The Chaos of Software 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

In this paper we present a new perspective on the problem of complexity in software, using sound mathematical concepts from information theory such as Shannon's Entropy [31]. We study the complexity of the development process by examining the logs of the source control repository for large software projects. We hypothesize that the process of developing code is a good indicator of the current and future problems in the code and the project. A complex process will have negative affects on its outcome, such as producing a complex system or delaying releases. We validate our work by studying the evolution of six large open source projects (three operating systems, a window manager, an office productivity suite, and a database).

1 INTRODUCTION

> *"Complexity is the business we are in and complexity is what limits us."*. Fred Brooks, The Mythical Man-Month, p226 [25].

Large software systems are critical assets which provide a competitive advantage to their owners. Software systems need to evolve gracefully to fulfill customers' changing needs and requirements, otherwise they will fail [22]. To ensure such graceful evolution of software systems, developers need to reduce and control the complexity associated with software systems.
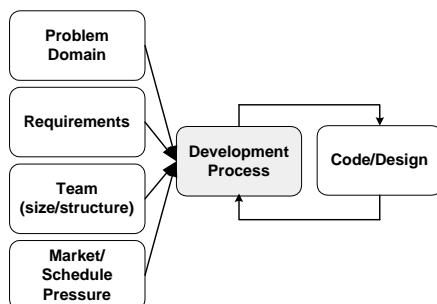

Figure 1: View of Complexity Flow in a Project

Complexity appears in many aspects pertaining to a software project such as the current code base and its design, the customer's requirements, the team structure and size, the de-

velopment process, market pressure, and problem domain. Figure 1 shows a simplistic view of how complexity in one aspect of the project flows from one part to the next. For example a complex set of requirements, a very large software team or a large set of requirements will increase the complexity and chaos in the development process. This will eventually have effects on the source code of the project. Also the code and the design of the project can as well affect the development process, for example a complex design or spaghetti code will complicate the development process.

Controlling the complexity of a project is a concern of project managers as they strive to deliver future releases on time and under budget. Researchers have proposed many metrics to measure the complexity of a software system and assist in reducing it. Though there are dozens of complexity metrics [33], they do not provide reliable indicators of the state of chaos in a project and are often costly to calculate [29, 13].

Previous work has focused on the development of complexity metrics mostly based on the source code of the software system and the interaction between its various components. While this approach is promising, we believe it has limitations. It focuses on the source code, which is the final output of the process.

Ideally we would like to monitor and control complexity as early as possible instead of detecting it in the code after it occurred. For example, it would be more beneficial to determine that the customer's requirements are excessively complex or that the development process is excessively complex, as early detection will help in better planning and preparation.

Code based measures do not quantify the complexity faced by developers adding features to the code and project managers monitoring the progress of the project. For example an operating system may have a very complex memory manager yet the memory manager may be so versatile and bug free that it requires no changes. Metrics based on source code analysis would indicate a high complexity. These would suggest that the code is too hard to maintain and evolve, yet in reality the code may be able to support well the current and future needs of its users as indicated by its succinct and bug

free evolution. Other metrics based on bug counts in the system have been proposed to measure the complexity of the system, unfortunately they do not indicate if the the bugs are due to the complexity of the project's source code, its architecture, or its requirements.

In this paper, we propose a metric to measure the complexity of the development process. This approach promises to address the concerns we highlighted. By monitoring the complexity of the development process, we provide early warnings about potential difficulties that will eventually cause an increase in the complexity of the code. We hypothesize that the complexity of the development process is a good indicator of the overall complexity of the project. A complex development process will produce code that is hard to control for timely and bug free releases.

We study the source code change history to predict patterns of change, and trends in the development process. These trends are good indicators of the complexity of the software system. We produce graphs that show micro and macro evolution trends of the software system, software developers case use these graphs to monitor the progress of their project and control its complexity. If the complexity of the software is above specific thresholds they need to react to them to prevent a downward spiral of the software system's ease of change and maintainability.

Organization of Paper
The paper is organized as follows. Section 2 discusses software change. We present source control repositories and give an overview of the type of information stored in them. We survey previous research use of such stored information. Section 3 gives an introduction to information theory and the mathematical concepts we use to model the development process' complexity and its evolution. Once we have the concepts from information theory presented, we are ready to explain our model in section 4. We start with a simple model, then we expand it into a more detailed and complete model in section 5. Section 6 shows the architecture of our software analysis framework. Section 7 provides some results from our study of the change history for six large open source system. Section 8 lists related work in the field of software evolution, entropy, and open source systems. Section 9 summarizes our results and presents plans for future work.

2 SOFTWARE CHANGE
Over time software systems undergo many modifications to implement the various functionality required to fulfill customers requirements and stay competitive in the market. As a software system ages, its complexity and size grow with many developers working simultaneously on the code base. Source control systems such as CVS [7, 14] or Perforce [26] help coordinate team development for large complex projects. They permit developers to work simultaneously on the same software system while ensuring that

their modifications do not interfere with work done by other team members. Source control systems provide a history of changes to the code of the software system.

The source code of the system is stored in a source repository – for each file in the software, the source repository records details such as the creation date of the file, modifications to the file over time along with the size and a description of the lines affected by the modification. Furthermore, the repository associates for each modification the exact date of its occurrence, a comment typed by the developer to indicate the reason for the change, and in some cases a list of other files that were part of the change described by the developer's comment. Such detailed records permit the roll back of the code to any point in time to either retrieve an old version of the code or to abandon new changes that were found to be irrelevant or buggy.

This detailed description of the history of code development provides a rich opportunity to perform empirical studies on the evolution of source code. The level of the detail in the repository and the consistent use of the source control system throughout the life of a project provide a detailed data source for several types of analysis. In addition, this data collection process is unintrusive and companies do not incur any extra costs as the source control system is already used as part of the development process of large software systems.

Researchers have described the many benefits of using the development history to gain a better understanding of bugs in source code, to locate hidden dependencies, or to assist in searching and browsing source code. Eick *et al.* used this information to study bugs and decay in a large telephony system [11, 12]. Graves *et al.* showed that the number of modifications to a file is a good predictor to the fault potential of the file [17]. Gall *et al.* examined the source repository of another telephony system to detect logical coupling between the different components of the software system [15]. Michail *et al.* have shown that comments associated with each change provide a rich and accurate indexing for source code when developers need to locate source code lines associated with a specific feature [6].

In this paper, we hypothesize that a software system becomes complex to manage and maintain when its change history becomes too complex to comprehend. 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. We develop a model to measure the complexity of the information contained in the source code change history. If the change history is too complex, then we expect that software developers and managers will face difficulties in understanding it, this will lead to complex code. The complex code will then degrade the development process. Eventually the software will become buggier and unmaintainable; and the process will become too complex. The software must be re-written

or re-engineered. In many respects, our hypothesis reflects Brooks' remarks on software development [25]. In particular, Brooks warned of the decay of grasp for what is going in a complex system. If the development team is no longer in touch with the code and the changes to the code, their knowledge of the system over time will deteriorate and the quality of the system will worsen. He also cautioned the effects of the team size and requirements on the success of a project.

3 INFORMATION THEORY

In 1948, Shannon laid down the basis of *Information Theory* in his seminal paper - *A mathematical theory of communication* [31]. 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 emits 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/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 $\sum_{k=1}^{n} p_k = 1$.

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 BASIC MODEL FOR THE EVOLUTION OF ENTROPY

If we view the development process of a software system as a system which emits data, and we define the data as the 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 a basic model for the entropy of software development and its evolution. The following section extends the model to be more elaborate and complete.
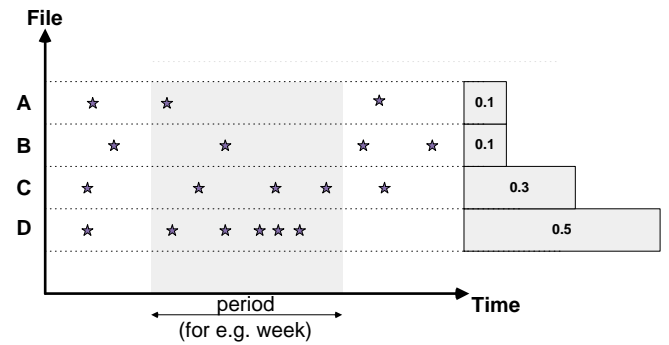
Basic Model

Figure 2: The Entropy of a Period of Development

Suppose we have a software system which consists of 4 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 modifications that are not bug fixes, for now. Thus, we first filter out all modifications that are bug fixes. We perform our filtering based on the text description associated with the modification.

Once these modifications are filtered out, we can plot over time for each file the moments the file was modified. As we can see in Figure 2, we put stars to indicate that for a specific file, it was modified on a particular moment in time. We repeat this process for each file in the system. We now define a period of time, for example a week, or a month. For that period of time j, we can define a file modification probability distribution P_j. P_j shows the probability that $file_i$ is modified in period j. 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 grey shaded 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 any file is equal (*ie.* $file_k = \frac{1}{n}, \forall k \in \{1, 2, ..., n\}$) 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 lim-

ited number of file have been modified. The development team grasp of what is going on in the software system will decay.

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 many successive periods in time, and view the evolution of a software system as the repeated transformation of the development process from one state to the next. Looking at Figure 3, we see the P_j's calculated for 4 consecutive periods with their respective entropy. Now we are able of monitoring the evolution of chaos/entropy in the development process. If the project and the development process are not under control nor managed well, then the state of the system will head towards maximum entropy and chaos.
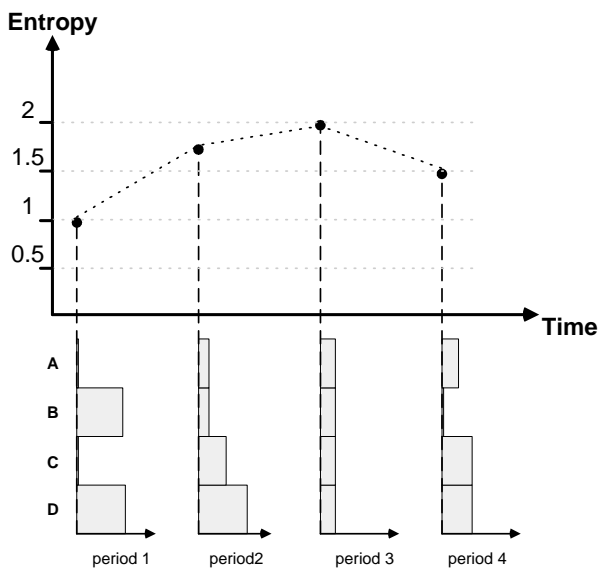
Figure 3: The Evolution of the Entropy of Development

The manager of a large software project should aim to control and manage the entropy. A graph like the one shown in Figure 3 would provide an accurate and up-to-date view on 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 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, a complex source code would cause a consistent rise in entropy over an extended period of time, till problems causing the rise in entropy/complexity are ad-

dressed and resolved.

Files As a Unit Of Measurement

In our model we use the file as our unit of code to build the modification probability distribution P for each period. Our choice of files was 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 except in some notable situations. For example in the VIM text editor [30], we found two files $misc1.c$ and $misc2.c$ which comprise a substantial amount of the source code. In that case using a file as a unit for building P would produce a low misleading entropy as most modifications may be associated with these files. That said the evolution of entropy graph will still indicate any variation in entropy in the system over its lifetime. Furthermore, a random break down of large files in the system to smaller files and associating modifications to the appropriate smaller files to build P may overcome this problem.

It is interesting to note that the notion of entropy can be used to locate such offending large files which violate traditional encapsulation principles. We are currently still investigating this hypothesis. Ignoring the rest of the software system, If we were to choose a single large file, break it into smaller chunks, and use these smaller chunks as our unit of entropy measurement, we would expect:

- If the large file is a coherent conceptual unit then changes would occur uniformly across the smaller chunks of the large file. This would produce a high internal entropy over time.
- But if the content of the large file are not conceptually related well, then we would see a concentration of modifications to specific chunks over others. This would produce a low internal entropy over time.

5 EXTENDED MODEL FOR THE EVOLUTION OF ENTROPY

In this section, we refine our basic model to address some of the characteristics and challenges associated with the evolution of large software systems. In the basic model we used a fixed period size to measure the evolution of entropy. Also we assumed that the size of the software system remains fixed over time. In the following sections we extend our model to deal with these two limitations.

Evolution Periods

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

Time based periods: This is the simplest model and it is

the one presented in the basic model. 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 choose in our experiments a 3 month period.

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 prevent 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 the previous 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 600 modifications limit.

Moving Window Period Sampling

To improve the continuity of our sampling to graph the evolution of a large project, we employ a moving window technique for our period creation. 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 1,2,3. Then the second period would have data from months 2,3,4. The third period would have data from months 3,4,5. 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. We need to adjust our entropy calculations to deal with the varying size of a software system.

To compare the entropy between the various periods in the life of the software system, we define H, ***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, $0 \leq H \leq 1$. 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 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'. 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 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 2 different techniques to create a period, then we have 2 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, 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 that 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 ANALYSIS FRAMEWORK

Figure 4 shows the pipeline architecture of our entropy evolution and analysis framework. It is broken into 3 main phases:
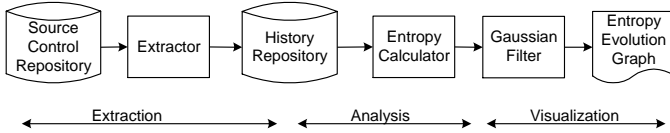
Figure 4: System Architecture

Extraction Phase: First we extract the details needed for our analysis from the source control repository. The extracted data is much smaller than the whole repository which contains the text of each source file over time. For example, the source code repository may be stored in multiple files and contain details such as the specific lines involved in each change and the content of these lines. All these details are not needed for our analysis, they are removed to reduce the size of our *History Repository* and speed up our analysis. Using this extraction method we are able to develop extractors for other source control repositories easily and use them in our analysis framework. We currently support the CVS source control system but other extractors can be easily developed. We chose to develop the CVS extractor first, as all the open source systems we studied used the CVS system to manage their source code repositories.

Analysis: This phase analyzes the content of our extracted *History Repository*. First, we define the periods in the project's lifetime (using either a time or a modification limit based periods). Then we calculate the modification probability distribution for each period. Then, we calculate the entropy for each distribution and may use a traditional or an adaptive system sizing to adjust the entropy and get a standardized entropy.

Visualization: Finally, we generate graph such as the ones in shown in Figures 3 and 5. To reduce the jaggedness of the graph we perform a Gaussian Smoothing on the generated graph. A Gaussian Smoothing is traditionally used to remove details and noise in images [16]. The Gaussian Smoothing removes localized changes and lets the graph show the more prominent trends.

7 CASE STUDIES

To validate our approach we analyzed the evolution of several large open source systems. Table 1 summarizes the details for these software systems. The oldest system is over ten years old and the youngest system is five years old. Due to size limitation we only give graphs for the Postgres [27] database system and a graph for the KDE [19] system.

| Application Name	Application Type	Start Date	Subsystem Count	Prog. Lang.
NetBSD	OS	21 March 1993	25	C
FreeBSD	OS	12 June 1993	33	C
OpenBSD	OS	18 Oct 1995	28	C
Postgres	DBMS	9 July 1996	116	C
KDE	Windowing System	13 April 1997	32	C++
Koffice	Productivity Suite	18 April 1998	85	C++

Table 1: Summary of the Studied Systems

Postgres is a sophisticated open-source Object-Relational DBMS supporting almost all SQL constructs. Its development started in 1986 at the University of California at Berkeley as a research prototype. Since then it has become an open source software with a globally distributed development team. It is being developed by a community of companies and people co-operating to drive the development of one the world's most advanced Open Source database software (DBMS). In our case study we use data beginning with 1996 when it became an open source project.
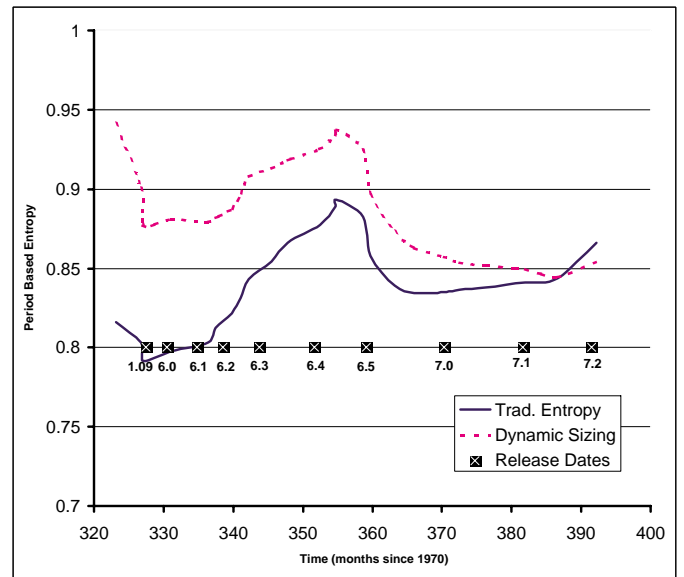
Figure 5: Period Based Entropy Evolution Graph for Postgres

We present two graphs from our study of Postgres. The first graph, shown in Figure 5, shows the period based entropy evolution of Postgres. We show the release dates for each major release of Postgres on the time line so we can correlate them with the variation in the entropy. We notice a continuous increase in entropy from release 1.09 to release 6.4. Reading through the documentation of the Postgres system, we correlate this to the period in which the develop-
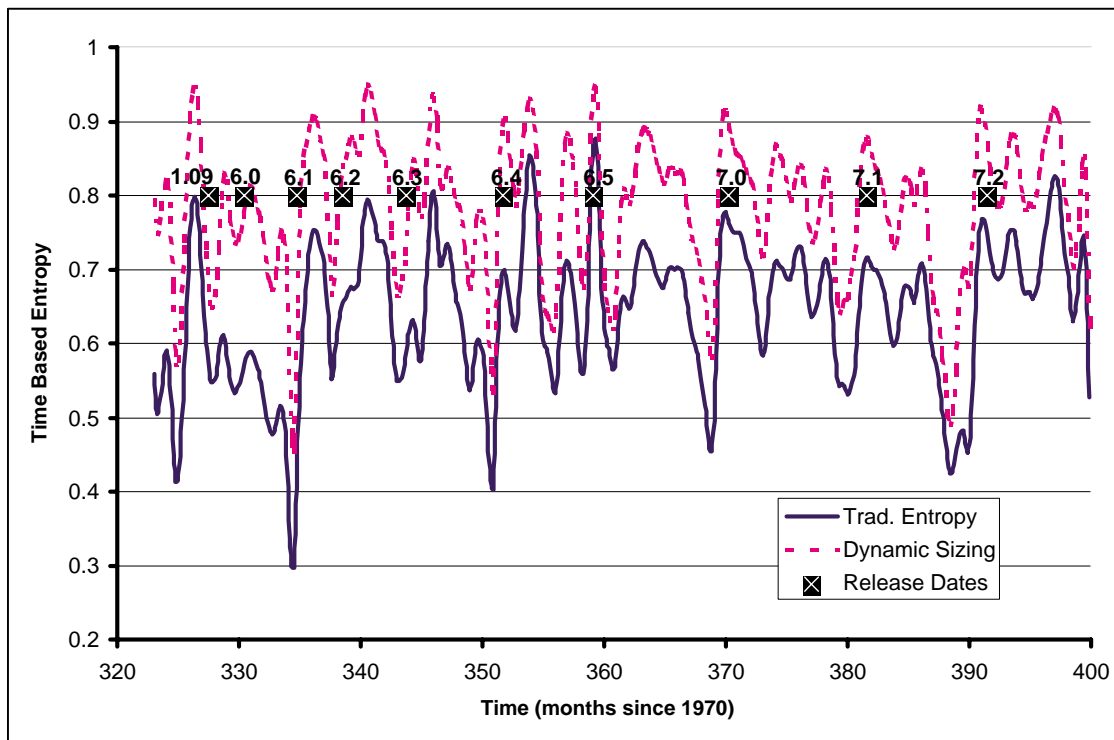
Figure 6: Time Based Entropy Evolution Graph for Postgres

ment team acknowledged that the code base they inherited from the University of California was not robust enough as a production level system. Instead it was a research prototype which they kept on enhancing and patching from release to release. From release 6.4 to release 6.5, we see a very sharp increase in entropy. We associate this sharp increase in entropy to major refactoring done to get a much better understanding and clean up of the code base. We believe this is an example of a controlled entropy increase – a spike in entropy then a drop to an entropy level that is much lower than pre-spike entropy levels. We have seen this occurring in our experiments usually due to major refactoring or redesign. The release notes for 6.5 supports our position:

> *"This release marks a major step in the development team's mastery of the source code we inherited from Berkeley. You will see we are now easily adding major features, thanks to the increasing size and experience of our world-wide development team."*

The effort exerted in release 6.5 apparently made the development of releases 7.0 and 7.1 an easier task with a correspondingly lower entropy measures. In release 7.2, the focus was on adapting the database to support high load. For that release we see an increase in entropy as modifications occur in many places in the code.

We noticed that the dynamic sizing and traditional entropies are highly correlated but with lower values for traditional sizing entropy. The same holds for Figure 6, which uses time based periods.

In Figure 6, we see a different view of the evolution of Postgres. Whereas Figure 5 showed a macro view of the evolution of entropy , Figure 6 shows a micro view of the evolution of entropy. Figure 6 shows the variation in entropy as the software project goes through the different phases in individual releases. For example, in Figure 6 looking at the period from release 6.5 to release 7.0, we see a rise in entropy then a drop in entropy. The increase in entropy is due to the usual task of development where many files are modified to implement the features for the release. Once the features are implemented small localized modifications are done. Concerned that entropy is only measuring the number of modifications and not a true measure of chaos in the development process, we performed a correlation test to verify if entropy and number of modifications are highly correlated and found the correlation to be statistically insignificant.

To get a better understanding of the patterns of entropy change and how they relate to software release dates, we took the continuous entropy values for the period based entropy evolution of Postgres (shown in Figure 5) and we clipped these values to a discrete band. If the entropy is below 0.6, we clip it to 0.4, otherwise we clip it to 0.8. We plotted the newly calculated discrete entropy as shown in Figure 7. We found that the pattern of a rise in entropy then a drop is
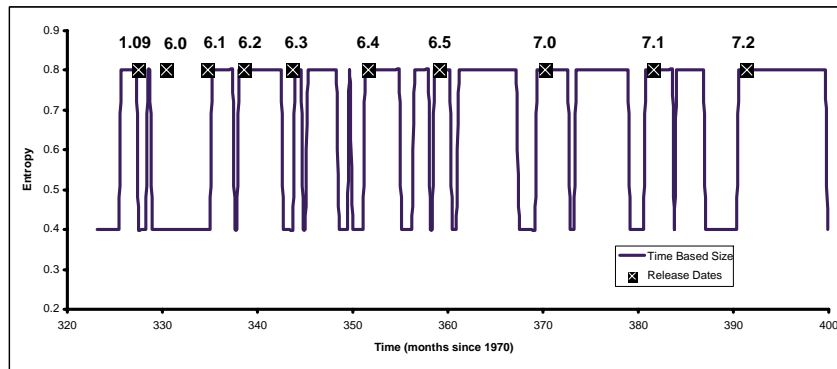
Figure 7: Discrete Period Based Entropy Evolution Graph for Postgres

a consistent pattern for each release, as one would expect. These findings hold for the other systems we examined as well. Also we found that a release never follows on a rise of entropy instead it always follows a drop in entropy. It is interesting that for some releases we can see only one peak in entropy whereas for other releases we can see two to three peaks before the system is released. We attribute this to implementing two or three rather large features in a single release - currently we do not have a way to validate this hypothesis yet.
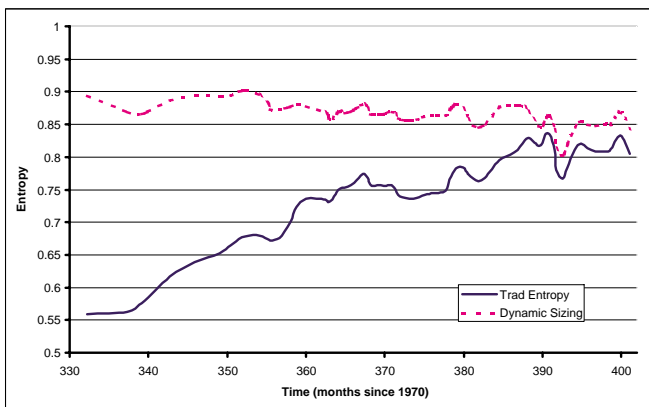
Figure 8: Period Based Entropy Evolution Graph for KDE

Another system we examined in our case study is the KDE (K Desktop Environment) system. The KDE project is an Open Source graphical desktop environment for Unix workstations. It seeks to fill the need for an easy to use desktop for Unix workstations, similar to the desktop environments found under MacOS or Microsoft Windows. With several hundred developers working on it, it is currently over 2.6 million lines of code.

Whereas in Postgres we saw a controlled rise in entropy due to refactoring and re-engineering, in Figure 8 we find an alarming rise in the evolution of traditional entropy for the KDE project. When we view the dynamic sized entropy we notice that the entropy has always been at a very high level.

We conjecture that this has caused the KDE project to suffer major delays as it aims to stabilize the system for release. Miguel de Icaza, the founder of GNOME and Ximian, describes the reasons for delays in KDE project to [8]:

> *"KDE 2.0 was another project that was delayed for a long time because of the nature of the changes they made. The KDE project aimed too high for their KDE 2.0 release: they did incorporate and later they had to drop "OpenParts" support which was a major change to their system. KDE 2.0 was also a very ambitious project, and the nature of the changes delayed the project for a whole year."*

Using our entropy graphs, the high entropy levels would have warned of the significant risks associated with the project. The project leaders could have reacted to the dangers and attempted to control the chaos by reducing the scope of the project or re-thinking their designs/architecture.

8 RELATED WORK

In section 2, we presented an overview of previous work which analyzed source control repositories to gain a better understanding of the software system. Recent work by Barry *et al.* used a volatility rankings 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.* used the source modification records to assist in predicting the efforts in large software systems for AT&T [24]. Other than the work done by Michail *et al.* [6], previous research has focused on studying the source code repositories of closed systems. This focus on closed systems may limit the applicability of the results as they may be dependant on the studied system or organization as the results are usually validated usually using just one system. By focusing on open source systems we are able to get a much larger set of systems to validate our findings. 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 we base our entropy calculation on change statistics associated with the source code, previous studies [1, 3, 4, 5, 18, 32] base their entropy calculation on the source code text. For example, 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 as a whole instead of just focusing on computing the complexity of the source code. We conjecture that detecting chaos in the development process will serve as an early warning measure to prevent the code from becoming too complex over time.

Previous work on software evolution, in particular work by Lehman *et al.* [20, 21, 22] on closed source systems and Godfrey *et al.* [23] on open source systems focus on size (number of modules) and LOC as the principal measure of evolution between the releases of a software system. Instead we focus on measuring the entropy/chaos in the development process. Lehman's second law deals with complexity in the evolution of large software systems – "*As an E-type system evolves its complexity increases unless work is done to maintain or reduce it*". The 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. In our presented case studies, we quantified the idea of complexity/entropy in the development process using our new model. Then we measured that chaos and saw examples of the occasional maintenance activities done to reduce 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 [28]. It has been used for edge detection and image searching in large image database [10]. Also, it has been used for text classification and many text based indexing techniques [9].

9 CONCLUSIONS AND FUTURE WORK
In this paper, we presented a new perspective on the complexity of software. We examined the complexity and chaos associated with the development process. We view the development process as a system with an unknown output, in other words we are uncertain about the files that will be modified by the process over time. Using the ideas of uncertainty and entropy from information theory, we measure how much information exists in the development process. We hypothesize that too much information will require more effort for project members to keep track of the development process over time. Thus the higher the entropy of the system, the more complex the system's code becomes over time as developers grasp of all that is going in the system decays and the team looses its shared image of the software system.

We verified our models and hypotheses using data derived from six large open source projects. Source code repositories provide a rich source of high quality empirical data to validate software engineering results. The repositories are always available for any large project and there are no extra costs associated with collecting the data, as it is always available. We designed our framework so closed source repositories can be analyzed using the same framework.

Using our entropy graphs, managers can monitor with great detail the evolution of complexity in their software system and work hard on controlling it. In the case studies presented in the paper, we saw for the Postgres system a controlled entropy increase due to refactoring. We also saw in the KDE project a continuous rise in entropy. The controlled entropy rise had the beneficial effect of reducing the entropy of the development process for the following releases. Whereas the uncontrolled entropy rise has caused major delays (in years) to the KDE project.

In future work, we plan to correlate our development entropy measures to the bug counts in the system. We would like to know if a complex development process will cause a chaotic appearance of bugs in the software system or if they are two independent entities. We also plan to compare the results of our model to other well known complexity measures such as McCabe cyclomatic complexity. In addition, we would like to validate that our entropy measure satisfies the criteria for a code complexity measure as detailed in [32].

Furthermore, we are examining the use of our entropy model to locate refactoring opportunities in large software systems and to find similarity in development trends over time. We believe that finding similar development periods to the current development period will assist in project and resource planning. Also locating code refactoring will be a valuable tool for developers in large software system to reduce the complexity of their source code and eventually the complexity of developing the code.

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] M. de Icaza. Recent Free Software that Has Suffered Large Delays. Available online at <http://primates.ximian.com/~miguel/gnome-2.0/delays.html>

[9] I. Dhillon, S. Manella, and R. Kumar. Information Theoretic Feature Clustering for Text Classification.

[10] 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.

[11] 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.

[12] 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.

[13] N. E. Fention and M. Neill. A Critique of Software Defect Predction Models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

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

[15] 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.

[16] R. Gonzalez and R. Woods. *Digital Image Processing*. Addison-Wesley Publishing Company, 1992.

[17] 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.

[18] W. Harrison. An Entropy-Based Measure of Software Complexity. *IEEE Transactions on Software Engineering*, 18(11):1025–1029, Nov. 1992.

[19] KDE Homepage - Conquer your Desktop! Available online at <http://www.kde.org>

[20] M. M. Lehman. Programs, Life Cycles and Laws of Software Evolution. *IEEE Transactions on Software Engineering*, 68:1060–1076, 1980.

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

[22] 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.

[23] Michael W. Godfrey and Qiang Tu. Evolution in Open Source Software: A Case Study. In *IEEE International Conference on Software Maintenance (ICSM00)*, pages 131–142, 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 (ICSE 2003)*, pages 274–284, Portland, Oregon, May 2003.

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

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

[27] PostgresSQL Homepage. Available online at <http://www.postgresql.org>

[28] S. Savari and C. Young. Comparing And Combining Profiles. In *Second Workshop on Feedback-Directed Optimization (FDO)*, 1999.

[29] M. J. Sheppaerd. A Critique of Cyclomatic Complexity as a Software Metric. *Software Engineering Journal*, 25(5):30–36, 1988.

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

[31] S. Weaver. *The Mathematical Theory of Communication*. Urbana: University of Illinois Press, 1949.

[32] E. J. Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, Sept. 1988.

[33] H. Zuse. *Software Complexity Measure And Methods*. Watler de Gruyter, 1991.