Collecting data from distributed FOSS projects

Fabian Fagerholm
Department of Computer Science
University of Helsinki
Fabian.Fagerholm@helsinki.fi

Juha Taina
Department of Computer Science
University of Helsinki
Juha.Taina@helsinki.fi

ABSTRACT

A key trait of Free and Open Source Software (FOSS) development is its distributed nature. Nevertheless, two project-level operations, the *fork* and the *merge* of program code, are among the least well understood events in the lifespan of a FOSS project. Some projects have explicitly adopted these operations as the primary means of concurrent development. In this study, we examine the effect of highly distributed software development, as found in the Linux kernel project, on collection and modelling of software development data. We find that distributed development calls for sophisticated temporal modelling techniques where several versions of the source code tree can exist at once. Attention must be turned towards the methods of quality assurance and peer review that projects employ to manage these parallel source trees. Our analysis indicates that two new metrics, *fork rate* and *merge rate*, could be useful for determining the role of distributed version control systems in FOSS projects. The study presents a preliminary data set consisting of version control and mailing list data.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics; D.2.9 [**Software Engineering**]: Management—*Software configuration management*; H.4 [**Information Systems Applications**]: Types of systems—*Decision support (e.g. MIS)*

General Terms

Human Factors, Management, Measurement

Keywords

free software, open source software, FOSS, software metrics, version control, distributed software development

1. INTRODUCTION

Free and Open Source Software (FOSS) has gained a strong foothold in the computer software industry. It challenges the entire field of software production in several areas, including technical aspects of software engineering, organisational and managerial structures of software projects, and social structures that surround production and use of software in society. Furthermore, FOSS is often seen as a cost-reducing factor in both software production and use, having a disruptive effect on the software market. Several commercial companies produce FOSS products, have entered or wish to enter into cooperation with FOSS projects, or wish to replicate positive aspects of FOSS in general. One particular area of interest is agility: robustness, low project management overhead, and sustained development speed in rapidly changing circumstances are closely tied into the working habits and tools of FOSS development.

Studying, describing, analysing and understanding FOSS is important both from academic and practical viewpoints. Several studies have described areas in which FOSS can bring advantages to software development projects [4, 14, 8]. Some studies have attempted to highlight details that seem to apply to FOSS in general [5, 11, 15]. A trait of FOSS is its tendency to rapidly abandon tools and methods that are cumbersome and rapidly adopt tools and methods that prove useful for the moment. In recent years, FOSS has undergone an evolution towards larger-scale and more distributed development, and earlier studies no longer reflect the current state of the art in FOSS development. Also, some techniques and methods described for collecting FOSS metric data are not directly applicable to distributed development.

In this paper, we describe how increased distribution affects FOSS development and collection and modelling of metric data. As an example, we examine a preliminary data set describing version control and mailing list data collected from the Linux kernel project.

2. RELATED WORK

Asklund and Bendix described the state of the art in FOSS configuration management in 2002 [1]. They noted that the area has traditionally been a manager activity where the software development process stages are directed and controlled from a top-down perspective. In contrast, FOSS configuration management is applied from a developer perspective. Configuration management tools are used to store and maintain software components and their history and to coordinate simultaneous changes to the product. Both configuration and working methods are included in this view, and the goal is to enable a group of developers to be as efficient as possible.

Godfrey and Tu examined the Linux kernel and saw that it has increased dramatically in size over a period of six years [6, 7]. They concluded that this indicates an ability of FOSS development to sustain super-linear growth. Izurieta and Bieman applied part of this study to the FreeBSD kernel and found that the claim of super-linear growth in FOSS has no support in general [9]. Polančič et al., Michlmayr, and Crowston et al. all draw attention to project-level aspects of FOSS development [15, 11, 12, 4, 5]. They observe both the layers and hierarchies that project participants are organised in and the limits in numbers of participants that project structures can support.

Koch and Schneider have developed a formal data model for information obtained from FOSS projects [10]. Their model encompasses people involved in the project, the discussion guiding it, and the actual source code produced. The model is based on observation of mailing list activity and activity within the GNOME CVS source code repository. Their experiment displays important characteristics that allows capturing some defining aspects of FOSS development. First, the sample granularity is quite small. For source code, a sample is a single CVS commit action, instead of an entire release. Second, each action is associated with an *actor* that performs it. Koch and Schneider demonstrate that this is an important factor when modelling FOSS projects. Third, Koch and Schneider include discussion lists in the model, and although they do not analyse discussion contents, they do find temporal correlation between discussion list activity and CVS activity. Fourth, Koch and Schneider apply timeline analysis throughout their methodology. Each action is placed on a time-line with very fine granularity, allowing the authors to make effort estimations and track progress on a very accurate scale.

Conklin et al. describe OSSmole, a repository of research data and analyses of FOSS data [3]. They note that a large portion of research on FOSS has prioritised presentation of results but have masked or discarded background information and research context needed to correctly interpret the results. Their experience with FOSS data collection shows that while a large mass of information is available because actions are always digitally recorded, the practical challenges to utilise this data are significant.

Christley and Madey describe a number of data mining techniques that can be used to manage some practical challenges when analysing a large dataset with many different kinds of data from many different projects [2]. They concentrate on analysis of well-defined activities such as checking out source code from repositories, modifying a source code file or updating a local copy of the source code. They show examples of algorithms for data analysis and identify activity patterns that correspond to development processes.

3. PROBLEM

Mining publicly available data from FOSS projects presents two important challenges. First, as previously noted, the analysis itself is a difficult problem. Finding or designing a data model that fits more than a few FOSS projects is hard because there is so much variation among projects. Data cleaning and format normalisation is time-consuming because the possible patterns are virtually endless and require repeated adjustments. Deep analysis frequently requires information network algorithms with NP complexity. Second, the data sources are changing. The tools used by FOSS projects are moving from a centralised storage and communication architecture towards a distributed architecture. It is no longer enough to obtain data from a single network location that remains stable for a long time; data acquisition must dynamically discover new sources during the analysis and cope with overlapping time-lines. The tool changes also affect the first problem – as the data sources themselves change, the analysis inevitably becomes more difficult.

We experienced the challenges of data capture from distributed version control systems as we studied quality factors in FOSS projects. Our original goal was to capture and analyse a data set with both great breadth and depth – number of variables and amount of samples. We wanted to model and analyse data from several complementing information spaces, allowing us to examine possible correlation between them and find more comprehensive explanations for the quality-related phenomena that we observed. We also explicitly wanted to explore the challenges involved in automating data collection and analysis. For our experiment, we used a data model partially derived from Koch and Schneider's, but expanded it and made it more general.

Although there is a large body of work analysing many different aspects of metric data collection and analysis, it is mostly based on a traditional view of FOSS development where a centralised, network-accessible source code repository is used by participants. However, several large FOSS projects have already abandoned or started to abandon this development model due to both project scaling issues and technical drawbacks of centralised systems. These projects have adopted distributed version control systems where each individual developer holds his or her own source trees and code sharing can take place in a more flexible way.

The scaling issues have been clearly visible in the Linux kernel project. Originally, the project did not use any source code management system [1]. Collaboration relied on patch files that were sent by email between developers or to public mailing lists. This caused a bottleneck in development, not due to the underlying model but due to the amount of manual work involved in processing and integrating patches[1]. In February 2002, the project started to use the BitKeeper version control system, with positive effects in the form of increased productivity. In April 2005, BitKeeper was abandoned[2] and later that year the project started using `git`, a tool written to fit the needs of that project. Several other FOSS projects have started using `git` as well. Efforts are

[1] Linus Torvalds: *Re: source mgt. requirements solicitation*, mailing list message on the GCC Project Mailing List, December 2002.

[2] Linus Torvalds: *Kernel SCM saga*, mailing list message on the Linux Kernel Developers' Mailing List, April 2005.

being made to store not only source code but also bug information in distributed data repositories.

As we applied our data collection tools on the Linux kernel project, we noticed that although we could collapse the complex version control system history into a single timeline by obtaining samples from only one of the source code trees, we did lose many explaining factors. We looked at the project from a single perspective – a perspective that does not reflect how the project participants see their project.

Let us briefly describe and define the distributed development model employed by the Linux kernel project. The source code is stored using the `git` version control system. Each stored instance, a source tree, can be defined as a *fork* of the source code. To begin development, a developer makes a *clone* of an existing fork. There is no actual primary copy – forks are technically equal and although humans may assign special meaning to them, their only relation is the common point in time when one fork has been cloned from another. The clone action also constitutes a fork action on the project level: the clone is a disconnected copy which gains an independent history identical to that of the original up to the point where it was forked.

The crucial point of the distributed development model is not forking but the way forks, or parts of them, are *merged.* Merges in `git` are performed as pull operations; even though the initial fork was made by cloning data from a specific location, merging is not performed by pushing back changes and resolving conflicts that may have arisen because of other merges. Instead, anyone may pull changes from a particular fork and resolve conflicts in their own source code tree. On the project level, a merge temporarily brings two independent histories together in a single point. Using pull merging has some important implications: the developer who made the fork does not have to be active in merging, as long as the fork is made available on the Internet. For metric data collection, however, this implies that new data mining techniques are required. A single time-line no longer reflects the real evolution of the source code.

4. EXPERIMENT

We first conducted an experiment prototype, in which we made feasibility tests for obtaining data and taking measurements from it. We experimented with approximately ten different FOSS projects, including the Linux kernel project. We wrote simple tools to collect data from several sources and calculate or simulate calculation of metric values. The focus in these prototypes was to solve technical issues with the data sources.

We selected three projects from the initial set of projects: Linux 2.6, an operating system kernel, GIMP, a graphics program, and Blender, a 3D content creation suite. Additionally, we included source code from versions of the Linux kernel ranging approximately from February 2002 to April 2005, as a comparison to the current Linux kernel tree. This code falls into the period when Linux was maintained using BitKeeper and the code has later been imported into a separate, special-purpose `git` repository to preserve history. We refer to this code as Linux-historical. Due to space limitations, we will only present results from the Linux kernel project here. We grouped the data sources into *information spaces.* The data sources are shown in Table 1.

	Linux 2.6	**Linux-historical**
VCS type	git	git
VCS URL	git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux-2.6.git	git://git.kernel.org/pub/scm/linux/kernel/git/tglx/history.git
BTS type	Bugzilla	Ad-hoc / none
BTS URL	http://bugzilla.kernel.org	Various / none
Relevant mailing lists	LKML	LKML
Mailing list archives	http://userweb.kernel.org/∼akpm/lkml-mbox-archives/	http://userweb.kernel.org/∼akpm/lkml-mbox-archives/

Table 1: Identified data sources for Linux and Linux-historical.

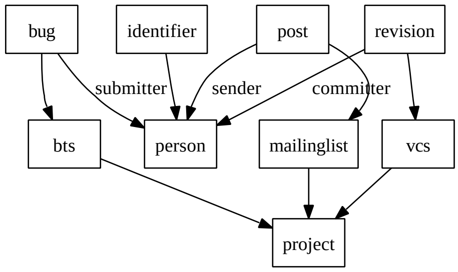

Figure 1: Data model for experiment.

To acquire data from each data source, we wrote special programs based on the earlier prototypes. All programs insert the results of their computation into an SQL database. The values of each computation can then be retrieved efficiently along with information about its context – from which project it was obtained, to which point in time the value is connected, and so on. The data model used in the database is described in Figure 1.

The first program extracts information from mailing list archives. It detects the sender, subject, message identifier and date of each message, and performs data cleaning on these items. Table 2 shows the number of messages and the time required to import them into the database. The second program obtains bug reports from bug tracking systems. We omit the analysis of this data since there is no symmetric, machine-readable mapping between a source code *commit* event and, for example, a bug *close* event. Also, bug tracking space data is frequently lagging behind source code space data, which could distort exact temporal correlation.

The third program obtains source code from network-accessible version control systems and runs metric calculations on it. We wrote support for Subversion and `git` systems, the first representing a traditional, centralised version control system, and the second representing a distributed system. The `git` part presented some challenges due to its

Mailing list	Messages imported	Time required
LKML	646 001	approx. 7 h

Source code repository	Revisions imported	Full metrics interval	Time required
Linux	64 707	every 300 revisions	approx. 44 h
Linux-historical	63 428	every 3000 revisions	approx. 14 h

Table 2: Import statistics for mailing list and version control data. The times required are approximate wall-clock measurements, rounded to the nearest hour, of the import job running on a 1.2 GHz PC.

distributed nature. Source trees are forked every time a developer clones another developer's tree and can be merged back either directly or via other developers. Several separate time-lines may exist simultaneously. Following the global source code changes is complicated and in some cases impossible. We chose to observe development through the main tree maintained by Linus Torvalds. We also included the Linux-historical tree as a comparison. Table 2 shows the number of revisions imported, the metric run interval, and the time required to perform this data acquisition.

All programs make use of data already stored in the database to identify the actor whose action resulted in a mailing list message, a submitted bug, or a source code revision. As artefacts are analysed, the database is automatically consulted to see if the associated actor-identifier has already been recorded. If so, we assume that the actor is the same person. Thus, we were able to make some correlation between actions in the different information spaces, although we did not use all possible data sources.

On inspection of discovered correlations, we found that the identification tokens have too little overlap between the different systems. In the Subversion system, the actor is identified only by a user name local to the main Subversion server. We found that this user name was frequently different from the name or email address used elsewhere, so unambiguously connecting actors in all three data spaces was not possible. However, `git` encourages the use of an email address as identification token, and thus it was often possible to connect actions in version control system data with actions in data from other information spaces. In some cases, developers had chosen other forms of tokens, which again interfered with correlation. We considered an approach where the analysis would be re-run each time a new identifier token is added, using tokens from previous runs to bootstrap identification. Unfortunately the time required to run that many iterations would have exceeded reasonable limits. Another approach considered was to manually produce the identifier tokens, but we rejected this because the time required would have been significant, and we wanted to see how well this completely automatic approach would work. We note now that the order of data source analysis is relevant; if we had started with the version control systems, we could have obtained greater overlap by using the local part of the actors' email addresses as well as the whole email address as an identifier in later stages. The identifiers obtained from the version control systems seem to be most consistent and have the least amount of errors and variations.

5. ANALYSIS

Our analysis shows that even by looking only at the metrics commonly reported, such as commit frequency, posting frequency, and project participant structure, there are indications that the project has made changes to its organisational model and thus achieved increased output. However, deeper analysis is needed to discover what has actually changed. Observing activity in the distributed version control system reveals both how the action patterns differ from centralised systems, and suggests some new metrics that could be calculated.

The commit frequency shows that activity has been varying between a few hundred commits to nearly 4000 commits per 30-day interval. It is difficult to detect a trend, but activity does seem to be cyclic, more or less following the development phases in use in the project. Releases are preceded by a short stabilisation time when only bug fixes are incorporated. The post frequency shows a rising trend. Discussion seems to be increasing, which could be a sign that more design work and coordination is performed as the code base grows. Alternatively, it could be a result of the steps taken to distribute development – as the barrier of participation decreases, the pace of discussion increases.

We detected a total of 22 236 distinct persons posting to the Linux kernel developers' mailing list and a total of 646 001 messages. The same kind of distribution applies to mailing list participation and bug submission as to source code commits. These findings are consistent with results in other studies; a majority of all work is done by a small number of contributors, while the majority of contributors contribute only once.

We were also able to obtain information on project structure, in terms similar to those defined by Crowston et al. and Mockus et al. [4, 5, 13]. The project structure is assumed to be layered with core developers, co-developers, active users, readers, and passive users. We chose to divide participants into these categories by measuring their contribution to the source code of the project. The number of commits per author follows a power law, with most commits being contributed by a small number of authors. Core developers were expected to have contributed the largest number of commits, with co-developers having contributed less but still enough to implement some small feature or make some important change in program logic such as optimisation or refactoring. Active users were assumed to have made sporadic contributions, mostly cosmetic fixes or small bug fixes.

Estimating the number of commits for core developers at 500 or more and for co-developers at 50 or more, we find that core developers constitute approximately half a percent of all committers and co-developers slightly more than 6% (Table 3). It should be noted that this definition of developer classes is only an approximation of socially assigned roles within the project.

Compared to Linux-historical, the core team has shrunk by

Developer class	Size: Linux	Size: Linux-historical	Size: difference	Percentage of committers: Linux	Percentage of committers: Linux-historical	Percentage of committers: difference
Core developers ($\geq$ 500 commits)	18	26	-8	0.54%	1.65%	-1.11%
Co-developers ($\geq$ 50 commits)	213	125	88	6.38%	7.93%	-1.55%
Active users	3106	1426	1680	93.08%	90.42%	2.66%

Table 3: Developer classes in Linux and Linux-historical.

more than one percentage unit, and the co-developers set has shrunk by slightly more than one and a half percentage units. The active user set has in turn grown by more than two and a half percentage units. This could indicate that there is a trend toward more distributed development where a shrinking core is moderating a growing mass of changes. The implication is that use of `git` has enabled development to become more distributed, reducing the load on core and co-developers, but also increasing the need for a hierarchical peer-review process.

When analysing version control data more closely we find that it is worth describing how the action patterns are different in a distributed system compared to a centralised one. Figure 2 shows a simplified part of version control history for a centralised system and a distributed system. In the centralised system, each work cycle begins with a *checkout* action, where the developer obtains a working copy of the source tree. After making local modifications, the developer *commits* the changes back to the repository. If conflicting modifications have been made in the meantime, the developer will have to resolve the conflict by updating the changes to apply to the newer version. In the distributed system, more complex patterns are possible.

Our example shows that the source tree begins its life as revision *A1*, stored in developer *A*'s repository. Developer *A* makes a change and commits it to the repository. At this point, developer *B* forks the source tree, creating a clone of revision *A2*, labelled *B1*. After this, developer *A* commits a change, creating revision *A3*. Meanwhile, developer *B* commits a change, creating revision *B2*. This revision has no other connection to developer *A*'s tree than the common ancestry with revision *A2*.

At this point, developer *C* forks developer *B*'s tree, cloning revision *B2* into revision *C1*. After that, developer *B* makes three commits, creating revisions *B3*, *B4* and *B5*. Developer *A* now merges revision *B5* back into the first source tree. Developer *B* does not need to be involved in this operation. The result of the merge is revision *A4*. Developer *A* then makes another modification, creating revision *A5*. Developer *C* has now modified revision *C1* and committed the modifications as revision *C2*. Developer *A* decides to merge these changes as well, and pulls in the changes as revision *A6*. Finally, developer *A* commits more changes, creating revision *A7*.

As the example shows, the action patterns in the distributed version control system are far more complex even in this quite normal scenario. Centralised version control systems

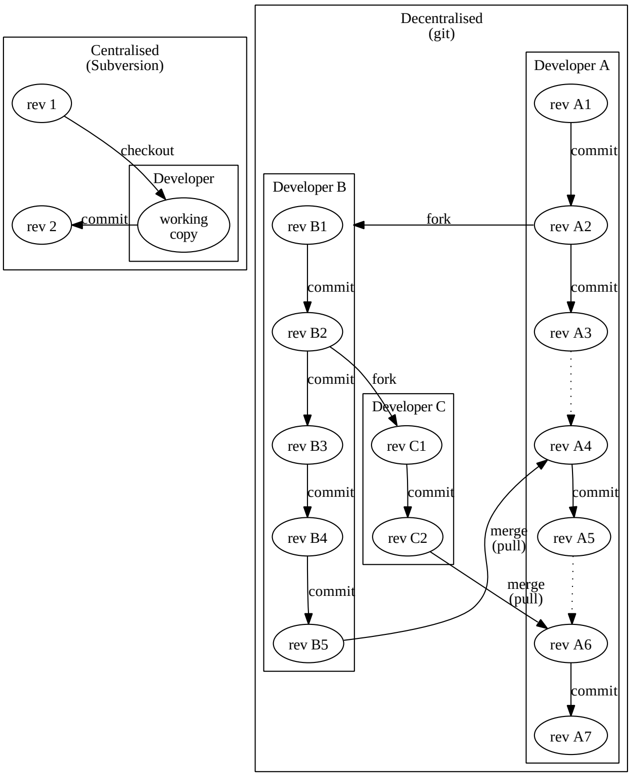

Figure 2: Examples of differences in workflow between centralised VCS (Subversion) and distributed VCS (`git`).

do have capabilities that go beyond the simple checkout-modify-commit cycle, but the simplest workflow in the `git` system can already result in action patterns that are far more complicated, because forks can occur from any source code tree and merges can be pulled into any source code tree at any time.

6. CONCLUSIONS

We have presented some of the challenges involved in collecting data from distributed FOSS projects. Our experiment suggests that some existing methods can be applied with meaningful results. However, new metrics are needed to specifically measure characteristics of distributed software development.

As FOSS projects adopt distributed version control systems, the amount of public data is likely to increase because the number of source trees increases and access to data is a prerequisite for the distributed workflow. This leads to increased challenges in studying data. Data sets will grow in size and must model several time-lines at once instead of only one. As we have shown, it is possible to see signs of changed organisational structures in projects by observing traditional metrics such as commit frequency, post frequency, and size of developer classes.

Because forks and merges are so fundamental to distributed version control systems, as shown in Figure 2, we suggest two metrics that could be useful for studying distributed development: *fork rate* and *merge rate*. They determine how how often fork and merge actions are performed in relation to commits. Differences in these metrics could be used to classify source trees; high sustained numbers for both could identify coordination trees while low numbers could identify development trees, for example.

Another area that requires further study is action patterns in distributed version control systems. For example, are forks usually merged back into the same tree as they originated from or is it common to have widely branched fork trees which are merged into each other with no clear main tree? These patterns may reveal how projects implement distributed development and how their organisational structure correlates with the development process.

We believe that the distributed development paradigm will increase the importance of a framework for research on public software development data. Detailed and accessible descriptions of data retrieval methods, metric calculation algorithms, tools to facilitate research, and standardised data sets would allow researchers to focus on one problem at a time and still perform the entire set of operations needed to get meaningful, comparable results.

7. REFERENCES

[1] U. Asklund and L. Bendix. A Study of Configuration Management in Open Source Software Projects. *IEE Proceedings – Software*, 149(1):40–46, 2002.

[2] S. Christley and G. Madey. Analysis of Activity in the Open Source Software Development Community. In *Proceedings of the 40th Hawaii international conference on system sciences*, page 166, Waikoloa, HI, USA, 2007. IEEE Press.

[3] M. Conklin, J. Howison, and K. Crowston. Collaboration using OSSMole: a repository of FLOSS data and analyses. In *Proceedings of the 2005 international workshop on mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[4] K. Crowston, H. Annabi, J. Howison, and C. Masango. Effective work practices for software engineering: free/libre open source software development. In *Proceedings of the 2004 ACM workshop on interdisciplinary software engineering research*, pages 18–26, New York, NY, USA, 2004. ACM Press.

[5] K. Crowston and J. Howison. Assessing the Health of Open Source Communities. *IEEE Computer*, 39(5):89–91, 2006.

[6] M. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. In *Proceedings of the international conference on software maintenance*, page 131, Washington, DC, USA, 2000. IEEE Computer Society.

[7] M. Godfrey and Q. Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the 4th international workshop on principles of software evolution*, pages 103–106, New York, NY, USA, 2001. ACM Press.

[8] G. Gousios and I. Samoladas. Software Quality Observatory for Open Source Software: Overview of the state of the art, deliverable report. Technical report, January 2007.

[9] C. Izurieta and J. Bieman. The evolution of FreeBSD and Linux. In *Proceedings of the 2006 ACM/IEEE international symposium on empirical software engineering*, pages 204–211, New York, NY, USA, 2006. ACM Press.

[10] S. Koch and G. Schneider. Results from Software Engineering Research into Open Source Development Projects Using Public Data. Discussion paper for Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft, Wirtschaftsuniversität Wien, 2000.

[11] M. Michlmayr. Software Process Maturity and the Success of Free Software Projects. In K. Zielinski and T. Szmuc, editors, *Software Engineering: Evolution and Emerging Technologies*, pages 3–14, Krakow, Poland, 2005. IOS Press.

[12] M. Michlmayr. *Quality Improvement in Volunteer Free and Open Source Software Projects: Exploring the Impact of Release Management*. PhD thesis, University of Cambridge, March 2007.

[13] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.

[14] J. W. Paulson, G. Succi, and A. Eberlein. An Empirical Study of Open-Source and Closed-Source Software Products. *IEEE Transactions of Software Engineering*, 30(4):246–256, 2004.

[15] G. Polančič, R. V. Horvat, and T. Rozman. Comparative assessment of open source software using easy accessible data. In *Proceedings of the 26th international conference on information technology interfaces*, volume 1, pages 673–678, Slovenia, 2004.