Empirical Software Engineering, 9, 197–210, 2004.

Open-Source Change Logs

KAI CHEN kai.chen@vanderbilt.edu
Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37235

STEPHEN R. SCHACH srs@vuse.vanderbilt.edu
Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37235

LIGUO YU liguo.yu@vanderbilt.edu
Department of Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN 37235

JEFF OFFUTT ofut@ise.gmu.edu
Department of Information and Software Engineering, George Mason University, Fairfax, VA 22030

GILLIAN Z. HELLER gheller@efs.mq.edu.au
Department of Statistics, Macquarie University, Sydney, NSW 2109, Australia

Editor: Audris Mockus

Abstract. A recent editorial in *Empirical Software Engineering* suggested that open-source software projects offer a great deal of data that can be used for experimentation. These data not only include source code, but also artifacts such as defect reports and update logs. A common type of update log that experimenters may wish to investigate is the `ChangeLog`, which lists changes and the reasons for which they were made. `ChangeLog` files are created to support the development of software rather than for the needs of researchers, so questions need to be asked about the limitations of using them to support research. This paper presents evidence that the `ChangeLog` files provided at three open-source web sites were incomplete. We examined at least three `ChangeLog` files for each of three different open-source software products, namely, GNUJSP, GCC-g++, and Jikes. We developed a method for counting changes that ensures that, as far as possible, each individual `ChangeLog` entry is treated as a single change. For each `ChangeLog` file, we compared the actual changes in the source code to the entries in the `ChangeLog` file and discovered significant omissions. For example, using our change-counting method, only 35 of the 93 changes in version 1.11 of Jikes appear in the `ChangeLog` file—that is, over 62% of the changes were not recorded there. The percentage of omissions we found ranged from 3.7 to 78.6%. These are significant omissions that should be taken into account when using `ChangeLog` files for research. Before using `ChangeLog` files as a basis for research into the development and maintenance of open-source software, experimenters should carefully check for omissions and inaccuracies.

Keywords: Open-source software, change log, GNUJSP, GCC-g++, Jikes.

1. Introduction

The success of open-source software is having a growing impact on the software industry. Many open-source software products, such as Linux (SourceForge, 2002) and the Apache Web server (Welcome, 2002), now have a significant share of their

respective software markets. Leading software companies, including IBM (2002) and Sun (JavaOne, 2002), have begun to embrace the open-source model, and are actively taking part in the development of open-source software products.

Empirical software engineers tend to be enthusiastic about open-source software. It has traditionally been hard for software engineering researchers to obtain source code to use as experimental data. Many software companies believe that the company will lose some of its competitive advantage if it were to divulge information about the software development process used by that company, let alone the source code itself. Software companies use intellectual property laws to keep their source code secret; only the binary code is typically sold in the marketplace. Software companies provide information publicly about their software almost exclusively for the purpose of boosting sales, whereas the data needed for software engineering research are kept strictly private. And even when companies are willing to give data to researchers, the data provided are almost always out of date. In addition, companies often impose restrictive stipulations on what can or cannot be done with the data. Also, some software companies are concerned that letting outsiders know that they find and fix software faults will cause customers to have less confidence in their products.

The appearance of open-source software is changing all of these factors. A recent editorial in *Empirical Software Engineering* stated (Harrison, 2001):

> As empirical software engineers, we should embrace this development [open-source software]. Suddenly one of the greatest obstacles in the way of empirical software engineering has been cleared! Not only is source code available, but also defect reports, update logs, etc. For a change, we can now focus on the analysis rather than the data collection.

As empirical software engineers, we found this suggestion intriguing and hope to use open-source software artifacts in our research. However, it has to be noted that open source software artifacts are not created for the benefits of researchers, but to support the software development process. Thus we started by asking the question: ''How useful are the data for research purposes?'' Of course, this is a multi-faceted question that could be answered in terms of any of the available artifacts. One of the easiest types of artifacts to study is that of ''update logs,'' thus we chose to start there. Update logs happen to come in several forms, most commonly CVS log entries and change log files. Programmers tend to deposit files into CVS after making large numbers of changes (dozens and sometimes even hundreds) and comments are often rather sparse. To get a complete picture of the changes, an investigator would need to undertake a detailed study of the differences between the two versions of the source code. Change log files, on the other hand, are typically used to document changes at a more detailed level, including reasons why the changes were made. Thus, we believed that investigators are likely to focus on change-log files as being easier to process, particularly in terms of obtaining semantic information, and we decided to begin our investigation there.

The `ChangeLog` is a file in which software engineers can record detailed information about changes to the software. Not every open-source project maintains a `ChangeLog` file, but it is available at some open-source software web sites. However, we were concerned about the accuracy of `ChangeLog` records. Specifically, we wished to determine whether the records in a `ChangeLog` file are complete and correct enough to be used as a basis for research into open-source software.

To evaluate this question, we decided to study specific open-source software products. We selected the `ChangeLog` files of three open-source software products, GNUJSP (2002), GCC-g++ (GCC Home Page, 2002), and Jikes (IBM, 2002). For each product, we examined the `ChangeLog` files appertaining to at least three versions, and checked the actual changes in the source code itself against the entries in each `ChangeLog` file. Section 2 discusses change logs. There are many ways to count and evaluate changes, so the method we used is described in detail in Section 3. Sections 4, 5, and 6, respectively, describe the `ChangeLog` files for GNUJSP, GCC-g++, and Jikes that we analyzed. Our conclusions appear in Section 7.

2. Change Logs

A change log is a file in which software engineers can record changes between consecutive versions of a software product. Change logs are useful for both development and maintenance. The GNU coding standards (2002) give the following explanation for why we need change logs:

> You can think of the change log as a conceptual "undo list" which explains how earlier versions were different from the current version. People can see the current version; they don't need the change log to tell them what is in it. What they want from a change log is a clear explanation of how the earlier version differed.

A change log is a maintenance record over the life of the software product. A good change log includes every change without exception, detailed information about the reason for making each change, the location at which the change was made, the name of the person who made the change, contact information for that person, and the date on which that change was made. From the viewpoint of empirical software engineering, a complete and correct change log can be a valuable source of data for research on software maintenance.

The Free Software Foundation's GNU project gives clear requirements for the change log, including stylistic standards, as one component of the overall software documentation. The GNU project also explicitly states why it is important to have good change logs (GNU Coding Standards, 2002):

> Keep a change log to describe all the changes made to program source files. The purpose of this is so that people investigating bugs in the future will know about the changes that might have introduced the bug. Often a new bug can be found by

looking at what was recently changed. More importantly, change logs can help you eliminate conceptual inconsistencies between different parts of a program, by giving you a history of how the conflicting concepts arose and who they came from.

Open-source software projects provide a wide variety of resources from which information regarding changes can be deduced. The most obvious is the source code itself. Tools like `diff` can determine precisely what changes were made between one version and the next. This kind of information can also be obtained from version control tools such as CVS (Domain Home Page, 2002; Mockus et al., 2000, 2002). CVS relies on `diff` but records multiple changes that are grouped and allows annotations to be entered about the changes. However, unless every change has been annotated, it is often hard to discern the reason for a change. Furthermore, multiple changes are often checked into CVS at the same time and it can be hard to decide if they represent one or several changes. Mailing list archives could contain that information, as could fault tracking tools such as Bugzilla (2002) and GNATS (2002), but it is hard to extract the information and even harder to relate the comments to changes in the source code. When using `diff`, we treated changes that appeared to have a close relationship with each other as a single change. Although this is somewhat subjective, our belief is that it is less error-prone than solely relying on CVS.

If there is a change log, however, then there is no need to laboriously attempt to deduce what changes were made and the reason for each change; that is precisely the information that is recorded in a change log. A number of open-source web sites maintain a change log in a file called `ChangeLog`, which is thus an appealing source of information for researchers who are investigating changes. For this reason, we considered it important to investigate the completeness of `ChangeLog` files.

3. Method

There are many ways to evaluate `ChangeLog` entries. We decided to compare actual differences in the source code with entries in the `ChangeLog` file. We used `lxr`, the Linux cross-referencing tool (SourceForge, 2002), to determine the precise differences between two successive software versions. We then compared these differences with the records in the `ChangeLog` file to check the completeness of the `ChangeLog` file. Although some subjectivity is inevitable, we tried to limit this subjectivity as much as possible, as discussed below.

Descriptive statistics of three open-source software products, GNUJSP (2002), GCC-g++ (GCC Home Page, 2002), and Jikes (IBM, 2002), are shown in Table 1. Although GNUJSP and GCC-g++ are both GNU products, the former is small whereas the latter is medium-sized. Jikes is another medium-sized open-source product, which is supported by IBM. Also, GNUJSP is written in Java, GCC-g++ is written in C, whereas Jikes is written in C++. These products were chosen to be diverse in terms of size and language. We were not able to consider other attributes

Table 1. Summary of projects investigated.

Name	Language	Size	Description
GNUJSP	Java	Small	Implementation of Sun's Java Server pages
GCC-g++	C	Medium	Compiler for C++
Jikes	C++	Medium	A Java compiler that supports dependency analysis for incremental builds

such as number of contributors, age of the product, and amount of external support for the project in this study.

We categorized each change between successive versions as a corrective change, an enhancement (Schach, 2002), a code rearrangement, or a comments change. A corrective change is intended to fix a fault in the source code. Enhancement includes changes to improve the effectiveness of the product (including new features and efficiency) and to adapt the product for a different operating environment. A code rearrangement change modifies the sequence of the source code statements with no change in functionality (as judged by the evaluator). Any type of change to a comment, including a correction, was classified as a comments change.

In order to measure the completeness of the `ChangeLog` data, we checked whether or not each change in the source code was recorded in the `ChangeLog` file. We categorized each change and computed the percentage of omissions in each category. Also, the total number of changes in the source code and total number of changes in the `ChangeLog` file were computed to give an overall picture of the completeness of the `ChangeLog` file in question.

Counting the number of changes was not always straightforward. The changes fell into two groups, namely, changes for which there was an entry in the `ChangeLog` file, and changes that we discovered by applying `lxr` to the source code itself. This presented two kinds of difficulties. The first was detecting changes in `ChangeLog` files and the second was detecting changes from differences in versions (`lxr`). Whereas using the `lxr diffs` (or alternatively CVS logs) can ensure that every change was detected, the comments in CVS often do not provide enough information to analyze. In general, each entry in the `ChangeLog` file was treated as a single change. Sometimes, however, a single `ChangeLog` entry appeared to correspond to more than one change. For example, item 35 in the `ChangeLog` file for GCC–g++ version 3.01 reads:

```
''35* class.c (dfs_accumulate_vtbl_inits): Just point to the
  base we're sharing a ctor vtable with. Merge code for cases 1
  and 2.
  (binfo_ctor_vtable): New fn.
  (build_vtt_inits,   dfs_build_secondary_vptr_vtt_inits):
  Use it''
```

After carefully studying the source code, we concluded that this constituted three separate changes.

Another, almost opposing issue is when multiple entries are made to solve one problem or introduce one feature. Whereas it would be possible to try to categorize these as one "transaction-level" change, it is often impossible to discern the difference between multiple changes logged at the same time from a single change with multiple parts without being fully cognizant of the programmer's thinking and intent.

Where there was neither a `ChangeLog` entry nor any comments in the source code to explain why a change was made, we again studied the source code. Changes that appeared to have a close relationship with each other were treated as a single change.

There are other ways of counting changes, such as the number of changed files, the number of CVS check-ins, or the number of CVS modifications to individual files. The goal of our work was to count omissions in change logs, so we designed a counting mechanism that, as far as possible, would allow us to treat each change-log entry as a single change, thereby simplifying the counting process. In contrast, had we decided to count, say, the number of changed files, we would have had to study each change-log entry in detail to determine which files were referenced in that change log, and then decide whether each such reference was made within the context of change.

In a further effort to control subjectivity, we performed each classification twice and computed the cross-rater reliability. Two different researchers (from the authors, Chen and Yu) independently categorized each set of changes. The evaluations were compared using Cohen's Kappa test (Cohen, 1960). Cohen's Kappa coefficient κ (Cohen, 1960) is an index of cross-rater reliability. That is, κ is a measure of the extent to which two raters agree. Let P_O denote the proportion of ratings on which the two raters agree, and P_e denote extent of agreement expected by chance, that is, the expected proportion of ratings that agree, assuming the raters are statistically independent. Then κ is defined as

$$\kappa = \frac{P_O - P_e}{1 - P_e}$$

That is, κ is the ratio of the observed excess over chance agreement to the maximum possible excess over chance agreement. We used the Kappa threshold values computed by El Emam (1998); these values appear in Table 2.

Table 2. Threshold values for Cohen's Kappa statistic (El Emam, 1998).

Kappa value	Strength of agreement
< 0.45	Poor (bottom 25%)
0.45–0.62	Moderate (bottom 50%)
0.63–0.78	Substantial (top 50%)
> 0.78	Excellent (top 25%)

4. Change Log for GNUJSP

GNUJSP is distributed under the terms of the GNU General Public License (2001). It is a free implementation of Sun's Java Server Pages. At the time our research was conducted, the home page of GNUJSP contained 13 versions of the source code. The earliest version on the web was 0.9.0, which was published on August 27, 1998. The latest version we found on the web was 1.0.1, which was published on October 5, 2000. Most of the source code is written in Java. To simplify the analysis, only `.java` source code files were considered. There are four `.java` files in version 0.9.0, totaling 938 lines of source code. The results of our analysis on four versions are shown in Table 3. The columns each refer to a new version of GNUJSP, for example, the first column represents changes from version 0.9.1 to 0.9.2. Only one change was recorded between versions 0.9.2 and 0.9.3, so these data were omitted.

The number of changes in each separate category is relatively small, so we discuss only the overall percentage of changes. In version 0.9.2, 11 of the 14 changes (78.6%) were omitted. In version 0.9.4, however, only 1 of the 13 changes (7.7%) was omitted from the `ChangeLog` file. Versions 0.9.5 and 0.9.6 have omission rates of 3 out of 9 (33.3%) and 6 out of 31 (19.4%), respectively. Overall, 31.3% of the changes (21 out of 67) were omitted from the GNUJSP `ChangeLog` files.

Two different evaluators categorized the changes separately and the result is shown in Table 4. The Kappa value is 0.91 for GNUJSP. As shown in Table 2, this means that the strength of the agreement is considered excellent.

Table 3. Change-log analysis for GNUJSP.

					Weighted mean
Version investigated	0.9.2	0.9.4	0.9.5	0.9.6	
Previous version	0.9.1	0.9.3	0.9.4	0.9.5	
No. of changes in the source code	14	13	9	31	
No. of changes in the change log	3	12	6	25	
Overall percentage of change omissions	78.6%	7.7%	33.3%	19.4%	31.3%
No. of corrective changes in the source code	2	1	7	7	
No. of corrective changes in the change log	0	1	5	2	
Percentage of corrective change omissions	100.0%	0.0%	28.6%	71.4%	52.9%
No. of enhancement changes in the source code	4	12	1	14	
No. of enhancement changes in the change log	3	11	0	14	
Percentage of enhancement change omissions	25.0%	8.3%	100.0%	0.0%	9.7%
No. of code rearrangements in the source code	4	0	0	5	
No. of code rearrangements in the change log	0	0	0	4	
Percentage of code rearrangement omissions	100.0%	—	—	20.0%	55.6%
No. of comments changes in the source code	4	0	1	5	
No. of comments changes in the change log	0	0	1	5	
Percentage of comments change omissions	100.0%	—	0.0%	0.0%	40.0%
No. of comments that cannot be located	1	0	1	2	

Table 4. Cross-rater reliability analysis for GNUJSP.

	Evaluator 1				
Evaluator 2	Corrective	Enhancement	Rearrangement	Comment	Total
Corrective	16	1	0	1	18
Enhancement	1	30	1	0	32
Rearrangement	0	0	8	0	8
Comment	0	0	0	9	9
Total	17	31	9	10	67

5. Change Log for GCC-g++

GCC is one of the most successful GNU projects. The term "GCC" was initially an acronym for "GNU C Compiler." After its success as a compiler for C, GCC was extended to other languages, including C++, Objective C, Fortran, Java, and Ada. The meaning of GCC was then changed to "GNU Compiler Collection." GCC-g++ is the compiler for C++ in that collection. On the GNU Web site (GCC-g++ Source Code, 2002), we found the source code for 10 versions of GCC-g++, from version 2.95 (July 13, 1999) through version 3.1 (May 15, 2002). Most of the source code was written in C so, for simplicity, we analyzed only the C source files in the GCC directory. Also, we did not examine the other directory, libstdc++-v3, which contains the GNU Standard C++ Library. In version 3.0, there are 36 C source code files in the GCC directory, including both `.c` and `.h` files, totaling 97,333 lines of code.

The results of our analysis are shown in Table 5. Overall, only 10.8% of the changes (10 out of 93) were omitted from the `ChangeLog` file. The percentage of omissions decreased from 11.6% for version 3.01 to only 3.7% for version 3.03. (Similarly, we also observed an improvement with time in the Jikes project, as described in the next section.)

As before, two different evaluators categorized the changes independently and the result is shown in Table 6. The Kappa value is 0.86; again, the strength of agreement is excellent.

6. Change Log for Jikes

Jikes, an IBM-supported open-source software product, is a command-line compiler for Java, available under IBM's Public License (developerWorks, undated). The strength of Jikes over its peers is its ability to perform dependency analysis, thereby supporting incremental builds within Java source code. We examined changes to the source code but not to non-source files such as configuration and `make` files. To ensure that the data are comparable, we considered `ChangeLog` entries relating to only the source code as well.

Most of the source code is written in C++. At the time we performed this research, the source code for 15 versions of Jikes (versions 1.1 through 1.15) could be

Table 5. Change-log analysis for GCC-g++.

				Weighted mean
Version investigated	3.01	3.02	3.03	
Previous version	3.0	3.01	3.02	
No. of changes in the source code	43	23	27	
No. of changes in the change log	38	19	26	
Overall percentage of change omissions	11.6%	17.4%	3.7%	10.8%
No. of corrective changes in the source code	25	14	14	
No. of corrective changes in the change log	22	13	14	
Percentage of corrective change omissions	12.0%	7.1%	0.0%	7.5%
No. of enhancement changes in the source code	11	6	10	
No. of enhancement changes in the change log	10	5	10	
Percentage of enhancement change omissions	9.1%	16.7%	0.0%	7.4%
No. of code rearrangements in the source code	3	1	0	
No. of code rearrangements in the change log	3	1	0	
Percentage of code rearrangement omissions	0.0%	0.0%	—	0.0%
No. of comments changes in the source code	4	2	3	
No. of comments changes in the change log	3	0	2	
Percentage of comments change omissions	25.0%	100.0%	33.3%	44.4%
No. of comments that cannot be located	2	2	0	

downloaded from the Jikes homepage (IBM, 2002). The first version was published on January 25, 2001, and the latest version on May 5, 2001. In version 1.10, there are 74 C++ source code files. This figure includes both `.cpp` and `.h` files, totaling 92,233 lines of code (as counted by `lxr`).

The results of our analysis are shown in Table 7. As previously mentioned, the number of omissions decreased with time, from 62.4% in version 1.11 to 11.0% in version 1.15. Overall, the omission percentage was 24.5% (176 out of 719) for the three versions we examined. With regard to the four types of changes we measured, the percentage of comments changes that were omitted was 86.8% for version 1.11 (33 out of 38), 93.6% for version 1.13 (73 out of 78), but only 10.2% (15 out of 147) for version 1.15. One hundred percent of the code rearrangement changes were

Table 6. Cross-rater reliability analysis for GCC-g++.

	Evaluator 1				
Evaluator 2	Corrective	Enhancement	Rearrangement	Comment	Total
Corrective	53	4	1	2	60
Enhancement	0	23	0	0	23
Rearrangement	0	0	3	0	3
Comment	0	0	0	7	7
Total	53	27	4	9	93

Table 7. Change-log analysis for Jikes.

				Weighted mean
Version investigated	1.11	1.13	1.15	
Previous version	1.10	1.12	1.14	
No. of changes in the source code	93	281	345	
No. of changes in the change log	35	201	307	
Overall percentage of change omissions	62.4%	28.5	11.0%	24.5%
No. of corrective changes in the source code	27	30	67	
No. of corrective changes in the change log	14	28	63	
Percentage of corrective change omissions	48.1%	6.7%	6.0%	15.3%
No. of enhancement changes in the source code	27	171	122	
No. of enhancement changes in the change log	16	167	112	
Percentage of enhancement change omissions	40.7%	2.3%	8.2%	7.8%
No. of code rearrangements in the source code	1	2	9	
No. of code rearrangements in the change log	0	1	0	
Percentage of code rearrangement omissions	100.0%	50.0%	100.0%	91.7%
No. of comments changes in the source code	38	78	147	
No. of comments changes in the change log	5	5	132	
Percentage of comments change omissions	86.8	93.6	10.2%	46.0%
No. of comments that cannot be located	1	5	10	

overlooked in versions 1.11 and 1.15, but the numbers of such changes were relatively small (one and nine, respectively).

As before, two different evaluators categorized the changes independently. The result is shown in Table 8. The Kappa value is 0.88; once more this reflects, as shown in Table 2, that the agreement between the evaluators is excellent.

7. Analysis and Conclusions

This paper has made two contributions. The first is a methodology to extract, count, and classify changes from `ChangeLog` files and released versions. The second is the presentation of data that show that `ChangeLog` files are incomplete for research purposes.

Table 8. Cross-rater reliability analysis for Jikes.

	Evaluator 1				
Evaluator 2	Corrective	Enhancement	Rearrangement	Comment	Total
Corrective	99	24	1	0	124
Enhancement	22	293	0	0	315
Rearrangement	2	1	11	0	14
Comment	1	2	0	263	266
Total	124	320	12	263	719

Using our methodology, we detected omission percentages of between 3.7% (version 3.03 of GCC-g++) and 78.6% (version 0.9.2 of GNUJSP) in the `ChangeLog` files we examined. Averaging the overall weighted means (i.e., the numbers in the last column of the fifth row of Tables 3, 5, and 7), we found that the overall average percentage omission was 22.2%. Even the lowest figure of 3.7% can introduce errors into empirical studies, and 22.2% is at least disturbing for an average percentage error.

In addition to checking the completeness of the `ChangeLog` files, we also checked the correctness of each `ChangeLog` entry. It was our determination that almost all the entries were correct, although correctness may not be good enough if the data are not complete. At the very least, the researchers must be aware of the limitation.

It should be noted that `ChangeLog` files (and other open source artifacts) were not created for research purposes, but rather to support software development by easing tasks like maintenance and testing. It could certainly be argued that it is not fair for researchers to expect so much. It is undoubtedly true that, if investigators want to use `ChangeLog` files for experimental purposes, they must be extremely careful because of the incompleteness.

Our data are based on three open-source software products, thus the question of whether they are "typical" of open-source products is a threat to external validity. Although we are not aware of any way to determine whether an open-source product is typical, we have no reason to expect these products to be atypical. However, if this threat introduces a bias, it would be in the positive direction, i.e., we could have just happened to have stumbled across `ChangeLog` files that are better than others. The somewhat negative nature of our results indicating that `ChangeLog` files are not always complete serve as an existence "proof" that incomplete `ChangeLog` files exist, therefore the threat to external validity is reduced.

Based on our data, we suggest that before a log can be used as a basis for research, either it should be shown that the log in question is complete and correct, or it should be augmented with more complete sources of data such as CVS logs. In the case of a `ChangeLog` file, the only way to do this is by downloading the corresponding source code (or examining the version control files, if available) and checking that every change appears in the `ChangeLog` file and is correctly described. In the light of the large variability we observed in omission percentages from version to version within each project, this check has to be done for the `ChangeLog` file of every version of a given product. In view of this problem, particularly when coupled with the fact that complete data can be deduced from other sources of information (source code or version control files), it is our opinion that using the `ChangeLog` files alone can introduce internal threats to the validity of many experiments.

The omission percentages presented in this paper were computed using our change-counting method, as presented in Section 3. Other change-counting methods are possible (for example, the number of files changed) and generally result in different values for the omission percentage. For example, suppose that, using our change-counting method, the omission percentage for the change log for a specific version of an open-source product is 56%. Suppose further that 40 files are changed,

but only 12 of them are referenced in the change log within the context of change. Had we used the number of files changed as the change-counting method instead, the omission percentage would have been 70%.

However, even though the value of the omission percentage varies depending on the change-counting method used, our conclusions are not restricted to our own change-counting method. On the contrary, our conclusions hold for any change-counting method that reflects omissions in change logs.

This work examined the `ChangeLog` files of only three products. On the one hand, the data were sufficient for showing that there are substantial omissions in many `ChangeLog` files. On the other hand, the sample size is probably too small to be able to make any sort of generalization regarding `ChangeLog` files. In particular, it would be premature to put forward a mechanism to explain why the older GCC-g++ `ChangeLog` files exhibit fewer omissions than those of the other two products. However, we are currently examining other open-source products. With a larger sample, we will be able to put forward hypotheses to explain what we have observed, and then test those hypotheses statistically.

The field of open-source software is still developing. Although `ChangeLog` files may well serve the purposes for which they are intended (supporting software development), researchers need to take great care when using them. If the results of their experimentation depend on the `ChangeLog` files being complete, their results may be called in question.

Acknowledgments

This work was sponsored in part by the National Science Foundation under grant number CCR-0097056.

References

Bugzilla Project Home Page. October 2, 2002. www.mozilla.org/projects/bugzilla.

Cohen, J. 1960. A coefficient of agreement for nominal scales. *Educ. Psych. Meas.* 20: 37–46. developerWorks Open Source. [undated]. oss.software.ibm.com/developerworks/oss/license10.html.

Domain Home Page. 2002. www.cvshome.org.

El Emam, K. 1998. Benchmarking Kappa for Software Process Assessment Reliability Studies. *International Software Engineering Research Network Technical Report ISERN-98–02*.

GCC Home Page—GNU Project—Free Software Foundation (FSF). May 29, 2002. www.gnu.org/software/gcc.

(GCC-g++ Source Code). May 15, 2002. ftp://ftp.gnu.org/pub/gnu/gcc.

GNATS—GNU Project—Free Software Foundation (FSF). November 9, 2002. www.gnu.org/software/gnats.

GNU Coding Standards—Table of Contents—GNU Project—Free Software Foundation (FSF). May 31, 2002. http://www.gnu.org/prep/standards_toc.html.

GNU General Public License Home Page—GNU Project—Free Software Foundation (FSF). July 15, 2001. www.gnu.org/licenses/gpl.html.

GNUJSP—A free Java Server Pages implementation. February 21, 2002. www.klomp.org/gnujsp.

Harrison, W. 2001. Editorial: Open source and empirical software engineering. *Empirical Software Engineering* 6(3): 193–194.
IBM—developerWorks—Open Source Software – Jikes' Home. April 21, 2002. oss.software.ibm.com/developerworks/opensource/jikes.
JavaOne: Sun wades into open-source waters with Java. March 26, 2002. *Infoworld*, www.infoworld.com/articles/hn/xml/02/03/26/020326hnjavasource.xml.
Mockus, A., Fielding, R. T., and Herbsleb, J. 2000. A case study of open source software development: The Apache server. *Proc. International Conf. on Software Engineering*, pp. 263–272.
Mockus, A., Fielding, R. T., and Herbsleb, J. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. on Software Engineering and Methodology* 11: 309–346.
Schach, S. R. 2002. *Object-Oriented and Classical Software Engineering*, 5th edition. Boston MA: WCB/McGraw-Hill.
SourceForge: Project Info—LXR Cross Referencer. 2002. sourceforge.net/projects/lxr.
Welcome!—The Apache Software Foundation. 2002. www.apache.org.

Kai Chen is a Ph.D. student in the Department of Electrical Engineering and Computer Science at Vanderbilt University. He received his MS in Computer Science from Vanderbilt University. His current research includes maintenance and development of open-source software, formal methods, modeling language design, hybrid embedded system design, model verification and model-based integration of embedded software.

Liguo Yu is a Ph.D. student in the Department of Electrical Engineering and Computer Science at Vanderbilt University. His research topic concentrates on the maintainability of the Linux kernel and open-source software development. Before working on Software Engineering, his research focused on modeling and system identification, and fault detection and isolation of hybrid systems.

Stephen R. Schach is an Associate Professor in the Department of Electrical Engineering and Computer Science at Vanderbilt University in Nashville, Tennessee. Steve is the author of over 115 refereed publications. He has written 10 software engineering textbooks, including *Object-Oriented and Classical Software Engineering*, Sixth Edition, published by WCB/McGraw-Hill in 2004. He consults internationally on software engineering topics. Steve's current research interests are empirical software engineering, software maintenance, and open-source software engineering. He obtained his Ph.D. from the University of Cape Town in South Africa.

Jeff Offutt is an Associate Professor of Information and Software Engineering at George Mason University and holds part-time visiting positions at NIST and Skövde University. Jeff's current research interests include software testing, analysis and testing of web applications, software maintenance and object-oriented program analysis. He has published over eighty refereed papers. Jeff was program chair for ICECCS 2001 and is on the editorial boards for the *IEEE Transactions on Software Engineering*, the *Journal of Software Testing, Verification and Reliability*, the *Journal of Software and Systems Modeling* and the *Software Quality Journal*. Jeff's Ph.D. is from the Georgia Institute of Technology. He previously held a faculty position in the Department of Computer Science at Clemson University.

Gillian Z. Heller is an Associate Professor in the Department of Statistics at Macquarie University, Sydney, Australia, where she has been for the last 10 years. Her B.Sc. (Hons) and Ph.D. degrees are in Mathematical Statistics, from the University of Cape Town, South Africa, and her M.Sc. in Operations Research is from the University of South Africa. Gillian's research interests are in discrete distribution theory, with applications in biostatistics.