2015 12th Working Conference on Mining Software Repositories

Why Power Laws?

An Explanation from Fine-Grained Code Changes

Zhongpeng Lin and Jim Whitehead
University of California, Santa Cruz, USA
Email: {linzhp, ejw}@soe.ucsc.edu

***Abstract*—Throughout the years, empirical studies have found power law distributions in various measures across many software systems. However, surprisingly little is known about how they are produced. What causes these power law distributions? We offer an explanation from the perspective of fine-grained code changes. A model based on preferential attachment and self-organized criticality is proposed to simulate software evolution. The experiment shows that the simulation is able to render power law distributions out of fine-grained code changes, suggesting preferential attachment and self-organized criticality are the underlying mechanism causing the power law distributions in software systems.**

I. Introduction

Power law distributions have been found in many areas, such as the magnitude of earthquakes, the size of human settlements, the intensity of wars, etc. [1] The abundance of open-source software repositories permits statistical analyses of software systems and their evolution, which reveals the prevalence of similar distributions in software: in change sizes [2], in-degree and out-degree in dependency networks [3], number of subclasses [4], and so on. In fact, if one were to analyze the distribution of another measure of software, it would be most surprising to find it *not* following a power law or other heavy-tailed distribution.

Given the multiplicity of empirical studies across many different kinds of software systems, the power law distributions are unlikely to be accidental. Some mechanism is at work, but which one? Developers might naturally come to mind, since it is their hands that write the code that exhibits the many power laws. Yet, assuredly, no developer intentionally sets out to create these power law distributions. We have never heard a developer say, "the distribution of file sizes on this project is diverging from power law, so we should make changes to fix this!" Indeed, accomplishing such a policy would involve known bad practices. For example, power law tails in file size distributions indicate the existence of large files, a code smell [5], even in projects believed to be well-maintained, such as those studied by Herraiz et al. [6]. Rather than the deliberate result of intentional activity, power laws must be an emergent phenomenon caused by software evolution dynamics not yet fully understood.

It is challenging to explore the causes of power law dynamics. As an emergent phenomenon, they are caused by the cumulative effect of a software change process working over time. Achieving a high degree of control over a real world change process is challenging, and expensive. There is no cost-effective way to set up a controlled multi-year software evolution experiment in actual settings. Instead, we use software evolution simulations as a way to focus on interesting factors within a complex overall process, and study possible causes of power law distribution. Simulation has the advantage of permitting a high degree of control over software change processes, and the ability to quickly change parameters and see the impact of these changes over a multi-year evolution.

The causes of power law distribution have been studied in non-software domains, with several generative mechanisms proposed [1]. The two generative mechanisms that appear to have the best explanatory power for describing software power laws are *preferential attachment* and *self-organized criticality*, described below. Preferential attachment—also known as a Yule process—has been used to explain power law distributions in many different areas, such as the expansion of the World Wide Web [7]. During a preferential attachment process, "entities get random increments of a given property in proportion to their present value of that property" [4]. For example, the probability of a new link pointing to a web page is proportional to the in-degree of the web page. As preferential attachment continues, the distribution of the given property (e.g., in-degree of web pages) will eventually follow a power law distribution.

Some other power law distributions, such as earthquakes and extinction of species, can be explained with self-organized criticality (SOC) [8]. In SOC, many small changes create cumulative effects, leading the system into a series of critical states, also called *punctuated equilibrium*, where different sizes of avalanches occur. The evolution of such complex systems consists of equilibriums, where small changes accumulate, and punctuations, where large changes, or avalanches, disturb the system. The sizes of avalanches follow power law distributions. Bak [8] argues that there is no need to have different models for equilibriums and avalanches; they are both outcomes of self-organized criticality.

Bak's classic example is a sandpile. Imagine that sand trickles down at a constant rate onto a flat floor thus forming a pile. For most of the time, individual grains do not move after they land, and new grains only cause nearby gains move slightly if at all (equilibrium). As sand grains accumulate, the pile becomes steeper and steeper, eventually leading to a threshold where dropping more grains onto the pile could trigger a sand slide (avalanche), in which many grains are

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.14

displaced, some of them far away from the newly added grains. Over the process, different sizes of sand slides occur, following a power law distribution [8].

Similar to a sandpile, the evolution of self-organizing systems consists of elementary changes, like the movements of individual sand grains. These changes are straightforward individually, but collectively they produce complex patterns, such as power law distributions, that cannot be explained with simple aggregation of elementary changes. To understand the origin of these complex patterns, one has to start with individual "grains" and their interactions. Since software source code is semi-structured text that can be deterministically parsed into abstract syntax trees (ASTs), the "grains" in our study are the AST nodes. The fine-grained code changes on AST nodes are the subject of our simulation.

Based on preferential attachment and self-organized criticality, we propose a generative model to simulate source code changes in order to answer the following research question: *How can individual fine-grained code changes lead to power law distributions?*

Preferential attachment in code changes is known to be a possible cause of the power law distributions in several static code measures [4], and evidence shows that software evolution may also be a self-organized critical process [9]. However, to date there has been no research capable of simulating an evolving software system that exhibits power laws *simultaneously* in static measures, such as method calls, and evolutionary measures, such as change size.

The remaining content of this paper starts by showing power law distributions in change size (Section II). The simulation model is presented in Section III with simulation results in Section IV. Section V discusses the results and presents our thoughts on future work, followed by threats to the validity (Section VI). Section VII relates our work to others before the paper is concluded in Section VIII.

II. Power Law Distributions of Change Size

To provide an empirical background, we show that the change sizes in software projects follow power law distributions, which is important evidence for self-organized criticality in software evolution.

A. Data Collection and Analysis Approach

The data used in this section come from four open source Java projects. We used CVSAnalY[1] to collect data from their Git repositories. Table I shows the start and end dates of the data collection, as well as the number of commits collected. Only commits in their master branches were collected for this study. However, some of these commits might have been submitted in other branches before being merged into the master branch. As Git keeps the branching and merging history, the commits in the master branch of a project form a directed acyclic graph, in which multiple commits may derive from the same commit. As a result, we had to use the branching graph in Git to decide the previous commit of a given commit, rather than simply sorting the commits by their time stamps. On the other hand, a commit may be the result of merging two previous revisions due to parallel development. Merging commits are ignored in this study, as they normally do not bring meaningful new changes to software other than repeating changes from two branches and resolving conflicts.

[1] http://metricsgrimoire.github.io/CVSAnalY/

TABLE I
Data sources

Project	Start Date	End Date	Commits
jEdit	1998-09-27	2012-08-08	6486
Eclipse JDT	2001-06-05	2013-09-09	19321
Apache Maven	2003-09-01	2014-01-29	9723
Google Guice	2006-08-22	2013-12-11	1198

TABLE II
Parameters and p values of commit size distributions

Project	x_{min}	α	$p_{power\ law}$	$p_{log-normal}$
jEdit	252	2.59	0.55	$\ll 0.0001$
Eclipse JDT	122	2.11	0.03	$\ll 0.0001$
Apache Maven	115	2.33	0.32	$\ll 0.0001$
Google Guice	120	2.42	0.56	$\ll 0.0001$

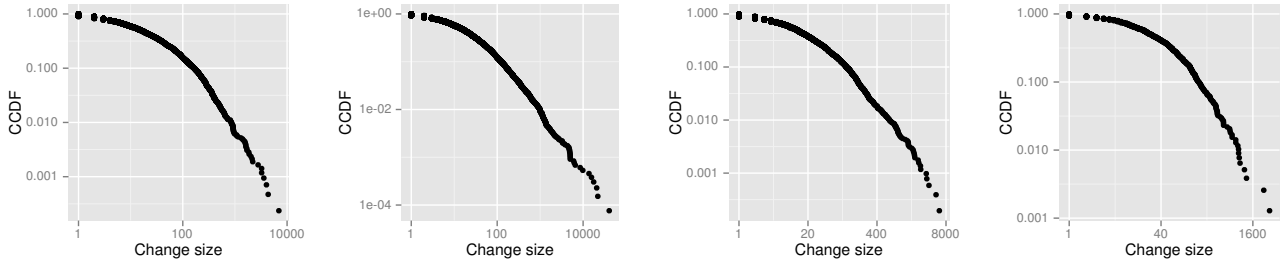
After collecting the raw data, we used ChangeDistiller [10] to extract fine-grained code changes by comparing the ASTs in each commit to those in its previous one. Fine-grained code changes extracted are changes to AST nodes down to the statement level.

Change size in this study measures the number of fine-grained code changes made during a period of time, such as the changes in a commit or a month. For both commit size (change size of a commit) and monthly change size, we first visualize the distributions with their complementary cumulative distribution function (CCDF), as proposed by Newman [1]. Then, following the statistical approach of Clauset et al. [11] as implemented in poweRlaw [12], we estimate the parameters of power law distributions that best fit the data, test the goodness of fit for the distributions, and compare power law with other heavy-tailed distributions to see if any other distribution provides a better fit.

B. Commit Size

On the CCDF of the commit size distribution (Fig. 1), the dots in the right side of each plot closely follow a straight line, which is an indication of power law distribution.

To test our hypothesis statistically on each data set, we first estimated the transition point x_{min}, where the straight line starts, as well as the exponent α of the power law. The x_{min} and α for each project are shown in Table II. With these parameters, we performed Kolmogorov-Smirnov tests to decide whether it was possible that the right tails were generated by power law distributions. As can be seen from the $p_{power\ law}$ column of Table II, most hypothesis tests report p values greater than 0.05 except Eclipse JDT. Therefore, we can only reject power law in the commit size distribution of Eclipse JDT.



(a) (b) (c) (d)

Fig. 1. CCDF for commit size distribution: (a) jEdit; (b) Eclipse JDT; (c) Apache Maven; (d) Google Guice.

TABLE III
COMPARING POWER LAW WITH ALTERNATIVE DISTRIBUTIONS FOR GOODNESS OF FIT TO COMMIT SIZE DATA

Project	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
	$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
jEdit	-0.251	0.802	3.43	0.000603	1.99	0.0468
JDT	-1.62	0.105	5.27	$\ll 0.0001$	2.36	0.0183
Maven	-0.176	0.860	5.79	$\ll 0.0001$	2.57	0.0102
Guice	-0.649	0.516	2.09	0.0365	3.75	0.000176

TABLE IV
PARAMETERS AND p VALUES OF MONTHLY CHANGE SIZE DISTRIBUTIONS

Project	x_{min}	α	$p_{power\ law}$	$p_{log-normal}$
jEdit	2765	3.17	0.63	0.0689
Eclipse JDT	12303	3.59	0.91	0.00157
Apache Maven	794	1.79	0	0.00105
Google Guice	249	1.87	0.14	$\ll 0.0001$

Although we cannot reject power law in most cases, other distributions might fit the data better. To rule out this possibility, we conducted likelihood ratio tests to compare power law with log-normal, exponential and Poisson distributions, which often have similar tails. In the tests, a positive $\mathcal{R}$ indicates power law fits the data better than the alternative, while a negative $\mathcal{R}$ indicates otherwise. The p value indicates how significant the corresponding $\mathcal{R}$ is.

Table III shows that power law fits the data better than exponential and Poisson distributions in all data sets, while log-normal distribution seems to fit the data better, although the difference is insignificant. To further test whether the data could be possibly drawn from log-normal distributions, we performed Shapiro-Wilk tests on the logarithms of change sizes. The resulting p values, reported as $p_{log-normal}$ in Table II, reject log-normal distribution in all cases.

C. Monthly Change Size

The procedure from the previous subsection was performed to analyze the distributions of monthly change size for the four projects and the results are presented in Fig. 2, and Tables IV and V.

The overall outcome of the monthly change size analysis is the similar to commit size distribution with a few exceptions. With regard to the monthly change size of Apache Maven project, the power law hypothesis was rejected and log-normal distribution fitted the data significantly better than power law in the follow-up likelihood ratio test. This trend can be visually verified in Fig. 2c, where the dots form a smooth curve rather than a straight line. Nevertheless, the result of the Shapiro-Wilk test rejected log-normal distribution as the true distribution of the data set, rendering the distribution unknown. In other data sets, power law showed a good fit at the right tails of their monthly change size distribution, and no other distributions in our study have a better fit.

TABLE V
COMPARING POWER LAW WITH ALTERNATIVE DISTRIBUTIONS FOR GOODNESS OF FIT TO MONTHLY CHANGE SIZE DATA

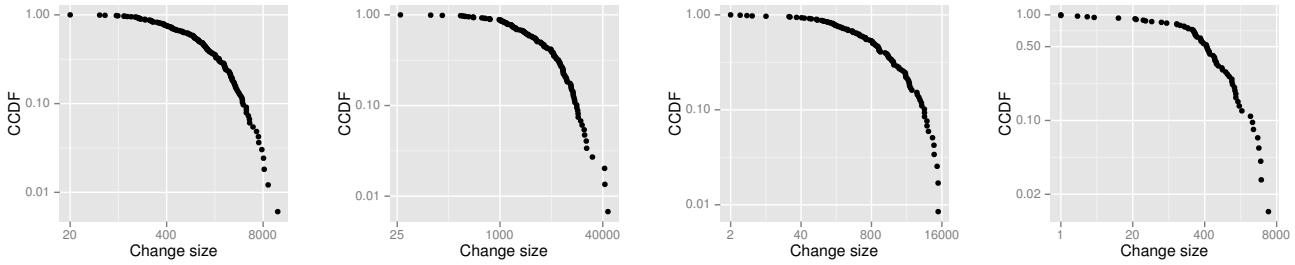
Project	pl vs. log-normal		pl vs. exponential		pl vs. Poisson	
	$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
jEdit	-0.794	0.427	1.65	0.0983	2.31	0.0208
JDT	-0.0280	0.978	1.974	0.0484	4.79	$\ll 0.0001$
Maven	-2.08	0.0374	0.408	0.683	2.83	0.00442
Guice	-1.38	0.167	0.692	0.489	3.50	0.000469

D. Summary

This section revealed that fine-grained code changes are yet another software measure whose distribution can be described by a power law. Combined with findings in earlier studies [2], [9], we can see that the distribution of change size follows power law regardless of the granularity of the changes, be it at AST-node level, line level or file level. This scale-free quality is an important feature of power law distributions. The possible dynamics in source code changes that collectively shape power law distributions will be explored in the next sections.

III. CODE CHANGE SIMULATION

The simulation starts with a simple piece of Java source code with one class and one method, which is parsed into an AST. Then a series of simulated changes are made to the AST. Changes simulated include adding classes, adding methods, calling methods from other methods, deleting methods, and adding statements to methods. At the end of the simulation, the resulting ASTs are printed to source code files. This section first introduces how the simulation model realizes



(a) (b) (c) (d)

Fig. 2. CCDF for monthly change size distribution: (a) jEdit; (b) Eclipse JDT; (c) Apache Maven; (d) Google Guice.

preferential attachment and SOC when making these code changes. Then the model is presented, followed by the setup of our experiment.

A. Preferential Attachment

Preferential attachment is used to simulate the growth of software, including adding dependencies among its components. Although there are several types of component dependencies in software, only method calls and class inheritances are simulated in this study. Based on preferential attachment, the following rules are used during the simulation:

- The likelihood of a method to be called is proportional to the number of times it is being called,
- The likelihood of a method or a class to grow is proportional to its size, and
- The likelihood of a class to be subclassed is proportional to the number of its subclasses.

It has been demonstrated that these simple rules, if running alone, are able to produce power law distributions in class size, number of method calls, and subclasses [1]. To reproduce punctuated equilibrium characteristics, to the preferential attachment rules (above) we need to add additional rules.

B. Self-organized Criticality

An SOC process is often characterized by its avalanches, e.g., sand slides in the sandpile model introduced in Section I. Avalanches are analogous to source code changes of different sizes during software evolution, as shown in Section II. In the model, we create rules to facilitate avalanches of different sizes, and see if the rules on fine-grained code changes could lead to SOC behavior in the simulated software evolution.

In order to simulate the avalanches in software evolution, we borrow a concept from biological evolution: *fitness*. For a given method in a software class, fitness is a measure of how well it endures throughout a software evolution, with values ranging from near 0 (very susceptible to change) to 1 (very stable). Initial fitness values are randomly assigned to methods in simulation runs. During simulation, methods with the lowest fitness values are selected for either a update operation, or for deletion. When a method is changed, it gets a new random fitness value. In addition, when a method signature is changed or deleted, all its callers have to change as well, and thus are assigned new fitness values.

During a simulation run, a threshold fitness value, f_0, is established, and every commit will have no methods with fitness below threshold. Then a commit in our simulated software evolution would comprise fine-grained code changes between a step S_i in our simulation model when all fitness values are above f_0, and the closest subsequent step S_j when the minimal fitness value of all methods, $f_{min}(S_j)$ is back to above f_0 again. Steps between S_i and S_j constitute an avalanche, with its size being the number of changes made in between.

During an SOC process, the f_{min} would never exceed the self-organized threshold f_c. As a result, setting f_0 above f_c would lead to no commit after the initial fluctuations. When f_0 is set just below f_c, the avalanche (commit) sizes follow a power law distribution [13]. f_c can be identified in the simulation result if SOC is produced.

C. Simulation Model

The rules for preferential attachment and SOC are implemented as four interweaving processes in the simulation model, corresponding to creating, calling, updating and deleting methods, denoted as P_c, P_r, P_u and P_d, respectively. Each process is associated with a probability p_i, where $\sum_{i=1}^{4} p_i = 1$. For each step in the simulation, one of the four processes is chosen to run according to their probabilities.

P_c works as follows:

1. Create a empty method M, and assign it with a random fitness value between 0 and 1;
2. With probability q_1, add M to a new class, which is created as follows:
   (a) Create a empty class C;
   (b) With probability q_2, make C a subclass of an existing class in the system, chosen with a probability proportional to the number of its current subclasses;
3. With probability $1 - q_1$, add M to an existing class in the system, chosen with a probability proportional to the number of its current methods.

P_r runs with following operations:

1. Choose a method S as the caller with a probability proportional to the size of its body, measured by the number of statements;
2. Choose a method T as the callee with a probability proportional to the number of its current references;
3. Call T from S, and assign S with a new random fitness number.

P_u inserts a statement into the method with the least fitness value and assigns a new random fitness value to the method. To delete a method with P_d, the following operations are performed:

1. Find the method M with the lowest fitness value;
2. Remove all references to M:
   (a) If the caller M_c becomes an empty method after removing references to M, add M_c to V;
   (b) Else update M_c with a new fitness value;
3. Remove M from its class C;
4. If C is now empty, delete C;
5. Repeat operations 2-4 for all methods in V.

Note that P_d could lead to ripple effects, causing many methods to be changed or deleted.

D. Experiment Setup

In our experiment, p_{1-4} were heuristically set to 0.1, 0.4, 0.45, and 0.05 respectively, while q_1 and q_2 were set to 0.1 and 0.8 respectively. During each simulation run, 200,000 steps were executed, each executing one of P_c, P_r, P_u and P_d. Twenty simulation runs were performed, with results reported in the next section. For each simulation run, we recorded the change size for each step, as well as the number of classes, methods, method calls and lines of code at the end of the simulation.

For reproducibility, the source code of the simulation is made available on Github[2].

IV. Simulation Results

All simulation runs in our experiment produced similar results, yielding 1484 classes, 9234 methods, 12900 method calls, and 53940 lines of code on average. In addition, the simulations produce very similar distributions of change size, class size, number of method calls, and subclasses. Due to space constraints, and for ease of description, below we report on the distributions produced by a single randomly chosen simulation run.

A. Change Size

The simulated software evolution exhibited SOC behavior. The scatter plot of $f_{min}(S)$ at different steps during the chosen simulation is shown in Fig. 3. At the beginning of the simulation, f_{min} could be any value between 0 and 1. Over simulated time, the range of f_{min} values reduced as the system grew. After around 20,000 steps, the system became stationary, and

[2]https://github.com/linzhp/Codevo3

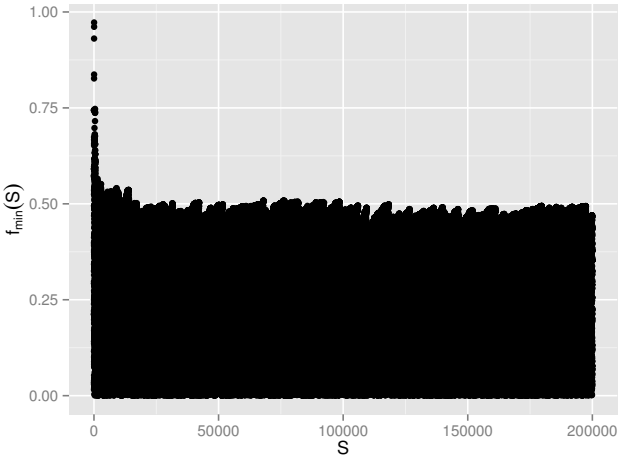


Fig. 3. Minimal fitness value at different steps. The x-axis represents different steps, y-axis is the minimal fitness value of all methods at the end of each step.

TABLE VI
Parameters and p values of distributions produced by simulation

Measure	x_{min}	α	$p_{power\ law}$	$p_{log-normal}$
Change size	2	1.59	0.93	$\ll 0.0001$
# of callers	11	4.42	0.94	$\ll 0.0001$
Class size	80	2.65	1	$\ll 0.0001$
# of subclasses	2	2.68	0.05	$\ll 0.0001$

f_{min} never went beyond a self-organized threshold between 0.5 and 0.51.

With f_0 set to 0.5, we obtained 1277 commits of different sizes. It can be seen from Fig. 4a that the CCDF of commit size distribution follows a straight line, indicating a possible power law distribution.

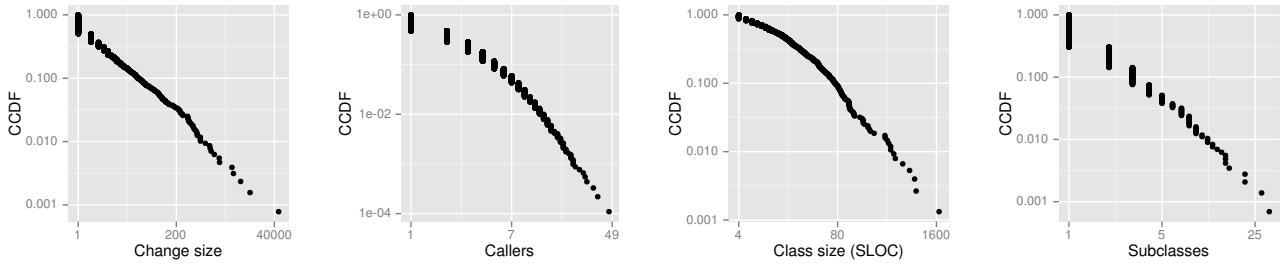
Following the procedure from Section II, we tested the goodness of fit to power law and compared power law with other alternative distributions. Tables VI and VII show that it is risky to rule out a power law distribution, which fits the commit size distribution better than exponential and Poisson distributions. Although a log-normal distribution seems to have a better fit, a follow-up Shapiro-Wilk test rejects it.

B. Method Calls, Class Sizes and Subclasses

As the simulation produced Java source code with valid syntax, static analyses can be performed. It can be seen from Fig. 4 that the distributions of method in-degrees (number of callers), class size measured by SLOC and the number of subclasses all have tails closely following power law, consistent with the empirical studies in [3], [6] and [4]. The results of these hypothesis tests are shown in Tables VI and VII.

C. Summary

From the simulation results, it can be seen that the code change mechanisms in Section III-C, in spite of their sim-



(a) (b) (c) (d)

Fig. 4. The CCDFs of distributions in simulation result: (a) Commit size distribution with f_0 set to 0.5; (b) In-degree (number of callers) distribution of methods. To avoid taking logarithm of 0, the in-degree for every method has been increased by 1; (c) Class size distribution; (d) Number of subclasses distribution. The number of subclasses has been increased by 1 for every class.

TABLE VII
COMPARING POWER LAW WITH ALTERNATIVE DISTRIBUTIONS FOR GOODNESS OF FIT TO SIMULATION RESULTS

Measure	power law vs. log-normal		power law vs. exponential		power law vs. Poisson	
	$\mathcal{R}$	p	$\mathcal{R}$	p	$\mathcal{R}$	p
Change size	-1.07	0.285	1.93	0.0532	3.19	0.00142
Number of callers	-0.565	0.572	0.771	0.441	1.59	0.112
Class size	0.00647	0.948	2.67	0.00764	2.12	0.034
Number of subclasses	-1.21	0.227	3.02	0.0025	1.58	0.114

plicity as compared to the complexity of real world software development, are capable of producing the power law distributions observed in software systems.

V. DISCUSSION AND FUTURE WORK

This section discusses the findings and our thoughts about future extensions of this work.

A. Punctuated Equilibrium and SOC During Simulations

The change size distribution in Section IV-A exhibits a punctuated equilibrium behavior: a large number of small changes mixed with a small number of large ones. Most changes are localized, and the rest of the system remains stable. New methods often have lower fitness, thus evolve faster and are also more likely to be deleted. However, they tend to be relatively isolated from the other parts of the system, so their changes are more localized. As the system evolves, more method calls are added. Occasionally, when a highly depended-upon method is changed, all its callers are changed too, with new fitness values assigned to them, triggering a ripple effect. Some stable methods are assigned low fitness values and become actively evolving. It takes many steps before all low fit methods evolve to a better fitness or are deleted, and the system is back to equilibrium again.

Self-organized criticality in software evolution implies that large, system-wide changes, though infrequent, are inevitable. A significant external (hardware, requirement, market etc.) change is not required to trigger large software changes. As the size and complexity of the system grows, large changes can come in the form of proactive refactorings or painful *shotgun surgeries* [5]. This, though, does not imply that refactorings are not necessary. As Turnu et al. [4] discovered, the power law distribution can take different α values at different times. So if a refactoring is overdue, shotgun surgeries could become more frequent, driving α down.

Power law distributions of change size imply that it is very difficult to predict change size. This is because the mean of power law distributions is undefined when α is less than 2, and the variance is infinite when α is less than 3. As we can see from Tables II and IV, α values often fall below 3, and sometimes even below 2. This implies that the central limit theorem does not apply to such distributions, and hence the sample mean and variance is a poor estimate of the population mean and variance. Combined with the *scale-free* nature of power law distributions [1], there is no "typical monthly change size" or "typical number of changes in a commit", that we can use for prediction.

Limitations exist in our simulations, however. The change size distributions in the simulated evolutions followed power law throughout the full range (Fig. 4a), while in reality only the tails follow the power law (Fig. 1), starting at x_{min}, which is greater than 100 in all four projects studied. Towards the left of x_{min}, change size follows different distributions, possibly log-normal distributions, like Herraiz et al. [6] found in file size distributions. More work is needed to understand the forces that deviate change size from power law distribution at the lower end.

The fitness value of a method simplifies the driving forces of software evolution in our simulations. It is not clear yet how the value could be obtained in real software evolution.

B. Preferential Attachment During Simulations

The use of preferential attachment in growing the software system during the simulations turned out to be successful, as

it both helps to organize the system into critical states to allow punctuations, and produce degree distributions similar to real software systems.

The preferential attachment of method calls and subclasses is based on the assumption that existing method calls and subclasses indicate the usefulness of a method or class. If a method or class is useful at present, it is likely to be used in the future too. The preferential attachment of methods to classes can be thought of as modeling the effort to place a new method closer to the methods and data it uses (e.g., in order to increase cohesion and reduce coupling). So the larger a class is, the more likely one or more of its methods and attributes will be used in a new method. So the new method is more likely to be added to the class.

Software can be modeled with the dependency network of its components. How well the network generated by preferential attachment in this study resembles the dependency networks in real software systems deserves more investigation. The properties of software dependency networks discovered by Myers [14] provide good pointers for future improvements to our model.

C. Software Engineers as a Generative Process

The simulation we present is an abstract model of the kinds of changes software developers make on a daily basis as they work on a large project. In actual project work, instead of following a statistical random process, developers make rational decisions on which files to modify, and how many to modify in a single change. Despite not trying to create them, somehow the accumulated effect of all of these rational choices results in the observed power law distributions. It would be satisfying to connect these small scale rational decisions to the random generative model presented herein, and show how intentional decisions accumulate to produce power laws.

VI. Threats to Validity

The possible threats to the validity of this study are:

1) Construct Validity: The threats to construct validity of this study reside in the question of how well the rules and processes used to produce power laws in our simulation reflect the reality of software development. We believe all of the rules are based on reasonable assumptions about software development. Although more empirical studies are needed to verify these assumptions, we were not able to find any evidence to the contrary either.

2) Internal Validity: Since many details in software evolution are removed from our simulation model, leaving only the processes directly related to our study, it is unlikely for other mechanisms than our proposed ones to produce the distributions in our experiment.

3) Statistical Validity: We have to admit that, technically, not being able to reject power law does not confirm that power law *is* the true distribution of any the data set in Section II and IV. However, several different statistical procedures were taken to reduce the risk of our assumption about power law.

VII. Related Work

The prevalence of the power law distribution has attracted researchers from different areas to explore underlying mechanisms. Newman [1] summarizes six generative mechanisms for power law distributions, among which are preferential attachment and self-organized criticality. Integrating preferential attachment and self-organized criticality in one simulation model has not been tried to the best of our knowledge.

Models for self-organized criticality have been proposed in many different areas. Our model is inspired by the Bak-Sneppen model for biological evolution [13] and the forest-fire model [15]. The Bak-Sneppen model was also modified and used to describe software evolution by Gorshenev et al. [2]. Instead of simulating the evolution, they performed a mathematic analysis for a simpler version of their model, and proved that the change size, in number of lines changed, follows a power law distribution. Cook et al. [16] also tried to adapt Bak-Sneppen to simulate software growth. They were able to produce punctuated equilibrium but failed to produce any power law behavior, which essentially means their system did not organize itself into a critical state.

Modeling software systems as complex networks and simulating the network evolution are not unprecedented either. Myers [14] proposed an abstract model based on string splitting to simulate refactoring. The model is able to capture many salient features of software systems, such as in-degree and out-degree distributions in software dependency networks, as well as the relation between degrees and clustering coefficient in different nodes. His model could be adapted to complement our model. Turnu et al. [4] used a preferential attachment process to produce the distributions of some object-oriented measures. They also proposed methods to estimate parameters of a preferential attachment process from empirical data.

From a very different perspective, Hatton [17] built a statistical mechanical model to prove that, given a fixed total number of bugs and system size, a software system is most likely to be organized in such a way that its component sizes will follow a power law distribution. However, such assumptions may not hold for many software systems. For example, it was found that the size of Linux kernel grew super-linearly [18].

VIII. Conclusion

The distribution of many software system measures have been found to follow a power law distribution, at least in their tails. Few explanations have been provided for such distributions.

In this article, we have presented a model to simulate fine-grained source code changes based on preferential attachment and self-organized criticality. Our experiment shows that the model, running under some reasonable rules, is able to modify software and produce power law distributions in several measures. The results indicate that preferential attachment and self-organized criticality may be the driving forces behind the power law distributions of measures such as file size, change size and in-degrees of methods.

REFERENCES

[1] M. E. J. Newman, "Power laws, Pareto distributions and Zipf's law," *Contemporary Physics*, vol. 46, pp. 323–351, 2005.

[2] A. Gorshenev and Y. Pis'mak, "Punctuated equilibrium in software evolution," *Physical Review E*, vol. 70, no. 6, p. 067103, Dec. 2004.

[3] P. Louridas, D. Spinellis, and V. Vlachos, "Power laws in software," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 1, pp. 1–26, Sep. 2008.

[4] I. Turnu, G. Concas, M. Marchesi, S. Pinna, and R. Tonelli, "A modified Yule process to model the evolution of some object-oriented system properties," *Information Sciences*, vol. 181, no. 4, pp. 883–902, Feb. 2011.

[5] M. Fowler and K. Beck, *Refactoring: Improving the Design of Existing Code*, 1st ed., ser. Addison-Wesley Object Technology Series. Reading, MA: Addison Wesley, 1999.

[6] 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: SciTe Press, 2011.

[7] R. Kumar, P. Raghavan, S. Rajagopalan, D. Sivakumar, A. Tomkins, and E. Upfal, "Stochastic models for the Web graph," *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pp. 57–65, 2000.

[8] P. Bak, *How Nature Works: the science of self-organized criticality*. New York, NY, USA: Copernicus, 1996.

[9] J. Wu, R. C. Holt, and A. E. Hassan, "Empirical Evidence for SOC Dynamics in Software Evolution," in *2007 IEEE International Conference on Software Maintenance*. IEEE, Oct. 2007, pp. 244–254.

[10] 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*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[11] A. Clauset, C. R. Shalizi, and M. E. J. Newman, "Power-Law Distributions in Empirical Data," *SIAM Review*, vol. 51, no. 4, pp. 661–703, Nov. 2009.

[12] C. S. Gillespie, "Fitting heavy tailed distributions: the poweRlaw package," *Journal of Statistical Software*, 2014.

[13] P. Bak and K. Sneppen, "Punctuated equilibrium and criticality in a simple model of evolution," *Physical Review Letters*, vol. 71, no. 24, pp. 4083–4086, Dec. 1993.

[14] C. Myers, "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs," *Physical Review E*, vol. 68, no. 4, p. 046116, Oct. 2003.

[15] B. Drossel and F. Schwabl, "Self-Organized critical Forest-Fire Model," *Physical Review Letters*, vol. 69, no. 11, pp. 1629–1632, 1992.

[16] S. Cook, R. Harrison, and P. Wernick, "A simulation model of self-organising evolvability in software systems," in *Proceedings of the 2005 IEEE International Workshop on Software Evolvability*, 2005, pp. 17–22.

[17] L. Hatton, "Power-Law Distributions of Component Size in General Software Systems," *IEEE Transactions on Software Engineering*, vol. 35, no. 4, pp. 566–572, Jul. 2009.

[18] M. Godfrey and Q. Tu, "Evolution in open source software: a case study," *Software Maintenance, 2000. Proceedings. International Conference on*, pp. 131–142, 2000.