Empirical Studies of Open Source Evolution

J. Fernandez-Ramil[1], A. Lozano[1], M. Wermelinger[1] and A. Capiluppi[2]

[1] Computing Department, The Open University, United Kingdom
[2] Department of Computing & Informatics, Lincoln University, United Kingdom

Summary. This chapter presents a sample of empirical studies of Open Source Software (OSS) evolution. According to these studies, the classical results from the studies of proprietary software evoltion, such as Lehman's laws of software evolution, might need to be revised, if not fully, at least in part, to account for the OSS observations. The book chapter also summarises what appears to be the empirical status of each of Lehman's laws with respect to OSS and highlights the threads to validity that frequently emerge in these empirical studies. The chapter also discusses related topics for further research.

1 Introduction

Software evolution is the phenomenon of software change over years and releases since inception to the final decommissioning of a software system. The evolution of a large software system poses many challenges. Software development and maintenance can be improved by taking into account the findings of empirical studies of evolving software systems.

With the emergence of the open source paradigm, software evolution researchers have access to a larger number of evolving software systems for study than ever before. This has led to a renewed interest in the empirical study of software evolution. Some surprising findings in open source have emerged that appear to diverge from the classical view of software evolution. In this book chapter we examine this and argue that the divergence can be explained using simple system theory concepts. We also propose research topics for further advance in this area.

1.1 Classical Views of Proprietary Software Evolution

In the late 1960s and early 1970s Lehman and his collaborators pioneered the empirical study of the evolution of software. They examined a number of

proprietary systems, including the IBM 360-370 operating system [1]. In the late 1970s and early 1980s they studied measurement data from several other systems. Their initial fascination was with the phenomenon of *large program growth dynamics*. But later they realised that the phenomenon was not only a property of large systems, partly because *largeness* cannot be unambiguously defined. What they saw was a process of change in which software systems acquire additional functionality and other characteristic that could be legitimately called *software evolution*. Their intention wasn't to make an analogy with Darwinian evolution, particularly when the differences between software and biological entities are so notorious, but to highlight that software artefacts experienced progressive enhancement in functional power and other changes in characteristics that could be legitimally called evolution.

Lehman realised that software evolution, the continual change of a program, was not a consequence of bad programming, but something that was inevitably required to keep the software up-to-date with the changing operational domain. Continual software change was needed for the stakeholders' satisfaction to remain at an acceptable level in a changing world. This matched well with the software measurements that he and colleagues had collected. This realisation was so compelling that this observation was termed the *law* of continuing change. The use of the term *law* was justified on the basis that the phenomenona they described was beyond the control of individual developers. The forces underlying the laws were believed to be as strong as those of the laws of demand and supply. Other empirical observations were encapsulated in statements and similarly called laws. Initially three laws were postulated, followed by five that were added at various points later, giving a total of eight.

Despite the strong confidence on the validity of the laws, the matter of universality of the laws was not sufficiently well defined. Anyone could always recall a program that was developed, used only once or twice and then discarded. Hence, the first requisite for evolution is that there is a continual need for the program, i.e. there is a community of users for which running the program provides some value. Lehman's analysis, however, went deeper and led to the realisation that, strictly speaking, the laws only applied to a wide category of programs that Lehman called *E-type systems* [1], where the "E" stands for *evolutionary*. An E-type system is one for which the problem being addressed (and hence, the requirements and the program specification) can't be fully defined. E-type software is always, to some extent, incomplete and addresses open problems (open in the sense that the change charter has arbitrary boundaries that may change at any time and that the specification can always be further refined or modified in some other way). The immediate consequence is that for an E-type program there is always a possible or perceive need for change. Another characteristic of E-type system is that the installation of the program in its operational domain *changes the domain*. The evolution process of an E-type program becomes a *feedback system* [1]. This

is illustrated in Fig. 1.

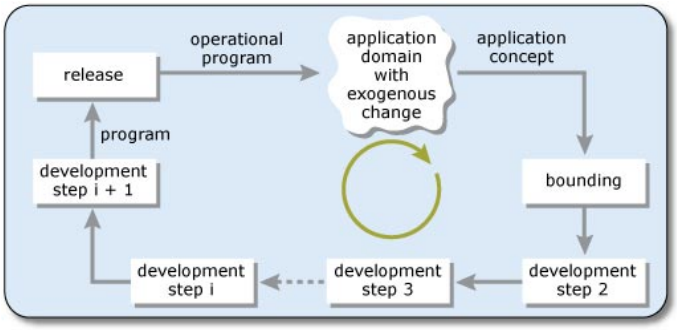

Fig. 1. Lehman's view of E-type Software Process. Taken from [2]

E-type systems contrast with S-type programs, where the "S" stands for *specified*. In S-type programs the specification is complete and should be expressible formally using Mathematics. In S-type programs mathematical arguments are used to prove that the program fully satisfies its specification. S-type programs represent the domain within which the application of formal verification methods is more meaningful and likely to be effective. However, the vast majority of systems used in businesses and by the general public are of the type E. Hence the importance of this type and the laws that are claimed to be descriptions of their evolutionary characteristics. In its original classification [1], Lehman also identified another type, called P-type (*problem*). The P-type related to programs that are based on heuristics rather than mathematical proof. They are generally characterised by some trade-offs in their requirements and their results are satisfactory only to certain level (not absolutely correct as in the case of S-type programs). If a P-type program is actively used in a real-world application it is likely to acquire, at least to some extent, E-type properties. Traditionally, the software evolution research has concentrated on E-type programs since it is the most common of the three.

Initially the topic of empirical study of software evolution did not reach much momentum beyond Lehman's immediate circle of collaborators. There were two independents studies in the 1980s: one confirmatory by Kitchenham [3] and one that was mainly critic and unsupportive by Lawrence [4]. Lawrence [4] took a statistical approach and found support for one of the at that time five laws. Three of the laws were not supported by his tests and he was not able to formulate one of the laws into proper statistical tests. The key contribution of Lawrence's study is that laws are informal statements and that their formal testing against data involve their formalisation. Because each law can be formalised in more than one different way, it may lead to more than

one test for each law.

Despite these empirical challenges and the common perception that software is not restricted by any natural laws, the wider software engineering community seemed to progressively realise that Lehman's laws were a legitimate, possibly the most insightful so far, attempt to describe why software evolves and what evolutionary trends software is likely to display. The laws appeared to match common experience and were discussed in popular software engineering textbooks and curricula [5, 6]. The laws should be considered, at the very least, hypotheses worth further studying.

In the late 1990s and early 2000s a fresh round of empirical studies by Lehman and colleagues took place (e.g. [7]). These involved five proprietary systems that were studied in the FEAST projects with results widely publicised [8]. FEAST led to the refinement of some of the laws, which, as we said, are currently eight in number. The laws were no longer isolated statements: the phenomena they describe are interrelated. The project realised that empirical data related to some of the laws were easier to extract than for others. Despite the difficulties, the laws were generally supported by the observations and seen as the basis for a theory of software evolution. The laws, in a recent post-FEAST wording, are listed in Table 1.

As can be seen in Table 1 a recent refinement of the fourth law included the text "The work rate of an organisation evolving an E-type software system tends to be constant over the operational lifetime of that system *or segments of that lifetime*", with the most recent addition in italics. This apparently minor addition recognised explicitly in the laws for the first time the possible presence of discontinuities in the lifetime of a software system and was a consequence of the observation in FEAST of breakpoints in growth and accumulated changed trends. Other researchers [9, 10] arrived to similar independent views that software evolution tends to be discontinuous. Aoyama [9] studied the evolution of mobile phone software in Japan over a period of four years in the late 1990s. During this time mobile phones went through a fast evolution from voice communication devices to mobile internet Java-enabled terminals. The code base studied by Aoyama increased its size by a factor of four in four years within which the software experienced significant structural changes at particular points. In Aoyama's view, dealing with discontinuities in evolution is an unresolved challenge. The immediate consequence is that it may not be sensible to simply project evolution trends, such as growth or change rate, to predict the future of a system.

In connection to the idea of discontinuity, an important addition to the description of how proprietary systems evolve came from two software maintenance researchers, Bennett and Rajlich [10], in the form of their *staged model of the software lifecycle*. A key idea contributed by Bennett and Rajlich is

that there are distinctive phases or stages, as illustrated in Fig. 2. While skipping unnecessary details, the staged model indicates that systems tend to go through distinctive phases, termed *initial development*, *evolution*, *servicing*, *phase-out* and finally *close-down*, and that the management of such of these phases involves specific challenges. Bennett and Rajlich chose to call *evolution* to one of their phases, possibly because according to them it is within this phase that software is actively enhanced and changed. During the so-called servicing phase, only minor fixes are implemented to keep the system running before phasing it out.

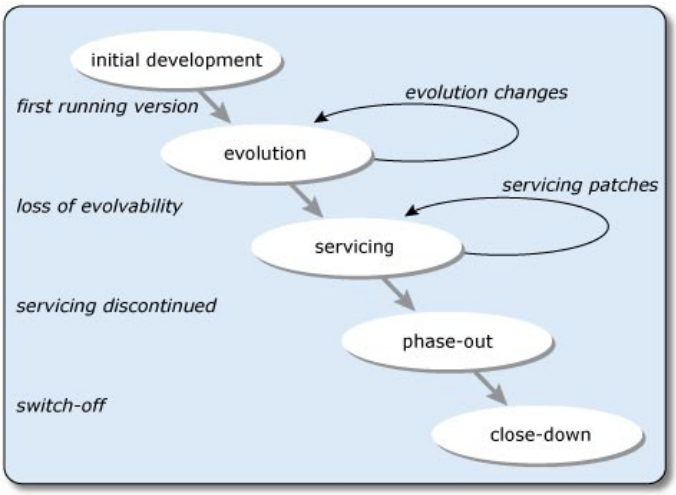

Fig. 2. Staged model of the software life-cycle[10]. Taken from [2]

This section has presented a brief account of the situation with regards to empirical studies of proprietary system evolution. With the emergence of open source, software evolution researchers can access vast amounts of software evolution data which is now available for study. Some of the initial findings (e.g. [11]) were concerning because they suggested that OSS evolutionary patterns can be different to the ones suggested by the laws and generally expected in proprietary software evolution. This and other OSS studies will be examined in the remainder of this chapter with the aim of providing the reader with an overall picture of the past and current empirical OSS evolution research.

1.2 The Emergence of Open Source

The emergence of *open source software* (OSS) and *free software* [3], has provided researchers with access to large amounts of code and other software artefacts (e.g. documentation, change-log records, defect databases, email conference postings) that they can use in their studies. For example, using OSS data researchers are able to test certain hypotheses about the effectiveness, of a software engineering technique or the validity of theory. OSS has become an established approach to distribute software as a *common good* . This is the free software ideal defended by the Free Software Foundation and others. It is often emphasised that in free software, "free" is used as in "freedom", not as in "free beer". The following quotation from the Debian website (one of the largest Linux distributions) captures well the open source philosophy:

> "While free software is not totally free of constraints...it gives the user the flexibility to do what they need in order to get work done. At the same time, it protects the rights of the author. Now that's freedom."[4]

The OSS paradigm is well documented in the literature with many studies describing it(e.g. [12]). Its main characteristic is that source code is shared with only some restrictions (e.g. normally any changes can only be released as OSS and under the same license restrictions). Contributors are working in their free time with their own computing resources even though businesses are getting increasingly engaged in some OSS projects for market and other reasons. The actual OSS process is highly streamlined, with the code being the main artefact for sharing knowledge and understanding amongst contributors. Release notes, email lists, defect databases and configuration management facilities are frequently provided by a project. However, one should not expect to find in OSS other advanced software engineering tools such as, for example, requirements engineering tools. One could say tha tOSS is *code-centric* and it is unlikely that documents and other artefacts will be present such as a formal or informal requirements specification, a program specification or some formal representation of the architecture of a system. To different degrees, individuals can control the evolution of a system. They operate as gate keepers or main architects under a set of policies or rules which may change from community to community and from project to project. Normally there are two evolving streams of code that are interrelated, the so-called stable or apt for distribution stream, and the developmental, which is the one currently being changed and enhanced. From time to time, development releases become stable and are distributed. There are not external constraints on the timing of releases and in many projects there is no formal testing (e.g. no test cases are available).

---

[3] In this chapter we use "open source" and "free" as synomyns, even though there are slight differences in meaning (see their glossary entries).

[4] http://www.debian.org/intro/free (as of Nov 2006)

Since the late 1990s the OSS-related contributions to the empirical software evolution literature have increased. It is useful to distinguish here two type of studies. On the one hand, there are *technology-oriented* papers. These constitute what Lehman called the "*how* view of evolution". These papers address a particular technical problem in implementing or supporting software evolution processes and propose a technique to address such problem. On the other hand, one encounters *empirical studies* that gather and analyse observations of the OSS evolution phenomenon and an attempt to their modelling and explanation, trying to find out what Lehman termed the "*what and why* view of evolution". These empirical studies aim at characterising software evolution, identifying general or particular evolutionary patterns and either to increase our understanding of the phenomenon or to inform good evolution practice. The empirically-oriented papers that we have selected for our discussion examine code evolution, i.e. sequences of code versions or releases, and provide empirical observations related to classical view of software evolution. These include OSS functional growth patterns and OSS compliance or not with Lehman's laws.

The structure of this chapter is as follows. Section 2 presents and summarises a number of relevant empirical studies of open source evolution. Section 3 compares the findings with regards to the evolution in OSS and proprietary systems and discusses the currently stressed relationship between empirical studies of OSS evolution and the classical theories of software evolution. The main threats to validity of the surveyed studies are summarised in Section 4. Section 5 concludes this chapter and indicate topics for further work.

2 Empirical Studies of Open Source Evolution

Pirzada's PhD thesis [13] was the first study that singled out differences between the evolution of Unix operating system and the systems studied by Lehman *et al.* (e.g. [1]). Pirzada's work was still in the pre-Internet days and open source was yet to arrive. However, Pirzada's study should be credited with arguing, probably for the first time, that differences in development environments, in this case, academic vs industrial development, could lead to differences in the evolutionary patterns. If Pirzada was right we should expect differences between OSS and proprietary evolution. However, study of OSS evolution started 10 years or so later. In the next sections we summarise some of the most relevant empirical studies of OSS evolution to date.

2.1 The Linux kernel study by Godfrey and Tu [11]

Godfrey and Tu [11] studied the growth trend of the popular OSS operating system Linux, for which Unix was a precursor, over six years of its lifespan,

with data covering the period 1994 to 1999. Development of Linux started as a hobby by Linus Torvalds in Finland. The system was then publicly released and experienced an unprecedented popularity with hundreds of volunteers contributing to Linux. In 2000 more than 300 people were listed as having made significant contributions to the code. Godfrey and Tu found that Linux, a large system with about 2 million LOCs at that time, had been growing superlinearly. This essentially meant that the system was growing with an increasing growth rate. These authors found that the size of the Linux followed a quadratic trend. This type of growth was fully in line with Lehman's sixth law, but the superlinear rate contradicted some consequences of the second law, such as growth rate slow down as complexity increases. It also appeared to contradict laws three (self-regulation) and fifth (conservation of familiarity).

Godfrey and Tu found that the growth rate was higher in a one particular sub-system of Linux that holds the device drivers as can be seen in Fig. 3. Drivers enable a computer to communicate with a large variety of external or internal hardware devices such as network adapters and video cards. Their explanation for Linux high growth rate was that drivers include code which tends to be independent of each other. Another significant part of the Linux code base was the replicated implementation of features for different CPU types, giving the impression that the system was larger than it really was. The core or kernel of Linux represents only a small part of the code repository. These authors recommended, in line with previous researchers [14], that evolution patterns should be visualised not only for the total system but also individually for each subsystem.

Godfrey and Tu's study was later replicated by Robles *et al.* [**?**] using independently extracted data from the Linux repository and who also identified a superlinear growth trend.

2.2 The Comparative Study by Paulson *et al.* [15]

Paulson *et al.* [15] compared the evolution of three well-known and successful OSS (the Linux operating system *kernel*, the Apache HTTP web server, and the GCC compiler) and three proprietary systems in the embedded real time systems domain (the proprietary systems were described as "software protocol stacks in wireless telecommunication devices"). They chose to look at the Linux kernel because in their view it was more comparable to their three proprietary systems than the Linux system as a whole. The five hypotheses studied were: (1) OSS grows more quickly than proprietary software, (2) OSS projects foster more creativity, (3) OSS is less complex than proprietary systems, (4) OSS projects have fewer defects and found and fix defects more rapidly, and (5) OSS projects have better modularization. The measurements used to test these hypotheses were as follows:

1. For hypothesis 1, related to size (or growth): number of functions and in lines of code (LOCs) over time.

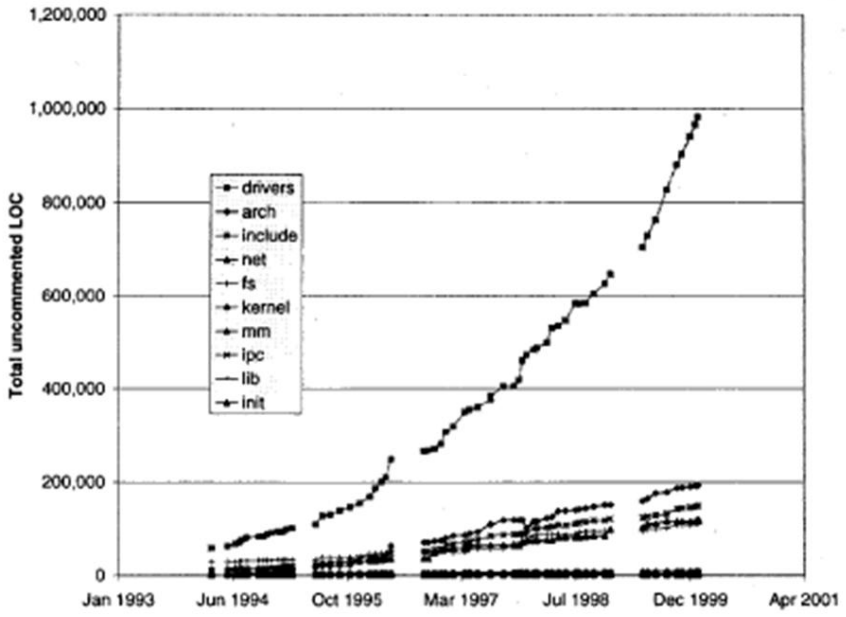

Fig. 3. Growth of Linux's major subsystems (development releases only). Source [11].

2. For hypothesis 2, related to creativity: functions added over time.
3. For hypothesis 3, related to complexity: overall project complexity, average complexity of all functions, average complexity of added functions.
4. For hypothesis 4, related to defects: functions modified over time, percentage of modified functions with respect to total.
5. For hypothesis 5, related to modularity: correlation between functions added and modified.

Only hypotheses (2) and (4) were supported by the measurements. However, with respect to hypothesis 2, it could be an over simplification to assess creativity by simply looking at the number of functions added over time, without taking into consideration the number of developers. With respect to hypothesis 4, one would have expected some direct measure of defects or defect density, instead of simply looking at functions. For these reason we conclude that these two hypotheses are not easy to investigate based on the measurements chosen and raise some questions.

The investigation of the other three hypothesis seem to have been much more straight-forward. Paulson *et al.* found that the growth of the six systems analyzed was predominantly linear. They compared their results with the averaged data by two other groups of researchers (see Fig. 4), finding that

the slopes in the data by others matched well into the pattern they found. Paulson *et al.* also found, using three different complexity measures, that the complexity of the OSS projects was higher than that of the proprietary systems, concluding that the hypothesis that OSS projects are simpler than proprietary systems was not supported by their data. As said, one further aspect investigated was modularity. They looked at the growth and change rates, arguing that if modularity is low, adding a new function will require more changes in the rest of the system than if modularity is high. No significant correlation was found between the growth rate and change rate in proprietary systems, but such correlation was present in OSS projects. Hence, no support was found to the hypothesis that OSS projects are more modular than proprietary systems.

Whereas Godfrey and Tu (see section 2.1) found superlinear growth in Linux, Paulson *et al.* detected linear growth. These two findings do not necessarilly contradict each other because the former study was looking at Linux as a whole, while the latter focused on the kernel, which is one of its subsystems and does not include drivers.

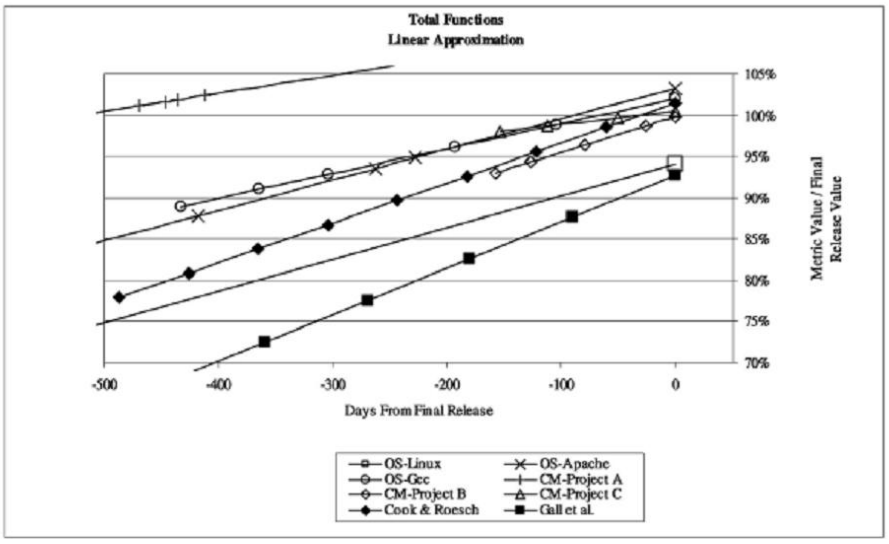

Fig. 4. Total size of systems studied by Paulson *et al.* and by other researchers (linear approximations). Source [15].

2.3 The Study of Stewart *et al.*[16]

Stewart *et al.* [16] explored the application of a statistical technique called *functional data analysis* (FDA) to analyze the dynamics of software evolution in the OSS context. They analysed 59 OSS projects in order to find out whether their structural complexity increases over time or not. Two measurements of complexity were considered: coupling and lack of cohesion. The higher a program element is related to others, the higher the coupling. The higher the cohesion was, the stronger the internal relationships within an element of a program. They consider that generally there is trade-off between the two measurements (i.e. increasing cohesion leads to a decrease in coupling). For this reason they use the product of the two attributes "coupling × lack of cohesion", as their measurement of interest. These authors found that FDA helped to characterize patterns of evolution in the complexity of OSS projects. In particular, they found two basic patterns: projects for which complexity either increased or decreased over time. When they refined their search for patterns they actually found four patterns, as shown in Fig. 5. The name give to each of these patterns (and the number of projects under each) were *early decreasers* (13), *early increasers* (18), *midterm decreasers* (14) and *midterm increasers* (14).

Another differentiating factor, not represented in Fig. 5, was the period of time, shorter or longer, during which projects appeared to be most active. These researchers explored factors that might explain such patterns, as both functional growth and complexity reduction are desirable evolution characteristics. They discuss that contrary to their hypotheses, neither the starting size or the increase of size was significantly different between increasing and decreasing complexity clusters. Moreover, there was not a significant difference in the patterns on the average release frequency between increasing and decreasing complexity clusters. The authors hypothesise that the results may relate to the number of people involved in the project. Generally a correlation is expected between the number of contributors and the complexity. Projects with low complexity may initially attract and retain more people than others, but if they become very popular, their complexity may later increase. This may explain the midterm complexity increase pattern observed. However, in this study the number of contributors was not measured and this is suggested as further work.

2.4 The Study by Herraiz *et al.* [17]

Herraiz *et al.* [17] examined the growth of 13 OSS systems. This sample included some of the largest packages in the Debian/Linux distribution. They concluded that the predominant mode of growth was superlinear. The choosing of the large and popular Debian/Linux distribution was an attempt of achieving a representative sample of successful OSS projects. After various

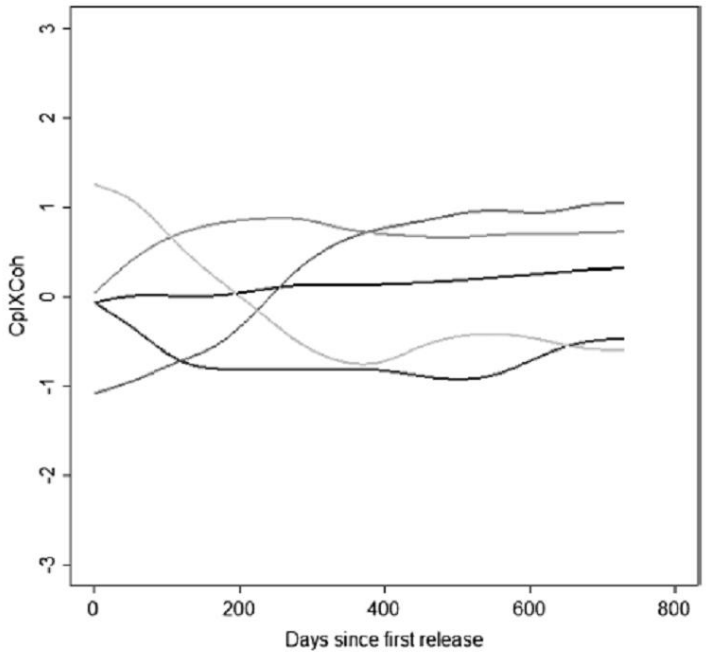

Fig. 5. Four clusters (mean functions) detected by Stewart *et al.* in their sample of 59 OSS projects [16].

technical considerations, 13 projects were selected for study. Mathematical models were fitted to the growth trends and the best fits were selected, determining that six projects where experimenting superlinear growth, four projects displayed linear growth and three projects were sublinear. The size measurements were made using number of files and number of lines or statements in the source code (SLOCs), with both measurements giving similar results. This research, that looked at Linux growth data from 1991 to 2003 or so, confirming that Linux have been still growing superlinearly since Godfrey and Tu's study [11] six years before. Table 2 lists the names of the OSS systems studied, their growth rates and the identified overall growth trends. For the growth rates, what's only relevant is their sign[5], positive, approximate zero or negative, which indicates predominantly superlinear, linear or sublinear growth.

[5] Herraiz *et al.* [17] fitted a quadratic polynomial to the SLOC and number of files data and looked at the coefficient of the quadratic term as an indication of the overall trend.

Number (year)	Name	Statement
I (1974)	Continuing change	An E-type system must be continually adapted otherwise it becomes progressively less satisfactory in use, (and) more difficult to evolve
II (1974)	Increasing complexity	As an E-type system is evolved its complexity increases unless work is done to maintain or reduce the complexity
III (1974)	Self regulation	Global E-type system evolution is regulated by feedback
IV (1978)	Conservation of organisational stability	The work rate of an organisation evolving an E-type software system tends to be constant over the operational lifetime of that system or segments of that lifetime
V (1991)	Conservation of familiarity	In general, the incremental growth (growth rate trend) of E-type systems is constrained by need to maintain familiarity
VI (1991)	Continuing growth	The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over the system lifetime
VII (1996)	Declining quality	Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining
VIII (1971/96)	Feedback system	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

Table 1. Laws of E-type Software Evolution

Project	Growth rate (SLOCs)	Growth rate (files)	Category
Amaya	1.45	-0.0055	linear
Evolution	-31.89	-0.17	sublinear
FreeBSD*	15.16	0.056	linear
Kaffe	77.13	0.71	superlinear
NetBSD*	152.74	1.04	superlinear
OpenBSD*	401.20	2.01	superlinear
Pre tools	4.31	0.044	superlinear
Python	18.43	-0.062	linear
Wine	50.06	0.064	linear
wxWidgets*	587.56	0.29	superlinear
XEmacs	-259.44	-0.60	sublinear
XFree86	-412.28	-1.47	sublinear
Linux*	186.21	0.71	superlinear

Table 2. Growth rates and overall growth trend in some Debian packages. Growth rates are semiannual (For projects with * monthly, rather than seminannual, growth rates are indicated.) Taken from [17].

2.5 The Study by Wu *et al.*[18, 19]

Wu *et al.* [18, 19] analyzed the evolution of three OSS systems (Linux, OpenSSH, PostgreSQL). One of the contributions of this work is to have put forward evidence that reinforces the observation that OSS evolution goes through periods of relatively stability where small, incremental changes are implemented, separated by periods of radical restructuring, where architectural changes take place. These are changes that may occur in relatively short periods of time and that virtually transform the architecture of an evolving system and the subsequent evolution dynamics. Fig. 6 presents one of the results derived by Wu [19] for Linux using the *evolution spectograph* [18] visualisation technique. This type of graph shows the time on the x-axis, whereas the y-axis is mapped to a single element (e.g. a file) in the system. Files are ordered on the y-axis based on their creation date, from the bottom upwards. Every horizontal line in the graph describes the behaviour of a property (e.g. number of dependencies) over time for each element. Whenever the property changes for an element at a point in time, that portion of the horizontal line is painted with strong intensity. If the property does not change or changes little, the intensity gradually decreases and the line fades away. Changes in colour intensity that can be seen vertically denote many elements having changes in that property. When vertical lines appear on the spectrograph, these indicate massive changes across the system. As one can see in Fig 6, there is evidence for at least four major Linux restructurings, identified with the release codes in the figure.

2.6 The Study of Capiluppi *et al.* [20, 21, 22]

Capiluppi *et al.* [20, 21, 22] studied the evolution of approximately 20 OSS systems using measurements such as growth in number of files, folders and functions; complexity of individual functions using the McCabe index [**?**]; number of files handled (or touched) [1] and amount of *anti-regressive work* [1].

Segmented Growth Trends

One example of the systems studied is Gaim, a messenger that runs under Linux, Windows, MacOS X and BSD. The growth trend of this system, in number of files and folders, is presented in Fig. 7.

In Gaim, one cannot easily identify which is its overall growth pattern. From day 1 to day 450 or so the growth pattern is superlinear. Then, growth essentially stops until day 1200, after which growth is resumed at a linear rate. It is difficult to predict what type of curve (linear, sublinear, or superlinear) will come out if this data is fed into a curve fitting algorithm. Gaim provides evidence of the fragmented nature of software growth patterns: growth patterns can be abstracted differently depending on the granularity of the

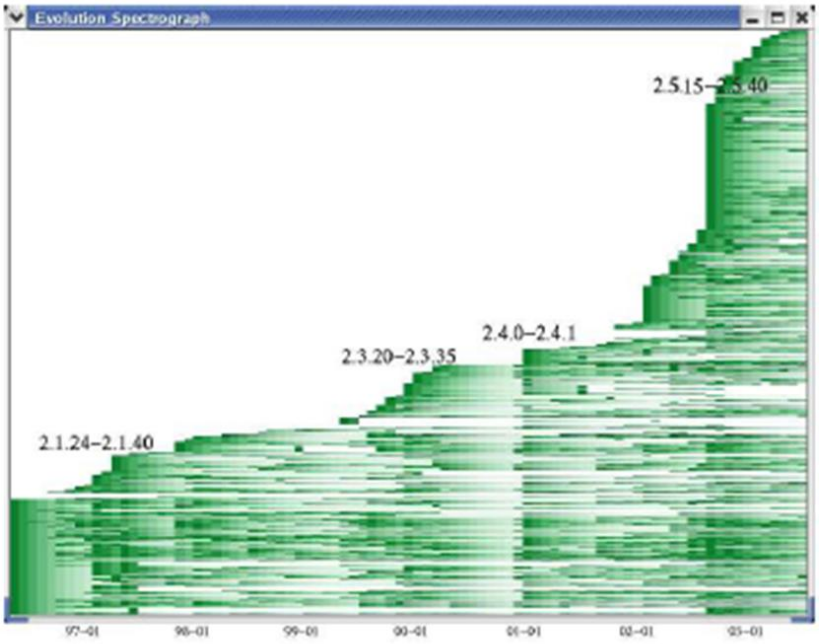

Fig. 6. Outgoing dependency changes in Linux [19].

observations. Another OSS system studied, Arla, showed a positive sublinear growth followed by stagnation (Fig. 8.

While the growth pattern of Arla is smoother than that of Gaim, overall it is a sublinear growth pattern. Nevertheless, it can also be seen as an initial superlinear trend, up to day 125, then followed by a sublinear trend, up to day 400 or so, followed by a short period of no growth, then followed by linear growth until day 1,000, and, more recently, a period of no growth. As in the Gaim case, in Arla, the interpretation of a fragmented growth trend as an arbitrary sequence of superlinear, linear and sublinear trends is plausible.

Both Fig. 7 and 8 display the growth in number of folders which overall follows the file growth trend but tends to be more discontinuous, with the big jumps possibly indicating architectural restructuring or other major changes, as when large portions of code are transferred from another application. There is tendency for large jumps (e.g. growth greater than 10 percent) in number of folders to precede a period of renewed growth at the file level and it appears that one could use, to certain extent, the folder size measurement to identify periods of restructuring, even though it does not always work.

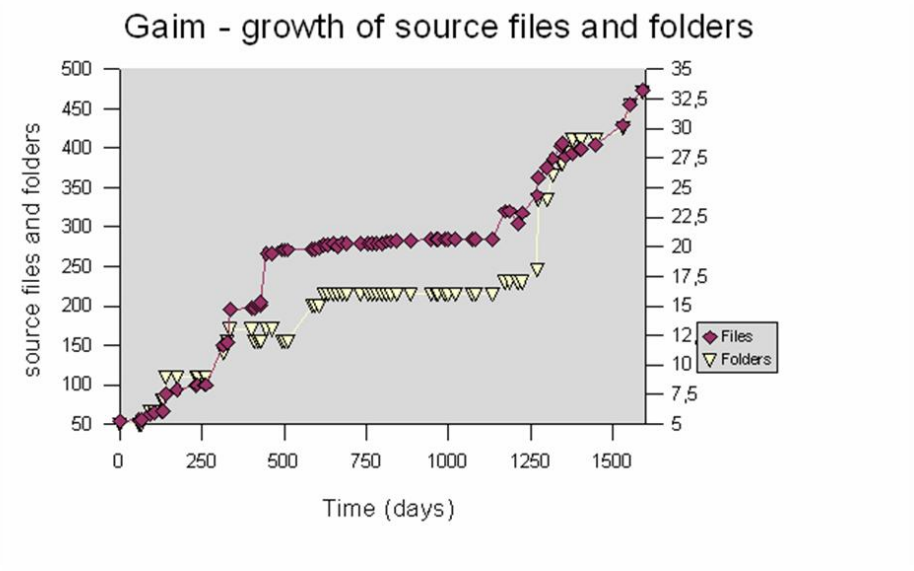

Fig. 7. Growth of the OSS Gaim system both in number of files and number of folders [**?**].

Refactoring Work in OSS

A contribution of these studies [22] was the provision of metric evidence that the so called anti-regressive work, which is related to what has been more recently called *refactoring* [23], actually takes place in some OSS projects. Refactoring involves re-writing of portions of the code which appear to be too complex, without changing the functionality and it is a popular and well accepted practice in agile methods. However, there is no "official" approach to refactoring in OSS projects, with some OSS projects more concerned with the understandability and complexity of the code than others. From a small sample of systems studied by Capiluppi *et al.*, it appears that in general OSS projects invest on average only a small portion of the effort in refactoring, even though some large peaks of refactoring activity can be present. In two OSS systems (Arla and Mozilla) for which anti-regressive work was measured, the portion of changes that can be considered as conducting complexity reduction work, were less than 25 percent of the total changes in a given release. This is illustrated in Fig. 9 that presents the approximate amount of anti-regressive work in Arla. The figure shows high variance in the work devoted to refactoring with high peaks but low average [22]. Note that the presence of a peak in anti-

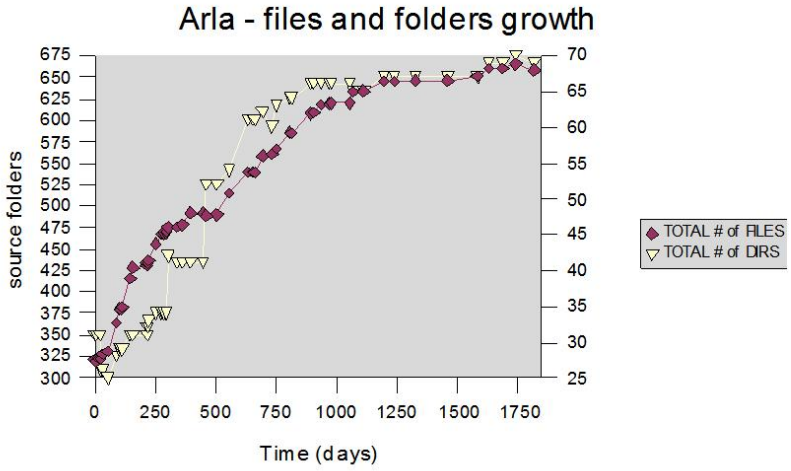

Fig. 8. Growth of the OSS Arla system both in number of files and number of folders [**?**].

regressive work does not imply that the activity for that month or period was predominantly refactoring. New functionality could have been added during the same interval.

2.7 The Study by Smith *et al.* [24, 25]

One important aspect, not considered by Lawrence [4] in his critic, is that the phenomena described by all the laws operate in the real-world in a parallel fashion. The important point to make here is that *testing each law in isolation and independently of the other laws and their assumptions can lead to erroneous results*. This is way, in our opinion, simulation models remain as the most promising way of empirically validating the laws. In this line of work, Smith *et al.* [25] examined 25 OSS systems by looking at the following attributes: functional size, number of files touched and average complexity. The research question was to test whether the growth patterns in OSS were similar to those predicted by three simulation models previously studied [26]. This was an indirect way of testing the empirical support for some of Lehman's laws, as these models were three different interpretations or refinements of some of Lehman's laws, in particular those related to system growth and complexity.

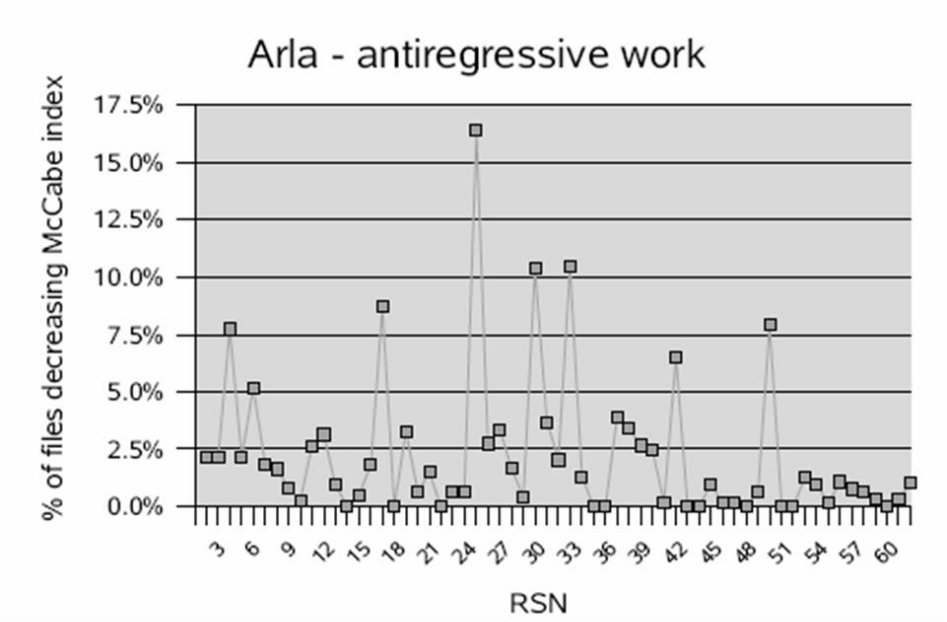

Fig. 9. Estimated amount of complexity reduction work as a percentage of all the files touched in a given release [22].

As previouly indicated in Section 1.1, simulation models are a good way to test the empirical validity of the laws as a whole. This is important because the laws interact with each other. Moreover, because the laws are informally stated in natural language, their formalisation is varied and leads to multiple simulation models.

This work used *qualitative abstraction*. The key idea is to abstract from the detail of the data and focus on a high level characteristic (e.g. overall pattern of growth). One possible way of applying qualitative abstration is finding out whether a trend is superlinear, linear, or sublinear by checking the value of the first and second differences in a time series. The symbols used are presented in Fig. 10.

Since growth trends in OSS systems display discontinuities, a characteristic already discussed in Section 2.6, the authors allowed for a sequence of multiple growth trends to be considered. Fig. 11 shows the results obtained for 25 systems. Two types of growth trends were considered for each system: size in files per release, called *un-scaled* trend, and a trend where the incremental growth in number of files was divided by the number of files touched during the interval, called *scaled* trend. The scaled trend was intended in order to

Sign of first difference dm/dt	Sign of 2nd difference ~d^2 m/dt^2	Symbol	Segment name
[0]	[0]	—	Constant
[+]	[0]	/	Linear growth
[—]	[0]	\	Linear decrease
[+]	[+]	⌡	Super-linear increase
[+]	[—]	⌠	Sub-linear increase
[—]	[—]	⌡	Super-linear decrease
[—]	[+]	⌠	Sub-linear decrease
[0]	[+]	∪	Minimum reached
[0]	[—]	∩	Maximum reached

Fig. 10. Symbols used to represent abstracted trends and the corresponding signs for the first and second differences of the variable [24].

remove the effect of the effort applied, hoping that any impact of the evolving complexity will be more evident. In fact, however, both scaled and un-scaled patterns were quite similar, as can be appreciated in Fig. 11.

The results in Fig. 11 show a variety of segment sequences (or patterns). These 25 OSS systems display greater variability in their segmented sequences of growth than the proprietary systems studied in [26]. In the OSS systems, increasing patterns predominated over non-growth or decreasing patterns. None of three qualitative simlation simulation models, built and run using a tool called QSIM, was able to predict the OSS observed trends, with the latter being richer and more complex than those predicted by the models. (The interested reader is referred to [24] for details on how this type of analysis was carried out.) This has led to the development of a *multi-agent model* to study how size, complexity and effort relate to each other in OSS [25]. In this model, a large number of contributors, represented in the model as agents, generate, extend, and re-factor code modules independently and in parallel. To our knowledge, this was the first simulation model of OSS evolution that included the complexity of software modules as a limiting factor in productivity (second law), the fitness of the software to its requirements (seventh law), and the motivation of developers (a new factor). Evaluation of the model was

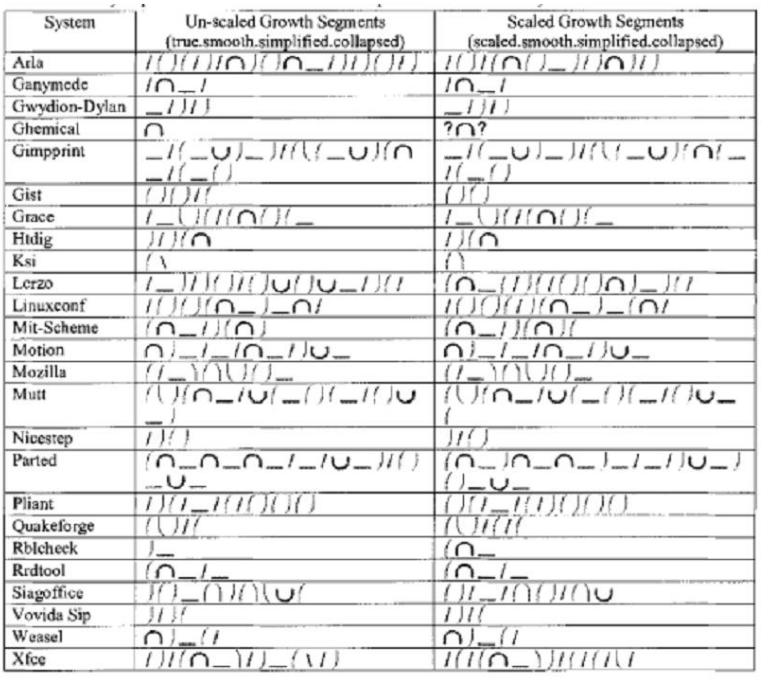

Fig. 11. Qualitative behaviours for system growth identified in empirical data from 25 OSS systems [24].

done by comparing the simulated results against four measures of software evolution (system size, proportion of highly complex modules, level of complexity control work, and distribution of changes) for four OSS systems (Arla, Gaim, MPlayer, Wine). The simulated results resembled the observed data, except for system size: three of the OSS systems showed alternating patterns of super-linear and sub-linear growth, while the simulations produced only superlinear growth. However, the fidelity of the model for the other measures suggests that developer motivation, and the limiting effect of complexity on productivity, are likely to have a significant effect on the development of OSS systems and should be considered in further simulation models of OSS development [25].

3 Comparing the Evolution of Open and Closed Source Software Systems

From the above discussion we can observe the following:

- The laws were proposed when most of the systems were developed in-house by a dedicated group of engineers working in the same place, under

some form of hierarchical management control and following a waterfall-like process. The software systems of the 70s and 80s were in many cases monolithic and there was little reuse from other systems. OSS challenges many of these assumptions[6].

- The laws are difficult to test empirically, because they are informal statements. One can formalise them making assumptions but many different formalisations are possible. Moreover, the phenomena described by the laws happens in parallel, with some of the laws related to the others. This calls for the use of techniques such as simulation models to test the laws. Qualitative simulation and multi-agent simulations are promising techniques.
- Growth patterns of OSS systems seem to be less regular than those of proprietary systems studied in the past[7] This could be due to the open system, in the system-theoretic sense, nature of OSS systems: contributors can come and go from wherever in the world, code can be easily duplicated or transferred from one application to the other. There are less restrictive rules than in traditional organisations. All these appear to contribute to a richer and more chaotic phenomenon.
- OSS evolutionary trends are in general more difficult to predict than those of traditional systems. Paradoxically, this does not imply more risk for those using OSS. Since they have access to the source code, they have a degree of control on the evolution of a system that users of proprietary systems do not have. OSS users can eventually implement their own features and fix defects, or even create and evolve their own stream if they need to.
- There is evidence for discontinuity in OSS evolution (see Section 2.6). Evolutionary stages are present in OSS but these have not been fully characterised. Models such as the one by Bennett and Rajlich [10] will need to be revisited to accommodate OSS observations.

Table 3 is an attempt to summarise the applicability of each of the laws to successful OSS projects, based on the empirical evidence so far collected. The laws do not apply to very many OSS projects which remain in the initial development or proposal stage. Some of the possible reasons for a project to become successful have been investigated in [27] and this is an important topic for the understanding of OSS evolution.

It is worth mentioning here that the laws refer to common properties across evolving E-type systems *at a very high level of abstraction*. For example, under the laws, the fact that two software systems display functional increase over

[6] Current proprietary systems are less monolithic and waterfall is progressivelly replaced by other process models. This is likely to affect the validity of the laws even for proprietary systems.

[7] Ideally one would like to compare contemporary proprietary and OSS system evolution and not contemporary OSS with out-of-date proprietary systems, but access to new proprietary systems is limited.

Number	Name	Empirical Support	Comment on applicability to Open Source Evolution
I	Continuing change	YES	Seems to apply well to those OSS projects which have achieved maturity. Many projects do not pass the initial development stage. However, even successful projects experience periods of none or little change.
II	Increasing complexity	?	Evidence is so far contradictory. There are some OSS systems that show increasing complexity and others of decreasing complexity. There is evidence of complexity control but it is not clear how this affects the overall complexity trend. Structural complexity has many dimensions and only a handful of them have been so far measured.
III	Self regulation	?	Not clear whether this law applies to OSS or not. For example, the influence of individuals like Linus Torvalds in the evolution of a system is very significant. On the other hand, there are forces from the entire multi-project eco-system which may affect the growth, change and other rates.
IV	Conservation of organisational stability	?	There are different degrees and types of control by small groups of lead developers and how their policies and loose organisation affect evolution is still not understood. Segmented growth suggests less stability than in proprietary systems. Mechanism that influence the joining in and departure of contributors need to be better understood.
V	Conservation of familiarity	?	Literally all, including users, can access the code and the documents available. Need to familiarise with a new release might be less relevant in OSS than in proprietary systems because many users are at the same time contributors and have a more in-depth knowledge of the application or participated in the implementation of the latest release.
VI	Continuing growth	YES	The law seems to describe well mature and successful OSS where despite irregularity in patterns there is a tendency to grow in functionality. Some successful OSS systems like Linux display super-linear growth. However, many OSS projects also display none or little growth.
VII	Declining quality	Possibly, but not tested yet	This law is difficult to test because it depends on the measurement of quality. At least it should consider in addition to defect rates, the number of requirements waiting to be implemented at a given moment in time. These variables are difficult to study in OSS since, in general, there are no formal requirements documents.
VIII	Feedback system	YES, but different type of feedback system	This law seems to apply well to OSS evolution. However the nature of the feedback in proprietary and open source systems may be different, leading to more variety, perhaps more chaotic behaviour, in OSS evolution.

Table 3. Applicability of the laws of software evolution to mature and successful OSS projects

time or over releases, means that they share one property: positive functional growth. Growth is a rather straight-forward and global characteristic that can be studied across a large number of systems. However, there is empirical research where investigators are looking to much more detailed characteristics (e.g. types of design problems in software systems), perhaps looking for statistical regularities in these, which might be more challenging to generalise across systems than the simple properties which are the concern of the laws. This also means that two systems may share some properties at a high level of abstraction but when one studies the details they might be highly different. One needs to keep the issues of the level of abstraction in mind when one is referring to common or different characteristics across software systems. The same applies when one is discussing whether software evolution is predictable or not. Some characteristics at a high level of abstraction may be predictable but as we get concerned of more detailed properties (e.g. the precise evolution in requirements that a software application will experience in two years time), characteristics are likely to be much more difficult to predict.

4 Threats to Validity

Empirical studies are frequently subject to some threats to validity and it is seen as a duty of authors to discuss these to the best of their knowledge [28]. The validity of the results of the empirical studies of OSS evolution, and in some cases also of proprietary software evolution, is constrained by a number of factors such as:

Incomplete or erroneous records: Chen *et al.* [29] found that in three different OSS systems studied, the omissions in change-logs ranged from 3 to almost 80 percent and conclude that change-logs are not a reliable data source for researchers. This is obviously a concern because some studies may use change-logs as a data source. Other data sources may be subject similarly to missing or mistaken entries. Quantification of the error (or uncertainty) due to missing, incomplete or erroneous records tends to be difficult and, unfortunately, not common. This is factor that requires increasing attention in order for empirical studies of software evolution to become more disciplined, scientific and relevant.

Biased samples: When projects selected for study were not randomly chosen there is a risk of having selected more projects of some type than others. For example, we know that only a small percentage of OSS projects achieve a mature and stable condition where there is a large number of contributions. The vast majority of OSS projects do not reach such stage [27]. Similarly many software projects are cancelled for one reason or another before initial delivery to users and hence never achieve evolution. Strictly speaking, one should be referring to many studies as empirical studies of *successful* software evolution.

Errors in data extraction: Data extraction from raw sources (e.g. code repositories and configuration management systems) can be complex and error prone. Assumptions may have made that are not clearly indicated. Data extraction and parsing and visualisation tools may contain errors.

Data extraction conventions: Whereas classic studies of proprietary systems use time series, where each measurement was taken for a given release, most of OSS studies follow a contemporary trend of using time series based on actual time of the measurement. Some authors like [15] have argued that this is more appropriate. However, it remains the question to what extent the release sequence is more or less informative than actual dates (rel-time) and how these different data can be compared.

External validity: In many studies it is not clear how the systems studied were selected and to what extent the systems analyzed are representative of typical OSS, or whether such a typical OSS actually exists. Some empirical evidence [27] suggests that the type of application influences the stability and success on an OSS project. Whether and how application domains relates to evolutionary patterns remains an open question for further research.

Granularity: There is evidence that evolutionary behaviour at the total system level and at the level of individual subsystems is different [11]. This may affect the internal validity of any results. Moreover, there is little knowledge on how the behaviour observed at the total system level relates to the behaviour observed at the subsystem level.

Initial development: Many OSS projects are started as closed-source projects before made available as OSS on the Internet. Little is known about what happens during this initial phase and how it influences the later evolution phases. Most of the empirical data do not capture this hidden initial development phase, which is possibly more similar to proprietary initial development than the later time when a system becomes OSS.

Confounding variables and co-evolution: There might exist other known or unknown variables that impact on the observed behaviours different to those considered in the studies. This could be due to measurement difficulties, because the researchers could not take additional variables into account for practical reasons or because these additional variables are unknown. One example of these is the amount of code that is duplicated, sometimes called *code cloning*, or ported from another system. This is an example of *network externality* [30]. Scacchi *et al.* [30] refer to OSS as a "software eco-system". In such eco-system one should not study individual systems, but one should look at the complex co-evolution of multiple software projects in order to make sense of the evolutionary trends.

The above list is not complete and other factors may also become threats to validity. Future studies will need to consider and handle these factors in detail. For the moment, we assume that the empirical results are the best description we have at hand of OSS evolution. The fact that some studies

have been replicated or point towards the same type of phenomena, however, enhances the validity of the current OSS empirical research, despite the many threats that we have mentioned in this section.

5 Conclusions and Further Work

In this chapter we have been concerned with evolution, that is, all that happens after the first release of a software, simply because it is an extremelly important part of the lifecycle of an application. OSS has made software evolution accessible for wider study. Empirical studies of open source software is a vast area and this chapter has discussed a small sample of studies that are concerned with the evolution of OSS, which is, as someone put it, what happens when one looks at the dynamic changes in software characteristics over time. By studying how OSS changes over time one might understand better the specific challenges of OSS evolution and how to address them in different ways, by inventing specific tools, for example. The many "static" empirical studies of OSS to date [30] have, therefore, not been of immediate interest in this chapter.

Empirical studies of OSS evolution, the focus of this chapter, tell us that the classical results from the studies of proprietary software evolution, which have laid a foundational stone in our collective understanding of software evolution, need to be reconciled with some of the evidence coming from OSS. From Table 3 it is clear that the OSS evolution phenomenon is not completely inconsistent with the laws, but it is opening up new questions which challenge the assumptions of the laws and it could well be that we are facing a paradigm-shift in our understanding of software evolution. Scacchi *et al.* [30] have put forward the view that OSS evolution should be viewed as an ecosystem. If this were so, we would need to get a better understanding of the personal attitudes, rules and "good practice" that make the OSS eco-system work successfully. Multi-agent simulation models (e.g. [25]) may be particularly useful here and perhaps the software evolution and biological evolution analogies, discussed in the 70s and 80s [1], may need to be revisited. We add a precautionary note here since fundamental differences are likely to remain between the two domains: software evolution is done by people using programming languages and technologies that themselves evolve, unconstrained by any physical laws, while biological evolution is constrained by the physical and chemical properties of molecules such as the DNA.

In Section 4 a number of important threats to the validity of the empirical studies of OSS evolution were indicated. A key issue is to find out which should be the first-class entity in the software evolution research. While classical studies of software evolution concentrated in a single software system as the first-class entity, in OSS (and in some proprietary environments too)

there is high code re-use and software evolves within interrelated multi-project environments. Because many OSS software systems can be strongly related through re-use and the importing and exporting of code, various systems co-evolve and influence the evolution of each other. This suggest that we should conduct future empirical studies of families of OSS systems.

For software evolution researchers who are seeking to contribute to OSS evolution, perhaps one of the areas in which they could contribute is, rather than trying to apply concepts that were conceived with proprietary processes in mind, seeking to identify the evolutionary characteristics of OSS and address them directly within the characteristics of OSS. For example, one interesting aspect is the issue of growth discontinuity and, in particular, the stages during which OSS systems appear to have stopped evolving, only to regain momentum some time later. What can be done in terms of tools, for example, to preserve the knowledge about the "mysteries" of the system during such periods?

Another interesting question is to what extent OSS evolution can be planned and controlled and the OSS evolutionary stages predicted. Many corporate and other customers (e.g. governments) would feel attracted to OSS if they had more confidence in the predictability of the evolution of these systems without having to create their own evolution stream of the system. More predictability is needed for OSS to become serious contenders of popular proprietary applications. However, are the OSS communities ready to become more business-like in setting targets and commitments?

Finally, even from a superflous analysis it is becoming evident that understanding of OSS evolution requires a multi-disciplinary approach that involves economics, social science and other disciplines in addition to computing. This all means plenty of challenges for developers and researchers and need to establish links to other research communities (e.g. information systems, economics, complexity science, psychology) with whom wider questions and interests could be shared. This is a pre-requisite to any major progress in the understanding of OSS evolution and its improvement.

References

1. Lehman, M.M., Belady, L.A.: Program Evolution: Processes of Software Change. Apic Studies In Data Processing. Academic Press (1985)
2. Fernandez-Ramil, J., Hall, P.: Maintaining and evolving software. M882 Course on Managing the Software Enterprise, Learning Space, The Open University, http://openlearn.open.ac.uk/course/view.php?id=1698 (2007) Work licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 2.0 Licence.
3. Kitchenham, B.A.: System evolution dynamics of VME/B. ICL Tech. J. (1982) 42 – 57

4. Lawrence, M.: An examination of evolution dynamics. In: Proc. Int'l Conf. Software Engineering (ICSE), IEEE Computer Society Press (1982) 188 – 196
5. Pfleeger, S.L.: Software Engineering: Theory and Practice. Prentice-Hall (1998)
6. Sommerville, I.: Software Engineering. 6th edn. Addison-Wesley (2001)
7. Lehman, M., Ramil, J.: An overview of some lessons learnt in FEAST. In: Proc. WESS. (2002)
8. FEAST: Feedback, evolution and software technology projects website. http://www.doc.ic.ac.uk/ mml/feast/ (2001)
9. Aoyama, M.: Metrics and analysis of software architecture evolution with discontinuity. In: Proc. Int'l Workshop on Principles of Software Evolution (IWPSE), Orlando, Florida (2002) 103 – 107
10. Bennett, K.H., Rajlich, V.T.: Software Maintenance and Evolution: A Roadmap. In: The Future of Software Engineering. ACM Press (2000) 75–87
11. Godfrey, M.W., Tu, Q.: Evolution in open source software: A case study. In: Proc. Int'l Conf. Software Maintenance (ICSM), Los Alamitos, California, IEEE Computer Society Press (2000) 131–142
12. Raymond, E.S.: The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary. Revised edn. O'Reilly & Associates, Inc., pub-ORA:adr (2001)
13. Pirzada, S.S.: A Statistical Examination of the Evolution of the Unix System. PhD thesis, Department of Computing, Imperial College, London (1988)
14. Gall, H., Jazayeri, M., René. Klösch, Trausmuth, G.: Software evolution observations based on product release history. In: Proc. Int'l Conf. Software Maintenance (ICSM), IEEE Computer Society Press (1997) 160–166
15. Paulson, J.W., Succi, G., Eberlein, A.: An empirical study of open-source and closed-source software products. IEEE Computer Society Trans. Software Engineering **30** (2004) 246–256
16. Stewart, K.J., Darcy, D.P., Daniel, S.L.: Opportunities and challenges applying functional data analysis to the study of open source software evolution. Statistical Science **21** (2006) 167–178
17. Herraiz, I., Robles, G., Gonzalez-Barahona, J.M., Capiluppi, A., Ramil, J.F.: Comparison between slocs and number of files as size metrics for software evolution analysis. In: Proc. European Conf. Software Maintenance and Reengineering (CSMR), Bari, Italy (2006)
18. Wu, J., Spitzer, C.W., Hassan, A.E., Holt, R.C.: Evolution spectrographs: Visualizing punctuated change in software evolution. In: Proc. Int'l Workshop on Principles of Software Evolution (IWPSE), Kyoto, Japan, IEEE Computer Society Press (2004) 57–66
19. Wu, J.: Open Source Software Evolution and Its Dynamics. PhD thesis, University of Waterloo, Ontario, Canada (2006)
20. Capiluppi, A., Morisio, M., Ramil, J.F.: The evolution of source folder structure in actively evolved open source systems. In: Proc. IEEE Symp. Software Metrics, IEEE Computer Society Press (2004) 2–13
21. Capiluppi, A., Morisio, M., Ramil, J.F.: Structural evolution of an open source system: A case study. In: Int'l Workshop on Program Comprehension. (2004) 172–182
22. Capiluppi, A., Ramil, J.F.: Studying the evolution of open source systems at different levels of granularity: Two case studies. In: Proc. Int'l Workshop on Principles of Software Evolution (IWPSE), IEEE Computer Society (2004) 113–118

23. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA (1999)
24. Smith, N., Capiluppi, A., Ramil, J.F.: A study of open source software evolution data using qualitative simulation. Software Process: Improvement and Practice **10** (2005) 287 – 300
25. Smith, N., Capiluppi, A., Ramil, J.F.: Agent-based simulation of open source evolution. Software Process: Improvement and Practice (**11**) 423 – 434
26. Ramil, J.F., Smith, N.: Qualitative simulation of models of software evolution. Software Process: Improvement and Practice **7** (2002) 95 –112
27. Comino, S., Manenti, F.M., Parisi, M.L.: From planning to mature: on the determinants of open source take off. Technical report, Trento University, Dept. of Economics (2005)
28. Kitchenham, B.A., Pfleeger, S.L., Hoaglin, D.C., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE Computer Society Trans. Software Engineering **28** (2002) 721 – 734
29. Chen, K., Schach, S.R., Yu, L., Offutt, A.J., Heller, G.Z.: Open-source change logs. Empirical Software Engineering **9** (2004) 197–210
30. Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S., Lakhani, K.: Understanding free/open source software development processes. Software Process: Improvement and Practice **11** (2006) 95 –105