# Using Fine-Grained Code Change Metrics to Simulate Software Evolution

Zhongpeng Lin
Department of Computer Science
University of California, Santa Cruz, USA
linzhp@soe.ucsc.edu

Jim Whitehead
Department of Computer Science
University of California, Santa Cruz, USA
ejw@soe.ucsc.edu

## ABSTRACT

Software evolution simulation can provide support for making informed design decisions. In this research, we explored the distributions of fine-grained code change (FGCC) metrics and used them to build a simple simulator to evolve an existing source code file. The simulator generates synthetic changes to modify the source code analogous to how the code evolves in actual settings. By comparing the simulated evolution with the actual one, we found that the number and types of synthetic changes have no significant difference from those of the actual changes. Furthermore, the simulator is able to produce syntactically correct Java code, allowing us to analyze its static code metrics. The analysis shows that the distributions of method and field counts both have short tails at their left side, making it helpful in estimating the lower bounds for software growth. However, the actual method count falls below the distribution range produced by the simulation runs, indicating more sophisticated simulators are needed.

## Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement; D.2.8 [**Software Engineering**]: Metrics

## General Terms

Measurement, Theory

## Keywords

Fine-grained Code Changes, Simulation, Software Evolution

## 1. INTRODUCTION

Software gets continually changed, according to a well accepted Law of Software Evolution [7]. When people design a piece of software, they often have future changes in mind. Therefore, they try to make the design flexible. However, when future changes are different from what the designers anticipated, a good design could become unwieldy. For example, when Visitor design pattern [4] is used, it is perceived that the inheritance structure of the data elements is stable, while the operations to the elements may change often. The design pattern confines the changes to add or delete an operation to a single class. Nevertheless, if one needs to add a new data element type, all visitor classes need to be updated. To date, there is insufficient empirical support for choosing the right software designs.

A better design decision could be made if we are able to simulate future changes to a piece of software, allowing developers to see the ramification of different designs in software evolution. To that end, we are trying to build a simulator that takes the source code of a certain design and generates plausible changes to modify the source code. The resulting source code may not be meaningful or runnable, but it is structurally similar to the actual code in the future. Therefore, a variety of static code metrics can be extracted from it, so as to evaluate the impact of the design to the future code structure.

In order to simulate future changes, we need changes from the past. The availability of the technique to extract fine-grained code changes (FGCCs) [3] provides an opportunity to study such changes. FGCCs are software changes extracted by comparing the abstract syntax trees (ASTs) before and after the changes. An FGCC has information about the type of the AST node (e.g., `if` statement, method) being changed, the parent and children of the node, and the change type (e.g., method renaming, statement removal). It also has information about the position of the node in the source code, allowing for even finer-grained inspection if necessary. With the fine-grained information about changes made in the past, we can learn the change patterns and generate changes according to the pattern to simulate future changes.

In this exploratory study, we build a simple simulation model that uses the distribution of different kinds of changes in the past to generate changes according to that distribution. The simulator first needs to know the number of changes made during a certain time unit, a.k.a. code churn. In previous studies, code churn is often based on the number of source code lines modified. We redefine code churn based on FGCCs and use commits in version control systems as the time unit.

The first research question is dedicated to understand the code churn:

*RQ1. What is the distribution of FGCC based code churn?*

In addition, if a software evolution simulator were to use

*WETSoM'14*, June 3, 2014, Hyderabad, India
ACM 978-1-4503-2854-8/14/06
http://dx.doi.org/10.1145/2593868.2593871

FGCC metrics from the past to predict the future, it is necessary to know:

*RQ2. How well do past FGCC metrics predict future FGCC metrics?*

If metrics of past FGCCs are good predictors of future changes, a simulator could generate plausible changes and automatically evolve a piece of source code. However, plausible changes may not guarantee plausible source code, which leads to the next research question:

*RQ3. How similar to the actual source code is the source code with simulated changes?*

In this paper, we measure the numbers of fields and methods in the source code resulted from simulation runs, and compare them to the actual numbers.

## 2. RELATED WORK

Simulation is an approach to examine software evolution in previous studies. Most existing simulation models for software evolution can only predict some aggregated metrics, such as size [8], effort [10]. Stopford and Counsell [9] proposed a framework for simulating structural software evolution, which is closest to the simulation in this study, although their simulation is only down to method level. In contrast, the simulator built in this study generates source code changes down to statement level, and the artifact resulted from a simulation run is a piece of syntactically correct source code.

Change Distilling [3] is a tree differencing algorithm (implemented as an open source project called ChangeDistiller) to extract FGCCs and classify them according to their taxonomy [2]. The simulator in this study learns from past FGCCs extracted by ChangeDistiller, and generates changes to modifiy ASTs in order to produce the source code for studying the result of the simulated software evolution.

Herraiz et al. [6] conducted an empirical study over a large number of C source code files, and found that the size and complexity metrics of those files follow double Pareto distributions. Their model implies that "any model addressing software growth should produce this Pareto distributions." However, based on the Eclipse dataset, Zhang et al. [11] found that the program size follows a log-normal distribution. With a larger dataset, Herraiz et al. [5] inspect the tails of the code size distributions, and found the tails follow the power law, while the distribution bodies are log-normal, making the overall size distribution double Pareto. It is possible that FGCC metrics have log-normal or double Pareto distributions.

## 3. DATA ANALYSIS

This section introduces the data source and presents the distributions of FGCC based metrics of the data source.

### 3.1 Data Source

In this study, we used a Java source file as the subject. The chosen file was the source code of `TextArea` class in the `org.gjt.sp.jedit.textarea` package of jEdit. From October 2006, when it was extracted from another class, to August 2012, there were 148 commits modifying the file, among which 123 commits were composed of non-trivial changes, which involved more than code formatting. During this period, the file was changed incrementally, and grew from 6069 LOC to 6707 LOC, making it a good subject to start with.

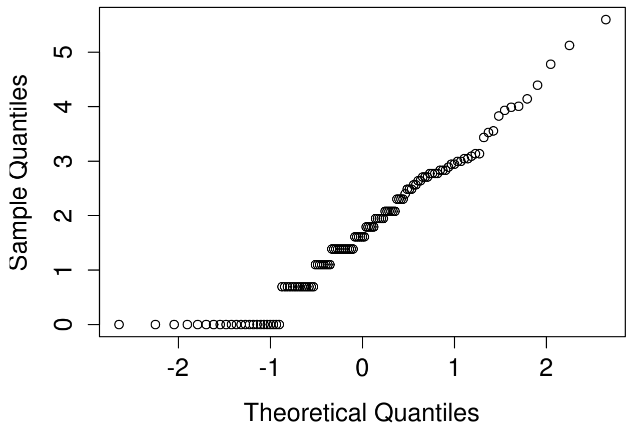

**Figure 1: Q-Q plot comparing distribution of the base-2 logarithm of code churn (y-axis) to standard normal distribution (x-axis)**

### 3.2 Distribution of FGCC Metrics

The simulator in this study needs to know the number and the kinds of changes to generate for each synthetic commit. It can be learned from the distributions of FGCC metrics.

We first visually compared the code churn distribution with a log-normal distribution. The `qqnorm` function in R was used to draw a Quantile-Quantile (Q-Q) plot to compare the logarithm of code churn with the standard normal distribution. In Figure 1, most of the points fit well to a straight line, indicating the code churn may follow a log-normal distribution. To verify whether log-normal distribution can be the actual one, we conducted a Kolmogorov-Smirnov test and obtained a $p$-value of 0.1734. The $p$-value is not significant enough to reject the null hypothesis that the code churn distribution is log-normal. Next, we compared code churn distribution with exponential distribution and Poisson distribution, which have similar shapes to log-normal in most part. The Kolmogorov-Smirnov tests returned $p$-values less than 0.01, which means that the code churn distribution can neither be exponential nor Poisson.

As can be seen from Figure 1, some dots on the tails deviate from the straight line. Deviating tails were also observed by Herraiz et al. [6]. They found those tails following power law distributions. Using the approach proposed by Clauset et al. [1], implemented in the R package `poweRlaw`, we estimated the $x_{min}$ and $\alpha$ for the power law distribution that best fits the data. The goodness-of-fit test gave a $p$-value of 0.23, meaning that power law cannot be ruled out as a possible distribution on the right tail. We further compared the log-normal and power law distributions with a likelihood ratio test, and found that power law fits the data slightly better than log-normal, but the difference is not significant ($\mathcal{R}$: 0.62, $p$-value: 0.53).

Consequently, log-normal and power law are both possible distributions of code churn. As there is no significant difference, for simplicity we use the parameters of the log-normal distribution to characterize the code churn based on FGCCs.

Based on the change types defined by Fluri et al. [2], we further define the term *change species* as a finer specification of FGCCs. The specification of a change species has three parts:

- change type
- the type of the AST node being changed
- parent node type. When a change type is moving a node (e.g., COMMENT_MOVE), this part is the type of the new parent node.

From the TextArea class, 184 change species were detected. The most frequent change species were:

1. inserting a variable declaration to a method
2. deleting a method call from a method
3. adding a method

These change species and their frequencies form a discrete distribution from which the simulator decides the kinds of changes to generate.

## 4. SIMULATION AND EVALUATION

In this section, the distributions of FGCC metrics are used to build a simple simulator. The simulator is evaluated using the changes of the TextArea class.

### 4.1 Simulation Method

Starting with a piece of source code, the simulator first parses the source code into an AST. Then a number of commits are generated and applied to the AST, transforming it into another AST, which can be printed out to a source code file.

Before each commit is generated, a number is sampled from the distribution of code churn, and used as the number of FGCCs in the commit. As the change species form a discrete distribution, the probability for each species can be estimated from the frequency of the species in the change history: when a change species appears more frequently in the past, it is more likely to be chosen by the simulator.

If the chosen change species is an insertion operation, the simulator generates a new AST node of the changed node type specified in the change species. For other operations, the simulator first randomly chooses an AST node of the changed node type from the current AST. When a change species has a parent type, a parent node of that type is also randomly chosen. Finally the simulator applies the change to the chosen node(s).

### 4.2 Training and Test Sets

The first 20% of the commit history of TextArea was used as the training set to build the simulator, and the following 20 commits were used as the test set. The revision of TextArea at the end of the training commits was used as an input to the simulator. This revision is denoted as $A$. The simulator then ran 500 times. Each time, the simulator parsed the source code of $A$ into an AST and generated 20 commits to sequentially modify the AST. The resulting revision is denoted as $B_i$, with $i$ being the simulation run number, ranging from 1 to 500. The actual revision at the end of the test set is denoted as $B'$.

### 4.3 Evaluation

The simulator was evaluated from three angles:

EV1 The number of generated changes compared to the actual number of changes in the test set

**Table 1: The distribution of field and method counts in the 500 revisions resulted from the simulation runs**

| | # of Fields | # of Methods |
|---|---|---|
| Minimum | 65 | 262 |
| 1st Quartile | 66 | 268 |
| Median | 66 | 271 |
| Mean | 66.63 | 272.4 |
| 3rd Quartile | 67 | 276 |
| Maximum | 88 | 374 |

EV2 Change species distribution in the generated changes compared to the actual distribution in the test set

EV3 The structure of $B_i$ compared to that of $B'$

Since the number of changes in each simulated commit is sampled from the code churn distribution in the training set, the number would eventually converge to the original distribution with a large number of simulation runs. Consequently, we only need to compare the code churn distributions between the training commits and the test commits to evaluate EV1. As the logarithm of the change count in each commit has a distribution close to normal, the EV1 can be evaluated by comparing two normal distributions: the training set versus the test set. One-way ANOVA was used for such comparison, resulting in a $p$-value of 0.729, implying that there was no significant difference in code churn between the training set and the test set. Therefore, as long as a simulator uses the code churn distribution in the training set to decide the number of changes to generate, the code churn in the simulated evolution is not significantly different from that in the actual evolution.

Next we evaluated EV2. Again, since the change species in the simulated commits were drawn from the distribution of change species in the training set, the evaluation of EV2 is reduced to comparing the change species distributions in the training and test sets. We used chi-square test to determine whether the two discrete distributions were different, resulting in a $p$-value of 0.206. This implied that there is no significant difference in the distribution of change species between the simulated software evolution and actual one.

As there is no strong evidence from EV1 or EV2 that the FGCC metrics in the test set are different from the training set, we built a simulator based on the FGCC metrics in the training set. The simulator ran 500 times and generated 500 revisions, $B_1$ to $B_{500}$. We then parsed these revisions and found that all of the revisions are syntactically correct Java code, allowing for analysis for the number of methods and fields each revision contained. The statistics of the field and method counts in $B_1$ to $B_{500}$ are shown in Table 1. Before the simulation, $A$ had 65 fields and 257 methods. After 20 actual commits, $B'$ had 67 fields, which was close to the numbers of fields in the resulting revisions of most simulation runs. However, all simulations resulted in source code with more methods than $B'$, which had only 254 methods.

We further investigated why all simulation runs increased the method count while it was decreased in actual evolution. It turned out there were 13 times more method removal changes in the test set than that in the training set, while the test set only had 10% more changes than the training set (training: 470, test: 516). The outcome showed that the simple simulator in this study did not work well with such a small data set.

Meanwhile, in the field and method count distributions (Table 1), the minima are much closer to the median and mean than the maxima, indicating that their probability density functions have short tails on the left side and long tails on the right side. Such distributions show the potential of the simulator in estimating the lower bounds of the field and method counts after a period of time.

## 5. DISCUSSION

In this section, we discuss our findings with regard to the research questions.

*RQ1. What is the distribution of FGCC based code churn?*

The code churn distribution of `TextArea` was can be either a double Pareto or log-normal distribution. Empirical studies on a larger data set are needed to know which distribution fits better. Future research may investigate the distribution of other FGCC based metrics as well.

*RQ2. How well do past FGCC metrics predict future FGCC metrics?*

When treating the code churn distribution as a log-normal distribution, the distribution was very similar in the training set and the test set. As a result, we used the log-normal distribution in the training set to decide the number of changes in each simulated commit. Given that some points deviates from the log-normal distribution at the tails, it is unclear how such simplification affected the simulation results.

The relation of change species distributions between the training and test sets was weaker, although the difference was not significant.

*RQ3. How similar to the actual source code is the source code with simulated changes?*

Due to the difference in change species distributions between the training and test sets, and an over simplified simulator, the method counts after simulations were significantly more than the count in $B'$, but most field counts were close to the actual one in $B'$.

## 6. THREATS TO VALIDITY

Since this is a exploratory study, we are aware of the following threats to validity:

**Statistical validity** Theoretically, not being able to reject the null hypothesis cannot confirm the null hypothesis. Despite the difference between power law and log-normal distributions in modeling the tail of code churn distribution is insignificant, it may become significant with larger data set. EV1 and EV2 face similar threats as well.

**Construct validity** Although we are able to reject exponential and Poisson distributions to be the distribution of code churn, there are may be other distributions that model code churn better than log-normal and power law.

**External validity** This study is based on a single file with limited revisions. More studies are need to verify its generalizability.

## 7. CONCLUSION

In this research, we studied the FGCC metrics distributions and built a simple simulator to generate code changes based on the distributions. The simulator is able to generate changes with similar metrics comparing to the actual changes from the test set. According to the parsing result, all simulation runs resulted in syntactically correct code. After parsing the resulting source code, promising distributions in field counts and method counts were found; both had short tails on the left side, making it feasible to estimate lower bound of the counts. However, a simple software evolution simulator based on FGCC metrics distributions did not predict the number of methods very well. Nevertheless, this study shows that an improved FGCC simulator may be able to foresee software evolution in some contexts, helping software developers to design for the upcoming changes.

## 8. REFERENCES

[1] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-Law Distributions in Empirical Data. *SIAM Review*, 51(4):661–703, Nov. 2009.

[2] B. Fluri and H. Gall. Classifying Change Types for Qualifying Change Couplings. In *14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 35–45. IEEE, 2006.

[3] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change Distilling:Tree Differencing for Fine-Grained Source Code Change Extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, Nov. 2007.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1994.

[5] I. Herraiz, D. German, and A. Hassan. On the distribution of source code file sizes. In *International Conference on Software and Data Technologies*, Seville, Spain, 2011. SciTe Press.

[6] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a Theoretical Model for Software Growth. In *Fourth International Workshop on Mining Software Repositories*, pages 21–21. IEEE, May 2007.

[7] M. M. Lehman. Laws of Software Evolution Revisited. *Computing*, 1149:108–124, 1996.

[8] M. M. Lehman, G. Kahen, and J. F. Ramil. Behavioural modelling of long-lived evolution processes–some issues and an example. *Journal of Software Maintenance and Evolution: Research and Practice*, 14(5):335–351, Sept. 2002.

[9] B. Stopford and S. Counsell. A framework for the simulation of structural software evolution. *ACM Transactions On Modeling And Computer Simulation*, 18(4):1–36, 2008.

[10] P. Wernick and M. Lehman. Software process white box modelling for FEAST/1. *Journal of Systems and Software*, 46(2-3):193–201, Apr. 1999.

[11] H. Zhang, H. Tan, and M. Marchesi. The distribution of program sizes and its implications: An eclipse case study. In *1st International Symposium on Emerging Trends in Software Metrics*, pages 1–10, 2009.