Studying the Chaos of Code Development

Ahmed E. Hassan and Richard C. Holt
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Canada
{aeehassa,holt}@plg.uwaterloo.ca

ABSTRACT
As large software systems evolve, controlling their complexity is a major challenge for many companies, as they strive to deliver future releases on time and within budget. Several source code based metrics have been proposed to assist in determining the complexity of code to help control development costs and outcome.

In this paper we offer a novel view on the problem of complexity in software. We present a complexity metric that is based on the process followed by the developers to produce the code instead of on the code directly. We conjecture that a chaotic/complex development process negatively affect its outcome, the source code. We validate our hypothesis empirically using data derived from the development process history of six large open source projects (three operating systems: *NetBSD*, *FreeBSD*, *OpenBSD*; a window manager: *KDE*; an office productivity suite: *KOffice*; and a database management system: *Postgres*).

1 INTRODUCTION

The complexity of a software project has always been a central focus by stakeholders involved in producing software systems. Stakeholders, such as developers and managers, aim to keep the complexity of the project under control. Their natural conviction is that complexity has many negative effects on the project. Developers focus on reducing the complexity of the source code. A less complex source code base is easier to maintain, and should have less faults over time. Requirements engineers focus on reducing the complexity of the requirement provided by the customers. A simple set of requirements will ensure a simple and easy implementation that is delivered on time and within budget. An architect focuses on producing a less complex design for the system to ensure its successful evolution as it adapts to the changes needs of customers. A manager focuses on ensuring that the overall project complexity is limited in all its facets from requirements, all the way to source code. Controlling complexity helps the software system evolve gracefully to fulfill customers' changing requirements. Failing to control the complexity, the software system will no longer satisfy the needs of its users and will be abandoned [21].

The idea of complexity has different connotations for each of the stakeholders in a project, but they all agree that limiting the complexity has many beneficial effects. In our work, we focus on the complexity of the modifications done to the source code of a software system. We refer to this complexity as the *complexity* or *chaos of the code development process*. We conjecture that a software system with a chaotic code development process is undesirable. It will produce code that has many faults and the project overall will face many delays [16]. To define the chaos in the code development process we study the code modifications, and their pattern. We employ concepts from information theory to rigourously define our intuition about complexity.

Our interest in studying and quantifying the complexity of the code development process stems from our belief that this process plays a central role in a software project. The process is responsible for producing the code needed to satisfy the requirements of the customer while dealing with the complexity and challenges associated with the current code and the other facets of the project. Figure 1 shows various facets in a software project. Each facet represents a source of complexity in a software project. All facets are tightly intertwined. They interact heavily and affect each other. Complexity in one facet flows to another facet and affects it. Schedule pressure, requirements, and team structure affect the complexity of the code development process. As the complexity of the code development process increases, it negatively affects the source code whose complexity increases as well. Over time the complexity of the source code and its design will increase in return the complexity of the development process increases, creating a feedback loop.

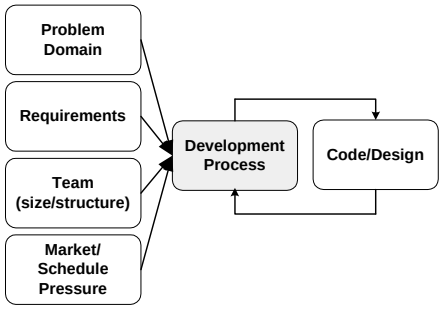

Figure 1: Flow Of Complexity Between the Facets of a Software Project

Referring to Figure 1, there are many facets where we can measure the complexity of the project. Most research related to complexity has focused on the design of metrics which are based on the source code. For example, metrics are defined to measure the number of tokens in the code or the Cyclomatic complexity of the looping and conditional constructs in the code [23]. Some researchers have focused on the development of methodologies and techniques to measure the complexity of the design and architecture [33, 28, 17, 39]. Other researchers have focused on quantifying the complexity of the requirements. Though many of these approaches have provided promising results, we believe they have certain limitations.

Measuring the complexity of the requirements is usually done early in the project before the requirements are fully understood and detailed. We believe that in many cases these measurements are too remote from the code and may not reflect the actual complexity that the project will have to deal with. For example, even though requirements may be simple, the source code may be overly complicated. Complexity metrics based on the source code have their limitations too, in that they measure the complexity after it has occurred. They do not provide an early warning system to avoid future complexity. Instead they are mainly indicators that the source code is in need of refactoring, repair, or is no longer maintainable and must be re-engineered.

As a software system evolves, the unwarranted difficulties associated with adapting it to the evolving requirements are a good indicator of the complexity associated with the source code in particular and the project's overall state. Ideally, a metric should provide a good measure of the complexity that the overall project is dealing with. The metric should be based on a better and more detailed information about the current development tasks at hand. At the requirement level, too little is known. At the code level, it is usually too late to react to the measurement results. We believe that a complexity metric based on the complexity or chaos of the code development process promises to address many of the aforementioned concerns.

Overview of Paper

In this paper, we present a novel view of complexity in a software project. We focus on the complexity of a central facet of the project, the ongoing code development process. We study the pattern of modifications to the code during the development and conjecture that:

> *A chaotic code development process negatively affects its outcome, the source code.*

Section 2 gives our view of the code development process. We turn our attention to source control systems used by large software projects. Source control system provide a convenient repository of data to study the code development process. We survey the various data stored in source control repositories. Then we present an overview of research which uses this data to gain a better understanding of large software systems and to predict faults.

Section 3 presents the mathematical concepts needed for measuring the chaotic nature of the code development process. In particular, we introduce information theory and Shannon's entropy.

Section 4 introduces our first and simplest model for code development process complexity – *The Basic Code Development (BCD) Model*. We then proceed to give a more elaborate and complete model in Section 5 – *The Extended Code Development (ECD) Model*.

In [16], we evaluated a preliminary model similar to the ECD model, presented in Section 5, using an observational study. We built an explanatory theory that showed that our preliminary model is able to clarify events in the history of a project such as delays in releases. In Section 6, we provide a more mathematically sound and elaborate validation of our ECD model. We first reformulate the ECD model and develop a finer grained model – *The File Code Development (FCD) Model*. The FCD Model measures the effects of the chaotic nature of development process on individual source code files. We then show using regression analysis that our FCD Model provides a good indicator of the number of faults occurring in an evolving software system better than simply using the number of the modifications to a file as an indicator.

Section 7 showcases related work in the field of software evolution, entropy, and open source systems. Section 8 summarizes our results and presents plans for future work.

2 CODE DEVELOPMENT PROCESS HISTORY

We use the term *code development process* to mean the pattern of modifications to the source code of a project. The modifications are done by the developers to implement new features and repair faults. By studying these patterns of modifications and quantifying their degree of complexity over time (using our defined model), we hope to achieve a better understanding of the evolution of complexity in the source code in particular and the overall project.

Luckily, large software project generally contain a convenient record keeping system which tracks modifications to the source code over time. Source control systems are used extensively by large software projects to control and manage their source code [32]. They help coordinate the development process between the various members of the team and provide the ability to restore the state of the source code to its state at any given time in the past. For example, developers can retrieve a source code file that is no longer part of the project or roll back to a previous version of a file if they discovered that their changes are inappropriate or are too complex to maintain and understand. Furthermore, source control systems provide tools to reconcile changes made by

developers working simultaneously on the same file. The most frequently used source control systems are RCS [32], CVS [7, 12] and Perforce [29].

The repository of a source control system contains various details about the development history of each file in the project. It contains the creation date of the file, its initial content and a record of each modification done to the file. A *modification record* stores the date of the modification, the name of the developer that performed the changes, the line numbers that were changed, the actual lines of code that were added or removed, and a detailed message entered by the developer explaining the reasons for the change.

In this paper, we use a lexical technique, similar to [25], to automatically divide modifications into three types based on the content of the detailed message attached to a modification:

Fault Repairing modifications (FR): These are all the modifications which contain terms such as bug, fix, or $repair$ in the detailed message attached to modification. These modifications are not used in the calculation of the development process complexity. But they are used in the validation process presented in Section 6.

General Maintenance modifications (GM): These are modifications that are mainly bookkeeping modifications and do not reflect the implementation of a particular feature. These modifications are removed from our analysis and are never considered. For example, modifications to update the copyright notice at the top of each source file are ignored. Modifications that are re-indentation of the source code after being processed by a code beautifier pretty-printer are ignored as well.

Feature Introduction modifications (FI): These are the modifications that are not FR or GM modifications. These modifications are used in the the calculation of the development process complexity.

Data stored in the source repository presents a great opportunity to study the code development process and validate our hypothesis. The data collection costs are minimal as it is collected automatically as modifications are done to the source code. Other researchers have used the source code repository to explain and validate their ideas. For example, Eick $et\ al.$ studied the concept of code decay and used the modification history to predict the incidence of faults [10, 11]. Graves $et\ al.$ showed that the number of modifications to a file is a good predictor to the fault potential of the file [14]. In other words, the more the file is changed the higher the probability it will contain faults. Furthermore, they showed that more recent changes contribute more to the bug potential than older changes over time. Michail $et\ al.$ presented a case study for a source code searching tool that makes use of developer's comments associated with each modification to the code [6]. The tool uses the comments to index the source code to provide more accurate search results, when developers search for the location where specific features are implemented in the code. Gall $et\ al.$ examined the source repository of a telephony system to detect logical coupling between the different components of the software system, discovering that files, which implement similar concepts, have a higher tendency of being modified in close proximity in time to each other [13].

In this paper, we promote a new perspective on the complexity of the development process of a software project as it evolves to satisfy the needs of its customers. We conjecture that a software system becomes complex to manage and maintain when its code development history becomes too complex to easily comprehend and track. A software system which has to endure highly scattered modifications to its code base as it implements the requirements of its customers, will have a high tendency of becoming a complex project. In contrast, a project where modifications are limited to specific spots in the code will have less complexity associated with it.

Various observations by Brooks support our intuition and our model [27]. In particular, Brooks warned of the decay of grasp of what is going in a complex system. A complex modification pattern will cause delays in release, high bug rates, stress and anxiety to all the personnel involved in the project. As the ability of team members to understand the changes to the system deteriorates so does their knowledge of the system. New development performed by them will be negatively affected. In short, a chaotic code development process is a good indicator of a complex code base and many other project problems.

In this section, we advocated the need to measure the complexity of the development process and the amount of predictability and chaos that is associated with code modifications as a system evolves. We have yet to explain how we can measure such complexity. In the following section we introduce concepts from information theory that will form the mathematical basis of our model.

3 INFORMATION THEORY

In 1948, Shannon laid down the basis of *Information Theory* in his seminal paper - *A mathematical theory of communication* [36]. Information theory deals with assessing and defining the amount of information in a message. The theory focuses on measuring uncertainty which is related to information. For example, suppose we monitored the output of a device which emitted 4 symbols, A, B, C, or D. As we wait for the next symbol, we are uncertain as to which symbol it will produce ($ie.$ we are uncertain about the distribution of the output). Once we see a symbol outputted, our uncertainty decreases. We now have a better idea about the distribution of the output; this reduction of uncertainty has given us information.

Shannon proposed to measure the amount of uncertainty/en-

tropy in a distribution. The ***Shannon Entropy***, H_n is defined as:

$$H_n(P) = -\sum_{k=1}^{n}\Big(p_k * \log_2 p_k\Big),$$

where $p_k \geq 0, \forall k \in 1, 2, ..., n$ and $\sum_{k=1}^{n} p_k = 1$.

For a distribution P where all elements have the same probability of occurrence ($p_k = \frac{1}{n}, \forall k \in 1, 2, ..., n$) , we achieve maximum entropy. On the other hand for a distribution P where one of the elements i has probability of occurrence $p_i = 1$ and all other elements have probability of occurrence equal to zero ($ie.\ p_k = 0, \forall k \neq i$), we achieve minimal entropy.

By defining the amount of uncertainty in a distribution, H_n describes the minimum number of bits required to uniquely distinguish the distribution. In other words, it defines the best possible compression for the distribution (ie. the output of the system). This fact has been used to measure the quality of compression techniques against the theoretically possible minimum compressed size.

4 THE BASIC CODE DEVELOPMENT MODEL

If we view the development process of a software system as a system which emits data, and we define the data as the FI modifications to the source files, we can apply the ideas of information theory and entropy to measure the amount of *uncertainty/chaos/randomness* in the development process. This section presents the Basic Code Development (BCD) Model for the entropy of software development and its evolution. The following section extends the model to be more elaborate and complete.

Basic Model

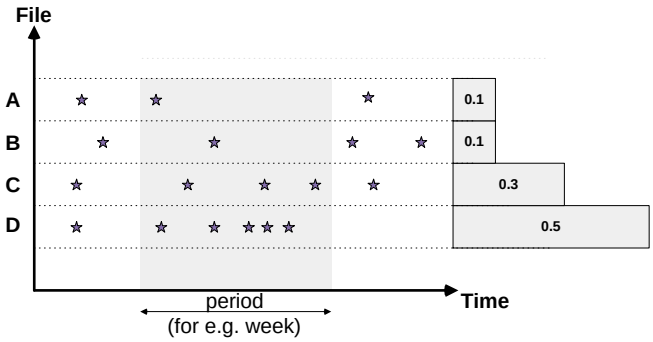

Figure 2: The Entropy of a Period of Development

Suppose we have a software system which consists of four files. If we were to examine the development history of this system that is stored in a source repository, we will find for each file the dates for each modification to the file and the reason for modifying the file. We only concentrate on FI modifications. The FI modifications are extracted automatically and the other types of modification are ignored.

Once the FI modifications are extracted, we can plot for each file the moments the file was modified. As can be seen in Figure 2, we put stars to indicate that for a specific file, it was modified on a particular moment in time. We now define a period of time, for example a week, or a month. For that period of time, we can define a file modification probability distribution P. P gives the probability that $file_i$ is modified in a period. For each file in the system, we count how many times it was modified during the period and divide by the total number of modifications in that period for all files. For example, in Figure 2, in the highlighted grey period we have 10 modifications for all the files in the system. $file_A$ was modified once so we have a $p(file_A) = \frac{1}{10} = 0.1$. For $file_B$ we get $p(file_B) = \frac{1}{10} = 0.1$, for $file_C$ we get $p(file_C) = \frac{3}{10} = 0.3$, and so on. On the right side of Figure 2, we can see a graph of the file modification probability distribution P for the shaded period.

If we monitor the modifications to the files of a software system and find that the probability of modifying $file_A$ is 1 and all other files is zero, then we have minimal entropy. On the other hand, if the probability of modifying each file is equal (ie. $file_k = \frac{1}{n}$) then the amount of entropy/chaos in the system is at its maximum. Intuitively, if we have a software system that is being modified across all of its files, the developers and the managers will have a hard time keeping track of all the modifications. The number of bits needed to remember all these modifications in their heads will be much larger than the bits needed when a small number of files have been modified. The development team grasp of what is going on in the software system will decay.

Instead of simply using the number of modifications to the file, we use the number of lines modified over the period to build the file modification probability. The lines changed in a modification is the the sum of the lines added and deleted as described in the modification record.

Files As a Unit Of Measurement

In the BCD model we use the file as our unit of code to build the modification probability distribution P for each period. Other units of code can be used, such as functions or code chunks that are determined by a personal with high knowledge of the system. Our choice of files is based on the belief that a file is a conceptual unit of development where developers tend to group related entities such as functions, data types, $etc.$ Based on our experience in studying large software system we found this to be the norm with some notable exceptions. For example in the VIM text editor [34], we found two files $misc1.c$ and $misc2.c$ which comprise a substantial amount of the source code that is not related.

Evolution of Entropy

We can view the file modification probability distribution P_j for a period j, as a vector which characterizes the system and uniquely identifies its state. We can divide the lifetime of a software system into successive periods in time, and view the

evolution of a software system as a the repeated transformation of the development process from one state to the next. Looking at Figure 3, we can see the P_j's calculated for 4 consecutive periods with their respective entropy. This allows us to monitor the evolution of chaos/entropy in the development process. If the project and the development process are not under control nor managed well, then state of the system will head towards maximum entropy/chaos.

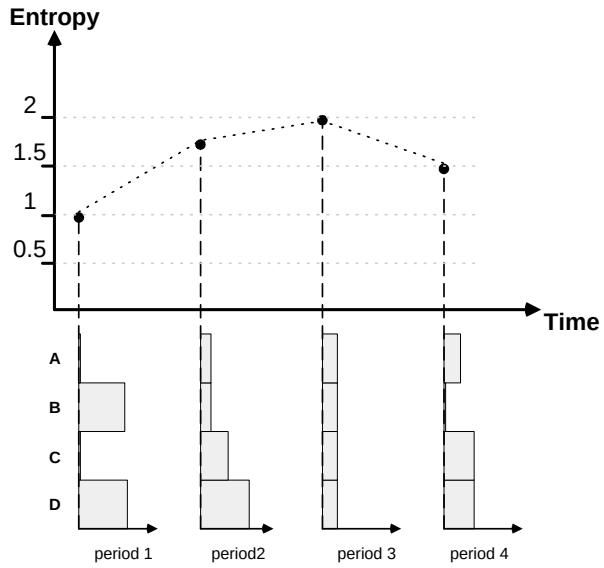

Figure 3: The Evolution of the Entropy of Development

The manager of a large software project should aim to control and manage the entropy. A graph like the one shown in Figure 3 provides an up-to-date view of the status and evolution of entropy in the system. Searching for unexpected spikes in entropy and investigating the reasons behind them would let managers plan ahead and be ready for future problems. For example, a spike in entropy may be due to an influx of developers working on too many aspects of the system concurrently, or to the complexity of the source code or to a refactoring or redesign of many parts of the system. In the refactoring case, the manager would expect the entropy to remain high for a limited time then drop as the refactoring leads to easier future modifications to the source code. On the other hand, complex source code may cause a consistent rise in entropy over an extended period of time, till issues causing the rise in entropy/complexity are addressed and resolved.

5 EXTENDED CODE DEVELOPMENT MODE

In this section, we extend our BCD model to address some of the characteristics and challenges associated with the evolution of large software systems. In the BCD model we used a fixed period size to measure the evolution of entropy. Also we assumed that the number of files in a software system remains fixed over time. The Extended Code Development (ECD) model presented in this section deals with these limitations.

Evolution Periods

In our BCD model, we presented the idea of using the file modification probability distribution as a vector to characterize each period in our study of the evolution of a software system. We used fixed length periods such as a month, or a year. We now present more sophisticated methods for breaking up the evolution of a software projects into periods:

Time based periods: This is the simplest technique and it is the one presented in the BCD model in Section 4. The history of development is broken into equal length periods based on calendar time from the start of the project. For example, we break it on a monthly or bi-monthly basis. A project which has been around for one year, would have 12 or 6 periods respectively. We chose a 3 month period in our experiments.

Modification limit based periods: The history of development is broken into periods based on the number of modifications to files as recorded in the source control repository. For example, we can use a modification limit of 500 or 1000 modifications. A project which has 4000 modifications would have 8 or 4 periods respectively. To avoid the case of breaking an active development week into two different periods, we attach all modifications that occurred a week after the end of a previous period to that period. To prevent a period from spanning a long time when little development may have occurred, we impose a limit of 3 months on a period even if the modification limit was not reached. We choose in our experiments a limit of 600 modifications.

Burst based periods: Based on studying the development history for several large software systems, we observed that the modification process is done using a bursty pattern. Over time, we see periods with many code modifications then they are followed by short periods of no or little code modifications. We chose to use that observation to automatically break up the history into periods. If we find a period of a couple of hours where no code modifications have occurred, we consider all the previous code modifications to be part of the previous period and we start a new period. In this paper, we focus mainly on using this period creation method in our analysis.

Moving Window Period Sampling

To improve the continuity of our sampling of the evolution of a large project, we employ a moving window technique for our period creation in when the time based or the modification based techniques are used. For example, using a time based period sizing of 3 months, we would break a project that is one year old into 4 periods. The first period would start at month 1 and go to month 3, the second period would go from month 3 till month 6, and so on. When we use a moving window for our periods, we start with a first period that has modification data from months 1, 2, and 3. Then

the second period would have data from months 2, 3, and 4. The third period would have data from months 3, 4, and 5. This overlap of period data permits the generated data to be smoother and accurate as it is more continuous. Instead of a discrete breakdown that just takes 4 snapshots in a year, we take 12 snapshots. We use a similar moving window technique for the modification limit based period sizing, namely, we have a window of 600 modifications and we move it 300 modifications for each period in our experiments.

Adaptive System Sizing
As a software system evolves, the number of files in it changes; increasing as new files are added, or split, and decreasing as files are removed, or merged. We need to adjust our entropy calculations to deal with the varying number of files in a software system.

To compare the entropy when there is a varying number of files in the the software system, we define H, which we will call ***Standardized Shannon Entropy*** as:

$$\begin{aligned} H(P) &= \frac{1}{Max\ Entropy\ for\ Distribution} * H_n(P) \\ &= \frac{1}{\log_2 n} * H_n(P) \\ &= -\frac{1}{\log_2 n} * \sum_{i=1}^{n} \Big(p_k * \log_2 p_k \Big), \end{aligned}$$

where $p_k \geq 0, \forall k \in 1, 2, ..., n$ and $\sum_{k=1}^{n} p_k = 1$. The standardized entropy H normalizes Shannon's entropy H_n, so that $0 \leq H \leq 1$. We now can compare the entropy of distributions of different size, such is the case when we examine the various periods of a software system as new files are added or removed. It is interesting to note that using standardized Shannon entropy H we can now compare the entropy between different software projects. Thus we can compare the evolution of two operating systems side by side or even an operating system and a window manager.

The Standardized Shannon Entropy, H, is dependent on the number of files in the software system, as it depends on n. Unfortunately, for many software system there exist files that are rarely modified, for example, platform and utility files. To prevent these files from reducing the standardized entropy measure, we defined a working set standardized entropy H' – *adaptive sizing entropy.* In H' instead of dividing by the actual current number of files in the software system, we divide by the number of recently modified files. We define the set of recently modified files using 2 different criteria:

Using Time: The set of recently modified files is all the modified files in the preceding x months, including the current month. In our experiments we used 6 months.

Using Previous Periods: The set of recently modified files is all the modified files in the preceding x periods, including the current period. We don't show results from using this model in this paper but in our experiments we used 6 periods in the past to build the working set of files.

As we have two different criteria to create a period based on size, then we have two different results based on the use of a time based or a modification limit period creation models.

An adaptive sizing entropy H' usually produces a higher entropy than a traditional standardized entropy H, than H, as for most software systems there exists a large number of files that are rarely modified and would not exist in the recently modified set. Thus the entropy would be dividing by a much smaller number. In some rare cases, where the software system has undergone a lot of changes/refactoring it may happen that the size of the working set is larger than the actual number of the files that currently exist in the software system, as many files may have been removed recently as part of a cleanup. In that rare case, an adaptive sizing entropy H' will be larger than a traditional standardized entropy H.

6 MODEL VALIDATION

A preliminary version of the ECD Model presented in Section 5 was used to explore problems encountered by several large open source projects during their evolution [16]. We correlated events in the project's history and comments in their release notes with variations in the complexity value calculated by our ECD model. Our observational study illustrated that our ECD model shows promise in being a good indicator of project's health and the amount of complexity associated with the code base.

In this section, we provide a more mathematically sound and elaborate validation of our ECD model using data derived from several large open source system, shown in Table 1.

Application Name	Application Type	Start Date	Subsystem Count	Prog. Lang.
NetBSD	OS	21 March 1993	235	C
FreeBSD	OS	12 June 1993	152	C
OpenBSD	OS	18 Oct 1995	265	C
Postgres	DBMS	9 July 1996	82	C
KDE	Windowing System	13 April 1997	108	C++
Koffice	Productivity Suite	18 April 1998	158	C++

Table 1: Summary for the Six Studied Systems

We aim to validate our conjecture that: *A chaotic code development process negatively affects its outcome, the source code.* These chaotic code changes which require modifications of many subsystems introduce more faults than localized changes. As presented, the ECD model produces a metric that quantifies the state of chaos/complexity in the code development process for the whole source code base. We

will now reformulate the ECD Model so the ECD system complexity metric applies to a single file or to a subsystem, which contains a set of files. We call that reformulation the *File Code Development (FCD) Model*.

THE FILE CODE DEVELOPMENT MODEL (FCD)

We conjecture that files that are modified during high complexity/chaotic development periods, as determined by our ECD Model, will have a higher tendency to contain faults. We define the *History Complexity Metric (HCM)* for each file in a software system. The HCM assigns to a file the effect of the complexity of a period, as calculated by our ECD model. A file that has been modified during periods of high complexity/entropy will have a high HCM value to indicate that the file will tend to be more complex and more prone to faults.

Given a period i, with entropy H_i where a set of files, F_i are modified with a probability p_j for each file $j \in F_i$, we define *History Complexity Period Factor ($HCPF_i$)* for a file j during period i as:

$$HCPF_i(j) = \begin{cases} p_j * H_i, & j \in F_i \\ 0, & otherwise \end{cases}$$

More elaborate definitions of $HCPF$ are possible but this is beyond the scope of this paper and our validation process. We use this simple $HCPF$, which assumes if a file is modified during a period i then its gains a percentage of the complexity associated to that period. The percentage is the probability of file j being modified during period i.

Now we define the *History Complexity Metric (HCM)* for a file j over a set of evolution periods $\{a, .., b\}$ as:

$$HCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} HCPF_i(j)$$

More elaborate definitions of HCM are possible but are beyond the scope of this paper. We use the simple HCM definition to indicate that complexity associated to a file keeps on increasing over time, as a file is modified.

We define the HCM for a subsystem S over a set of evolution periods $\{a, .., b\}$ as the sum of the $HCMs$ of all the files that are part of that subsystem:

$$HCM_{\{a,..,b\}}(S) = \sum_{j \in S} HCM_{\{a,..,b\}}(j)$$

If a file were to move from one subsystem to another during the studied evolution period, the moved file would contribute to the HCM of its old subsystem till the time it was moved. Then it would contribute to its new subsystem afterwards. Using the HCM for a subsystem, we proceed to validate that HCM is better indicator of faults in a software system compared to just using the number of modifications. This validation provides a concrete substantiation our ECD model of software complexity.

VALIDATION PROCESS

To validate that HCM is a better indicator of faults than the number of modifications, we built two Statistical Linear Regression ($SLR\ Model$) models for each software system in Table 1:

$SLR\ Model_A$**:** uses the number of FI modifications to determine the number of faults.

$SLR\ Model_B$**:** uses HCM calculated from the FI modifications, to determine the number of faults.

We measured the accuracy of each SLR model and determined that $SLR\ Model_B$ is a better indicator of faults.

Statistical Linear Regression Model Building

To build the two SLR models we employed a statistical procedure called *cross-validation* [31]. We used two different sets of data:

- A $training$ set to build each SLR model, and
- A $testing$ set to compute the accuracy of the built SLR model.

We used a statistical *cross-validation* procedure to test the accuracy of our built SLR models instead of using the same data for SLR model building and accuracy testing. This procedure gives a more realistic view of the accuracy of our SLR models [35]. We note that a similar approach was used by [18] and [14].

We based our study on the first five years in the life of each studied open source project. For the training set, we used data collected from the second and third year from the source control repository. For the testing set, we used data collected from the fourth and fifth year. We chose to ignore the first year in the source control repository, due to the special startup nature of code development during that year as each project initializes its repository.

The SLR models have the following form, where y is the dependant variable and x is the predictor/independent variable:

$$y = \beta_0 + \beta_1 x$$

For each model, y represents the number of faults in a subsystem. This is determined based on the number of FR modifications in the source control repository. As for x,

$SLR\ Model_A$**:** x represents the number of FI modifications to a subsystem.

$SLR\ Model_B$**:** x represents the HCM for each subsystem. The HCM is based on a the ECD bursty model that has a one hour quiet time between bursts.

To ensure the mathematical validity of our SLR models, we actually use the mathematical log of the number of modifications, and the mathematical log of faults instead for x. The use of a log transformation ($i.e.$ $log(number\ of\ modifications)$ stabilizes the variance in the error for each data point in the SLR model, a requirement for linear regression models which assume that the error variance is always constant [37].

The SLR model parameters are estimated using the $training$ data (year 2 and 3). Table 2 shows the SLR model for each software system, and their estimated β_0 and β_1 parameters. It also shows the R^2 statistic to indicate the quality of the fit. The better the fit, the higher R^2. A zero R^2 indicates that there exits no relationship between the dependant y and independent variable x. We notice that $SLR\ Model_B$ has a better fit for all of the studied software systems. For example, for $NetBSD$ we see an R^2 that is 0.6392 for $SLR\ Model_A$ vs. an R^2 that is 0.7535 for $SLR\ Model_B$, indicating that $SLR\ Model_B$ has a better fit.

Application	SLR Model	SLR Model Parameters		
		β_0	β_1	R^2
NetBSD				
	(A)	-1.53370	0.51029	0.6392
	(B)	0.36481	1.27236	0.7535
FreeBSD				
	(A)	-1.5070	0.6084	0.7094
	(B)	0.61611	1.34476	0.7949
OpenBSD				
	(A)	-0.69881	0.33439	0.5695
	(B)	-0.49550	0.68323	0.6653
Postgres				
	(A)	-0.69881	0.33439	0.5685
	(B)	0.07656	1.07858	0.7221
KDE				
	(A)	-2.33638	0.74495	0.771
	(B)	0.69540	1.35726	0.7898
Koffice				
	(A)	-1.12799	0.57641	0.6357
	(B)	0.88669	1.48474	0.7588

Table 2: SLR model Details for the Studied Software Systems

Statistical Linear Regression Model Accuracy Testing
Once the SLR models were built we validated their accuracy using the $testing$ data set. The $testing$ data set is based on the fourth and fifth year for each software system.

Once we estimated β_0 and β_1 for $SLR\ Model_A$ and $SLR\ Model_B$ for each system, we used our built SLR models to determine/predict the number of faults occurring in subsystems during the fourth and fifth years. Mathematically for each model with β_0 and β_1 as parameters, we get a $\widehat{y_i}$ for each x_i, where $\widehat{y_i}$ is the number of expected faults in the subsystem in the fourth and fifth years:

$$\widehat{y_i} = \beta_0 + \beta_1 x_i$$

We define the absolute prediction error as

$$e_i = \mid \widehat{y_i} - y_i \mid$$

where y_i is the actual number of faults that occurred in subsystem i during the fourth and fifth year.

Thus the total prediction error of an SLR model is:

$$E = \sum_{i=1}^{n} e_i^2,$$

for all n subsystems in the software system under study. Table 3 shows the prediction error E details for each of the SLR Models. Also the table shows the percentage of improvement in prediction when $SLR\ Model_B$ is used instead of $SLR\ Model_A$. We notice that the $SLR\ Model_B$, which is based on HCM has a much lower error that the $SLR\ Model_B$, which is based on the number of modifications.

As we have two SLR Models that we use to determine faults, we can compare their accuracy. We define the accuracy of the built SLR Model to be the amount of error in prediction. The less the prediction error the higher the quality of an SLR model. For example, if E_B is the total prediction error of $SLR\ Model_B$, E_A is the total prediction error for $SLR\ Model_A$ and $E_B < E_A$, then $Model_B$ has better predictions than $Model_A$. $SLR\ Model_B$ has better accuracy than $SLR\ Model_A$. Table 3 shows the improvement in prediction error between the two SLR Models. For example for the $NetBSD$ software system, $Model_B$ has a 35% improvement/reduction in error when compared to $Model_A$.

Unfortunately, simply comparing the total prediction errors (E_A and E_B) is not sufficient. Instead we need to validate that the prediction error difference is statistically significant.

To perform the statistical test of significance, we use a statistical paired t-test and formulate the following test hypotheses:

$$\mathcal{H}_\prime : \mu(e_{A,i} - e_{B,i}) = 0$$
$$\mathcal{H}_\mathcal{A} : \mu(e_{A,i} - e_{B,i}) > 0$$

where $\mu(e_{A,i} - e_{B,i})$ is the population mean of the difference between the absolute error of each observation pair. As the data size is large enough (the smallest software system has over 80 subsystems) and the t-test is robust for non-normally distributed data, we can safely use a t-test. Alternatively, a non-parameterized test such as a U-test can be used for smaller software systems [22].

If the null hypothesis H_0 holds then the difference in prediction is not significant. Thus we need H_0 to be rejected, with

Application	$SLRModel$	$Error(E)$	Improv. in E	$P(\mathcal{H}_0\ holds)$
NetBSD				
	(A)	261.0058		
	(B)	169.4185	-35.1%	2.359e-05
FreeBSD				
	(A)	209.3327		
	(B)	152.0989	-27.3%	0.002553
OpenBSD				
	(A)	123.0210		
	(B)	79.21801	-35.6%	5.112e-05
Postgres				
	(A)	161.1841		
	(B)	97.13253	-39.7%	0.0001031
KDE				
	(A)	200.3708		
	(B)	109.0434	-45.6%	7.396e-07
Koffice				
	(A)	191.5776		
	(B)	166.0808	-13.3%	0.1584

Table 3: SLR Model Prediction Error for Each Studied Software System

a high probability. Looking at Table 3, we find that a t-test on paired observations of absolute error was significant at better than 0.002 for all systems, except for $Koffice$ which was significant at 0.15. We are over 99% confident that the improvement in prediction error between $Model_A$ and $Model_B$ is statistically significant for all the studied systems except for $Koffice$. For $Koffice$ we are only 85% confident that the improvement in prediction error is statistically significant. Therefore, we can reject the null hypothesis H_0 and conclude that $Model_B$, which is based on HCM, has significantly better accuracy than $Model_A$, which is based on the number of modifications. This confirms our conjecture that our entropy model is a good indicator of the state of chaos in the system and its complexity.

In this section, we have shown that the ECD model and HCM are good indicators of faults in large software systems. They are much better indicators than simply using the the number of modifications to a file to determine the number of faults. This finding leads us to a new direction as we continue the validation of our ECD model. We hope in future work to examine the predictive power of the ECD model. Can the HCM metric predict the likelihood of faults occurring in the future of the project? We have just shown that it is a good indicator of faults, the predictive power will need to be evaluated and tested against other well knows fault predictors.

7 RELATED WORK

In section 2, we presented an overview of previous work which analyzes source control repositories to gain a better understanding of the software system. Recent work by Barry $et\ al.$ uses a volatility ranking system and a time series analysis to identify evolution patterns in a retail software systems based on the source modification records [2]. Also Mockus $et\ al.$ uses source modification records to assist in predicting the development efforts in large software systems for AT&T [26]. Other than the work done by Michail $et\ al.$ [6], previous research has focused on studying the source code repositories of closed systems. We believe that:

- This focus on closed source systems may limit the applicability of the results. The results may be dependent on the studied system or organization as only a single system is used in the validation.
- By focusing on open source systems we are able to get a much larger set of systems to validate our findings and are more confident about our results.

We hope in future work to validate our findings against a large closed source systems to determine if our approach holds for closed source as well.

Whereas our model quantifies the complexity of the development process as calculated from the source code modification statistics, previous studies [1, 3, 4, 5, 15, 38] quantify the complexity of the source code. For example, in previous models the distribution of special tokens in the source code or the control flow structure of the source are used to calculate the entropy. Our work aims to compute a measure of chaos in the development process instead of just focusing on computing the complexity of the source code. We conjecture that detecting chaos in the code development process will serve as an early warning measure to help prevent the code from becoming too complex over time.

Work by Lehman $et\ al.$ [19, 20, 21] and Godfrey $et\ al.$ [24] focus on studying the evolution of size (number of modules) and LOC between releases of software systems. Instead we focus on measuring the entropy/chaos in the development process. Lehman's second law suggests the need for occasional maintenance activities (such as refactoring) to reduce complexity, which arises as new features are added to the system. The addition of features and their associated assumptions about the real world lead to the increase of complexity/entropy in the system unless work is done to reduce it. In our presented case studies, we quantified the idea of complexity/entropy in the development process using our new model and have shown that chaos is a good indicator of faults due to increase in complexity. Whereas our presented model studies the evolution of complexity over time, Lehman advocates studying the evolution of software over releases/versions. Our model can be extended to use releases as a unit of observation instead of time. This would permit us to compare our findings to Lehman's laws of evolution.

Outside of the software engineering domain, the measure of entropy has been used to improve the performance of Just In Time compilers and profilers [30]. It has been used for edge detection and image searching in large image database [9].

Also, it has been used for text classification and several text based indexing techniques [8].

8 CONCLUSIONS

In this paper, we advocate a new view on the complexity of software. We conjecture that: *A chaotic/complex code development process negatively affects its outcome, the source code.* We presented a model to quantify the complexity in a software system over time based on the source code modification history as stored in the source control repository.

In [16], we evaluated our model using an observational study. We correlated our measurements to events in the history of the studied software systems. Events such as large refactorings or delays in releases were accompanied with increases in our ECD model measurement.

In this paper, we sought to perform a more formal and concrete validation of our ECD model. We conjectured that files which were modified during periods of high system complexity, as calculated by our ECD model, will have a high degree of complexity associated with them. These files will eventually have a large number of faults in them over time.

To validate our conjecture, we extended our ECD model to develop a metric that associated complexity values to each file in the software system. Then for each studied software system, we built two statistical linear regression models (SLR models). One SLR model correlated faults and FI modifications. The other SLR model correlated faults and our newly developed complexity HCM metric. We showed that the SLR model based on our complexity metric has a statistically significant better accuracy than the SLR model based on modifications. Thus we believe that our complexity metric is a good indicator of complexity in large software systems. If monitored by developers it should help them to be in control of the project and avoid delays and faults over time.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge the significant contributions from the members of the open source community who have given freely of their time to produce large software systems with rich and detailed source code repositories; and who assisted us in understanding and acquiring these valuable repositories.

We would like to thank Stephen M. Sheeler for many engaging and fruitful discussions as we developed our BCD model.

REFERENCES

[1] S. Abd-El-Hafiz. Entropies as measures of software information. In *IEEE International Conference Software Maintenance (ICSM 2001)*, pages 110–117, Florence, Italy, 2001.

[2] E. J. Barry, C. F. Kemere, and S. A. Slaughter. On the uniformity of software evolution patterns. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 106–113, Portland, Oregon, May 2003.

[3] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio. Evaluating software degradation through entropy. In *Eleventh International Software Metrics Symposium*, pages 210–219, 2001.

[4] N. Chapin. An entropy metric for software maintainability. In *Twenty-Second Annual Hawaii International Conference on System Sciences, Software Track*, pages 522–523, Jan. 1995.

[5] N. Chapin. Entropy-metric for systems with COTS software. In *Eighth IEEE Symposium on Software Metrics*, pages 173–181, 2002.

[6] A. Chen, E. Chou, J. Wong, A. Y. Yao, Q. Zhang, S. Zhang, and A. Michail. CVSSearch: Searching through source code using CVS comments. In *IEEE International Conference Software Maintenance (ICSM 2001)*, pages 364–374, Florence, Italy, 2001.

[7] CVS - Concurrent Versions System. Available online at `<http://www.cvshome.org>`

[8] I. Dhillon, S. Manella, and R. Kumar. Information theoretic feature clustering for text classification.

[9] M. Do and M. Vetterli. Texture similarity measurement using kullback-leibler distance on wavelet subbands. In *IEEE International Conference on Image Processing (ICIP)*, Vancouver, Canada, Sept. 2000.

[10] S. G. Eick, T. L. Graves, A. F. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Trans on Software Engineering*, 27(1):1–12, 1990.

[11] S. G. Eick, C. R. Loader, M. D. Long, S. A. V. Wiel, and L. G. V. Jr. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, Melbourne, Australia, May 1992.

[12] K. Fogel. *Open Source Development with CVS*. Coriolos Open Press, Scottsdale, AZ, 1999.

[13] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *IEEE International Conference on Software Maintenance (ICSM98)*, Bethesda, Washington D.C., Nov. 1998.

[14] T. L. Graves, A. F. Karr, J. S. Marron, and H. P. Siy. Predicting fault incidence using software change history. *IEEE Transaction of Software Engineering*, 26(7):653–661, 2000.

[15] W. Harrison. An entropy-based measure of software complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029, Nov. 1992.

[16] A. E. Hassan and R. C. Holt. The Chaos of Software Development. In *IEEE International Workshop on Principles of Software Evolution (IWPSE03)*, Helsinki, Finland, Sept. 2003.

[17] R. Kazman, L. J. Bass, M. Webb, and G. D. Abowd. SAAM: A method for analyzing the properties of software architectures. In *International Conference on Software Engineering*, pages 81–90, 1994.

[18] T. M. Khoshgoftaar and E. B. Allen. Empirical assessment of a software metric: The information content of operators. *Software Quality Journal*, 9:99–112, 2001.

[19] M. M. Lehman. Programs, life cycles and laws of software evolution. *IEEE Transactions on Software Engineering*, 68:1060–1076, 1980.

[20] M. M. Lehman and L. A. Belady. *Program Evolution - Process of Sofware Change*. Academic Press, London, 1985.

[21] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution the nineties view. In *Fourth International Software Metrics Symposium (Metrics97)*, Albuquerque, NM, 1997.

[22] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Annals of Mathematical Statistics*, pages 52–54, Dec. 1947.

[23] T. J. McCabe. A complexity measure. *IEEE Transactions Software Engineering*, 2(6):308–320, 1976.

[24] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *IEEE International Conference on Software Maintenance (ICSM 2000)*, pages 131–142, San Jose, California, Oct. 2000.

[25] A. Mockus and L. G. Votta. Identifying reasons for software change using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 120–130, San Jose, California, Oct. 2000.

[26] A. Mockus, D. M. Weiss, and P. Zhang. Understanding and predicting effort in software projects. In *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, pages 274–284, Portland, Oregon, May 2003.

[27] J. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley Professional, 1995.

[28] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, pages 1053–1058, Dec. 1972.

[29] Perforce - The Fastest Software Configuration Management System. Available online at `<http://www.perforce.com>`

[30] S. Savari and C. Young. Comparing and combining profiles. In *Second Workshop on Feedback-Directed Optimization (FDO)*, 1999.

[31] J. Shao. Linear model selection by cross-validation. *Journal of American Statistical Association*, 88:486–494, 1993.

[32] W. F. Tichy. RCS - a system for version control. *Software - Practice and Experience*, 15(7):637–654, 1985.

[33] M. VanHilst and D. Notkin. Decoupling change from design. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 58–69, USA, 1996.

[34] The VIM (Vi IMproved) Home Page. Available online at `<http://www.vim.org>`

[35] F. Walkerden and D. R. Jeffery. An empirical study of analogy-based software effort estimation. *Empirical Software Engineering*, 4(2):135–158, June 1999.

[36] S. Weaver. *The mathematical theory of communication*. Urbana: University of Illinois Press, 1949.

[37] S. Weisberg. *Applied Linear Regression*. John Wiley and Sons, 1980.

[38] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, Sept. 1988.

[39] S. Woods and M. Barbacci. Architectural evaluation of collaborative agent-based systems, 1999.