Predicting Faults Using the Complexity of Code Changes

Ahmed E. Hassan
Software Analysis and Intelligence Lab (SAIL)
School of Computing, Queen's University, Canada
ahmed@cs.queensu.ca

Abstract

Predicting the incidence of faults in code has been commonly associated with measuring complexity. In this paper, we propose complexity metrics that are based on the code change process instead of on the code. We conjecture that a complex code change process negatively affects its product, i.e., the software system. We validate our hypothesis empirically through a case study using data derived from the change history for six large open source projects. Our case study shows that our change complexity metrics are better predictors of fault potential in comparison to other well-known historical predictors of faults, i.e., prior modifications and prior faults.

1 Introduction

Managing the complexity of a project is a paramount goal while striving to meet user needs. The literature contains a wealth of metrics (*e.g.* [19]) which measure the complexity of the source code. However, little attention has been paid to measuring and controlling the complexity of the code change process. This process plays a central role in a project since it is responsible for producing the code needed to satisfy requirements, while dealing with the complexities and challenges associated with the current code base and other facets of the project such as its design, customer requirements, the team structure and size, market pressure, and problem domain. A software system with a complex code change process is undesirable since it will likely produce a system which has many faults and the project will face delays.

Four lines of prior work motivate our intuition about the importance of the code change process and historical code changes in predicting the incidence of faults:

1. Studies by Briand *et al.* [2], Graves *et al.* [11], Khoshgoftaar *et al.* [20], Leszak *et al.* [22], and Nagappan and Ball [26] indicate that prior modifications to a file are a good predictor of its fault potential (*i.e., the more a file is changed, the more likely it will contain faults*).
2. Studies by Graves *et al.* [11] and Leszak *et al.* [22], on commercial systems, and recently by Herraiz *et al.* [18] on open source systems show that most code complexity metrics highly correlate with LOC, a much simpler metric.
3. Studies, such as the one by Moser *et al.* [25], show that process metrics outperform code metrics as predictors of future faults.
4. Studies, such as the one by Yu *et al.* [37], indicate that prior faults are good predictors of future faults.

In prior work, we used concepts from information theory to define change complexity models which capture our intuition about complex changes. Events such as large refactorings or release delays were accompanied with increases in our proposed model measurements [14, 15]. Our earlier results lead us to the following conjecture:

> *A complex code change process negatively affects its product, the software system. The more complex changes to a file, the higher the chance the file will contain faults.*

In this paper, we extend our change complexity models and study the ability of our proposed model measurements to predict the incidence of faults in a software system. In particular, we compare the performance of predictors based on our complexity models with predictors based on the number of prior modifications and prior faults. Based on a case study using six large open source projects, our results indicate that our change complexity models are better predictors of fault potential in contrast to other historical predictors (such as prior modifications and prior faults).

Overview Of Paper. This paper is organized as follows. Section 2 gives our view of the code change process. Section 3 present Shannon's entropy which we use to quantify the complexity of code changes. Sections 4, 5, and 6 present the complexity models we use in our work. Section 4 introduces our first and simplest model for the complexity of code changes – **The Basic Code Change (BCC) Model**. We proceed to give a more elaborate and complete model in Section 5 – **The Extended Code Change (ECC) Model**.

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

Both these models calculate a single value that measures the overall change complexity of a project during a particular time period. In Section 6, we reformulate the ECC model to introduce a finer grained model – **The File Code Change (FCC) Model**. The FCC model maps the overall complexity to individual source files or subsystems. In Section 7, we empirically compare the performance of predictors based on the FCC model with the performance of predictors based on the number of prior modifications and prior faults using data from six large open source projects. We end Section 7 with a critical review of our findings and their applicability to other software systems. Section 8 presents related work. Section 9 summarizes our findings.

2 The Code Change Process

We use the term **code change process** to mean the pattern of source code modifications. Modifications are done by developers to implement new features and repair faults. By studying these patterns and quantifying their degree of complexity over time (using defined models), we hope to achieve a better understanding of the complexity facing developers who are evolving and working on a project.

Large projects extensively use source control systems to control and manage their source code [30]. Data stored in these repositories presents a great opportunity to study the code change process and validate our ideas. The data collection costs are minimal since it is collected automatically as modifications are done to the code.

The repository of a source control system contains various details about the change history of every file in a project. It contains the creation date of a file, its initial content and a record of every modification done to the file. A **modification record** stores the date of the modification, the name of the developer who performed the change, the number of changed lines, the actual lines of code that were added or removed, and a detailed message explaining the reasons for the change. We automatically analyze the content of the change message, using a lexical technique, similar to [23]. We divide modifications into three types:

1. **Fault Repairing modifications (FR)** which are done to fix a fault. FRs represent the fault repair process which likely differs from the code change process. In most projects, the change message, attached to an FR, would specify the ID of the fault being fixed or would use keywords such as "fix bug". FR modifications are not used in calculating the complexity of the change process, but are used for validating the results in our case study, which is presented in Section 7.
2. **General Maintenance modifications (GM)** which are mainly bookkeeping modifications and which do not reflect the implementation of a particular feature. Example GMs are modifications to update the copyright notice at the top of each source file and modifications to re-indent the code after being processed by a pretty-printer. GMs are removed from our analysis and are never considered. These changes are rather easy to identify in large projects since they usually involve a very large number of files and their change message would include keywords such as "copyright update", and "re-indent".
3. **Feature Introduction modifications (FI)** which add or enhance features. All modifications which are not FR nor GM are labeled as FI. FIs are used calculating the complexity of the code change process.

A software system which has to endure highly scattered modifications as it implements requirements, will have a high tendency of becoming a complex project. In contrast, a project where modifications are limited to specific spots will have less complexity associated with it. A complex code base, the addition of a large number of features within a short period of time, or a large number of developers simultaneously changing the source code of a project are some of the many reasons that could cause code modifications to be highly scattered. This scatter of modifications throughout the code, within a short time, makes it difficult for developers working on the project to keep track of its progress and the changes. For instance in [21], Lehman $et\ al.$ noted that the changed portion of a software system during a release tends to remain constant in relation to the rest of the system over time, and that a sudden increase in the scatter of changes during a release is likely to have adverse affect on the software system as noted in their OS/360 case study.

Various observations by Brooks support our intuition and our model [5]. In particular, Brooks warned of the decay of the grasp of what is going in a complex system. A complex modification pattern will cause delays in releases, high bug rates, stress and anxiety to all the personnel involved in a project. As the ability of team members to understand and track the changes to the system deteriorates so does their knowledge of the system. New development will be negatively affected. Similarly, Parnas warned of the ill-effects of *Ignorant Surgery*, modifications done by developers who are not sufficiently knowledgeable of the code [28]. Such ignorance may be due to the developers being junior developers or it may be due to the fast pace of development which prevents developers from keeping track of other changes. For instance, a study of the root cause of faults in a large telephony system found that over 35% of faults where due to problems such as change coordination, missing awareness, communication, and lack of system knowledge [22]. Information hiding and good designs attempt to reduce the need to track other changes, but as the scatter of changes increases so does the likelihood that developers will miss tracking changes that are relevant to their work and managers will have a harder time allocating testing resources or tracking the project's progress. In short, a chaotic change process is a good indicator of many project problems.

3 Information Theory

Information theory deals with assessing and defining the amount of information in a message [32]. 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 ($i.e.$ 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 or entropy 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 $\sum\limits_{k=1}^{n} p_k = 1$. 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 an element i has a probability of occurrence $p_i = 1$ and $\forall k \neq i : \; p_k = 0$, 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 ($i.e.$ the output of the system). This fact has been used to measure the quality of compression techniques against the smallest theoretically-possible compressed-size.

4 Basic Code Change Model

If we view the code change 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 to measure the amount of *uncertainty/randomness/complexity* in the change process.

4.1 Basic Model

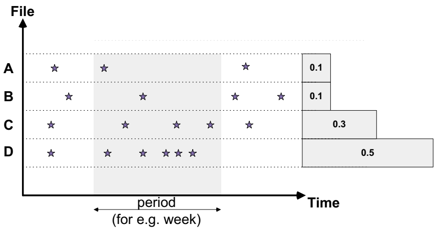

Figure 1. Complexity of a Change Period

Suppose we have a system which consists of four files. If we examine the change history of this system using the FI modifications, we can plot for each file the moments in time it was changed. As can be seen in Figure 1, we put stars to indicate when a specific file was changed. We now define a period of time, for example a week, or a month. For that period of time, we can define a file change probability distribution [1] P. P gives the probability that $file_i$ is changed in a period. For each file in the system, we count how many times it was changed during a period and divide by the total number of changes in that period for all files. For example, in Figure 1, in the highlighted grey period we have 10 changes 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 1, we can see a graph of the file change probability distribution P for the shaded period.

If we monitor the changes 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 changing each file is the same ($i.e.$ $file_k = \frac{1}{n}$) then the amount of entropy in the system is at its maximum.

Instead of simply using the number of changes to the file, we use the number of modified lines over a period to build the file change probability. Modified lines is the sum of added and deleted lines per the modification record.

Intuition. Consider these two modifications. In the first modification, the developer had to change over a dozen files to add a feature. When asked about the steps required to add the feature, she or he may not recall half of them. Whereas another modification to add a different feature required the changing a single file. Recalling the changes required for the latter feature is much easier. Intuitively, if we have a software system that is being changed across all or most of its files, developers will have a hard time keeping track of all these changes. Concerns about the complexity of tracking scattered changes have been expressed by others working on large software systems, such as telephony systems [33].

The BCC model quantifies the patterns of changes instead of measuring the number of changes or measuring the effects of changes to the code structure. Faults are introduced due to misunderstandings about the current structure and state of the system. By being aware of the current state of the system, developers are less likely to introduce faults and managers are likely to have an easier time monitoring the project. Entropy measures redundancy and patterns. Change patterns with low information content as defined by entropy are easier to track and remember by developers and others working on a project.

The BCC model, along with the next two models, only use the FI modifications. FR modifications are not used since they represent fault fixes which are likely to be more scattered and to touch areas that are not being developed during the current period. This property of fault fixes in-

[1] Our definition of distribution follows the frequentists school of thought on probability which considers the relative frequency of occurrence of an event, as a measure of its probability [34].

flates the entropy measurement for a period. Moreover, fault fixes are not likely to introduce new functionality, instead they are simply revisiting old changes which developers are already aware of and are less likely to need recalling them. Our models could be redefined to include FRs if need be.

The models quantify entropy for several modifications within a period not just for a specific modification. This choice of grouping several modifications is likely to inflate the entropy measurements, but we are more concerned with variations across periods instead of the absolute entropy values. In addition, by grouping modifications we can gauge the challenges that managers and developers need to cope with due to wide spread modifications. Nevertheless, the models could be adjusted to quantify the entropy of every modification.

Files As a Unit of Measurement. In the BCC model we use the file as our unit of code to build the change probability distribution P for each period. Other units of code can be used, such as functions or code chunks that are determined by a person with good 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 and data types. Based on our experience in studying large systems, we found this to often be the norm. In recent work [16] we were able to empirically support this belief by showing that the probability of two source code entities ($e.g.$ functions) changing together over time is high, if both entities are within the same file, at least for large open source software systems written in the C language.

4.2 Evolution of Entropy

We can view the file change 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 the repeated transformation of the code change process from one state to the next. Looking at Figure 2, we can see the P_j's calculated for 4 consecutive periods with their respective entropy. This allows us to monitor the evolution of entropy in the change process. If the project and the code change process are not under control nor managed well, then the system will head towards maximum entropy and chaos.

The manager of a large software project should aim to control and manage entropy. Monitoring 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 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 period then to drop as the refactoring eases future modifications to the code. On the other hand, a complex code base may cause a consistent rise in entropy over an extended period of time, until the issues causing the rise in change entropy/complexity are addressed and resolved as we observed when studying open source projects such as KDE [15].

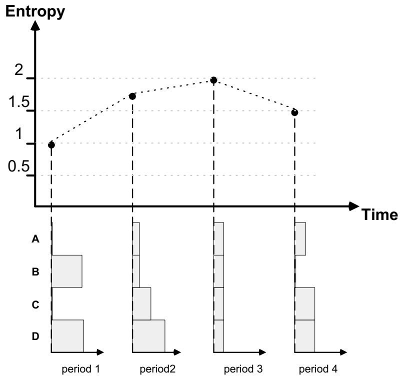

Figure 2. Evolution of Change Entropy

5 Extended Code Change Model

The BCC model presented in Section 4, assumes a fixed period size for entropy calculation, and assumes that the number of files in a system remains fixed over time. Both assumptions limit the use of the BCC model on large long lived software systems. The Extended Code Change (ECC) model, presented in this section, addresses these limitations.

5.1 Evolution Periods

Instead of using 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:

1. **Time based periods:** This is the simplest technique and it is the one presented in the BCC model in Section 4. The history of changes is broken into equal length periods based on calendar time from the start of the project. For example, we break the history on a monthly or bimonthly basis. A project which has been around for one year, would have 12 or 6 periods respectively. In prior work [15], we chose a 3 month period which represents a quarter. We believe that a quarter is a good amount of time to implement a reasonable amount of enhancements to a software system.
2. **Modification limit based periods:** The history of changes is broken into periods based on the number of modifications as recorded in the source control repository. For example, we can use a modification limit of 500 or 1,000 modifications. A project which has 4,000 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 where little development may have occurred from spanning a long time, we impose a limit of 3 months on a period even if the modification limit was not reached. In prior work [15], we chose a limit of 600 modifications.

3. **Burst based periods:** Based on studying the change history for several large software systems, we observed that the modification process is done in a bursty pattern. Over time, we see periods with many code modifications, these periods are followed by short periods of no or little code modifications. We chose to use that observation to automatically break up the change 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. This period creation method is used in [14] and in our case study in Section 7. The Burst based period creation method is the most general method, as we do not need to specify modification counts or time limits which may differ between projects or over time.

5.2 Adaptive System Sizing

Our entropy calculations, in Section 4, needs to account for the varying number of files in a software system. We define ***Normalized Static Entropy***, H, as:

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

where $p_k \geq 0, \forall k \in 1, 2, ..., n$ and $\sum_{k=1}^{n} p_k = 1$. The normalized static entropy H normalizes Shannon's entropy H_n, so that $0 \leq H \leq 1$. We can now compare the entropy of distributions of different sizes, 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 normalized static entropy H, we could compare the entropy between different software projects. For example, we could compare the evolution of two operating systems side by side or even an operating system and a window manager.

The Normalized Static Entropy, H, depends on the number of files in a software system, as it depends on n. For many software system there exist files that are rarely modified, for example, platform and utility files [21]. Developers do not need to worry about tracking changes to these files, since the probability of them changing is very low. To prevent these files from reducing the normalized entropy measure, we defined **Adaptive Sizing Entropy** (H') which is a working set normalized 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 two different criteria:

1. **Using Time:** The set of recently modified files is all files modified in the preceding x months, including the current month. In our experiments we used 6 months. Other values could be used. Our choice of six months as a window originates from our belief and our experience developing large software systems. We found that usually what is hot (relevant and development focus) at the beginning of the year tends not to be a concern towards the end of the year. This is mainly due to the fact that throughout the earlier part of the year most of the problems and features related to these files are addressed.
2. **Using Previous Periods:** The set of recently modified files is all files modified 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.

An adaptive sizing entropy H' usually produces a higher entropy than a traditional normalized entropy H, since 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 divided by a smaller number. In some rare cases, the software system may have undergone several changes/refactorings. In these cases, 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, since many files may have been removed recently as part of a cleanup [15]. In these rare cases, an adaptive sizing entropy H' will be larger than a traditional normalized entropy H.

6 File Code Change Model

The two previously presented models in Sections 4 and 5 produce a value which quantifies the entropy for each period in the lifetime of a software system. We now extend the ECC model to deal with assigning a complexity value to a file. By assigning a complexity value to a file we can later (*see* Section 7) study the ability of our entropy models in predicting the incidence of faults in specific files or subsystems.

We believe that files that are modified during periods of high change complexity, as determined by our ECC Model, will have a higher tendency to contain faults. Developers, performing changes during these periods, will not have a good grasp of the latest changes to the source code and the state of the project. We define a **History Complexity Metric (HCM)** for each file in a system. The HCM assigns to a file the effect of the change complexity of a period, as calculated by our ECC 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 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:

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

c_{ij} is the contribution of entropy for period i (H_i) assigned to file j. We explore three $HCPF$s by varying the definition of c_{ij}:

1. $HCPF^1$ with $c_{ij} = 1$: This factor assigns the full complexity value (H_i) to every modified file in a period ($j \in F_i$). This is the simplest model and assumes that all files changed during a period are affected by the full complexity of the period.
2. $HCPF^2$ with $c_{ij} = p_j$: This factor assigns a percentage of the complexity associated to a period (H_i). The percentage is the probability of file j being modified during period i. This metric assumes that files are affected based on their frequency of change during the period. The more a file is changed, the more it is affected by the complexity of a period.
3. $HCPF^3$ with $c_{ij} = \frac{1}{|F_i|}$: This factor distributes evenly the complexity associated to a period (H_i) between all modified files in that period. This metric assumes that files are equally affected with the complexity of a period. As more files are changed, the effect of a period's complexity on every changed file is reduced.

More elaborate definitions of $HCPF$ are possible but for this paper we chose to use these intuitive and simple definitions.

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

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

We use this simple HCM definition to indicate that complexity associated with a file keeps on increasing over time, as a file is modified. Using this simple HCM and our three $HCPF$ definitions, we have three HCM metrics namely: HCM^{1s}, HCM^{2s}, and HCM^{3s}, where the s superscript indicates the use of the simple HCM formula. In addition, we define a more elaborate HCM^{1d}, which employs a decay model using the simplest $HCPF$ ($HCPF^1$). In HCM^{1d}, earlier modifications would have their contribution to the complexity of the file reduced in an exponential fashion over time. Similar decay approaches have been used in [11, 17].

$$HCM_{\{a,..,b\}}(j) = \sum_{i \in \{a,..,b\}} e^{\phi * (T_i - Current\ Time)} HCPF_i^1(j),$$

where T_i is the end time of period i and ϕ is the decay factor.

We define the HCM for a subsystem ($i.e.$ directory) 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 moves subsystems during a 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 4 defined HCMs at the subsystem level (HCM^{1s}, HCM^{2s}, HCM^{3s}, and HCM^{1d}), we study whether our entropy HCM metric is a better predictor of faults compared to predictors based on the number of prior modifications or faults. We chose to compare the performance of our model against predictors using prior faults and modifications since prior research shows that these two types of predictors outperform other types predictors (e.g. ones based on complexity metrics) [2, 11, 20].

7 Case Study

We performed three experiments to predict future faults in the subsystems of large software systems:

1. **Modifications vs. Faults:** We compare the performance of predictors based on *prior modifications* with ones based on *prior faults*.
2. **Modifications vs. Entropy:** We compare the performance of predictors based on *prior modifications* with ones based on our *HCM entropy models*.
3. **Faults vs. Entropy:** We compare the performance of predictors based on *prior faults* with ones based on our *HCM entropy models*.

Application Name	Application Type	Start Date	Subsystem Count (*low level directories*)	Prog. Lang.
NetBSD	OS	March 1993	235	C
FreeBSD	OS	June 1993	152	C
OpenBSD	OS	Oct 1995	265	C
Postgres	DBMS	July 1996	280	C
KDE	Windowing System	April 1997	108	C++
KOffice	Productivity Suite	April 1998	158	C++

Table 1. Summary of the studied systems

Table 1 summarizes the details of the software systems we studied in our case study. We based our analysis on the first five years in the life of each studied open source project. We 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 and processes. Our case study follows an approach similar to [11], in particular:

1. We build Statistical Linear Regression ($SLR\ Model$) models for every software system in Table 1. These $SLR\ Model$s use data from the second and third years from the source control repository to predict faults in the fourth and fifth years of the software project. In total, we build six SLR models: 4 models for the HCM entropy metrics, one for prior faults, and one for prior modifications. All the built SLR models predict faults in subsystems during the fourth and fifth years.
2. We measure the amount of error in each model and compare the error between models. In particular, we compare the performance of
   (a) modifications versus fault models.
   (b) modifications versus entropy models.
   (c) faults versus entropy models.
3. We perform statistical tests to determine whether the difference in error is statistically significant or simply due to the natural variability of the studied data.

In the following subsections, we elaborate on these steps.

7.1 Linear Regression Models

To perform our experiments, we built six SLR models for each software system in Table 1. The built SLR models have the following form, $y = \beta_0 + \beta_1 x$, where y is the dependant variable and x is the predictor/independent variable.

For each model, y represents the number of faults in a subsystem. The number of faults is the number of Fault Repairing (FR) modifications which occurred in the subsystem during the fourth and fifth years. x represent specific metrics for each subsystem in the second and third years. Table 2 describes the value of x in each of the six SLR models.

SLR Model	**Value of x**
$\boldsymbol{Model_m}$	number of modifications for a subsystem.
$\boldsymbol{Model_f}$	number of faults for a subsystem.
$\boldsymbol{Model_{HCM1s}}$	HCM_{1s} value for a subsystem.
$\boldsymbol{Model_{HCM2s}}$	HCM_{2s} value for a subsystem.
$\boldsymbol{Model_{HCM3s}}$	HCM_{3s} value for a subsystem.
$\boldsymbol{Model_{HCM1d}}$	HCM_{1d} value for a subsystem.

Table 2. Value of x used to predict y (faults in years 4 and 5) for each subsystem.

All the HCM models are based on the ECC bursty model that has a one hour quiet time between bursts. The HCM_{1d} uses a decay factor (ϕ) of 10, which minimizes the error for the $SLR\,Model_{HCM1d}$ when correlating HCM_{1d} values in the second year to faults in the third year. To ensure the mathematical validity of our SLR models, we use the value of y and the mathematical log of the x values, instead of x. The use of a log transformation ($e.g.$ *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 [35]. The SLR model parameters (β_0 and β_1) are estimated using the fault data from the second and third years. Table 3 shows the R^2 statistic which measures the quality of the fit. The better the fit, the higher the R^2 value. A zero R^2 indicates that there exists no relationship between the dependant (y) and independent (x) variables. We notice that the C systems have a better fit in comparison to the C++ systems ($i.e.$ KDE and KOffice) for all the SLR models. The $SLR\ Model_{HCM1d}$ has the best fit of all the SLR models for all the studied systems.

App	R^2_f	R^2_m	R^2_{1s}	R^2_{2s}	R^2_{3s}	R^2_{1d}
NetBSD	0.57	0.55	0.54	0.53	0.61	0.71
FreeBSD	0.65	0.48	0.57	0.58	0.59	0.65
OpenBSD	0.45	0.44	0.54	0.55	0.54	0.57
Postgres	0.57	0.36	0.49	0.51	0.60	0.61
KDE	0.31	0.26	0.28	0.29	0.36	0.57
KOffice	0.30	0.27	0.33	0.33	0.27	0.41

Table 3. The R^2 statistic for the SLR Models

7.2 Prediction Error for the SLR Models

Once we estimate β_0 and β_1 for the SLR Models for every system, we measure the amount of prediction error. Mathematically for every model with β_0 and β_1 as parameters, we get a $\widehat{y_i}$ for every x_i, where $\widehat{y_i}$ is the number of predicted 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 years.

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. To achieve the goals of our study, we need to compare the prediction errors for the SLR models. For example, to determine if prior modifications are better than prior faults in predicting faults, we need to compare E_m with E_f, where E_m and E_f are the total prediction error for the $SLR\ Model_m$ and $SLR\ Model_f$ respectively. The best model is the one with the lowest total prediction error.

7.3 Statistical Significance of Differences

We use statistical paired tests to study the significance of the difference in prediction error between two SLR Models ($SLR\,Model_A$ and $SLR\,Model_B$). Our statistical analysis assumes a 5% level of significance (i.e. $\alpha = 0.05$). We formulate the following test hypotheses:

$$H_0 : \mu(e_{A,i} - e_{B,i}) = 0, \;\; H_A : \mu(e_{A,i} - e_{B,i}) \neq 0,$$

where $\mu(e_{A,i} - e_{B,i})$ is the population mean of the difference between the absolute error of each subsystem. If the null hypothesis H_0 holds (i.e. the derived p-value $> \alpha = 0.05$), then the difference is not significant. If the p-value $< \alpha = 0.05$ then we can with high probability reject H_0.

For our analysis, we conducted parametric and non-parametric paired statistical tests. For a parametric test, we used a paired t-test. For a non-parametric test, we used a paired Wilcoxon signed rank test which is resilient to strong departures from the t-test assumptions [29]. We studied the results of both tests to determine if there are any differences between the results reported by both types of tests. In particular for non-significant differences reported by the parametric t-test, we checked if the differences are significant according to the non-parametric Wilcoxon test. The Wilcoxon test helps ensure that non-significant results are not simply due to the departure of the data from the t-test assumptions. For the results presented below, both tests are consistent so we only report the values of the t-test.

7.4 Comparing Models

7.4.1 Modifications vs. Faults

App	$E_m - E_f$ (%)	$P(H_0\ holds)$
NetBSD	+11.7 (+04%)	0.67
FreeBSD	+71.2 (+48%)	0.00
OpenBSD	+03.7 (+02%)	0.84
Postgres	+47.2 (+49%)	0.02
KDE	+26.3 (+07%)	0.32
KOffice	+26.3 (+04%)	0.51

Table 4. The difference in prediction error and t-test results for the $SLR\ Model_m$ and $SLR\ Model_f$ for the studied systems

We want to determine if prior modifications are better than prior faults in predicting future faults; therefore, we compare the total prediction error for both the $SLR\ Model_m$ and $SLR\ Model_f$. The second column in Table 4 shows the percentage of difference in prediction error when the $SLR\ Model_m$ is used instead of $SLR\ Model_f$. The third column shows the results for the t-test which determines if the difference is statistically significant or if it is due to the natural variability of the data. The t-test on paired observations of absolute error was significant at better than 0.02 for the FreeBSD and Postgres systems (marked in grey in Table 4). For these two systems, we are over 98% confident that the increase in prediction error between $SLR\ Model_f$ and $SLR\ Model_m$ is statistically significant. Whereas for the other systems, the increase is not statistically significant indicating the performance of both models (prior faults or prior modifications) is similar.

These results indicate that prior faults should be used to predict faults instead of using prior modifications. Using a prior modifications predictor may cause an approximately 50% rise in prediction error over using a prior faults predictor.

7.4.2 Modifications vs. Entropy

App	$E_{HCM3s} - E_m$ (%)	$P(H_0\ holds)$	$E_{HCM1d} - E_m$ (%)	$P(H_0\ holds)$
NetBSD	-39.8 (-14%)	0.03	-106.5 (-36%)	0.00
FreeBSD	-47.4 (-22%)	0.02	-72.0 (-33%)	0.00
OpenBSD	-40.4 (-18%)	0.01	-53.8 (-23%)	0.00
Postgres	-52.7 (-37%)	0.04	-56.9 (-40%)	0.03
KDE	-52.1 (-13%)	0.01	-165.2 (-42%)]	0.00
KOffice	+03.3 (+01%)	0.83	-69.9 (-18%)	0.01

Table 5. The difference in prediction error and t-test results for the $SLR\ Model_m$, $SLR\ Model_{HCM3s}$, and $SLR\ Model_{HCM1d}$

We want to determine the value of the additional analysis in deriving our HCM entropy metrics which are derived from the number of modifications. We now compare the prediction quality of modifications and HCM metrics. We chose the simple $SLR\ Model_{HCM3s}$ and the decay $SLR\ Model_{HCM1d}$ to compare with the $SLR\ Model_m$. Both HCM models were the top two performing HCM models based on the R^2 statistic shown in Table 3. The second and fourth columns in Table 5 shows the percentage of difference in prediction error when the $SLR\ Model_{HCM3s}$, or the $SLR\ Model_{HCM1d}$ are used instead of $SLR\ Model_m$ respectively. The third and fifth columns in Table 5 show the results for the t-test which determines if the difference in prediction error is statistically significant. Greyed cells in Table 5 indicate that the shown results are statistically significant at $\alpha = 0.05$. All results are significant except for the $SLR\ Model_{HCM3s}$, for the KOffice system where there is a negligible, though not statistically significant, increase in prediction error (1%) for the simple HCM model.

These results indicate that both HCM (simple and decay) based models are statistically likely to outperform prior modifications in predicting future faults. The decrease in prediction error when using an HCM model ranges between 13% to 42% (32% on average) when compared to the prior modifications model.

7.4.3 Faults vs. Entropy

We have shown that our entropy metrics outperform prior modifications but prior faults outperform prior modifications in predicting faults. So we would like to study the performance of our entropy metric in comparison to prior faults (the best predictor up to now). We chose again

App	$E_{HCM3s} - E_f$ (%)	$P(H_0\ holds)$	$E_{HCM1d} - E_f$ (%)	$P(H_0\ holds)$
NetBSD	-28.14 (-10%)	0.26	-94.84 (-34%)	0.00
FreeBSD	+23.81 (+16%)	0.30	-00.79 (-01%)	0.97
OpenBSD	-36.59 (-16%)	0.02	-50.05 (-22%)	0.01
Postgres	-05.53 (-06%)	0.71	-09.71 (-10%)	0.55
KDE	-25.72 (-07%)	0.32	-138.87 (-38%)	0.01
KOffice	+19.20 (+05%)	0.34	-54.07 (-15%)	0.04

Table 6. Results for the $SLR\ Model_f$, $SLR\ Model_{HCM3s}$, and $SLR\ Model_{HCM1d}$

the top two performing models ($SLR\ Model_{HCM3s}$ and $SLR\ Model_{HCM1d}$) based on the R^2 statistic in Table 3. The second and fourth columns in Table 6 shows the percentage of difference in prediction error when the HCM models ($SLR\ Model_{HCM3s}$ or the $SLR\ Model_{HCM1d}$) are used instead of $SLR\ Model_f$ respectively. The third and fifth columns in Table 6 show the results for the t-test which determines the statistical significance of the difference in prediction error. Greyed cells in Table 6 indicate that the difference between prediction errors is statistically significant. For the $SLR\ Model_{HCM3s}$ model, only the cell for the OpenBSD system is grey indicating that the improvement in prediction error for this system is statistically significant. For OpenBSD, the simple HCM model statistically outperforms the prior faults predictor by 16%. The results for the other systems varies but the results are not statistically significant. For the $SLR\ Model_{HCM1d}$, all cells except the ones corresponding to FreeBSD and Postgres are grey. These results indicate that $SLR\ Model_{HCM1d}$ outperforms the number of prior faults in predicting future faults for all systems except for the FreeBSD and Postgres where the results are not statistically significant. For these two systems, even though the HCM decay model performs better, the performance improvement are not statistically significant.

These results indicate that models based on our entropy metrics are as good as (or even better) predictors of faults in comparison to prior faults for most studied software systems. The decrease in prediction error using an HCM model ranges between 15% to 38% when compared to the prediction error of a model based on prior faults.

Based on our three experiments we note that in almost all cases, except for E_{HCM1d} vs. E_m, no single model statistically outperforms all other models for all systems. Fault predictors are usually project specific and vary in performance from one project to the next (similar observations on commercial systems were noted by Nagappan *et al.* [27]). Nevertheless, we can discern the following general results:

1. Prior faults are better predictors of future faults than the prior modifications. These results on open source systems are similar to prior results reported on industrial systems by Graves *et al.* [11].
2. The HCM based predictors are better predictors of future faults than prior modifications or prior faults. These results are very promising since although many companies may not have a complete history of their faults, they often have a detailed record of code changes, as changes are readily available and automatically collected in code repositories. In practice, one can build multivariate models which combine our complex metric, prior faults, prior modifications, and other available complexity metrics instead of using a single predictor.

7.5 Threats to Validity

In our analysis we used the number of Fault Repairing (FR) modifications as recorded by the source control system and determined using an automatic lexical classification technique. In [13], we compared our automatic classifications to classifications done by six professional software developers on the same data used in this paper. Our analysis shows a high correlation ($\sigma > .8$) between a human and an automated classifier. When the humans were divided into two groups and were asked to correlate the same data, the inter-human correlation is as high as the human and automatic classification. In short, we feel that our analysis shows that the used data is as accurate as possible given the limited information available about the studied projects. Alternatively, we could have used data from defect management systems. Unfortunately, several of the studied systems do not have a defect tracking system. Also if we had access to defect systems, we could not map defects to particular parts of the code unless the modification records referenced every defect in the tracking system.

In our analysis we do not consider faults that may have been reported but never fixed, since we used the fault fixes instead of using the reported fault counts. There may exist subsystems in which a large number of faults have been reported yet they were never fixed during our period of analysis. We believe the chance of this occurring is low nevertheless it is a possibility. Furthermore, the number of fixed faults is likely to correlate with the number of reported faults.

Although we examined a large number of software systems, the systems used in our study are all open source systems which have several interesting characteristics that may not hold for commercial systems. The most notable characteristics are: a) The distribution of the development team around the world with members rarely meeting in person and relying heavily on electronic communications such as emails and newsgroups instead of in-person formal and informal (*e.g.*, water cooler and lunch time conversations). b)

The self selective nature of the team. Developers volunteer to work on the project and are free in picking which areas to contribute to. All these characteristics limit the generalization of our results. We believe that our results are generalizable to large open source systems with an extended network of developers spread out throughout the world. Our results are likely to generalize as well to commercial software systems which are developed by distributed teams, and probably even to commercial systems developed in a single location. We need to study a few commercial systems, before we can confidently generalize our results.

Finally, demonstrating that a complex code change process causes the appearance of faults requires more than simply showing statistical significant relations, instead we need to show temporal precedence as well. We need to show that the complex code change process caused the appearance of faults in the software system. Unfortunately, this is a rather hard task and may be difficult to demonstrate, as we believe that the complexity in the code change process interacts with all the other project facets in a feedback loop. A complex code base requires complex change process to maintain it and a complex change process produces a complex code base. Furthermore, a complex set of requirements may cause the change process to become a complex process which in turn may cause the appearance of faults in the software. Therefore to show true causality we would need to build a richer and detailed theory which can measure the effect of the feedback loop on the interacting facets in a software project. To validate this theory, we would need to perform controlled experiments with subjects. The results of such experiments would have a much weaker external validity ($i.e.$ would be hard to generalize). Our results do not show a causality relation but intuitively we believe that a complex code change process negatively affects the software system.

8 Related Work

Barry $et\ al.$ used a volatility ranking system and a time series analysis to identify evolution patterns in a retail software system based on the source modification records [3]. Eick $et\ al.$ studied the concept of code decay and used the change history to create visualization of the change history of a project [9, 10]. Graves $et\ al.$ showed that the number of modifications to a file is a good predictor of the fault potential of the file [11]. Leszak $et\ al.$ showed that there is a significant correlation between the percentage of change in reused code and the number of defects found in those changed components [22]. Mockus $et\ al.$ used source modification records to assist in predicting the development efforts in large software systems for AT&T [24]. Previous research has focused primarily on studying the source code repositories of commercial software systems for predicting faults or required effort. We believe that this focus on commercial source systems limits the applicability of the results since the findings may depend on the studied systems or organizations. Using open source systems we can study a much larger set of systems to validate our findings and are more confident about the generality of our results.

Whereas our model quantifies the complexity of the code change process as calculated from the source code modification statistics, previous studies [1, 4, 6, 12, 36] 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 the complexity of the code change process instead of just computing the complexity of the source code. We conjecture that detecting complex code changes will serve as an early warning measure to help prevent the occurrence of faults in a software system.

Outside of the software engineering domain, the measure of entropy has been used to improve the performance of Just In Time compilers and profilers [31]. It has been used for edge detection and image searching in large image databases [8]. Also, it has been used for text classification and several text based indexing techniques [7].

9 Conclusion

We conjecture that: *A complex code change process negatively affects its product, the software system. The more complex changes to a file, the higher the chance the file will contain faults*. We present models to quantify the complexity over time using historical code changes instead of source code attributes. Through a case study on six large open source projects, we show that the number of prior faults is a better predictor of future faults in comparison to the number of prior modifications. We also demonstrate that predictors based on our change complexity models are better predictors of future faults in large software systems in contrast to using prior modifications or prior faults.

References

[1] S. Abd-El-Hafiz. Entropies as measures of software information. In *Proceedings of the 17th International Conference on Software Maintenance*, pages 110–117, Florence, Italy, 2001.

[2] E. Arisholm and L. C. Briand. Predicting fault-prone components in a java legacy system. In G. H. Travassos, J. C. Maldonado, and C. Wohlin, editors, *ISESE*, pages 8–17. ACM, 2006.

[3] 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*, pages 106–113, Portland, Oregon, May 2003.

[4] A. Bianchi, D. Caivano, F. Lanubile, and G. Visaggio. Evaluating software degradation through entropy. In *Proceedings of the 7th International Software Metrics Symposium*, pages 210–219, 2001.

[5] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison Wesley Professional, 1974.

[6] N. Chapin. An entropy metric for software maintainability. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 522–523, Jan. 1995.

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

[8] M. Do and M. Vetterli. Texture similarity measurement using kullback-leibler distance on wavelet subbands. In *Proceedings of the 2000 International Conference on Image Processing*, Vancouver, Canada, Sept. 2000.

[9] 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 Transactions on Software Engineering*, 27(1):1–12, 2001.

[10] 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, May 1992.

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

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

[13] A. E. Hassan. Automated classification of change messages in open source projects. In R. L. Wainwright and H. Haddad, editors, *SAC*, pages 837–841. ACM, 2008.

[14] A. E. Hassan and R. C. Holt. Studying the chaos of code development. In *Proceedings of the 10th Working Conference on Reverse Engineering*, Nov. 2003.

[15] A. E. Hassan and R. C. Holt. The Chaos of Software Development. In *Proceedings of the 6th IEEE International Workshop on Principles of Software Evolution*, Sept. 2003.

[16] A. E. Hassan and R. C. Holt. Predicting Change Propagation in Software Systems. In *Proceedings of the 20th International Conference on Software Maintenance*, Chicago, USA, Sept. 2004.

[17] A. E. Hassan and R. C. Holt. The Top Ten List: Dynamic Fault Prediction. In *Proceedings of the 21st International Conference on Software Maintenance*, Budapest, Hungary, Sept. 2005.

[18] I. Herraiz, J. M. Gonzalez-Barahona, and G. Robles. Towards a Theoretical Model for Software Growth. In *Proceedings of the 4th International Workshop on Mining Software Repositories*, Minnesotta, USA, May 2007.

[19] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional, second edition, Sept 2002.

[20] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data Mining for Predictors of Software Quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5), 1999.

[21] M. M. Lehman, D. E. Perry, and J. F. Ramil. Implications of Evolution Metrics on Software Maintenance. In *Proceedings of the 14th International Conference on Software Maintenance*, Washington, DC, USA, 1998.

[22] M. Leszak, D. E. Perry, and D. Stoll. Classification and evaluation of defects in a project retrospective. *The Journal of Systems and Software*, 61(3):173–187, 2002.

[23] A. Mockus and L. G. Votta. Identifying reasons for software change using historic databases. In *Proceedings of the 16th International Conference on Software Maintenance*, pages 120–130, San Jose, California, Oct. 2000.

[24] 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*, pages 274–284, Portland, Oregon, May 2003.

[25] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of the 30th International Conference on Software Engineering*, 2008.

[26] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, 2005.

[27] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In L. J. Osterweil, H. D. Rombach, and M. L. Soffa, editors, *ICSE*, pages 452–461. ACM, 2006.

[28] D. Parnas. Software aging. In *Proceedings of the 16th International Conference on Software Engineering*, pages 279 – 287, Sorrento, Italy, May 1994.

[29] J. Rice. *Mathematical Statisitcs and Data Analysis*. Duxbury press, 1995.

[30] M. J. Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, 1(4):364–370, 1975.

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

[32] C. E. Shannon. A Mathematical Theory of Communication. *The Bell System Technical Journal*, 27:379–423,623–656, Jul, Oct 1948.

[33] N. Staudenmayer, T. Graves, J. S. Marron, A. Mockus, D. Perry, H. Siy, and L. Votta. Adapting to a new environment: How a legacy software organization copes with volatility and change. In *5th International Product Development Management Conference*, Como, Italy, May 1998.

[34] J. Venn. *The Logic of Chance*. Dover Publications, 1888, reprinted 2006.

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

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

[37] T. J. Yu, V. Y. Shen, and H. E. Dunsmore. An Analysis of Several Software Defect Models. *IEEE Transactions on Software Engineering*, 14(9):1261 – 1270, sep 1998.