Observational Studies of Software Engineering Using Data from Software Repositories

Daniel Pierce Delorey
Brigham Young University - Provo

OBSERVATIONAL STUDIES OF SOFTWARE ENGINEERING USING DATA FROM SOFTWARE REPOSITORIES

by

Daniel Pierce Delorey

A thesis submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computer Science

Brigham Young University

April 2007

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a thesis submitted by

Daniel Pierce Delorey

This thesis has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Charles D. Knutson, Chair
Date	Christophe Giraud-Carrier
Date	Scott N. Woodfield

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the thesis of Daniel Pierce Delorey in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date	Charles D. Knutson Chair, Graduate Committee
Accepted for the Department	Parris K. Egbert Graduate Coordinator
Accepted for the College	Thomas W. Sederberg Associate Dean, College of Physical and Mathematical Sciences

ABSTRACT

OBSERVATIONAL STUDIES OF SOFTWARE ENGINEERING
USING DATA FROM SOFTWARE REPOSITORIES

Daniel Pierce Delorey
Department of Computer Science
Master of Science

Data for empirical studies of software engineering can be difficult to obtain. Extrapolations from small controlled experiments to large development environments are tenuous and observation tends to change the behavior of the subjects. In this thesis we propose the use of data gathered from software repositories in observational studies of software engineering. We present tools we have developed to extract data from CVS repositories and the SourceForge Research Archive. We use these tools to gather data from 9,999 Open Source projects. By analyzing these data we are able to provide insights into the structure of Open Source projects. For example, we find that the vast majority of the projects studied have never had more than three contributors and that the vast majority of authors studied have never contributed to more than one project. However, there are projects that have had up to 120 contributors in a single year and authors who have contributed to more than 20 projects which raises interesting questions about team dynamics in the Open Source community. We also use these data to empirically test the belief that productivity is constant in terms of lines of code per programmer per year regardless of the programming language used. We find that yearly programmer productivity is not constant across programming languages, but rather that developers using higher level languages tend to write fewer lines of code per year than those using lower level languages.

ACKNOWLEDGMENTS

First and foremost, thank you to my wife, Natalie, and our daughters, Alexia and Abigail, for their patience and support during those times they didn't see me as often as we all wished.

I am grateful to Dr. Charles Knutson for the time he spent advising my research and editing this thesis.

I also thank Dr. Christophe Giraud-Carrier for his valuable insights and suggestions which have improved this thesis.

I acknowledge Alex MacLean and Scott Chun for their collaboration on the research reported here.

Finally, thank you to my parents, Kevin and Betsey Delorey, for their love and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Empiricism in Software Engineering Research

In the keynote address at the 18th International Conference on Software Engineering in 1996, Victor Basili of the University of Maryland wrote, "Software Engineering needs to follow the model of other physical sciences and develop an experimental paradigm for the field" [2]. Researchers in many areas of software engineering have responded to this call by increasing both the quantity and the rigor of the experiments used to validate their claims. Empirical software engineering experiments are reported in [1, 7, 10, 16, 25–27], as well as many others. There have also been a number of books and papers published to guide researchers in designing and implementing empirical software engineering experiments [2, 4, 18, 20, 33, 35].

There are, however, those who argue that software engineering research is still of little use to practitioners. For a more detailed treatment, see [32]. One argument is that software engineering experiments tend to be too small and too contrived for their results to be reliably extrapolated to real-world software development environments. Another argument is that researchers either focus on issues that are of little concern to practitioners or attempt to prescribe solutions without fully understanding their effects in actual development environments. To answer these concerns and make software engineering research more useful to practitioners, many more large-scale empirical studies based on real-world data are needed to provide insight into the current state of software engineering practice, to identify where improvements can be made, and to demonstrate why the solutions proposed by researchers generate improvements in those areas.

The most favorable type of empirical study is a controlled experiment. The results of appropriately designed and implemented controlled experiments can be generalized to the larger population from which the subjects are selected and can be used to infer a cause-and-effect relationship between the independent and dependent variables (assuming the other confounding factors have been controlled appropriately). Unfortunately, large-scale real-world controlled experiments are not feasible in the software engineering space. Researchers do not run small controlled experiments on graduate student subjects by choice, but rather because large controlled experiments of professional software engineers are prohibitively expensive. In addition, given the effects that observation has been shown to have on human behavior [13] and the number of potential confounding factors, it is questionable whether an effective controlled experiment in software engineering is possible *regardless of cost*.

When controlled experiments are impractical, observational studies can often be effective. The results of observational studies may be compelling because the number of subjects can be larger and the researcher exerts less control over the process. However, the results of observational studies cannot be generalized to a larger population, and they cannot be used to infer cause-and-effect relationships. In software engineering research, very large observational studies based on real-world data are possible, although the conclusions must often be based on imprecise or indirect metrics.

Recently it has been recognized that a wealth of data for observational studies of software engineering is available in software repositories [6] and the mining of software repositories has emerged as a promising area of research. Workshops on mining software repositories have been held concurrently with the International Conference on Software Engineering (ICSE) the last three years [8, 9, 12]. The research presented in these workshops has used data produced as a natural result of software development to answer many interesting and important questions about the state of Open Source Software development.

The advantages of using the data in software repositories for observational studies are clear and compelling. The data are historical and were produced via a real-world process that was not manipulated by researchers to produce contrived artifacts that were unnatural to the process in order to allow some predetermined measurement. Thus, the researcher does not contaminate the data and the subjects do not modify their behavior due to observation. In addition, the data are plentiful. There are over 100,000 projects hosted on SourceForge.net alone with publicly accessible data available on the web site, through the FLOSS Mole project [14], and through the SourceForge.net Research Data Archive [21].

The disadvantages of using software repository data in observational studies are also clear. Rarely do the data provide a precise direct metric to answer a question of interest. Instead researchers using these data must craft methods of approximating their ideal metrics using the indirect metrics data available.

1.2 Thesis Statement

We assert that we can gather data from software repositories for use in observational studies which provide insights into the state of software engineering practice and empirically test hypotheses about software engineering.

1.3 Thesis Layout

We gathered data from 9,999 Open Source projects hosted on SourceForge. We used these data to investigate the sizes of open source development groups and the activity of open source developers that contributed to projects between 2000 and 2005. We also use these data to statistically test an assertion made by Brooks [3] and others [30, 34] that programmers write the same number of lines of source code per year regardless of the programming language they are using.

Chapter 2 is a paper currently under external review in which we present the tools we developed to gather the data we use in our analyses. These tools are SFRA$^+$ which provides a rich interface for accessing the SourceForge Research Archive hosted at the University of Notre Dame [21] and `cvs2mysql` which extracts historical data from CVS repositories and produces an SQL script which can be imported into a MySQL database. In addition, we present some preliminary results obtained using these data to describe the relationships between developers and projects in the Open Source community.

Chapter 3 is a paper currently under external review in which we use the data gathered with the tools presented in Chapter 2 to test the assertion of constant programmer productivity regardless of programming language. We test the assertion for the ten most popular programming languages used in the SourceForge projects.

The two appendices are user manuals for applications we developed as part of this research. Appendix A is the user manual for SFRA$^+$ and Appendix B is the user manual for `cvs2mysql`.

Chapter 2

A Comprehensive Evaluation of Production Phase SourceForge Projects: A Case Study Using cvs2mysql and the SourceForge Research Archive

2.1 Introduction

In order for an empirical study of software engineering to have 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. Two of the most troublesome concerns when collecting data for empirical studies of software engineering (which are typically in direct conflict) 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 [15] 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 precision 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. At best, various documents and metrics are created or gathered, lending insight into the development history. At worst, source code is developed and can be examined for meaningful clues. We refer to these as *existing artifacts*. Of course, there are advantages and disadvantages to using these existing artifacts in empirical studies. The advantages are that they are plentiful, they are relatively inexpensive to collect, and they accurately represent the software development process that produced them without imposing artificial modifications. The disadvantages are that they do not allow the researchers control over the development process, they often do not provide direct metrics to answer the question of interest, and, as with all observational data, they can only be used to provide compelling evidence, not to infer cause and effect.

Many of the existing artifacts of software development processes 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 artifacts.

To demonstrate the wealth of information that can be extracted from existing repositories, we have collected artifacts from nearly 10,000 open source projects hosted on SourceForge. 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. Rather than being driven by centralized administration or vision, the behavior of open source software development groups emerges from the behavior of autonomous individuals.

In this paper we describe the process we followed and the tools we developed to collect data from software artifacts. We also give brief examples of the descriptive information we can extract from these data.

2.2 Related Work

Many of the benefits of using existing artifacts in empirical software engineering studies were identified by Cook et al. [6]. The authors emphasize the expense and intrusion imposed by traditional empirical methodologies in which the researchers identify metrics that could be used to answer their questions of interest and then modify the development process to produce the data necessary to calculate those metrics. 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. [17] 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 [11], 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 softChange 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. [29] 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 through the authors of CVSAnalY.

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 FLOSS Mole project [14] 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. [21] 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.

2.3 Data Collection

We used two sources of data in this project – the CVS repositories of SourceForge and the SourceForge Research Archive (SFRA) hosted at the University of Notre Dame. 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. We chose to focus on the CVS repositories because 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.

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 because the data was well-structured for the types of queries we had planned. Also, since the data represents 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` (see Appendix B) and SFRA$^+$ (see Appendix A), 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.

2.3.1 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 also 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 they produce for other projects.

Since our goals were not compatible with the data produced by these existing tools, we developed our own tool for collecting histories of files and revisions from CVS repositories and converting that data into MySQL import scripts. Our tool, which we named *cvs2mysql*, was developed in Python. We designed it to be cross platform compatible and it has been validated extensively on Windows XP, Cygwin, Red Hat Enterprise Linux, and Mac OSX. In order to use `cvs2mysql`, Python 2.4 or higher and a CVS client are required. The source code is currently available upon request and we plan to make it publicly available in the near future.

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 and process an existing sandbox assuming a log file has already been retrieved.

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 calculating initial file size as 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. [11], there is not a 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 (see Figure B.2) 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 defaults to the empty string, but may be set manually by the user; 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.

The SourceForge CVS Data

To validate the functionality of `cvs2mysql`, we used it to gather 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 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 for the 9,999 SourceForge projects that met our criteria and had usable CVS repositories between September 8, 2006 and September 21, 2006. These data were converted into 9,999 individual MySQL import scripts which we imported into a MySQL 5.0 database. These import scripts are currently available upon request and will be made publicly available along with the `cvs2mysql` source code.

2.3.2 SFRA Data Collection

The Source Forge 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 query 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 using Delphi 5, which we named the *SourceForge Research Archive Plus* (SFRA$^+$+). This application is available along with its source code upon request and will be made publicly available soon. However, a user name and password are required to access the SFRA and these must be obtained from the University of Notre Dame group [21].

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 available in the database. However, these ER diagrams do allow the user 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.

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.

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 are of particular interest to our study. We used the SFRA query tool to retrieve the contents of 12 of these tables—those linking users to projects and those 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.

2.3.3 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.

2.4 Applications of Collected Data

The data we have collected provide a historical view of the evolution of the Source-Forge community. By combining the data from the entire SourceForge database with the data from the CVS repositories of a large number of individual projects, we can get a better understanding of the lifecycles of SourceForge projects and the relation-

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%

Table 2.1: Projects Registered Per Year Compared to Projects Reaching Production Phase by August, 2006

ships between authors and projects. In this section we provide examples of ways in which these data may be used to describe the entire SourceForge community.

2.4.1 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. Figure 2.1 graphs the number of projects started per year based on the CVS logs. Note that the downward trend after 2004 is a remnant of our choosing to study only those projects in the Production/Stable or Maintenance phases of their lifecycle. 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 2.1. 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. In total, 23,838 distinct user names were recorded in the CVS data. Of these, 618 were not user names that were ever

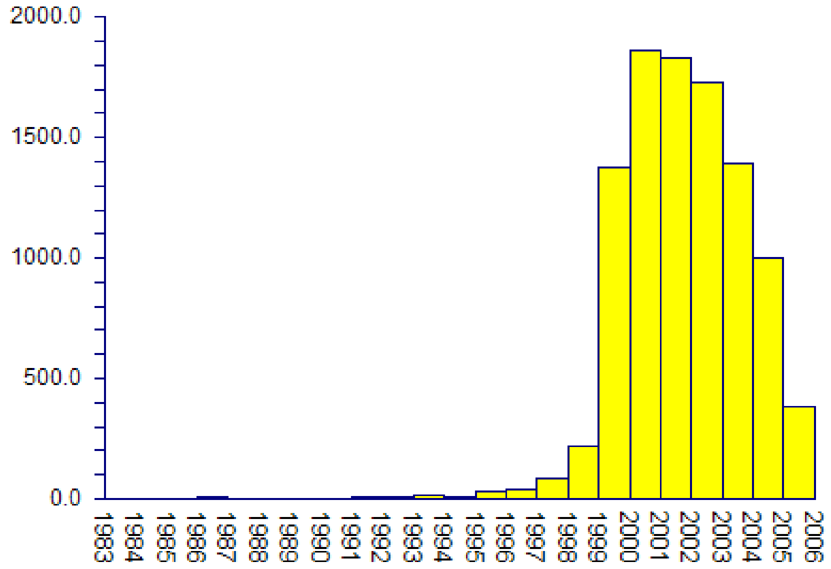


Figure 2.1: Count of projects started per year based on CVS logs

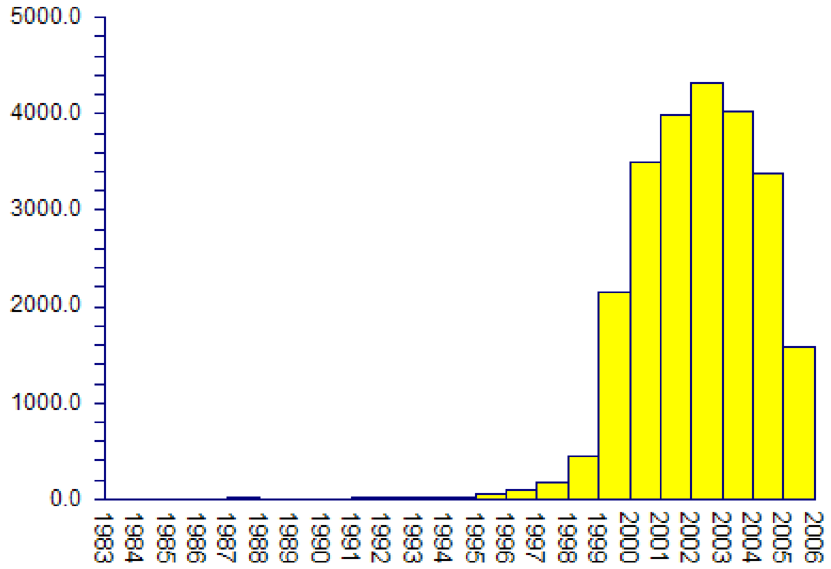


Figure 2.2: Count of authors joining per year based on CVS logs

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%

Table 2.2: SourceForge Users Registered Per Year Compared to Authors Included in Our Study

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. Figure 2.2 shows the number of authors joining one or more of our projects per year based on the CVS logs. In this case the downward trend after 2004 is interesting because it suggests that the number of developers per project was remaining approximately constant rather than increasing. Table 2.2 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.

2.4.2 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. Figures 2.3 and 2.4 show side-by-side box plots of the distributions of the

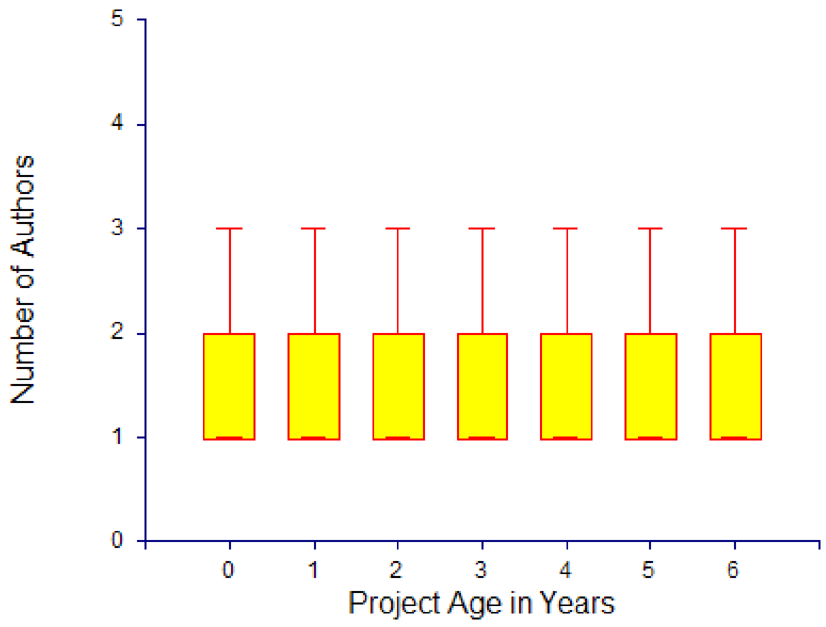


Figure 2.3: Distributions of authors per project by project age in years with outliers excluded

number of developers contributing to a project for each year, measured from the date of the first commit to the project's CVS repository. In Figure 2.3, the extreme values are excluded to show detail. In Figure 2.4, the extreme values are included to show range.

Identical patterns to the ones in Figures 2.3 and 2.4 are observed whether the time granularity is years, quarters, or months except that the range of the extreme values shrinks somewhat as the time period is decreased. The median number of developers per project is 1. Three quarters of the projects have between 1 and 2 developers. Table 2.3 lists the percentages of projects that have had three or fewer developers per month, quarter, year, and over their entire development.

The fact that almost 83% of the projects studied have never had more than three developers in their life time and almost 91% have never had more than three in a given month raises interesting questions for future research. Is there something fundamental about open source development that favors smaller development groups?

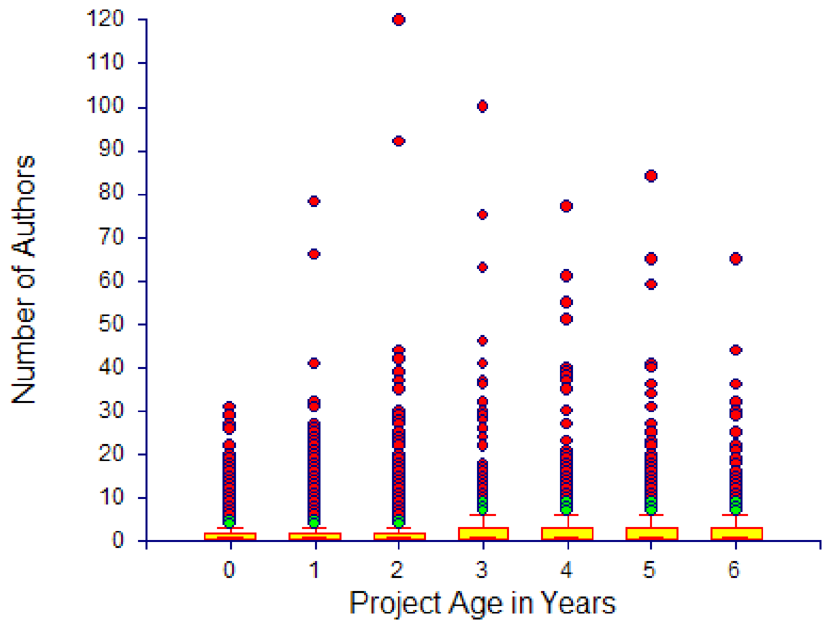


Figure 2.4: Distributions of authors per project by project age in years with outliers included

All Time	82.7%
Per Year	87.3%
Per Quarter	89.5%
Per Month	91.2%

Table 2.3: Percentage of Projects with Three or Fewer Authors

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

2.4.3 Analysis of Projects Per Author

The number of projects to which an author contributes can suggest 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 2.5 shows a side-by-side box plot of the distributions of the number of projects to which an author contributes for each year

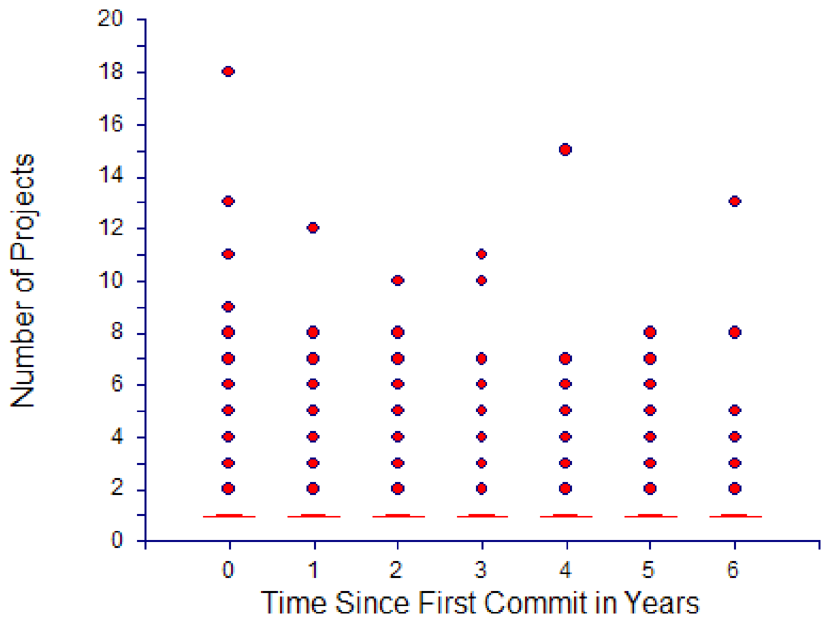


Figure 2.5: Distributions of projects per author by time in years since the author's first commit

All Time	86.6%
Per Year	90.5%
Per Quarter	92.5%
Per Month	94.2%

Table 2.4: Percentage of Authors Contributing to One Project

measured from the date of the first commit made by the author to the CVS repository of any of the projects studied.

As with the distributions in Figures 2.3 and 2.4, the project per author distributions do not change regardless of the level of time granularity. The median and 75th percentile is 1 project per author. Table 2.4 shows the percentages of authors who have contributed to only one project per month, quarter, year, and over their entire tenure.

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?

2.5 Conclusions

The use of data collected from the existing artifacts of unaltered software processes 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 data collected from the artifacts of an uncontrolled process 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 conclusion 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 20 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.

Chapter 3

Do Programming Languages Affect Productivity? A Case Study Using Data from Open Source Projects

3.1 Introduction

Brooks is generally credited with the assertion that annual lines-of-code programmer productivity is constant, independent of programming language. In making this assertion, Brooks cites multiple authors including [30] and [34]. Brooks states, “Productivity seems constant in terms of elementary statements, a conclusion that is reasonable in terms of the thought a statement requires and the errors it may include.” [3] (p. 94) This statement, as well as the works it cites, however, appears to be based primarily on anecdotal evidence. We test this assertion across ten programming languages using data from open source software projects.

3.2 Related Work

Various studies of productivity in software development have been reported, including [19, 22, 23, 28].

Empirical studies of programmer productivity differ in the productivity measures used, the types and quantities of data used, the explanatory factors considered, the goals of the study, and the conclusions reached.

The most common productivity metrics are lines of code per unit time [23] and function points per unit time [19, 22, 28]. While compelling arguments are made in the literature for both of these metrics, we use lines of code both because the assertion we are testing was stated in terms of lines of code.

Studies of software development productivity tend to rely on observational data collected from commercial projects. Maxwell et al. use data collected from 99 projects from 37 companies in eight European countries [23] and data gathered from 206 projects from 26 companies in Finland [22]. Premraj et al. use an updated version of the same data set with over 600 projects [28]. Liebchen et al. use a data set representing more than 25,000 projects from a single company [19]. Our data set was collected from the CVS repositories of 9,999 open source projects hosted on SourceForge.

The data sets used in these studies were each compiled manually with some level of subjectivity and transformation. Given this level of human involvement, the factors they consider are at a high level of abstraction. For example, the data set in [23] contains among its variables seven COCOMO factors, including required reliability, execution time constraints, and main storage constraints, each with discrete ordinal values between 1 and 6. Our data set contains only those features that can be calculated from the data in a CVS repository. As such, our data is limited conceptually but has the advantages of being concrete, objective, and simple to gather.

In each of the papers cited, the stated goal of the study was to identify the major factors influencing programmer productivity. The models developed in these studies were intended to be either predictive, explanatory, or both. Our goal is not to construct a predictive or explanatory model. Rather, we seek only to develop a model that sufficiently accounts for the variation in our data so that we may test the significance of the estimated effect of programming language.

	Project Rank	Author Rank	File Rank	Revision Rank	LOC Rank	Final Rank
C	1	1	2	2	1	1
Java	2	2	1	1	2	2
C++	4	3	4	4	3	3
PHP	5	4	3	3	4	4
Python	7	7	5	5	5	5
Perl	3	5	9	9	6	6
JavaScript	6	6	6	8	10	7
C#	9	9	7	6	7	8
Pascal	8	10	8	7	8	9
Tcl	11	8	10	10	9	10

Table 3.1: Top ten programming languages by popularity rankings

3.3 Data Collection

The data we use in our analysis comes from the CVS repositories of open source projects hosted on SourceForge. The tools we developed and methods we employed in collecting the data are described in Section 2.3.

As CVS manages individual changes (called *revisions*) it records the author of the change, the date and time the change happened, the number of lines that were added to and removed from the file, and a mandatory free-form message supplied by the author. These minimal data can be combined to produce a rich set of values describing the environment in which the change was made.

We collected data from the CVS repositories of 9,999 projects hosted on SourceForge. Our population for the data collection was the set of projects that met 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 gathered the entire history for each of the 9,999 CVS repositories and stored the resulting data in a MySQL relational database using a tool we developed called `cvs2mysql` (see Section 2.3.1). The resulting raw data contains records for 7,244,201 files and 26,559,460 changes to those files made by 23,838 developers.

3.3.1 Data Preparation

Of the more than 19,000 different file extensions represented in the SourceForge database, we identified 107 unique programming language extensions. In order to limit the scope of our study to the languages that are most widely used, we produced an ordered list of the most popular programming languages represented in the database. Popularity is defined here in terms of: 1) total number of projects using the language; 2) total number of authors writing in the language; 3) total number of files written in the language; 4) total number of revisions to files written in the language; and 5) total number of lines written in the language. We ranked each language using these five metrics and calculated the average ranking for each language. We then ranked the languages by their average rankings to determine an overall ranking. We chose to focus on the top 10 programming languages which are listed along with their rankings in Table 3.1. These 10 languages are used in 89% of all projects, by 92% of all authors, and account for 98% of the files, 98% of the revisions, 99% of the lines of code in our data set. The next three most popular languages are Prolog, Lisp, and Scheme, none of which can be easily compared to imperative and object-oriented languages on a line by line basis given the differences in programming paradigm.

We compare annual productions per programmer per language in an effort to limit the impact of normal variations in the amount of time individual programmers commit to development over smaller time periods. Data collection was limited to the time period from January 1, 2000 to December 31, 2005.

Our model of aggregating the lines written across authors, programming languages, and years assumes that every line committed to CVS by an author was written

- **Language Related Factors Per Year**
  - **For the Current Year**
    - Months since first recorded use
    - Active projects using this language
    - Active authors using this language
    - Current files written in this language
    - Total number of lines written in this language
  - **Aggregated Over Prior Years**
    - Total projects having used this language
    - Total authors having used this language
    - Total files written in this language
    - Total number of lines written in this language
- **Author Related Factors Per Year**
  - **For the Current Year**
    - Months since first contribution
    - Active projects with contributions
    - Number of programming languages used
    - Current files edited
    - Total number of lines written
  - **Aggregated Over Prior Years**
    - Total projects with contributions
    - Total number of programming languages used
    - Total files edited by this author
    - Total number of lines written by this author
- **Language Specific Author Related Factors Per Year**
  - **For the Current Year**
    - Months since first contribution
    - Active projects with contributions
    - Current files edited
  - **Aggregated Over Prior Years**
    - Total number of lines written
    - Total projects with contributions
    - Total files edited by this author
- **Temporal Factor**
  - Calendar Year

Table 3.2: Potential explanatory factors considered

by that author during the year in which it was committed. However, we identified six ways in which this assumption can be violated:

- Migration – An existing CVS repository created by multiple authors and/or over multiple years is migrated to SourceForge by a single author.
- Dead File Restoration – When a dead file is restored in CVS, the contents are not differenced against the pre-removal version.
- Multi-Project Files – Authors may contribute the same file to multiple projects.
- Gatekeepers – Gatekeepers receive credit for all the lines they commit even if they were not the author.
- Batch Commits – An author may work for more than a year before committing the changes.
- Automatic Code Generation – The tools an author uses to program may automatically generate lines of code which the author then commits to CVS.

While the data collected by CVS does not allow us to definitively identify all cases that violate our assumptions, we have taken steps to exclude as many offending cases as possible while sacrificing as few of the cases that do not violate our assumptions as is reasonable. To remove the migration cases, we excluded initial revisions for all files in our data set. To remove the dead file restoration cases, we excluded all revisions that followed a "dead" revision. After removing these, however, significant unrealistic outliers remained in our data set. To remove these outliers, we limited our population to those authors who had written fewer than 80,000 lines of source code in a single year. Since we believe that those authors who wrote more than 80,000 lines in a single year are exhibiting one of the non-population behaviors described above, we also exclude from our analysis the projects to which they contributed.

After limiting target programming languages and removing observations deemed to be outside our population, our target data contains records of 673,528

Factors Excluded Due to High Variance Inflation Factors (VIF Value)

Total authors having used the programming language in prior years (1860)
Total authors using the programming language in the current year (258)
Total projects having used the programming language in prior years (68)
Files written in the programming language in the current year (51)
Active projects using the programming language in current years (12)

Factors Excluded Due to Low Correlation with the Dependant Variable (Correlation)

Months since the first recorded use of the programming language (0.0071)
Calendar Year (0.0093)

Factors Excluded Due to Practically Insignificant Coefficients (Coefficient)

Total number of lines written in the language during the current year (0.0000)
Total number of lines written in the language during prior years (0.0001)

Factors Removed During Variable Selection Using the Cp Statistic

Total number of languages used by the author during prior years
Total number of files written in the language during prior years

Table 3.3: Explanatory factors excluded from our analysis

files, 4,198,724 revisions, and 16,197 authors. These data are aggregated across author, programming language, and year into 34,566 observations in our final data set.

3.4 Data Analysis

The goal of our data analysis is to determine whether there is evidence in the data we have collected that programming languages affect annual programmer productivity. Our dependant variable in this analysis is the lines of code committed to the CVS repositories of selected SourceForge projects by an individual author in a single year. Our independent variable is the programming language being used. We test all pairwise differences between the languages, adjusting our confidence intervals using the Tukey-Kramer Honest Significant Difference for multiple comparisons.

Clearly there are factors other than programming language that affect programmer productivity. Before testing the significance of the programming language effect, we must account for the effects of these confounding variables. We do this by including the confounding factors in a multiple linear regression analysis along with the independent variables so that their effects can be separated. The potential confounding factors we consider in this analysis are listed in Table 3.2. It is important to note that our goal is only to separate confounding effects before testing our independent variable. Our model is not intended to be predictive or explanatory. Therefore, we do not report the coefficients or the p-values of the confounding factors.

We develop our model by first excluding the programming language and considering only the confounding factors as independent variables. We systematically remove independent variables until we achieve the simplest model that still explains a significant portion of the variation in our data. To this model we then add the programming language factor and test its significance. The procedure for reducing the model is explained below.

We begin by removing independent variables that are highly correlated. Using correlated independent variables in a multiple regression leads to a condition known as multicolinearity which can affect the precision of estimates in unexpected ways. The Variance Inflation Factor (VIF) is a measure of multicolinearity. A VIF value grater than 10 is considered large. Using multicollinearity analysis we remove five of the independent variables. These variables along with their VIF values are listed in Table 3.3.

We next remove independent variables that have no explanatory power. To be useful as an independent variable in a multiple linear regression, a variable must have a linear relationship with the dependent variable. Correlation is a measure of linear relationship. Using the correlation between each independent variable and the dependant variable methods we are able to remove two of the independent variables. These variables along with their correlation coefficients are listed in Table 3.3.

Fitting a regression on the remaining variables we find that two of the variables have an estimate coefficient equal to or near zero. These coefficients are not statistically significant, but more importantly, they are not practically significant either, so they are removed. These variables along with their estimated coefficients are listed in Table 3.3.

Finally, the last step in reducing our model is to fit regressions using all possible subsets of the remaining variables and pick the model that best satisfies a model-fitting criterion. The model fitting criterion we use is the Cp statistic. The Cp statistic focuses directly on the trade-off between bias due to excluding important independent variables and extra variance due to the inclusion of too many variables. Using Cp selection on the remaining 16 independent variables, we find the model with the lowest Cp statistic in which all independent variables are significant contains 14 independent variables. The two independent variables excluded from this model are listed in Table 3.3.

Our final model contains 14 independent variables. Again, the goal of our analysis is not to create a predictive or an explanatory model but rather to control as much of the variation in the data as possible before testing the significance of the effect of programming language on average annual programmer productivity. Therefore, we do not explicitly present the independent variables included in our model to prevent the casual reader from interpreting our model as explanatory or predictive. For the curious reader, the independent variables included in our model can be determined using Table 3.2 and Table 3.3. The R^2 for our model is 0.80 meaning that it explains 80% of the variation in our data. All the independent variables are statistically significant at $p < 0.05$. The model is significant at $p < 0.0001$.

3.5 Results

To test the assertion that programmer productivity is constant in terms of lines of code per year regardless of the programming language being used, we fit a model consisting of the 14 independent variables selected in Section 3.4 to adjust for variation in programmer ability and programming language use. To this model, we add indicator variables for the programming languages we are considering. By running the analysis nine times and using a different language as the reference each time, we are

	JavaScript	Perl	Tcl	Python	PHP	Java	C	C++	C#
Perl	0.46								
Tcl	0.60	1.00							
Python	0.00	0.00	0.76						
PHP	0.00	0.00	0.08	0.72					
Java	0.00	0.00	0.02	0.18	1.00				
C	0.00	0.00	0.00	0.01	0.53	1.00			
C++	0.00	0.00	0.00	0.00	0.01	0.07	0.59		
C#	0.00	0.00	0.00	0.02	0.26	0.50	0.83	1.00	
Pascal	0.00	0.00	0.00	0.00	0.10	0.26	0.60	0.99	1.00

Table 3.4: Pair-wise language comparisons

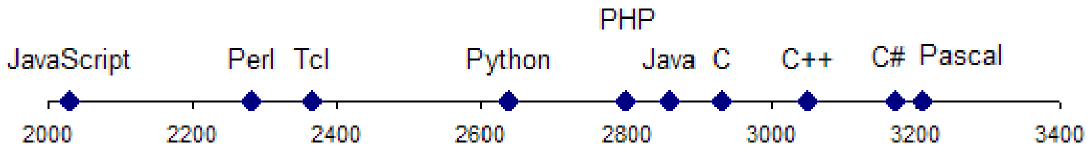


Figure 3.1: Estimated Average Productions

able to determine the estimated differences between the languages and the standard errors for each of those estimates which we then use to test the significance of the differences.

The null hypothesis for our tests is that there will be no difference in estimated average annual productions per programmer for any of the languages. However, we find evidence in the data to reject the null hypothesis for 24 of the 45 pair-wise comparisons. The *p*-values for the comparisons, adjusted using the Tukey-Kramer Honest Significant Difference for multiple comparisons are listed in Table 3.4. The shaded cells are the comparisons for which we reject the null hypothesis with 95% confidence or greater. To clarify the magnitudes of the differences, Figure 3.1 shows the estimated average annual productions for each language.

Using Table 3.4 and Figure 3.1 together we can observe groupings in the languages. Python, which sits near the middle of the range of estimated annual productions, for example, follows a different paradigm from the languages on each end of the range (JavaScript and Perl on the left and C, C++, C#, and Pascal on the right), but it is not significantly different from the other languages near the middle (Tcl, PHP, and Java). Further analysis may reveal that programming language paradigm influences programmer productivity.

3.6 Conclusions

We find significant evidence in our data that, even after accounting for variations in programmers and environments, programming languages are associated with significant differences in annual programmer productivity. The reader must be careful, however, not to infer a cause-and-effect relationship based solely on this study. Our analysis relies on observational data gathered from SourceForge.net CVS repositories. This is a strength in that the data represent an unaltered software development environment. However, it does limit the inferences we can make both in terms of cause-and-effect and generalization.

Nevertheless, the results of this study suggest a number of interesting avenues for future research. For example, there is a general progression in Figure 3.1 from newer, higher-level interpreted languages to older, compiled languages. This progression may imply a relationship between the level of abstraction of a language and the speed at which developers can write source code in that language. Brooks supported the assumption of constant productivity as "reasonable in terms of the thought a statement requires and the errors it may include." However, it is quite possible that today's higher-level languages require more thought per line or allow more errors per line than their predecessors. More research is needed to better understand the tradeoffs between the power provided by languages with higher levels of abstraction and the cognitive load placed on their users.

We expect that this model of using large-scale, longitudinal studies of Open Source projects to empirically test long-held assumptions in software engineering research will become more prevalent as the tools and methods for collecting and analyzing data from software repositories mature. Such studies are necessary in order to build a more firm foundation for understanding the similarities and differences between Open Source and other software development models.

Chapter 4

Conclusion

In this thesis we have argued in support of the use of data gathered from software repositories in observational studies of software engineering. We have presented the tools we developed to simplify the process of gathering data from the SourceForge Research Archive [21] and from CVS repositories [5]. We have also provided examples of how the data collected with these tools can be used in exploratory analyses as well as in observational studies.

4.1 Contributions

The contributions made by this research to the field of Computer Science, and more specifically to the study of Software Engineering, can be broken into three categories: 1) the contributions of the tools for use by the research community, 2) the contributions of the empirical studies which add to the knowledge of the research community, and 3) the contributions of proposed methodology to the ongoing development of an ”experimental paradigm for the field” [2].

4.1.1 Contributions of the Tools

The tools we have presented in this thesis, `cvs2mysql` and SFRA^{+}, are unique in their design and in their functionality. Rather than developing tools that are specific to our research, we have designed our tools to be general purpose. Each tool performs a single task simply. The results of the tools can be combined as desired by the user, but it is not necessary for users to perform all the steps of our analyses when using our tools.

`cvs2mysql` takes as input a CVS repository and produces as output an SQL import script containing all the historical data available from the repository. In addition to processing any CVS repository given the CVS root, `cvs2mysql` streamlines the processing of SourceForge CVS repositories by requiring only either a single project's unix group name or a file containing a list of project unix group names. The structure of the schema produced by `cvs2mysql` is remarkably simple consisting of only two tables. Also, a single SQL script is produced for each repository even if multiple repositories are processed during a single invocation of `cvs2mysql` and the SQL scripts produced for each repository are structured so that they can be imported into a properly structured database regardless of the contents of the tables. Existing data are not effected by importing additional `cvs2mysql` SQL scripts and the records in the SQL scripts are correctly linked despite the presence of the existing data. For a complete discussion of `cvs2mysql`, see Appendix B.

SFRA$^+$ replaces the existing interface of the SourceForge Research Archive and provides additional functionality which greatly increases the usability of this excellent resource. In addition to removing the SQL formatting restrictions imposed by the existing interface, SFRA$^+$ automatically retrieves, parses, and displays the results of queries. Beyond the querying functionality, SFRA$^+$ has the ability to reverse engineer the structure of the database and produce ER diagrams which are extremely helpful in discovering relationships between the data. SFRA$^+$ also exports query results to SQL scripts so that they may be imported into a MySQL 5.0 database and combined with other data sets such as those produced by `cvs2mysql`. For a complete discussion of SFRA$^+$, see Appendix A.

4.1.2 Contributions of the Empirical Studies

In this thesis we have presented one exploratory analysis and one observational study. Each of these demonstrated a way in which data from software repositories can be used.

The exploratory analysis focused on the relationship between authors and projects in Open Source development. We found that 82.7% of projects have never had more than three contributors in their entire existence and that 91.2% have never had more than three contributors in a single month. We also found that 86.6% of developers have never contributed to more than one Open Source project and that 94.2% have not contributed to more than one project in a single month.

The observational study tested the assertion made by Brooks [3] and others [30, 34] that annual programmer productivity in terms of lines of code is constant regardless of the programming language being used. In order to test this assertion, we first developed a model to control the variations in our data. Our model consisted of 14 independent variables and explained 80% of the variation in the data. Using this model, we then tested the significance of the programming language effect. We found that there were significant differences in the annual lines-of-code produced by developers using different programming languages.

While the studies reported in this thesis cannot necessarily be used to infer causality or to generalize the conclusions to a larger population, the data sets are sufficiently large that our results can be used to provide compelling evidence in support of causality and generalization.

4.1.3 Contributions of the Methodology

In this thesis we claim that the appropriate experimental paradigm for software engineering research is large-scale observational studies based on data gathered from artifacts produced by real-world software development processes rather than small controlled experiments conducted in artificial environments. Our reasons for this are two-fold. First, data gathered from existing software artifacts are more plentiful and more easily available than data collected during controlled experiments. Second, by using the natural by-products of the software development process, researchers can avoid the problems associated with monitoring and observing human subjects. While we acknowledge the drawbacks of observational studies, such as the inability to infer causation or to generalize the results, we assert that these are outweighed by the benefits of utilizing orders of magnitude more data and avoiding the negative effects of observing human subjects.

4.2 Future Work

One avenue of future work indicated by our research lies in the creation of additional tools. While CVS is currently the most popular version control system used by Open Source projects, it is not the only system in use nor is it likely to remain the most popular forever. Other systems such as Subversion and BitKeeper are increasing in popularity and are already used by some high-profile Open Source projects. In addition, more robust systems such as SourceSafe and ClearCase are often used by commercial development organizations. Future research that developed tools to extract data from version control systems other than CVS as well as research into a standardized schema that would allow data from various systems to be combined would be of great use.

Another option for future research based on this thesis would be to extend the empirical studies we present and to develop and test hypothesis for the questions they raise. For example, in this thesis we have only explored the relationship between authors and projects. Exploratory analyses of the relationships between authors and files and between multiple authors are likely to provide additional insights into the Open Source development community. In addition, we have raised numerous questions in the course of these studies. For example, the results of the observational study suggest that developers using higher-level languages, and especially languages with a more multi-paradigm feature set, tend to write fewer lines of code per year on average. This could suggest that those languages place a higher cognitive demand on their users, or it could suggest that those languages are more specialized than their imperative or object-oriented counter parts causing them to less generally utilized.

Finally, the bold claims we have made about the relative merits of observational studies and controlled experiments demand further scrutiny and comment from the research community. Surveys of the techniques used by social scientists to limit the impact of observation on human subjects and to gather measurements without disturbing existing systems will be especially impactful.

Appendix A

Documentation for the SourceForge Research Archive Plus

A.1 Overview

The SourceForge Research Archive (SFRA) is a collaboration between the Open Source Technology Group and the University of Notre Dame [21]. Monthly dumps of the SourceForge.net back-end database are archived at the University of Notre Dame and made available to academic researchers who sign a licensing agreement. The data are stored in a PostgreSQL database with each dump in a separate schema. The schemas contain between 73 and 139 tables. Access to the database is provided through a password protected web interface shown in Figure A.1.

The monthly dumps of the SourceForge.net database are an excellent source of SourceForge project data which allows researchers to avoid the time commitments and potential data corruption involved in gathering these data indirectly through the SourceForge.net website. However, the following limitations we encountered in using the existing SFRA interface render the system, if not unusable, at least less than desirable.

- Unconventional and seemingly unnecessary formatting requirements have been placed upon the user. For example, queries must be broken into three parts and entered into three separate text boxes on a web form. These text boxes are labeled Select, From, and Where; each must be filled with the content of the corresponding clause of the query excluding the key word from the label

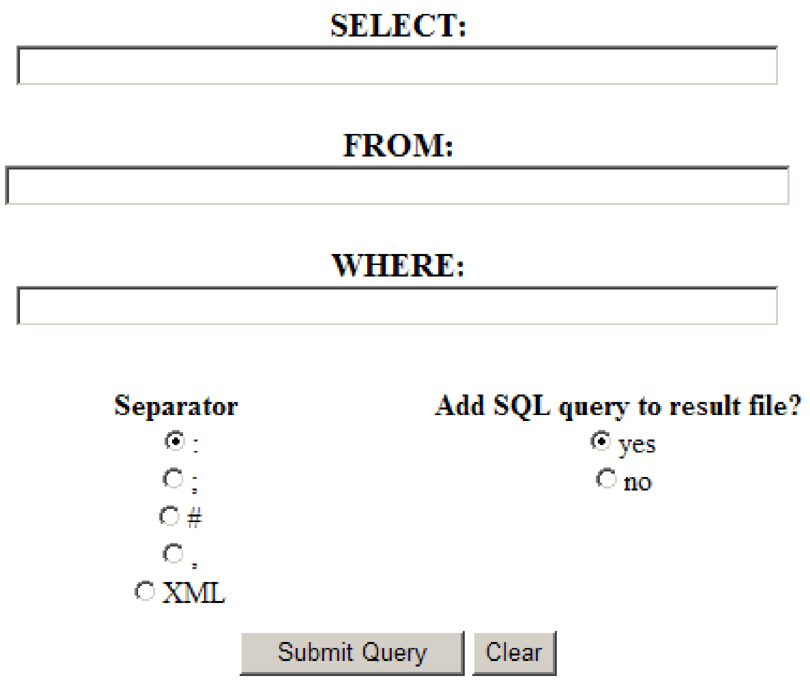


Figure A.1: The SourceForge Research Archive Web Interface

which is automatically added to the clause by the processing Perl script. Also, all table names used in the query must be prefixed with their schema name causing the user to enter the same schema name potentially many times.

- The complexity of the queries allowed by the web interface appears to be arbitrarily constrained to those with a `select` clause, a `from` clause, and a `where` clause. This constraint would preclude queries using `group by`, `having`, `order by`, and `limit` clauses, for example. By further testing the web interface and hypothesizing the behavior of the underlying Perl script, we determined that these more powerful queries could be run if the additional clauses were appended to the end of a valid `where` clause.

- Results are returned in delimited text files or in XML formatted text files. For the delimited files, no effort is made to replace the delimiters within the result fields and no header line is included, leading to difficulties in interpreting the results. The XML files are bloated, often to several times the size of the result set, by the repeated inclusion of the field names for each record in the results.

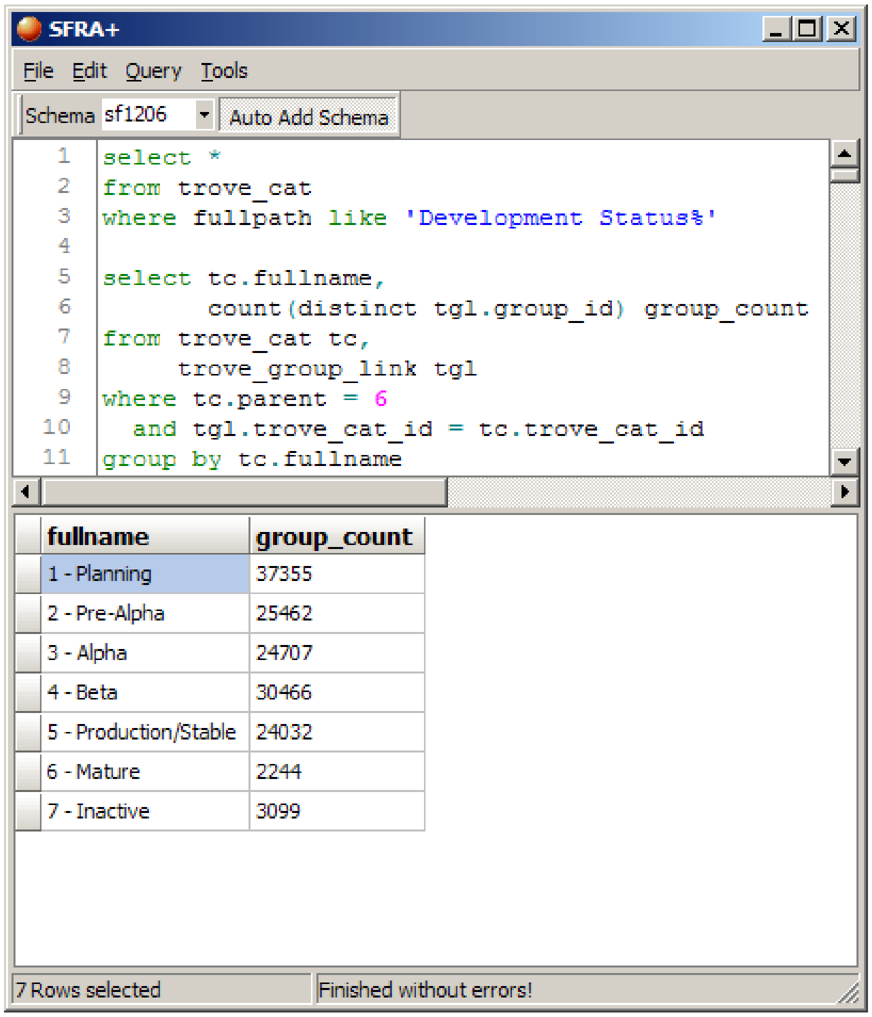
- Very limited documentation of the structures of the various schemas is provided. Despite the fact that the schema evolve, either slightly or significantly, between each monthly dump, only a single Entity-Relationship (ER) diagram is provided. This ER diagram is reported to represent the January, 2003 schema, however the ER diagram lists only 69 tables while the schema contains 139.

Given the difficulties we encountered in using the existing interface, we developed our own tool to streamline the process of understanding the structure of the database, querying the necessary tables, and analyzing the results of those queries. We have named our tool the SourceForge Research Archive Plus (SFRA$^+$).

A.2 SourceForge Research Archive Plus

The SourceForge Research Archive Plus (SFRA$^+$) is a Windows based graphical desktop application developed using Borland Delphi 5 that automates and extends the SFRA web interface. SFRA$^+$ allows a user to interact with a rich SQL editor while automatically interfacing with the SFRA and separating the user from the frustrations of its interface. SFRA$^+$ also helps the user to better understand the data available in the SFRA by automatically reconstructing the structure of the database and the relationships between the tables and displaying that information in an accessible graphical form.

The main window of SFRA$^+$ is shown in Figure A.2. There are five main regions in this window. Starting from the top of the window, they are the *main menu*, the *schema tool bar*, the *SQL editor*, the *result grid*, and the *status bar*. We discuss the last four regions, which we call the query interface, together before presenting the functions available through the menus.



```
select *
from trove_cat
where fullpath like 'Development Status%'

select tc.fullname,
       count(distinct tgl.group_id) group_count
from trove_cat tc,
     trove_group_link tgl
where tc.parent = 6
  and tgl.trove_cat_id = tc.trove_cat_id
group by tc.fullname
```

fullname	group_count
1 - Planning	37355
2 - Pre-Alpha	25462
3 - Alpha	24707
4 - Beta	30466
5 - Production/Stable	24032
6 - Mature	2244
7 - Inactive	3099

7 Rows selected

Finished without errors!

Figure A.2: The SFRA$^+$ Main Window

A.2.1 The Query Interface

The schema tool bar is located at the top of the query interface. From the drop-down menu, the user selects the default schema against which queries should be run. The toggle button to the right indicates whether the default schema should be prepended to all unprefixed table names when a query is run. Note that the default schema is only added to tables that do not have an explicit schema name provided by the user. This prevents the user from having to add the schema to tables from the default schema, but it does not preclude the user joining tables from separate schemas by including the other schema name in the query.

Below the schema tool bar, the SQL editor provides a large syntax-highlighting text area where the user can enter any valid PostgreSQL select statement formatted as the user prefers. The only formatting limitation imposed by SFRA$^+$ is that SQL statements may not contain blank lines. Blank lines are used by SFRA$^+$ to indicate separation between multiple SQL statements. This allows the user to simultaneously view and edit multiple statements in the editor. The contents of the editor are maintained between invocations of the program so that the user will not lose their work by inadvertently closing the program. The contents of the editor may also be saved to or loaded from text files with a `.sql` extension.

When a query is run using SFRA$^+$, the current statement in the SQL editor (the one in which the caret[1] is located) is parsed, prepared, formatted, and submitted to the SFRA web form. The statement preparation includes adding SQL commands to replace the delimiter characters in the text fields so that the result set can be automatically parsed.

As queries are executed, progress messages are displayed in the status bar. These messages indicate the success or failure of each step from statement parsing to result retrieval. The status bar also displays the total number of records retrieved and the total number displayed when a query is successfully completed. These numbers differ for large result sets. While all records are retrieved from the SFRA regardless of the size of the result set, a maximum of 4098 records are displayed in the result grid in order to improve performance.

After the results of a successful query are retrieved from the SFRA web interface, they are displayed in the result grid. The names of the individual fields are placed in the header row, and the row widths and column heights are adjusted so

[1]What we refer to here is the flashing vertical line in the editor that indicates the insertion point for the next typed character. This is also often referred to as a cursor.

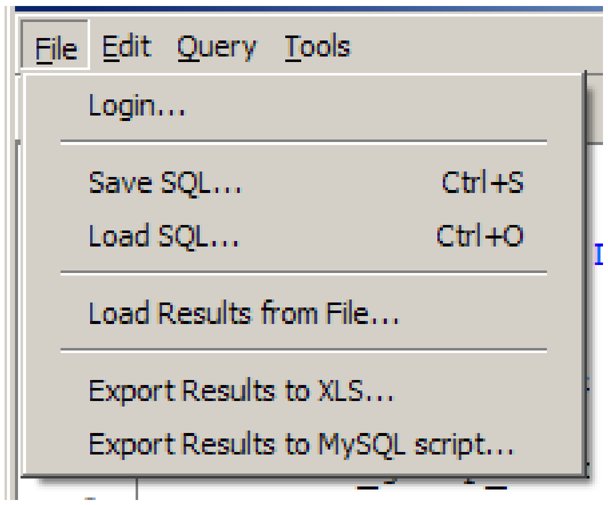


Figure A.3: The SFRA$^+$ File Menu

that all the field values are visible. The rows and columns can also be resized and reordered manually if necessary. In addition, field values may be copied or edited.

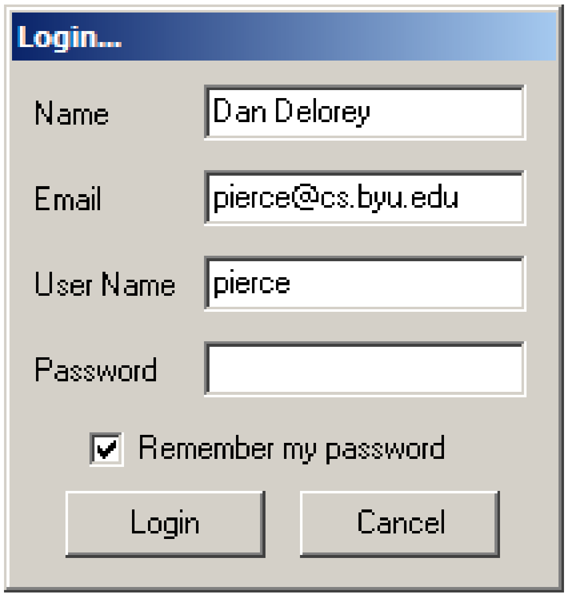
A.2.2 The Main Menu Functions

The main menu consists of four menus: 1) the File menu, 2) the Edit menu, 3) the Query menu, and 4) the Tools menu. The Edit menu contains only an Undo menu item and a Redo menu item which affect the SQL editor. We do not expect that these functions require further explanation. We will explain the functions provided by the other three menus in turn.

The File Menu

The File menu shown in Figure A.3 provides the functions that allow the user to manage SFRA$^+$ related files on their local machine. There are six items in the File menu grouped into four functional units.

The first item in the File menu raises the login window shown in Figure A.4 which allows the user to enter their SFRA authentication tokens. This login window is also displayed the first time the program is invoked. Academic researchers can obtain


Login...

Name: Dan Delorey

Email: pierce@cs.byu.edu

User Name: pierce

Password:

[✓] Remember my password

Figure A.4: The SFRA$^+$ Login Window

access through the University of Notre Dame group. Without valid credentials users will not be able to access the SFRA or the SFRA$^+$. By checking the "Remember my password" check box, users can avoid having to manually authenticate each time they access SFRA$^+$.

The second and third items in the File menu allow the user to save and load the contents of the SQL editor. The SQL statements are saved in a plain text file with a `.sql` extension. Any valid text file with this extension may be loaded regardless of its contents.

The fourth item in the File menu allows the user to load an existing result file into the result grid. Result files are are colon delimited text files with a `.dat` extension. A new result file is created by SFRA$^+$ each time a query is successfully executed. These result files are stored in the application directory and are named `results.dat`. To avoid the long delays associated with running complex queries or retrieving large result sets, users may find it convenient to archive certain result sets and later load them for further analysis without reexecuting the queries. It is important to note that result sets loaded using this menu item will not display field names in the header row since this information is not returned by the SFRA in the result files. There is a similar function in the Query menu that does allow result sets to be loaded with header information which will be explained in Section A.2.2

The fifth and sixth menu items in the File menu allow the user to export the result set to forms that allow further analysis of the data. SFRA$^+$ can export results sets into Microsoft Excel workbooks or into MySQL import scripts. When exporting to Excel, the field names are written as the first row of the worksheet. Note that because of an internal limit in Excel, result sets with more than 65,536 records or more than 256 fields may not be exported to Excel. When exporting to MySQL, SFRA$^+$ generates an import script that will create a table with the appropriate column definitions for the current result set, adds an insert statement for each record in the result set, and includes any post-processing commands such as constraint or index creation statements. The user is prompted to supply a name for the table that will hold the result set. The user is also given the opportunity to view and edit the table creation and post-processing statements before they are written to the import script.

The Query Menu

The Query menu shown in Figure A.5 provides the query functions. These functions involve the parsing and formatting of queries to prepare them for the SFRA web interface as well as the retrieval and display of results. There are three items in the Query menu.

The first item in the Query menu causes the current query to be prepared and formatted but not submitted to the SFRA. Instead the formatted query is displayed to the user in a three line output. Each line contains the exact text that would be submitted to the SFRA web form if the query were run. This functionality can help

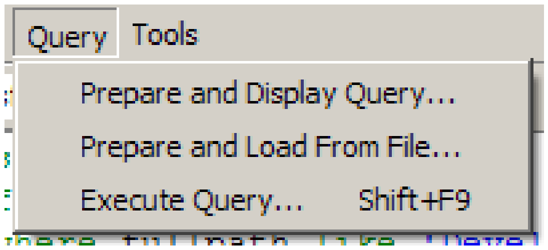


Figure A.5: The SFRA$^+$ Query Menu

the user understand what SFRA$^+$ is doing as it parses and modifies a query which can be particularly useful if unexpected results are being returned.

The second item in the Query menu is similar to the "Load Results from File..." item of the File menu except that it allows the user to identify a query that should be parsed to determine the field names and field types for the result set before loading it from the file. Ideally, the query used to load the result set would be the same query used to retrieve it. However, this function may be used to re-label a result set or to reduce or increase the number of columns in a result set. The number of fields and the names of the fields displayed in the result grid match those of the query being processed regardless of the number of fields in the result file. If the number of fields in the query (n) is less than the number of fields in the result file (m), then only the first n fields of the result set are displayed. If the number of fields in the query is more than the number of fields in the result file, then the last $n - m$ fields are displayed in the grid but are empty for all records.

The third item in the Query menu provides the main functionality of SFRA$^+$ by parsing the current query, submitting the prepared and formatted query to SFRA web interface, retrieving the result set, and displaying the results in the grid. This functionality is explained in detail in Section A.2.1

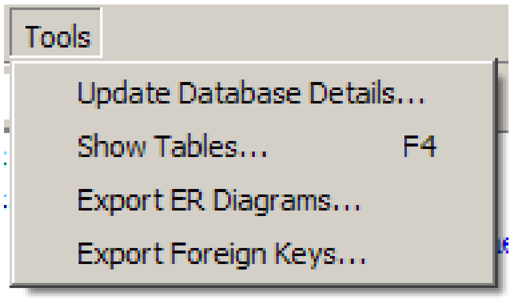


Figure A.6: The SFRA$^+$ Tools Menu

The Tools Menu

The Tools menu shown in Figure A.6 provides the functions that aid the user in understanding the available data. There are four items in the Tools menu.

The first item in the Tools menu retrieves data from the SFRA which is used to reverse engineer the structure of the SFRA database. The SFRA data are stored in a PostgreSQL database. PostgreSQL databases store the metadata used by SFRA$^+$ in the `pg_namespace`, `pg_class`, `pg_constraint`, `pg_attribute`, and `pg_type` tables. To improve performance and limit unnecessary transfers between the SFRA and client machines, SFRA$^+$ retrieves these data and stores them in an XML file on the local machine so they can be loaded each time the program is run. These data are used by SFRA$^+$ to determine field names and types for query results, to display table details, to reconstruct foreign key relationships, and to draw schema diagrams.

The second item in the Tools menu uses the data retrieved by the first item to display table details for the current default schema. The names of each table and the fields within each table are displayed in a collapsible tree structure providing the user quick reference when writing queries. This view is not intended to show the relationships between tables but instead corresponds directly to the schema details available through the SFRA web site.

The third item in the Tools menu provides the functionality to create ER diagrams for the schemas of the SFRA database automatically from the data in the locally cached XML file. The ER diagrams can be generated for multiple schemas at once, for a complete single schema, or for certain tables within a schema. If one or more complete schemas are selected, all tables and relationships are included in the diagram. If a partial ER diagram is chosen, the user may select the tables from the schema to be included in the diagram. In addition, for a partial ER diagram, the user may choose to include tables that are parents in a foreign key relationship with the tables selected, tables that are children in a foreign key relationship with the table selected, both, or neither. The complete ER diagrams are useful for observing changes in the schemas over time. The partial ER diagrams are helpful when trying to identify data related to a particular table. For all but one of the schemas in the SFRA database, the foreign key constraints are not enforced in the database. This prevents SFRA$^+$ from directly reconstructing these relationships from the metadata stored in the database. As a workaround, we have developed a set of heuristics which guess foreign key constraints based on field name patterns, table names, and data types. Figure A.7 shows an example of a partial ER diagram generated using SFRA$^+$.

The fourth item in the Tools menu exports the complete list of foreign key constraints detected for each schema in the locally cached XML file. A separate file is created for each schema. Each foreign key relationship is written on a single line with four tab delimited values: 1) the child table, 2) the foreign key field, 3) the parent table, and 4) the primary key field. These files are superior to the ER diagrams for quickly determining all the foreign key relationships for given table. They are also convenient for manually validating the heuristics being used to determine foreign key relationships.

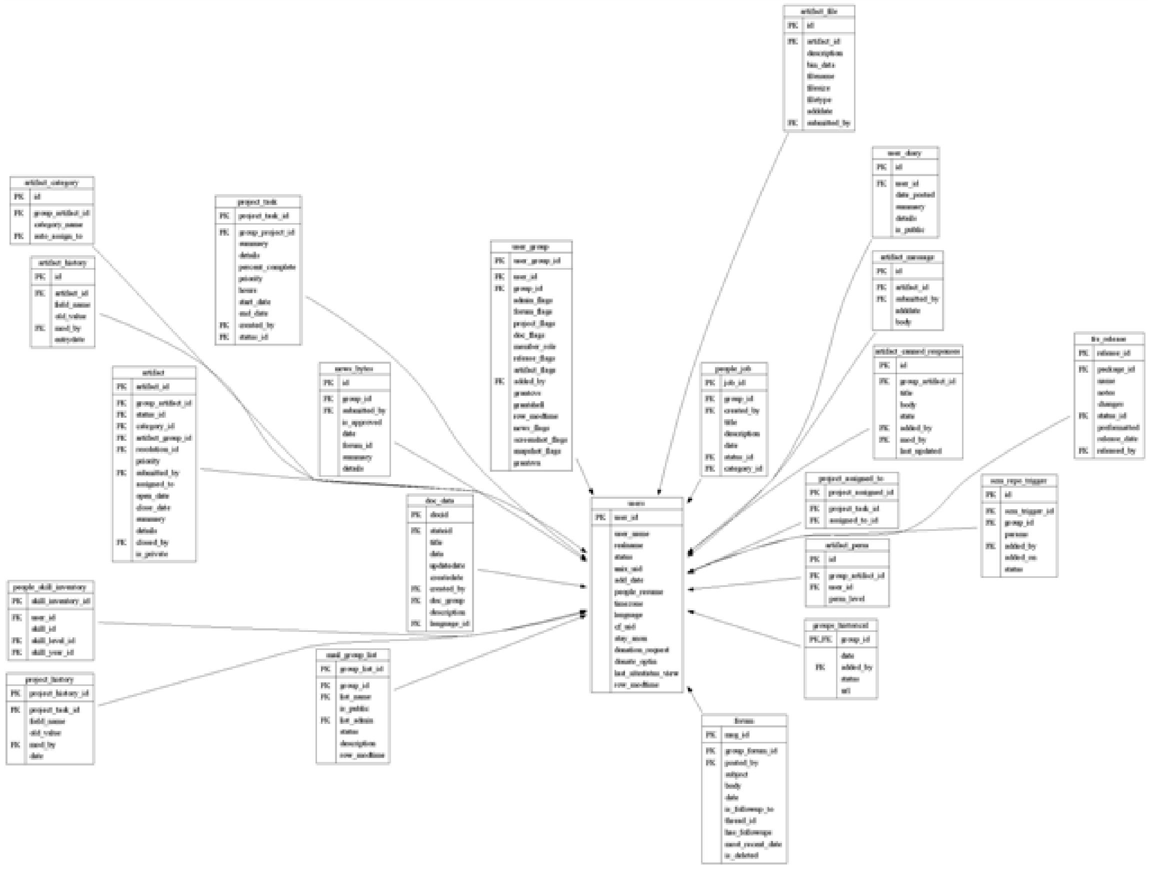


Figure A.7: Partial ER Diagram Showing the Users Table and Foreign Key Children

Appendix B

Documentation for cvs2mysql

B.1 Overview

The Concurrent Versioning System (CVS) is among the most widely used version management systems. Among the Open Source projects hosted on SourceForge.net 92.5% (155,293 projects) use CVS while only 4.4% (7432 projects) use its nearest competitor, Subversion. Extensive details about the inner workings of CVS can be found in the user's manual, commonly referred to as "The Cederqvist" in honor of the author of the program and the documentation [5].

As CVS archives versions of the files it manages, it maintains a history of changes to those files. These file histories may be retrieved from the CVS server in Revision Control System (RCS) [31] log format. These log files are useful for understanding the changes that have been made to a single file over its lifetime, but they are insufficient for more general project level analyses. In order to fully exploit the relationships between the data stored in the history logs of the various files managed by a CVS repository, it is necessary to transfer those data to a relational database. To facilitate this data transfer, we developed `cvs2mysql`.

B.2 cvs2mysql

`cvs2mysql` is a cross-platform application developed in Python that takes as input a CVS repository and produces an SQL script for importing the data into a MySQL database. We have extensively validated `cvs2mysql` by running it against more than 16,000 CVS repositories. In processing these repositories, we ran `cvs2mysql` in Windows XP, Cygwin, Red Hat Enterprise Linux, and Mac OSX environments to assure that the behavior was consistent across all these systems.

`cvs2mysql` comprises six separate script files. The main script is `cvs2mysql.py`. This script manages the user input, responds to command line options, prints a help message when necessary, and prepares the input for the other scripts. Three of the scripts, `project.py`, `file.py`, and `revision.py`, define classes which correspond to the three logical units encountered while processing a CVS log. The `sqlscript.py` script defines the class that generates the SQL scripts from the gathered data. The file `patterns.py` holds the regular expression patterns used in parsing the log files.

In order to use `cvs2mysql`, a Python distribution and a CVS client must be installed on the local machine. The python scripts include the necessary header to make them executable from the command line on typical system configurations under Windows, Cygwin, Linux and Mac OSX. They may also be run from within a Python interpreter if desired. The invocations of CVS made by `cvs2mysql` assume that the CVS client binary is in the executable path. If this is not the case, the `path_to_cvs` variable in the `patterns.py` script must be changed to indicate the correct path. Note that because of the inner workings of the Python `os.popen` command, this path may not contain any spaces on Windows systems.

B.2.1 Invoking `cvs2mysql`

As a command line application, `cvs2mysql` receives input from the user in the form of command line flags and command line arguments. In addition to the standard `-h` and `--help` flags which cause `cvs2mysql` to print usage instructions, there are three types of flags recognized by `cvs2mysql`: *input options*, *output location*, and *processing options*. The first two are mandatory while the third is optional.

Input Options

We designed `cvs2mysql` to process any CVS repository given the CVS root. In addition, because our primary use for `cvs2mysql` was to gather data from the CVS repositories of SourceForge projects, we extended `cvs2mysql` to process SourceForge project repositories given only the project's Unix name from SourceForge.net and to process multiple SourceForge projects given a text file with a single project Unix name per line. These input options are controlled by command line flags which can be specified in either short or long forms following typical conventions. The input options are mutually exclusive, meaning that exactly one of the input options, along with a valid argument, must be specified for each invocation of `cvs2mysql`.

- `-r REPOSITORY` or `--repository=REPOSITORY` to process a single repository
- `-p PROJECT_NAME` or `--project=PROJECT_NAME` to process a single SourceForge project
- `-f FILE_NAME` or `--file=FILE_NAME` to process multiple SourceForge projects from an input file

The second option is handled by the `cvs2mysql.py` script as an extension of the first option and the third option is handled as an extension of the second. That is, when processing a SourceForge project, the CVS root for the SourceForge CVS repository is generated programmatically from the project name and passed to the function that handles the `-r`. Similarly, when a file of SourceForge project names is processed, each is read from the file and passed to the function that handles the `-p`. The function that receives the CVS root creates a `Project` object for the repository and calls the `process` function of that object.

Output Location

As it processes a CVS repository, `cvs2mysql` creates both directories and files including a local copy of the contents of the repository (called a *sandbox*), a log file, and an SQL script. The user must specify the location in which these should be created using the `-o` or `--output` command line flags giving as an argument a directory using either an absolute or a relative path.

Processing Options

There are two types of processing options recognized by `cvs2mysql`. The first type alters the steps performed when processing a CVS repository. There are five main steps `cvs2mysql` performs when processing a CVS repository which are explained in detail below. Four of these steps may be skipped using the following command line flags. Note, however, that there are certain additional requirements involved with skipping the first or second step.

- `--no-checkout` — This option prevents `cvs2mysql` from retrieving a local copy of the repository. If this option is used, `cvs2mysql` verifies that a source directory for the repository is in the correct location inside the output directory. `cvs2mysql` aborts with an error if this option is used and no source directory is provided.
- `--no-log` — This option prevents `cvs2mysql` from retrieving a log file for the repository. If this option is used, `cvs2mysql` verifies that a log file for the repository is in the correct location inside the output directory. `cvs2mysql` aborts with an error if this option is used and no log file is provided.
- `--no-script` — This option prevents `cvs2mysql` from generating an SQL script.

- `--keep-files` — This option prevents `cvs2mysql` from deleting the sandbox and log file used during processing.

All combinations of these options are allowed by `cvs2mysql`; however, some combinations, such as the one that includes all four options, are less useful than others.

The second type of processing option allows the user to provide a value for the project name `cvs2mysql` uses to tag all the file records gathered from a CVS repository. By default, when processing a SourceForge repository, `cvs2mysql` tags all the files with the project's SourceForge Unix name. When processing any other CVS repository, the files are not tagged. Using the `-t` or `--tag` option, the user can specify a tag that is be used in place of these default values.

B.2.2 CVS Repository Processing

As mentioned above, `cvs2mysql` performs five main steps when processing a CVS repository: 1) retrieve a local copy of the repository from the server, 2) retrieve a log of the repository from the server, 3) parse the log file and extract the relevant data, 4) generate an SQL script for the repository, and 5) delete all the files created during processing except for the SQL script.

Retrieving the Repository Contents

A local copy of the CVS repository contents is retrieved from the server using a CVS `checkout` command. The `checkout` command is run with the `-r 1.1` option which causes revision 1.1 of each of the files to be returned. CVS revision numbers are defined as follows: "A revision number always has even number of period-separated decimal integers. By default revision 1.1 is the first revision of a file." [5] While it is possible to explicitly change the initial revision number or any subsequent revision number, 1.1 is the most common initial revision number. Files for which there is no revision 1.1 or for which revision 1.1 is not the initial revision are handled separately during log parsing.

We retrieve the initial revision of each file to simplify the process of determining the initial size of each file. While CVS does track the lines added to and removed from a file for all subsequent revisions, it does not record either the initial size of the file when it was added to the repository or the number of lines added to the file by the initial revision. While others have tried calculating the initial file size from the current size of the file by adding to the current size all the lines removed by previous revisions and subtracting all the lines added by previous revisions, we have found that this method is computationally expensive and likely to be inaccurate when files have been branched or removed from the repository at some point in the past.

If the CVS client fails to retrieve a complete local copy of the repository, `cvs2mysql` exits with an error and print a message alerting the user that processing for the repository has failed. Unless the `--keep-files` option has been used, `cvs2mysql` deletes any files that were created before exiting.

Retrieving the History Log

Our goal when retrieving the history log is to obtain a single log file with the RCS logs for every file in the repository. This is advantageous both because it simplifies the parsing of the logs and because it reduces the traffic between the client and the server. However, some repositories are so large or have such long histories that communications time out before the server can prepare and return the complete log file.

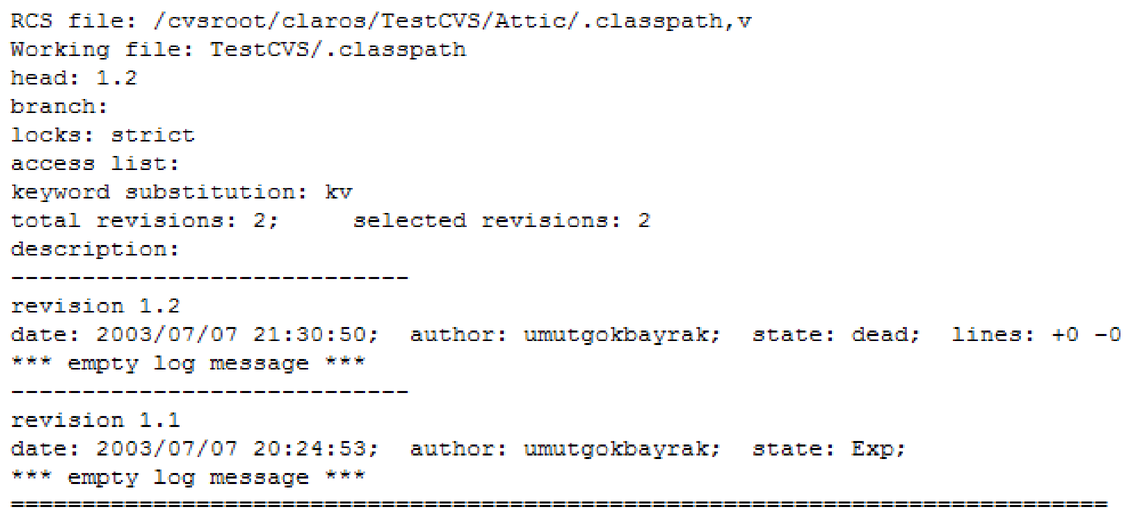
To allow the processing of large repositories and still retrieve only a single log file, `cvs2mysql` has a recursive error recovery mechanism. A CVS `log` command is then run on the top level directory of the repository. If the command fails, a cvs `log` command is run for each directory and file within the top level directory. This recursive processing continues until either the entire repository has been logged or logging fails for an individual file.

Logging begins with an empty log file on the local machine. The results of each `log` command are redirected to a temporary log file. If the command succeeds, the contents of the temporary log file are appended to the main log file. If the command fails, the temporary log file is overwritten by the subsequent `log` command. When the entire directory has been logged the main log file contains the entire log history for every file that has ever existed in the repository.

Parsing the History Log

As noted in [11], there is no published grammar for RCS logs. This makes parsing them particularly difficult. However, some discussion of RCS logs is provided in [24]. After reviewing this material, comparing the source code for both CVSAnalY [29] and softChange [11] (two other applications that parse CVS logs), performing more than a dozen detailed manual verifications of CVS logs parsed by `cvs2mysql`, and running `cvs2mysql` on more than 16,000 CVS repositories without errors, we are confident in the accuracy of our parsing algorithm.

An example CVS log is shown in Figure B.1. Clearly this simple log does not illustrate all the special cases that can occur. This example is meant only to show the common case to identify the data that can be collected from CVS logs. The log shows the complete history for one file from the Claros In Touch project hosted on SourceForge. This file, named `.classpath`, has two revisions. Both revisions were made by the same author. The first revision was made when the file was added to the repository and the second was made when it was removed from the repository. Neither revision has a descriptive message beyond the automatically generated "`*** empty log message ***`" and the file was never branched or tagged. All things considered,



```
RCS file: /cvsroot/claros/TestCVS/Attic/.classpath,v
Working file: TestCVS/.classpath
head: 1.2
branch:
locks: strict
access list:
keyword substitution: kv
total revisions: 2;     selected revisions: 2
description:
----------------------------
revision 1.2
date: 2003/07/07 21:30:50;  author: umutgokbayrak;  state: dead;  lines: +0 -0
*** empty log message ***
----------------------------
revision 1.1
date: 2003/07/07 20:24:53;  author: umutgokbayrak;  state: Exp;
*** empty log message ***
=============================================================================
```

Figure B.1: An Example of a CVS Log for a Single File

it is quite a boring log, but it nicely demonstrates the common case without taking up more space than is necessary.

The first line of the log provides the absolute path of the file on the CVS server. Notice that the path includes a directory named `Attic`. The Attic is a special directory within CVS repositories where removed files are kept. When a file is "removed" from a CVS repository using the CVS `remove` command, it is not actually deleted. Instead, it is move to the Attic. All of the history for the file is maintained and the file can be recovered either by using an explicit `checkout` or `update` command with the file name or by adding a new file to the repository in the same location with the same name as the original file.

The second line of the log gives the relative path of the file within a sandbox. This is the path to the file beginning in the directory where the CVS `checkout` command was run.

The third line of the log lists the most recent revision number. This revision is referred to as the head of the file and is the revision that is returned on a `checkout` command if no `-r` option is used.

The fourth line of the log lists the default branch for the file. An empty branch value indicates the file is committed on the main branch. If a default branch has been set for the file, this value has a string similar to a revision number but with an odd number of period-separated decimal integers for each branch. Branch strings are the common prefix of all the revision numbers on the branch.

The fifth and sixth lines of the log are used to control access to the file. The `locks` line lists revisions that have been locked along with the user name that locked the revision. Locks prevent other users from making changes to the file on the locked revision's branch. The `strict` directive at the end of the line guarantees that locks are strictly enforced. The `access list` line can be used to allow only certain users to edit the file by explicitly listing their user names. An empty `access list` allows any user with access to the repository to edit the file.

The seventh line of the log controls the process used when versioning the file. CVS automatically updates certain header comments in text files using keyword substitutions and standardizes line endings as is manages revisions. There are six available modes [5]. Of these, the most interesting for our purposes is the `-kb` mode which prevents CVS from replacing keyword values or standardizing line breaks. This mode is used to prevent CVS from corrupting binary files which may contain bit patterns that appear to be keywords or non-standard line endings when interpreted as ASCII text.

The eighth line of the log lists the total number of revisions that have been made to the file and the number of revisions recorded in the current log. These numbers may differ if the data-range option is used when running the CVS `log` command. This is never the case when using `cvs2mysql`.

The ninth line of the log indicates the beginning of the description of the file provided by the user when adding the file to the CVS repository. The description is free-form text and may span multiple lines. The end of the description is marked by the beginning of the revisions sections indicated by the pattern of '-' characters on line ten.

Beginning on line ten of the log, the revisions section contains the details of each revision of the file. Individual revision logs begin after the pattern observed on line ten of the log and end either at the beginning of the next revision log or at the end-of-file-log pattern of '=' characters seen on the last line of Figure B.1.

The first line of a revision log lists the revision number. As discussed above, this number is always an even number of period-separated decimal integers. CVS automatically increments the last decimal integer in the revision number when versioning files. Revision numbers may be set manually by the user, but they must still follow the same pattern. Revisions with the same prefix are considered to be on the same branch by CVS.

The second line of a revision log contains the date and time of the revision, the author of the revision, the state of the file after the revision, and the number of lines added to and deleted from the file by the revision. The format of the date and time is controlled by the CVS repository configuration files. The default is UTC format. The author value is the user name of the CVS user that committed the changes to the repository. The range of allowed state values is also controlled by the CVS repository configuration as discussed in [5]. For our purposes, the most interesting state is `dead` which indicates that the file was removed from the repository by the revision. The lines added and deleted values are always integer values.

The third line of a revision log begins the free-form message provided by the user that committed the changes. The message continues until the end of the revision log. Unlike the file description, the revision message is required by CVS and occupies at least one line in the log. The revision message may cover multiple lines.

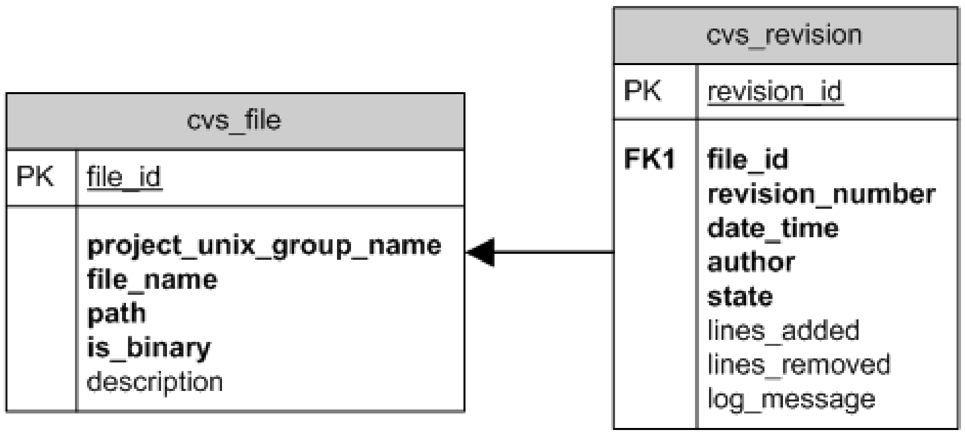


Figure B.2: Database Schema Used by `cvs2mysql`

Writing the SQL Script

The database schema we developed to store the data extracted from CVS logs is shown in Figure B.2. The structure of our schema is remarkably simpler than those produced by other tools [11, 29], yet it provides the same data. Our schema is designed to be easy to query and easy to join to other data sources such as [21] and [14].

To avoid redundancy, only those data from the CVS log that cannot be recalculated are written to the SQL script by `cvs2mysql`. For example, `RCS file`, `Working file`, and `state` from the most recent revision represent a set from which one value can be calculated given the other two. In this case, we store `Working file` (as `file_name` and `path` in the `cvs_file` table) and `state` (as `state` in the `cvs_revision` table) because these values are needed more often and are less amenable to recalculation. Other values from the CVS log that have been excluded to avoid redundancy are `head`, `total revisions`, and `selected revisions`.

In addition to redundant data, there are data in the CVS log that we discard because they are only partially available. For example, `branch` is time sensitive. It represents the default branch for a file at the time the CVS `log` command was run. However, historical data showing the changes in this value are not maintained by CVS. Thus, the data is incomplete and any analyses involving this value would be highly dependant on the point in time when `cvs2mysql` was used to process the repository. Other values from the CVS log that have been excluded due to partial availability are `locks` and `access list`.

All values not excluded due to redundancy or partial availability are included in the schema shown in Figure B.2.

The file specific data, which are `Working file` (which is broken into a file name and a relative path with in the repository), `keyword substitution` (which is encoded as a binary value indicating whether the file is binary), and the description, are stored in the `cvs_file` table. In addition, the `cvs_file` table has a column named `project_unix_group_name` which can be used to separate files from different projects when multiple SQL scripts have been imported into the same database. If the `-t` or `--tag` options were used when processing the repository, this field contains the user-specified flag. If these options were not used, for SourceForge projects processed using `cvs2mysql` this value is the "Project UNIX name" shown on the project's SourceForge web page. Using this value, the `cvs_file` table can be joined to tables from [21] and [14] as well as any other data source using SourceForge data which has tables indexed by a project's unix name.

The revision specific data, which are `revision`, `date`, `author`, `state`, `lines` (separated into lines added and lines removed), and the log message, are stored in the `cvs_revision` table. The `cvs_revision` table also has a `file_id` field which is a foreign key to the `cvs_file` table. Also, for SourceForge projects, the `author` field is the SourceForge user's "Login Name" shown on their Developer Profile. This value can also be used to join to tables in [21] and [14].

The SQL script produced by `cvs2mysql` when processing a CVS repository contains only `insert` statements. We have separated the table creation from the table population to simplify the process of importing data from multiple repositories into the same database. We have included functions in the `sqlscript.py` to produce the table creation script and an index creation script that creates the most commonly needed indices for the tables.

By taking advantage of the auto-increment primary keys in MySQL, the SQL scripts produced by `cvs2mysql` can be used to import their data into a properly structured database regardless of the contents of the `cvs_file` and `cvs_revision` tables prior to the insertion. File records are assigned new `file_id` values and revision records are linked to the correct file record. The only caution when importing data from multiple repositories is that if scripts representing distinct repositories have identical `project_unix_group_name` values, this field is no longer effective in separating the files from the two repositories. If this is a concern, the `-t` or `-tag` options should be used to explicitly tag the files with differentiable `project_unix_group_name` values.

Deleting Files

To make `cvs2mysql` truly cross-platform compatible, it was necessary to avoid all system specific API calls. This required us to use the `shutil.rmtree` function to remove the temporary files rather than using faster system-specific functions. The function recursively iterates through the directories removing the contents of the directory before removing the directory itself. Removal of a directory fails if the directory is not empty or if `rmtree` is denied access to a file or directory it is trying to delete. In such cases, the temporary files may be removed manually once `cvs2mysql` has finished processing.

References

[1] Victor R. Basili. The role of experimentation in software engineering: Past, current, and future. In *Proceedings of the 18th International Conference on Software Engineering*, pages 442–449, March 25-29, 1996 1996.

[2] Victor R. Basili, Forrest Shull, and Filippo Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, July/August 1999.

[3] Fredrick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering.* Addison Wesley, Boston, MA, 1995.

[4] Ruven E. Brooks. Studying programmer behavior experimentally: The problems of proper methodology. *Communications of the ACM*, 23(4):207–213, April 1980.

[5] Per Cederqvist. Version management with cvs. http://ximbiot.com/cvs/manual, 2007.

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

[7] Martha E. Crosby and Jan Stelovsky. How do we read algorithms? a case study. *IEEE Computer*, 23(1):24–35, January 1990.

[8] Stephan Diehl, Ahmed E. Hassan, and Richard C. Holt. Report on msr 2005: international workshop on mining software repositories. *SIGSOFT Softw. Eng. Notes*, 30(5):1–3, 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1095430.1095433.

[9] Stephan Diehl, Harald Gall, Martin Pinzger, and Ahmed E. Hassan. MSR 2006: the 3rd international workshop on mining software repositories. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 1021–1021, New York, NY, USA, 2006. ACM Press.

[10] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, July 2000.

[11] Daniel 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 .

[12] Ahmed E. Hassan, Richard C. Holt, and Audris Mockus. Report on msr 2004: International workshop on mining software repositories. *SIGSOFT Softw. Eng. Notes*, 30(1):4, 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1039174.1039188.

[13] John D. Holden. Hawthorne effects and research into professional practice. *Journal of Evaluation in Clinical Practice*, 7(1):65–70, 2001. doi: 10.1046/j.1365-2753.2001.00280.x. URL `http://www.blackwell-synergy.com/doi/abs/10.1046/j.1365-2753.2001.00280.x`.

[14] James Howison, Megan Conklin, and Kevin 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 .

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

[16] Magne Jorgensen. An empirical study of software maintenance tasks. *Journal of Software Maintenance: Research and Practice*, 7(1):27–48, Jan.-Feb 1995.

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

[18] Timothy Lethbridge, Susan Elliott Sim, and Janice Singer. Studying software engineers: Data collection techniques for software field studies. *Empirical Software Engineering*, 10(3):311–341, July 2005.

[19] Gernot Armin Liebchen and Martin Shepperd. Software productivity analysis of a large data set and issues of confidentiality and data quality. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005)*, 2005.

[20] R. Murray Linday and A. S. C. Ehrenberg. The design of replicated studies. *The American Statistician*, 47(3):217–228, August 1993.

[21] Greg Madey. Sourceforge research data archive. http://www.nd.edu/˜
oss/ Data/ data.html, 2005.

[22] Katrina D. Maxwell and Pekka Forselius. Benchmarking software development productivity. *IEEE Software*, pages 80–88, January 2000.

[23] Katrina D. Maxwell, Luk Van Wassenhove, and Soumitra Dutta. Software development productivity of european space, military, and industrial applications. *IEEE Transactions on Software Engineering*, 22(10):706–718, October 1996.

[24] Brian O'Donavan. *RCS Handbook*. 1992.

[25] Rachel Or-Bach and Ilana Lavy. Cognitive activities of abstraction in object orientation: An empirical study. *ACM SIGCSE Bulletin*, 36(2):82–86, June 2004.

[26] Lutz Prechelt. An empirical comparison of seven programming languages. *IEEE Computer*, 33(10):23–29, October 2000.

[27] Lutz Prechelt, Michael Philippsen, and Walter Tichy. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering*, 28(6):595–606, June 2002.

[28] Rahul Premraj, Martin Shepperd, Barbara Kitchenham, and Pekka Forselius. An empirical analysis of software productivity over time. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS 2005)*, 2005.

[29] Gregorio Robles, Stefan Koch, and Jesus 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 .

[30] W. M. Taliaffero. Modularity. the key to system growth potential. *IEEE Software*, 1(3):245–257, July 1971.

[31] Walter F. Tichy. RCS — a system for version control. *Software — Practice and Experience*, 15(7):637–654, 1985. URL `citeseer.ist.psu.edu/tichy85rcs.html`.

[32] James Tomayko. A historian's view of software engineering. In *Proceedings of the 13th Conference on Software Engineering Education and Training*, pages 103–110, March 6-8, 2000 .

[33] Claes Wholin, Per Runeson, Martin Host, Magnus Ohlsson, Bjorn Regnell, and Anders Wesslen. *Experimentation in Software Engineering: An Introduction.* Kluwer Academic Publishers, Boston, MA, 2000. ISBN 0-7923-8682-5.

[34] R. W. Wolverton. The cost of developing large-scale software. *IEEE Transactions on Computers*, C-23(6):615–636, June 1974.

[35] Marvin Zelkowitz and Dolores Wallace. Experimental models for validating technology, May 1998.