Visualization and Analysis of Open Source Software Evolution using an Evolution Curve Method

Robertas DAMAŠEVIČIUS

Software Engineering Department, Kaunas University of Technology
Studentų 50-415, LT-51368, Kaunas, Lithuania
e-mail: robertas.damasevicius@ktu.lt

Abstract. Design and evolution of modern information systems is influenced by many factors: technical, organizational, social. This is especially true for open source software systems (OSSS), when many developers from different backgrounds interact, share their ideas and contribute towards the development and improvement of a software product. The evolution of an OSSS is a continuous process of source code development, adaptation, improvement and maintenance. Studying changes to the various characteristics of source code can help us understand the evolution of a software system. In this paper, the software evolution process is analyzed using a proposed Evolution curve (E-curve) method, which is an implementation language independent method based on information theoretic metrics. The method allows identifying major evolution stages of an analyzed software system. The application of E-curves is illustrated for eMule, 7zip, and Grip OSSS.

Keywords: software evolution, open source software, program comprehension, software metrics, software archaeology, data mining.

1. Introduction

Software systems are not purely technical entities. They are designed, constructed and used by people. Therefore, they are components in larger socio-technical information systems (IS), which include technological and social structures [1]. Software design process is not purely a technical task as well, but also a social process embedded within organizational and cultural structures [2, 3]. Software programmers collaborate in teams and groups embedded within larger organizations. These social structures influence and govern their work behaviour and final work products such as source code and documentation. This is especially true for open source software systems (OSSS), when many programmers from different backgrounds interact, share their ideas and contribute towards the development and improvement of a software product. The actions and environment of software developers is rarely directly available for study. Often the only available material for analysis is the results of the programmers’ work such as source code. It can tell us about software design processes, its development history, and provide us with some information about its authors. The structure of software systems can be described in terms of technical relationships between software elements (components, classes, units, etc.) and their quantitative characteristics such as
software metrics. By analysing such characteristics we can analyze the trends and laws of IS evolution and software evolution, in particular.

Software evolution is a continuing process that concerns how a software system evolves over time with respect to some essential software properties. It encompasses software development activities, such as modification, adaptation or maintenance, which occur after the delivery of the first operational release to the users. Understanding the evolution of a software system is a complex task, especially, when it is composed of a large number of components and many versions of the software system must be considered. It has been reported that costs devoted to system maintenance and evolution now account for more than 90% of total software costs [4].

During the lifetime of a large software system usually many versions are produced. Analyzing changes to various characteristics of the source code of these versions and other release information can reveal trends and anomalies in the software development process. The empirical study of software evolution is important because it can help to understand system history, establish main phases of the evolution of the system, identify major factors influencing software change, predict possible evolution trends, as well as develop credible methods to measure, observe, model and analyze software evolution process [5].

The source code of a program at any point in time is the result of many different changes performed in the past, usually by several developers, especially in case of OSS. Free, open source software is providing huge quantities of data suitable to be used in the studies of software evolution. Many different aspects of its development process can be studied from the data available in public source code repositories. The recovery of essential details about an existing software system is sometimes called software archaeology [6, 7], which is an emerging direction in knowledge engineering and program comprehension.

Several papers have studied the evolution of software across several versions of the same system [8-12]. As a software system evolves, new code fragments are added, some parts are deleted, modified or remain unchanged, therefore, overall evolution is difficult to measure using common software metrics such as the number of functions, files or lines of code, McCabe’s cyclomatic complexity [13]. For estimation of software evolution complexity, specific evolution metrics can be used, e.g., System Design Instability (SDI) metric [14] measures the system-level design changes and combines four types of changes in an object-oriented system — the percentage of classes whose names have changed, percentage of added classes, percentage of deleted classes, and percentage of classes whose inheritance hierarchy have changed. L-metric [15] is based on Shannon entropy. It measures how much information exists in the development process. AICC-metric [16] and G-metric [17] are used for estimating system complexity. Hassan and Holt [18] apply the ideas of the information theory and entropy to measure the amount of uncertainty/chaos/randomness in the software development process. Neamtiu et al. [19] perform abstract syntax tree matching between versions of a program and determine the number of bijections (name maps) between old and new versions.

One of the major difficulties of software evolution analysis is huge amount of data that have to be analyzed. Hundreds of versions of thousands of files are common in a single software project. Understanding of this data can be eased using software visualization techniques such as colour-coded lines of code [20], association and sequence rules [21], percentage bars [22], evolution matrix [23], revision towers [24],
evolution spectrographs [25], Kiviat diagrams [26], and multivariate visualization techniques based on partitioning [27].

In this paper, we propose a novel software evolution visualization method, called Evolution curve (or E-curve), based on the information-theoretic metrics (Shannon entropy and Kolmogorov Complexity) of source code, which reflect the change of software system complexity and size during software evolution process. The aim is to identify major stages in evolution of a software system.

The structure of the paper is as follows. Section 2 discusses factors and forces that influence the process of software evolution. Section 3 presents the information theory approach to software evolution measurement. Section 4 describes the proposed E-curve method. Section 5 presents the case studies. Finally, Section 6 presents conclusions.

2. Factors and Forces of Software Evolution

*Open-source software systems* (OSSS) refer to software systems that are free to use and whose source code is freely accessible to anyone. In comparison to traditional IS, OSSS are mostly developed based on a less strict control and management model [28]. Usually an OSSS is started by a single developer, who wants to solve his or her own particular problem and makes the solution available to others for free. Because it is free, it often attracts many users, some of whom may become co-developers by extending or improving the initial system. The evolution of an OSSS begins in response to the needs of users and is carried out by contributing co-developers [29, 30]. Spontaneous collaboration is promoted and backed by a decentralized developer community using internet-based development systems such as SourceForge [31].

The OSSS evolution process is governed by some basic evolutionary mechanisms that also apply to the natural world. Basically, there are two groups of these mechanisms: 1) mechanisms that increase variation in software, and 2) mechanisms that decrease variation. All these mechanisms are influenced by psychological, intellectual, social and cultural, economic and business factors, which may act in any direction. For example, standardization as a business policy decreases variation, whereas product family-based model of software development increases variation.

To better understand the forces and factors that affect software evolution, Lehman [32] has formulated 8 laws of software evolution:

1. **Law of continuing change:** A system undergoes continuing changes until it is judged more cost effective to freeze and recreate it.
2. **Law of increasing complexity:** The complexity of a system increases with time, unless specific work is executed to maintain or reduce it.
3. **Law of statistically smooth growth:** Growth metrics of global system attributes may appear stochastic locally in time and space, but statistically they have well-defined long-range trends.
4. **Law of Organisational Stability:** The rate of development over the life of a software system is approximately constant and independent of the resources devoted to it.
5. **Law of Conservation of Familiarity:** The average incremental growth of a software system remains invariant as the system evolves.
6. **Law of Continuing Growth:** The functional content of software systems must be continually increased to maintain user satisfaction over their lifetime.
7. **Law of Declining Quality:** Quality of software systems declines unless they are rigorously maintained and adapted to operational environment changes.

8. **Law of Feedback System:** Software evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement over any reasonable base.

These laws been empirically proved by studying projects developed in traditional industrial software development environments [33]. For example, it has been established that most large software systems grow at an inverse square rate [32, 34]. However, a number of studies focused on OSSS have discovered some apparent contradictions to the Lehman’s laws [35, 36].

One of the aims of software evolution research is to establish the presence of stages in software evolution and to determine what exactly triggers them. Software evolution is commonly characterized as a slow process of incremental change. However, the researchers [37, 38] have observed that software systems also exhibit characteristics of sudden change during their evolution. In many systems different software evolution phases often overlap themselves. The breakpoints (or junction points) between different phases are called punctuations or *transitions* [25, 37]. These transitions may represent significant changes in the evolving system or in the process by which it is evolved, and separate two periods of stability (or equilibrium) in the evolution process. Transitions occur because as a system evolves, its structure must be regularly adapted to the changing requirements and environments. Within equilibrium periods, a system’s architecture stays relatively stable, and it remains capable of accommodating forecasted changes in requirements. Changes are small and incremental, and they rarely violate the principles imposed by the architecture. However, the architecture may exhibit symptoms of gradual decay, erosion or drift [39], which are normally caused by the accumulating effects of maintenance activities, such as bug fixes and feature modifications. The system architecture needs to be evolved to counter such erosion. This can occur through gradual software changes such as corrections and code cleanups, or by sudden qualitative change which occurs in a short time.

### 3. Information Theory Approach to Software Evolution Measurement

#### 3.1. Shannon Entropy

If a discrete information source (e.g., program source code) is treated as a Markov process, Shannon entropy $H$ [40] can be used to measure how much information is generated by such process. The information source generates a series of symbols $x_i$ belonging to an alphabet with size $N$. If symbols $x_i$ are generated according to a known probability distribution $p(x_i)$, the entropy function $H$ of a sequence $X$ can be defined:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \cdot \log_2 p(x_i)$$

(1)

Shannon entropy is a measure of the uncertainty associated with a random variable. If the message sequence (source code, in this case) consists of a series of symbols, the
quantity of information reaches a maximum, when the sequence satisfies the statistical criterion that all possible subsequences should appear with roughly equal probability.

If the probabilities of symbols are not equal at any position in a sequence then the information conveyed by the sequence is less than a maximum. The difference between the theoretical maximum of entropy and the information contained in a given sequence is called redundancy. The existence of redundancy in a sequence of symbols means that the sequence is not random, and that the sequence is longer than the minimum length necessary to transmit the same information. If the redundancy is reduced to zero, the message becomes random. Thus, the higher the entropy of the system, the more complex the system’s code becomes over time. Low entropy points to the existence of some repeated patterns of source code (e.g., code clones or snippets). Such repetitions may point to reuse of existing code by copying and pasting and indicate the need for code maintenance [41].

3.2. Kolmogorov Complexity

Kolmogorov Complexity [42] measures the ‘complexity’ (i.e., information content) of an object by the length of the smallest program that generates it. Generally, we have a domain object \( x \) and a description system (e.g., programming language) \( \phi \) that maps from a description \( w \) (i.e., a program) to this object. Kolmogorov Complexity \( K_{\phi}(x) \) of an object \( x \) in the description system \( \phi \) is the length of the shortest program in the description system \( \phi \) capable of producing \( x \) on a universal computer:

\[
K_{\phi}(x) = \min \{ \|w\| : \phi_w = x \} \tag{2}
\]

Kolmogorov Complexity \( K_{\phi}(x) \) is the minimal quantity of information required to generate \( x \) by an algorithm, and is the ultimate lower bound of information content. Unfortunately, it cannot be computed in the general case and must be approximated. Usually, compression algorithms are used to give an upper bound to Kolmogorov Complexity. Suppose that we have a compression algorithm \( C \). Then, a shortest compression of \( w \) in the description system \( \phi \) will give the upper bound to information content in \( x \):

\[
K_{\phi}(x) \leq C(x) = \min \{ \|C_w\| : \phi_{C_w} = x \} \tag{3}
\]

The compressibility of sequence \( x \) can be calculated as follows:

\[
\overline{C}(x) = \frac{\|C(x)\|}{\|x\|} \tag{4}
\]

A sequence is considered random, if it is incompressible (\( \overline{C}(x) \approx 1 \)), i.e. we cannot find a shorter algorithm for specifying it than the length of the sequence itself. When \( \overline{C}(x) < 1 \), this indicates the existence of some repeated patterns or similarities within a sequence.
The addition of new features to a software system leads to the change of basic software characteristics (complexity/entropy) in the system. These characteristics are related to the Lehman’s Second Law [32], which deals with complexity in the evolution of large software systems and suggests the need for maintenance activities to reduce complexity that increases as new features are added to the system. We use the change of software size and complexity as a means to determine different stages of evolution of a software system.

Here we propose a novel software evolution visualization method, called Evolution curve (E-curve). This method is inspired by Z-curve [43] and DNA walk [44] methods used in analyzing complex genetic sequences.

The E-curve is composed of a series of nodes $E_i = (x_i, y_i)$, whose coordinates are $x_i$ and $y_i$ ($i=1,2,\ldots,N$), where $N$ is the number of versions of the analyzed software system. The nodes $E_i$ are connected sequentially with straight lines.

The coordinates $x_i$ and $y_i$ are calculated iteratively as follows:

\[
x_i = \begin{cases} 
 x_{i-1} + 1, & \text{if } K_i > K_{i-1} \\
 x_{i-1}, & \text{if } K_i = K_{i-1} \\
 x_{i-1} - 1 & \text{if } K_i < K_{i-1}
\end{cases}
\]  

(5)

where $K_i$ is the Kolmogorov Complexity of the $i$-th version of a software system; 

\[
y_i = \begin{cases} 
 y_{i-1} + 1, & \text{if } H_i > H_{i-1} \\
 y_{i-1}, & \text{if } H_i = H_{i-1} \\
 y_{i-1} - 1 & \text{if } H_i < H_{i-1}
\end{cases}
\]  

(6)

where $H_i$ is the Shannon entropy of the $i$-th version of a system; $x_0 = 0$, $y_0 = 0$.

The two components of the E-curve, $x_i$ (relative information content) and $y_i$ (relative complexity), represent two independent (orthogonal) characteristics of software evolution of the studied system:

- the $x$-component of a E-curve $x_i$ represents the amount of information contained in a software system and is an estimation of software size;
- the $y$-component of a E-curve $y_i$ represents the information entropy of a software system and is an estimation of software complexity.

Based on the behaviour of the E-curve, we can identify 4 major stages of software evolution process:

- **Software Growth**, when $K_i > K_{i-1}$ and $H_i > H_{i-1}$. Software growth is characterized by rapid increase both in actual information content of a system (or software size) and its complexity.
- **Software Maintenance**, when $K_i > K_{i-1}$ and $H_i < H_{i-1}$. Software maintenance is characterized by increase in information content of a system and decrease of its complexity. In other words, software becomes simpler often at a cost of its size.
Software Improvement, when $K_i < K_{i-1}$ and $H_i > H_{i-1}$. Software improvement is characterized by decrease in information content of a system and increase in its complexity, i.e., software becomes more complex and generic.

Software Shrink, when $K_i < K_{i-1}$ and $H_i < H_{i-1}$. Software shrink is characterized by decrease both in information content of a system and its complexity. In other words, the functionality of a system is reduced. Sustainable software shrink is rarely observed in successful software projects.

5. Case Studies

5.1. eMule

eMule [45] is one of the biggest and most reliable P2P file sharing clients around the world. eMule is coded in Microsoft Visual C++ using the Microsoft Foundation Classes. eMule is free software, released under the GNU General Public License. The source code of eMule was first released at version 0.02 and published on SourceForge on July 6, 2002. Many developers contribute to the development of eMule, so the source code grows with every new version. The latest release, 047c, contains 222,680 lines of code. This system is actively developed by 5 developers. Current development status is “Production/Stable”.

For analysis, 68 versions of eMule source code were used starting from release no. 002 to release no. 047c, which span a period of over 4 years. Source code files of each version were concatenated and then analyzed using Shannon entropy and Kolmogorov Complexity metrics. For estimation of Kolmogorov Complexity, zzip compression tool was used, which implements Burrows-Wheeler transform (BWT) algorithm.

![Figure 1](image.png)

Figure 1: Evolution curve of eMule system

Figure 1 shows the Evolution curve constructed for the eMule software system. From the E-curve graph, we can identify 5 major successive stages in eMule evolution: 1) a growth stage until release no. 023b, 2) a rather short maintenance stage until
release no. 025b, 3) a second growth stage until release no. 030e, 4) a second maintenance stage until release no. 044b, followed by 5) a short improvement stage.

The characteristics of major transition points in eMule evolution process are summarized in Table 1.

Table 1: Characteristics of eMule system evolution transition points

<table>
<thead>
<tr>
<th>Ver. no.</th>
<th>Rel. no.</th>
<th>Source code size, B</th>
<th>Inf. content, B</th>
<th>Shannon entropy, b</th>
<th>E-curve node coordinates</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.02</td>
<td>130,092</td>
<td>21,545</td>
<td>5.4767</td>
<td>0 0</td>
<td>First release</td>
</tr>
<tr>
<td>26</td>
<td>0.23b</td>
<td>1,564,188</td>
<td>251,688</td>
<td>5.5493</td>
<td>25 11</td>
<td>End of 1st growth stage</td>
</tr>
<tr>
<td>30</td>
<td>0.25b</td>
<td>1,815,160</td>
<td>292,587</td>
<td>5.5439</td>
<td>27 7</td>
<td>End of 1st maintenance stage</td>
</tr>
<tr>
<td>47</td>
<td>0.30e</td>
<td>4,049,007</td>
<td>627,521</td>
<td>5.5728</td>
<td>42 16</td>
<td>End of 2nd growth stage</td>
</tr>
<tr>
<td>58</td>
<td>0.44b</td>
<td>5,943,455</td>
<td>913,924</td>
<td>5.5426</td>
<td>51 9</td>
<td>End of 2nd maintenance stage</td>
</tr>
<tr>
<td>68</td>
<td>0.47c</td>
<td>7,258,205</td>
<td>1,095,248</td>
<td>5.5549</td>
<td>53 15</td>
<td>Latest release</td>
</tr>
</tbody>
</table>

5.2. 7-zip

7-Zip [46] is an open source file archiver designed originally for the Microsoft Windows operating system, and later made available to other systems. 7-Zip project began in 2000 and is actively developed by 1 developer. Current development status is “Production/Stable”.

For analysis, 82 versions of 7-zip source code were used starting from release no. 2.30 beta 5 to release no. 4.52 beta, which span a period of over 5 years. The latest release, 4.52 beta, contains 1189 source files and 160,227 lines of code. Source code files of each 7-zip version were concatenated and then analyzed using Shannon entropy and Kolmogorov Complexity metrics. For estimation of Kolmogorov Complexity, bzip compression tool was used, which uses several compression techniques such as run-length encoding (RLE) and BWT.

Figure 2: Evolution curve of 7-zip system
Figure 2 shows the Evolution curve constructed for the 7-zip software system. From the Evolution curve graph, we can identify 3 major successive stages in 7-zip evolution: 1) a maintenance stage until version 2.30 beta 32, 2) a growth stage until version 3.11, and 3) a maintenance stage through the latest version 4.52.

As we can see from Figure 2, the implementation of the project’s functionality has already been finished at the time of creation of the first analyzed version of 7-zip and later the developer concentrated his efforts mainly on improvement and maintenance. This is confirmed by the history log of the system. Major event around version no. 29 (release 3.08 beta) was caused by “big source code reconstruction” as described in the history log.

The characteristics of major transition points in 7-zip evolution process are summarized in Table 2.

<table>
<thead>
<tr>
<th>Ver. no.</th>
<th>Rel. no.</th>
<th>Source code size, B</th>
<th>Inf. content, B</th>
<th>Shannon entropy, b</th>
<th>E-curve node coordinates</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.30b5</td>
<td>1,544,453</td>
<td>280,122</td>
<td>5.3699</td>
<td>0</td>
<td>First available version</td>
</tr>
<tr>
<td>28</td>
<td>2.30b32</td>
<td>2,179,727</td>
<td>426,105</td>
<td>5.3539</td>
<td>19</td>
<td>End of 1st maintenance stage</td>
</tr>
<tr>
<td>36</td>
<td>3.11</td>
<td>2,243,161</td>
<td>429,608</td>
<td>5.3319</td>
<td>21</td>
<td>End of growth stage</td>
</tr>
<tr>
<td>82</td>
<td>4.52</td>
<td>3,189,264</td>
<td>618,339</td>
<td>5.2903</td>
<td>43</td>
<td>Latest release, ongoing maintenance stage</td>
</tr>
</tbody>
</table>

5.3. Grip

Grip [47] is a GTK-based CD-player and CD-ripper/MP3 encoder. The Grip’s project began in 2000 and is actively developed by 1 developer. Its current development status is “Mature”.

![Figure 3: Evolution curve of Grip system](image)
For analysis, 36 versions of Grip source code were used starting from release no. 2.93 to release no. 3.3.1, which span a period of over 5 years. The latest release, 3.3.1, contains 20 source files and 1405 lines of code. Source code files of each Grip version were concatenated and then analyzed using Shannon entropy and Kolmogorov Complexity metrics. For estimation of Kolmogorov Complexity, \textit{zzip} was used.

Figure 3 shows the Evolution curve constructed for the Grip software system. From the Evolution curve graph, we can identify 5 major successive stages in Grip evolution: 1) a maintenance stage until version 2.98.0, 2) a growth stage until version 2.98.5, 3) a maintenance stage until version 3.0.5, 4) an improvement stage until version 3.1.4, and 5) a maintenance stage until version 3.3.1.

As we can see from Figure 3, the implementation of the Grip project’s functionality has already been finished and the developer mainly concentrates on its improvement and maintenance. The characteristics of major transition points in Grip evolution process are summarized in Table 3.

<table>
<thead>
<tr>
<th>Ver. no.</th>
<th>Rel. no.</th>
<th>Source code size, B</th>
<th>Inf. content, B</th>
<th>Shannon entropy, (b)</th>
<th>E-curve node coordinates (x_i, y_i)</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.9.3</td>
<td>194,539</td>
<td>33,926</td>
<td>5.4562</td>
<td>0, 0</td>
<td>First available release</td>
</tr>
<tr>
<td>5</td>
<td>2.98.0</td>
<td>345,480</td>
<td>58,021</td>
<td>5.3932</td>
<td>3, -5</td>
<td>End of 1st maintenance stage</td>
</tr>
<tr>
<td>10</td>
<td>2.98.5</td>
<td>363,153</td>
<td>62,630</td>
<td>5.4235</td>
<td>7, -1</td>
<td>End of growth stage</td>
</tr>
<tr>
<td>16</td>
<td>3.0.5</td>
<td>379,760</td>
<td>66,994</td>
<td>5.4274</td>
<td>9, -5</td>
<td>End of 2nd maintenance stage</td>
</tr>
<tr>
<td>25</td>
<td>3.1.5</td>
<td>282,878</td>
<td>47,054</td>
<td>5.4715</td>
<td>10, -4</td>
<td>End of improvement stage</td>
</tr>
<tr>
<td>36</td>
<td>3.3.1</td>
<td>315,805</td>
<td>52,492</td>
<td>5.4556</td>
<td>15, -5</td>
<td>Latest release, ongoing maintenance stage</td>
</tr>
</tbody>
</table>

**6. Conclusion**

Our case studies show that evolution process of open-source software can be coarsely divided into four stages: 1) software growth, where the size and complexity of developed software is increasing, 2) software maintenance, where the aim of the developer is to contain complexity and fix software bugs, 3) software improvement, where the aim is to contain software system size at a cost of increasing complexity, and 4) software shrink, when both software size and its complexity is trimmed.

These software evolution stages can repeat several times during the software system’s life cycle, or even overlap themselves. The need for such repeating stages can be justified by the Lehman’s Second Evolution Law, which claims that every growth of a large software system must be followed by necessary maintenance actions to reduce the growth of source code complexity.

We can identify these software evolution stages using the proposed Evolution curve (E-curve) method, which can visualize software evolution process based on the Kolmogorov Complexity and Shannon entropy characteristics of software source code. The E-curve also can identify the initial development status of the analyzed software
system: actively developed systems show long growth trends, while mature systems show maintenance and improvement trends.

Furthermore, the E-curve method is independent from the implementation language of the analyzed software system, because both metrics, Shannon entropy and Kolmogorov complexity, are computed on plain source code files with no regard to the syntax or semantics of the particular programming language.

Future work will focus on the extension of the E-curve method with other entropy measures such as block entropy and Rényi entropies, which could allow to reveal more interesting evolution events and transition points hidden in software source code.

References


