Knowledge-Based Software Engineering
A. Čaplinskas, H. Pranevičius, T. Nakatani (Eds.)
Technologija, 2010

AN APPLICATION OF GROWTH CURVE MODEL FOR PREDICTING CODE CHURN IN OPEN SOURCE DEVELOPMENT

Hirohisa AMAN[1] and Takahiro OHKOCHI
Graduate School of Science and Engineering, Ehime University, Japan

Abstract. Quantitative quality management and prediction are significant challenges in open source software development. While anyone could access to source code through the code repository, it is not easy to perform a quantitative assessment and/or prediction of the software quality. To provide a mean of quantitative management utilizing a code repository, this paper proposes a mathematical modeling of source code churn—a growth curve model based on a non-homogeneous Poisson process. An empirical work using 12 packages in Eclipse is performed, and the results show: (1) an inflection S-shaped curve model could be well-fitted to actual data, and (2) the proposed model could predict the code churn in the next 5 months with less than 10% error.

Keywords. Open source development, quantitative management, source code churn, prediction, growth curve model.

1. Introduction

In recent years, open source software(OSS) has been active in various fields such as operating system, Web server, embedded system, and so on. Some vendors use OSS in their products and/or services as key components[7]. However there is a difficulty in a quality assurance of OSS; Although anyone could access to a source code of OSS through the public code repository, it is not easy to perform a quantitative assessment and/or prediction of the OSS quality. While a source code is a detailed raw data of the development activities, the evolutionary change of the whole software could not be understood by simply observing each code. Such difficulty may lead to a barrier to industrial promotion of OSS utilization.

For an OSS project, a source code is a major product reflecting the development activities. To perform a quantitative management of OSS development, it is important to observe not only the source code but also its change-history showing the evolutionary process. Especially on change history, many repository mining techniques have been studied in recent days[8]. A repository mining is a powerful method to find some knowledge from a code repository. However it is a method for getting information through statistical analysis and/or machine learning, not for modeling the observed phenomena mathematically.

As another related work, software evolution in open source development has been studied as well. Godfrey and Tu[5] investigated a growth pattern of source code in Linux, and Xie et al.[15]

[1] Corresponding Author: Hirohisa AMAN, 3, Bunkyo-cho, Matsuyama-shi, Ehime, 790-8577 Japan, aman@cs.ehime-u.ac.jp.

also discussed the evolution patterns through their empirical study using many open source releases. Paulson et al.[11] investigated some growths in some major open source software including Linux, Apache and Gcc. While some software evolution patterns have been discussed, there are few studies focusing on a mathematical model of tendency in software evolution (code change).

Toward a mathematical modeling of the change history, we introduce another perspective on the analysis of code repository in this paper—a stochastic modeling. We consider a code change to be a stochastic event because many programmers would continue concurrently committing to the repository; in other words, it would be uncertainty when the next code change would be done.

In this paper, we propose a stochastic model of code change events (code churn) in order to formulate a novel method for explaining code churn mathematically. The contribution of this paper is to present some experimental trials showing that our mathematical model has a high level of ability to predict code churn in a large-scale OSS.

The rest of the paper is organized as follows: Section 2 describes the code churn in open source development, and introduces basic growth curve models. Then a stochastic modeling of code churn related to the growth curve model is proposed. Section 3 performs some empirical work using Eclipse. In the empirical work, some packages of source code are collected from Eclipse repository, and some trials are performed to fit growth curves to the actual code churn. Then the model's ability to predict code churn is evaluated. Finally, Section 4 presents our conclusion and future work.

2. Code Churn and Growth Curve Model

2.1. Code Churn in Open Source Development

In an open source development, source code changes are basic events reflecting the development activities. It is important to observe the code churn for a quantitative management of the open source development; term "code churn" includes code addition, deletion and modification. We measure the amount of code churn with the total changed LOC that is the total of added LOC (δ_a), deleted LOC (δ_d) and modified LOC (δ_m), excepting blank lines and comments:

$$\text{code churn} = \delta_a + \delta_d + \delta_m;$$

where δ_m is the greater LOC in corresponding fragments modified through version-upgrade. Generally a code modification corresponds to two different fragments—the older *x* lines were modified to the newer *y* lines. We define the amount of modified LOC as max(*x*; *y*). For example, Fig.1 shows a code churn in Java code fragments. The differences between the fragments could be captured by using GNU `diff`[4] [2], and Fig.2 shows the execution result. In this case,

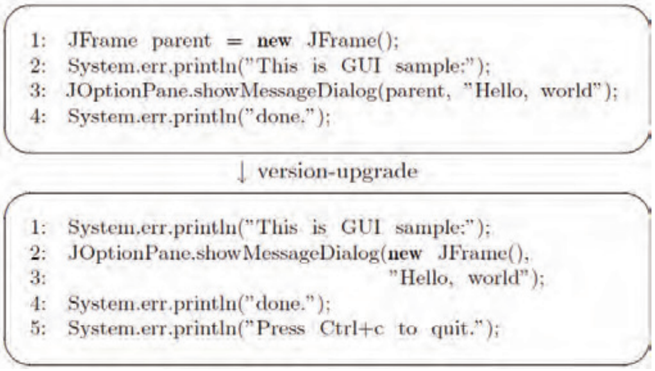
- Line 1 was deleted: $\delta_d = 1$;
- Line 3 was modified to new lines 3 and 4: $\delta_m = \max(1; 2) = 2$;
- New line 5 was added: $\delta_a = 1$.

Thus we get code churn = 1 + 1 + 2 = 4:

Most OSS projects have public code repositories to automatically manage their source code and code churn. While anyone could access to source code and code churn through the public repository, it is not easy to understand the developmental status. For instance, a major open source development site, SourceForge.net[12], gives some statistical information including the download history, tracker traffic statistics, and so on. However little statistics about code churn

[2] diff 2.8.1; the execution option was "-Bw."

could be easily obtained at such development site. While the number of version-upgrade and some simple `diff` results such as “`Changes since 1.3: +8 -6 lines`” could be obtained, it is not easy to organize all of such data about code churn scattered in the repository, and it is not easy to get useful statistics for a statistical analysis of code churn tendency.

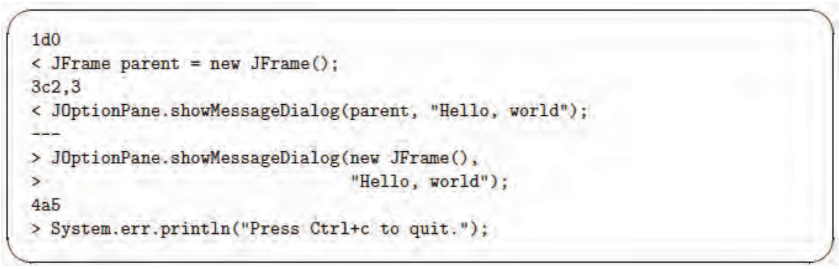


```
1:  JFrame parent = new JFrame();
2:  System.err.println("This is GUI sample:");
3:  JOptionPane.showMessageDialog(parent, "Hello, world");
4:  System.err.println("done.");
```

↓ version-upgrade

```
1:  System.err.println("This is GUI sample:");
2:  JOptionPane.showMessageDialog(new JFrame(),
3:                                "Hello, world");
4:  System.err.println("done.");
5:  System.err.println("Press Ctrl+c to quit.");
```

Figure 1. Example of code churn.



```
1d0
< JFrame parent = new JFrame();
3c2,3
< JOptionPane.showMessageDialog(parent, "Hello, world");
---
> JOptionPane.showMessageDialog(new JFrame(),
>                               "Hello, world");
4a5
> System.err.println("Press Ctrl+c to quit.");
```

Figure 2. Execution result of `diff` for fragments shown in Fig.1.

To get statistics about code churn, we collect monthly snapshots (source code) from the code repository, and examine monthly differences in the source code; we have developed a tool for collecting changed LOC, `DiffLineCounter.jar`[1]. Then we propose a mathematical model for explaining and predicting code churn from a stochastic viewpoint. We consider a code churn to be a stochastic event similar to a “customer arrival” in the queueing theory[10], and try to model code churn with a Poisson process model, especially a growth curve model.

2.2. Growth Curve Model

There has been considerable research on the software reliability growth model[3, 14] in which the cumulative numbers of faults or failures are explained with some mathematical models. Most of major models are based on the non-homogeneous Poisson process (NHPP) considering a fault (or failure) detection to be a stochastic event: Let $N(t)$ be the random variable representing the cumulative number of faults (or failures) detected by time t, and the probability of getting $N(t) = n$ is given as the Poisson distribution function,

$$\Pr\{N(t) = n\} = \frac{H(t)^n}{n!} e^{-H(t)} \qquad (n = 0, 1, 2, \ldots)$$

where

$$H(t) = \int_0^t h(x)dx \qquad (t \geq 0) .$$

Table 1. Basic NHPP models.

exponential model	delayed S-shaped model	inflection S-shaped model
$H(t) = a(1 - e^{-bt})$	$H(t) = a\left\{1 - (1 + bt)e^{-bt}\right\}$	$H(t) = a\,\dfrac{1 - e^{-bt}}{1 + ce^{-bt}}$

(a, b, c: constants)

H(*t*) is the mean value function of *N*(*t*), i.e., the expected number of faults (or failures) detected by time *t*, and *h*(*t*) is the fault (failure) detection rate at t. NHPP is a Poisson process with a time dependent fault (failure) detection rate, i.e., *h*(*t*) is variable over time; if *h*(*t*) is a constant, λ, then we get $H(t) = \lambda t$, and the stochastic process is homogeneous Poisson process(HPP), not NHPP. Table 1 shows the mean value functions of basic NHPP models. In Table 1, *a* is the expected number of all faults (failures), *b* is the parameter related to the fault (failure) detection rate, and *c* is the odds ratio of non-detectable faults to detectable faults[3]. Parameters *a*, *b* and *c* are estimated from actual data by using statistical method such as maximum-likelihood method or least-square method. We now propose modeling code churn in open source development, just as the above fault-detection: a change of code is considered as a stochastic event, and the amount of code churn is explained by a growth curve model based on NHPP. Although a strictly modeling should take considerations for the type and amount of each code churn, first we try to explore the availability of above macro models.

3. Empirical Work

In this section, we explore the trend of code churn from a practical open source development project, Eclipse[2]. Then we try to fit the above NHPP models to the actual data, and discuss the ability to predict the code churn.

3.1. Experimental Objects

Our experimental objects are 12 packages included in Eclipse, shown in Table 2, that have been actively developed with more code churn. Since a package is a set of source files that are related to each other structurally and/or semantically, we considered a package to be an appropriate unit of analysis for a code churn. We collected monthly code churn in those packages until Oct.1st, 2009. Although there are several releases of Eclipse in our data-collection period, we consider the cross-release issue would be of little concern, since all the target packages provide core functionalities of Eclipse, and such core functionalities would be continuously maintained through the releases.

3.2. Preliminary Work: Fitting

To find an appropriate mathematical model of code churn, we perform preliminary experiments for fitting the above three NHPP models to practical data of Eclipse. Figure 3 shows an instance of inflection S-shaped curve fitted to actual data in org.eclipse.core.runtime, whose mean value function is estimated as

[3] This is from the notion that some faults would be non-detectable in early testing stage since they were hidden by another faults. Such non-detectable faults would become detectable after the other detectable faults were fixed.

$$H(t) = 13382.07 \, \frac{1 - e^{-0.092t}}{1 + 12.7e^{-0.0921t}} \, .$$

Table 2. Packages used in the empirical study.

No.	package
1	org.eclipse.core.internal.content
2	org.eclipse.core.internal.jobs
3	org.eclipse.core.internal.localstore
4	org.eclipse.core.internal.plugins
5	org.eclipse.core.internal.preferences
6	org.eclipse.core.internal.registry
7	org.eclipse.core.internal.runtime
8	org.eclipse.core.resources
9	org.eclipse.core.runtime
10	org.eclipse.debug.core.model
11	org.eclipse.jdt.core.dom
12	org.eclipse.jdt.internal.compiler.parser

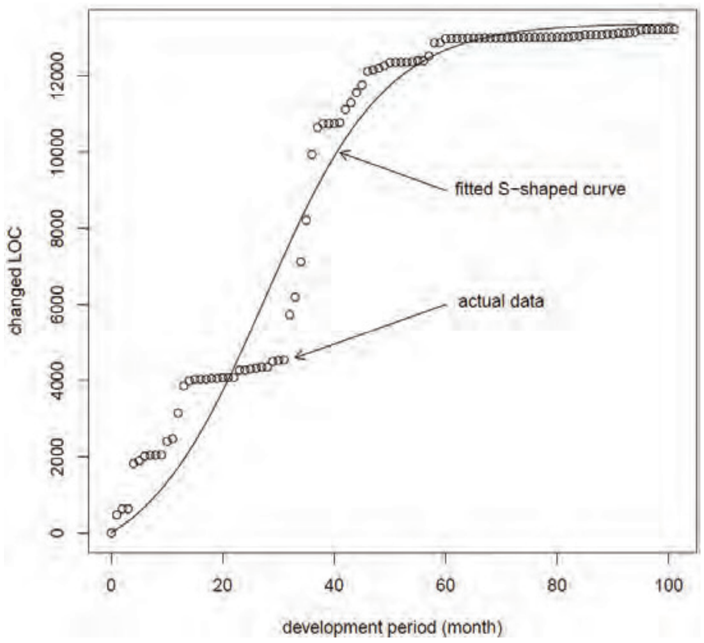


Figure 3. Fitting of an inflection S-shaped curve to actual data in "org.eclipse.core.runtime."

In Fig.3, the actual data is the monthly cumulative code churn (changed LOC) during the development, and the fitted curve is obtained through the least-square method. R[13] could automatically calculate the curve parameters so that the square error with actual data is minimal. In this example (Fig.3), while the early stage of development is roughly traced, the later stage is well-fitted and a convergence of the code churn may be modeled successfully. For all models shown in Table 1 and all packages shown in Table 2, we perform to estimate the curve parameters using R. See Table 4 in Appendix A for the list of estimated parameters.

Table 3 shows the mean magnitude of relative error (MMRE) as the results of fitting three models shown in Table 1; the less MMRE value indicates the better fitting. In our preliminary work, the inflection S-shaped model could show better fittings to the real code churn in Eclipse than the other models. We now consider the inflection S-shaped model to be a preferable model for explaining code churn, and we next try to use the model for predicting code churn in the following empirical work.

3.3. Procedure and Results: Predicting

We performed experiments for predicting code churn with the inflection S-shaped model. In our experiments, we predicted code churn in the next n month(s), for $n = 1, 2, \ldots, 24$. Notice that data before the inflection point of S-shaped curve could not build any stable prediction model, so we performed the experiments using data in later stage of the package development.

Table 3. MMRE in fittings.

No.	exponential	delayed S-shaped	inflection S-shaped
1	0.235	0.117	0.164
2	0.0487	0.0294	0.0284
3	0.149	0.182	0.16
4	0.0412	0.0381	0.0318
5	0.0457	0.0633	0.0426
6	5.44	2.12	0.615
7	0.324	0.146	0.0962
8	0.348	0.12	0.22
9	0.191	0.163	0.142
10	0.272	0.11	0.176
11	0.981	0.435	0.225
12	0.189	0.17	0.135
Total	0.689	0.308	0.17

Experimental procedure for predicting code churn in the next n months:

Given the cumulative code churn $c(t)$ in the first t months ($t = 1, \ldots, N$; N is the final month in the data collection. N varies from package to package since each package has a different time of starting the development.).

(1) Iterate the following (2) and (3) for $x = T_0; T_0 + 1, \ldots, N - x$; T_0 is the beginning month of the later developmental stage. We put $T_0 = 34$ from the actual data distribution.
(2) Build the inflection S-shaped model $H_x(t)$ using $c(t)$ ($t = 1, \ldots, x$), where $t = x$ is regarded as the current month on the package development.
(3) Predict code churn in the next n months, $c(t = x + n)$, and calculate the magnitude of relative error(MRE): MRE = $|c(x + n) - H_x(x + n)|/c(x + n)$.
(4) Calculate MMRE, the mean of MREs obtained in (3).

□

While MMRE = 0 is ideal for any prediction model, it would be not expected in the real world, and we should introduce a realistic threshold value of MMRE to be acceptable as a prediction model. Myrtveit et al.[9] have stated that MMRE = 0:25 is an acceptable level of an effort prediction model. This paper will consider that MMRE = 0:25 is a basic criterion of a prediction model as well. Moreover we will empirically regard MMRE $<$ 0:10 as a high-practical utility of a prediction model.

Figure 4 shows the prediction results in the above 12 packages. The X-axis represents the prediction time n as in “the next n months prediction,” and the Yaxis represents MMRE of the prediction. From Fig.4, we can see that the proposed models could predict code churn in the next 13 months with MMRE $<$ 0:25 for all packages. Moreover, the proposed models could predict code churn in the next 5 months with MMRE $<$ 0:10, and the results show a practical utility of the proposed ones.

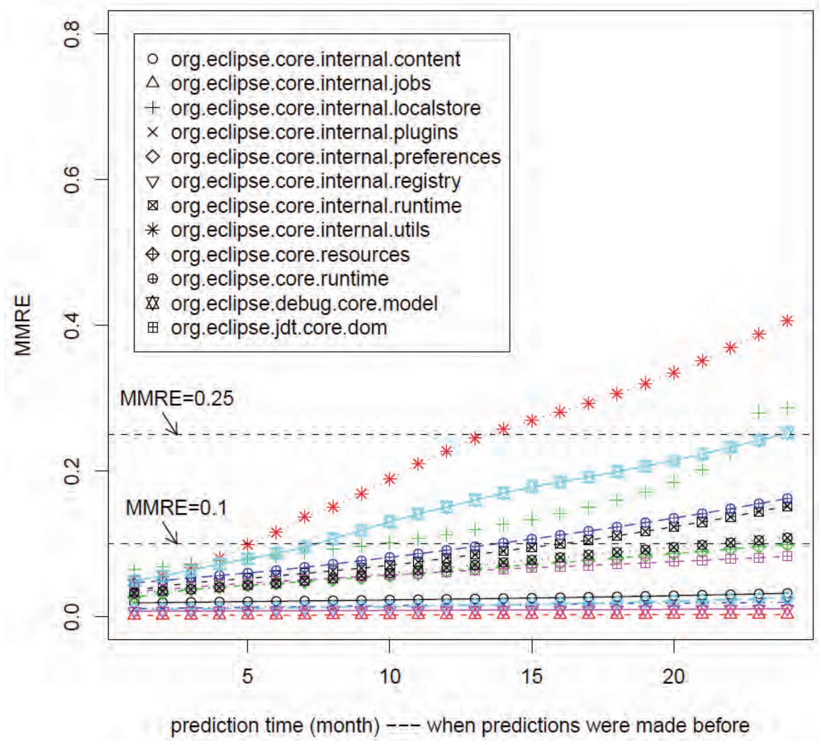


Figure 4. MMRE of predicted code churn in the next *n* month(s).

4. Conclusion and Future Work

This paper has considered a code churn in open source development to be a stochastic event, and proposed a prediction model based on the non-homogeneous Poisson process (NHPP). The empirical work using Eclipse source code has showed that: (1) an inflection S-shaped curve would be better model for fitting and predicting practical code churn; (2) the proposed model could predict code churn in the next 5 months with less than 10% error, and in the next 13 months with less than 25% error, repectively.

In the above small trial using Eclipse, the proposed approach have been supported: an inflection S-shaped curve model could predict code churn properly, and a similar tendency has been indicated for all experimental packages. We conclude that a NHPP-based model might be a promising model for predicting code churn in open source development.

To prove the generality of our proposal, we will perform further experiments for many other OSS packages as our future work, since the above empirical work was limited in only 12 core packages of Eclipse. Another future work includes an analysis of code churn itself. In the above empirical work, we omitted the contents and semantics of code changes. We will have to focus on code churn in detail with some related work, e.g.[6], for more accurate prediction model.

Acknowledgment

The authors would like to thank the anonymous reviewers for their thoughtful comments and helpful suggestions the first version of this paper. This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B), 2010, 22700035.

References

[1] H. Aman. DiffLineCounter, [Online] Available from: http://www.hpc.cs.ehime-u.ac.jp/˜aman/project/tool/DiffLineCounter.html [Accessed 31 March 2010].

[2] Eclipse.org, [Online] Available from: http://www.eclipse.org/ [Accessed 31 March 2010].

[3] A.L. Goel. Software Reliability Models: Assumptions, Limitations, and Applicability. *IEEE Trans. Softw. Eng.* 1985; SE-11(12):1411–1423.

[4] Free Software Foundation, Diffutils – GNU Project, [Online] Available from: http://www.gnu.org/software/diffutils/diffutils.html [Accessed 31 March 2010].

[5] M.W. Godfrey and Q. Tu. Evolution in Open Source Software: A Case Study. *Proc. In- ternational Conf. on Software Maintenance.* 2000; 131.

[6] S. Kim, E.J. Whitehead Jr. and Y. Zhang. Classifying Software Changes: Clean or Buggy?. *IEEE Trans. Softw. Eng.* 2008; 34(2):181–196.

[7] G. Lawton. The Changing Face of Open Source. *IEEE Computer* 2009; 42(5):14–17.

[8] MSR Home Page, [Online] Available from: http://www.msrconf.org [Accessed 31 March 2010].

[9] I. Myrtveit, et al. Reliability and Validity in Comparative Studies of Software Prediction Models. *IEEE Trans. Softw. Eng.* 2005; 31(5):380–391.

[10] R. Nelson. *Probability, stochastic processes, and queueing theory: the mathematics of com- puter performance modeling*. New York: Springer-Verlag; 1995.

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

[12] SourceForge.net, [Online] Available from: http://sourceforge.net/ [Accessed 31 March 2010].

[13] The R Project for Statistical Computing, [Online] Available from: http://www.rproject. org/ [Accessed 31 March 2010].

[14] A. Wood. Predicting Software Reliability. *IEEE Computer* 1996; 29(11):61–68.

[15] G. Xie, J. Chen and I. Neamtiu. Towards a Better Understanding of Software Evolution: An Empirical Study on Open Source Software. *Proc. International Conf. on Software Maintenance.* 2009; 51–60.

A. Parameters of growth curves fitted to actual data

Table 4 shows the estimated parameters of growth curves fitted to actual data in our preliminary work.

Table 4. Parameters of growth curves fitted to actual data.

No.		exponential	delayed S-shaped	inflection S-shaped
1	*a*	6037.38	5957.08	5972.24
	b	0.115	0.237	0.195
	c	—	—	1.92
2	*a*	9131.02	9015.27	9051.06
	b	0.122	0.259	0.190
	c	—	—	1.31
3	*a*	19600.46	13537.54	12646.03
	b	0.011	0.044	0.060
	c	—	—	7.31
4	*a*	9545.11	9446.24	9486.79
	b	0.098	0.204	0.142
	c	—	—	1.02
5	*a*	6942.22	6853.45	7051.96
	b	0.292	0.759	0.090
	c	—	—	−0.811
6	*a*	10645.46	10330.53	10101.12
	b	0.068	0.152	0.356
	c	—	—	81.2
7	*a*	23268.11	17309.13	15770.28
	b	0.013	0.049	0.091
	c	—	—	19.4
8	*a*	14897.73	13179.17	13240.86
	b	0.026	0.071	0.062
	c	—	—	2.45
9	*a*	16377.10	14043.96	13382.07
	b	0.021	0.059	0.092
	c	—	—	12.7
10	*a*	5009.16	4357.77	4389.72
	b	0.023	0.063	0.054
	c	—	—	2.41
11	*a*	163389.47	134594.52	120721.60
	b	0.020	0.061	0.257
	c	—	—	1035.11
12	*a*	108777.07	86055.38	79979.82
	b	0.016	0.053	0.087
	c	—	—	13.0