute? For the authors who contribute to multiple projects, especially those extreme outliers who are involved in up to 18 projects in a single year, are they able to split their time effectively and how do their contributions on an individual project basis compare to those of the developers dedicated to a single project?

5. CONCLUSIONS

The use of data collected from software artifacts can overcome some of the problems associated with empirical software engineering experiments based on contrived environments and altered processes. The data are plentiful, inexpensive to collect, and accurately reflect the process that created them.

However, the use of software artifacts in empirical research is not a panacea. We must remember that such data are observational and do not constitute a random sample. As such, the data may be used to provide compelling evidence but not necessarily to infer cause and effect or to generalize. We must, therefore, carefully report how and where the data were collected to avoid confusion about what conclusions may be drawn.

In the present study, we collected data from the CVS repositories of 9,999 open source projects hosted on SourceForge. Our study includes only those projects that are open source, use SourceForge hosted CVS repository as their version control system, and had reached the Production/Stable or Maintenance phases of their lifecycle by August, 2006. This set of projects is the inferential base for our conclusions.

We found that the vast majority of the projects we studied are developed entirely by three or fewer authors and that the vast majority of the authors contribute exclusively to a single project. However, there is large variation for the projects and authors that exceed these bounds. Some of the projects studied received contributions from more than 100 authors in a single year and some of the authors studied contributed to more than 15 projects in a single year.

The data we have collected can be used to study relationships beyond those we have presented in this paper. The CVS data is file and revision based, tracking the history of line changes over time, making it particularly well-suited to studies of the distributions of file types and of the rates of change per file.

As the methods for collecting and combining data from disparate sources mature, we expect to see more large-scale analyses comparing and contrasting software development efforts across the open source community. In addition, studies comparing data gathered from open source projects with those gathered in commercial and governmental software development settings will be of particular interest as they will help to calibrate and contextualize results based solely on open source projects.

6. REFERENCES

[1] J. Cook, L. Votta, and A. Wolf. Cost-effective analysis of in-place software processes. *IEEE Transactions on Software Engineering*, 24(8):650–663, August 1998.

[2] D. German. Mining cvs repositories, the softchange experience. In *Proceedings of the First International Workshop on Mining Software Repositories (MSR'04)*, pages 17–21, May 25, 2004.

[3] J. Howison, M. Conklin, and K. Crowston. Ossmole: A collaborative repository for floss research data and analyses. In *Proceedings of the First International Conference on Open Source Software*, July 11–15, 2005.

[4] R. Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley-Interscience, New York, NY, April 1991.

[5] S. Koch and G. Schneider. Results from software engineering research into open source development projects using public data. *Diskussionspapiere zum Ttigkeitsfeld Informationsverarbeitung und Informationswirtschaft*, 22, 2000.

[6] G. Madey. Sourceforge research data archive. http://www.nd.edu/oss/Data/data.html, 2005.

[7] G. Robles, S. Koch, and J. Gonzalez-Barahona. Remote analysis and measurement of libre software systems by means of the cvsanaly tool. In *Proceedings of the Second ICSE Workshop on Remote Analysis and Measurement of Software Systems*, May 24, 2004.