Understanding Open Source Software Evolution: Applying, Breaking, and Rethinking the Laws of Software Evolution

Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA USA 92697-3425
wscacchi@uci.edu
April 2003

Introduction

This chapter examines whether or how the evolution of open source software conforms to the laws of software evolution that have been in development for more than 30 years. The laws of software evolution and their development as the basis for a theory of software evolution represents a major intellectual contribution and challenge to the software engineering research community, and to the broader community of computer science. The principal developer and advocate for the laws of software evolution is Professor M.M. (Manny) Lehman from Imperial College in London, and over the years his work has been expanded and refined by a growing list of students and scholars of software evolution. However, the emerging trend of free or open source software (F/OSS) development, often focusing attention to popular software systems like the GNU/Linux operating system, the Apache Web server, the Mozilla Web browser, and many others, raises the question as to whether F/OSS conforms to or breaks the laws of software evolution as currently formulated. Finding conformance would be reassuring to the current outstanding effort representing decades of study, whereas finding breakdowns, inconsistencies, or failures might point to refutations of the laws/theory, or at least the need to rethink, refine, and reformulate the laws/theory to account for the evolution of F/OSS.

This chapter is organized around five themes. First, it presents a brief review of models and theories of evolution from domains outside of software. This is to help set the stage for understanding some of the challenges and alternative historical groundings that might be used to shape our collective understanding for how to start to think about software evolution, as well as the significance of theorizing about it.

Second, it presents a brief review of the laws and theory of software evolution and their underpinnings in order to capture a reasonable picture for understanding what these laws and theory do and do not attempt to explain.

Third, it presents selected data and evidence that has begun to appear in the last few years that seeks to characterize different patterns of evolution and change are being discovered through empirical study of F/OSS systems, projects, and communities.

Fourth, given the materials presented, attention then shifts to analyzing where, how, and why the evolution of F/OSS does or does not conform to the laws and theory of software evolution. Without revealing too much at this point, it is fair to say that there is evidence and patterns of data from studies of F/OSS that do not appear to conform to the laws and theory of software evolution, as developed by Lehman and colleagues.

Given the potential for evidence and data that may not conform to the laws and theory, then it becomes necessary to consider how the laws and theory might be revamped or rethought to better account for the data that characterizes both conventional closed source software and F/OSS systems. This is addressed in the Fifth theme appearing in the last section.

As such, the remainder of this chapter progresses through each of these themes in the order presented here.

Evolution Models and Theories

One way how to better understand how conventional and open source software may evolve is to first review of models and theories of evolution from domains outside of software. In this regard, theories of biological evolution can be considered a starting point. The evolution of culture, language, economy, and technology can be considered complimentary points of reflection into how to understand software evolution. Clearly, review of such matters can only be very brief, but it can nonetheless help to introduce useful concepts for thinking about software evolution. Finally, the role of software process improvement, innovation and technology transfer may also be considered as factors that may contribute to software evolution.

Biological theories of the evolution have been the subject of scientific inquiry and speculation for centuries, with the work of Charles Darwin on the origins of species and natural selection being the most widely known and cited theory. Darwin's theoretical analysis was based in part on his field studies of animal species on the Galapagos archipelago. Darwin's theorizing begat not only more than a century of scientific debate, but also one of religious and moral substance [Bowler 1989]. From such theorizing about Evolution emerges concepts of *developmental biology* that examine the role of genetics, reproductive (natural) selection, co-evolution (among co-located species) and adaptation to ecological circumstances shaping the lives of organisms. In contrast, *evolutionary biology* accounts for the influence of genetics, reproduction, lineage, speciation, and population growth and diffusion shaping the long-term or trans-generational lives of species of organisms. Biological *systematics* further helps draw attention to the "progress", direction, or (punctuated) equilibrium of evolution based on associating the developmental properties (e.g., agency, efficiency, and scope) of living organisms with those found in the fossil and geological record [Nitecki 1988, Gould 2002]. Finally, recent studies have begun to employ computational mechanisms and simulation as an experimental strategy for studying evolutionary biology, artificial life, and complex adaptive systems (cf. the Artificial Life journal).

Culture, language and economy, which arise from the social actions and interactions of people, may evolve in ways similar or different from that in the natural science of biology. Culture, for example, may rely on the development, diffusion, and assimilation of *memes* (i.e., concepts, compelling ideas, or cultural "genes") that embody recurring social practices, situations, beliefs, or myths that can be shared, communicated, or otherwise transported as a basis for their evolution [Gabora 1997]. "Open source" and "free software" are examples of memes that are related but different. Thus, rather than conjecturing physical (biological) conditions or circumstances, cultural evolution relies on social actions and narrative strategies that may be situated with respect to physical conditions and circumstances that enable the ongoing evolution of diverse cultures and cultural experiences. Language evolution [Christiansen and Kirby 2003] seems to share and span ideas from culture and biology with respect to efforts that associate language learning, perception, and semiotics with neurological mechanisms and human population dynamics. Elsewhere, topics like *competition*, *resource scarcity*, *population concentration/density*, *legitimacy*, and *organizational ecologies* appear as situated factors shaping the evolution of markets, organizations and economies, at least at a macro level [Hannan and Carroll 1992, Nelson and Winter 1982, Saviotti and Mani 1995]. Beyond this, the evolution of culture, language and econony are being explored experimentally using computational approaches [e.g., Gabora 2000]. Overall, this tiny sample of work draws attention to associations more closely aligned to evolutionary biology, rather than to developmental biology.

The evolution of modern technology has also become the subject of systematic inquiry. For example, in Abernbathy's [1978] study of the American automobile industry, he finds that the *technical system* for developing and manufacturing automobiles associates product design and process design within a *productive unit* (i.e., the manufacturing systems within a physical factory or production organization). Each depends on the other, so that changes in one, such as the introduction of new techniques into a productive unit, are propagated into both product design and production process layout/workflow. Hughes [1987] in his historical study of the technical system of electrification draws attention to the role of the *infrastructure* of electrical production and distribution as spanning not just equipment, mechanisms (e.g., power generators, sub-stations), cabling and power outlets, but also the *alignment* of producers, retailers, and consumers of devices/products together with the processes that depend on electrification for their operation. Meyer and Utterback [1994] were among the first to recognize that productive units and technical systems of production and consumption were increasingly organized around *product lines* that accommodate a diversity of product life cycles centered around the *dominant design* [Utterback 1994] or product architecture that dominates current retail markets. From an economic perspective, Nelson and Winter [1982] independently termed the overall scheme that associates and aligns products, processes, productive units with producers, retailers and consumers, a *technological regime*. Last, though the development, use, and maintenance of software is strongly dependent on computer hardware, there are now studies that examine how different kinds of computer hardware components exhibit evolutionary patterns across technological generations or regimes [e.g., Victor and Ausubel 2002, van de Ende and Kemp 1999].

Software programs, systems, applications, processes, and productive units continue to develop over time. For example, there is a plethora of software innovations in the form of new tools, techniques, concepts or applications, which continue to emerge as more people experience modern computing technology and technical systems. These innovations give rise to unexpected or unanticipated forms of software development and maintenance, due to, for instance, software systems that are dynamically linked at run-time instead of compile-time [Mens *et al.*, 2003, Kniesel *et al.*, 2002]. Software innovations are diffused into a population of evermore diverse settings and technical systems via technology transfer and system migration. Software processes are subject to ongoing experience, learning, improvement and refinement, though there is debate about how to most effectively and efficiently realize and assimilate such process improvements [Conradi and Fuggetta 2002, Beecham, Hall and Rainer 2003]. Software systems are also subject to cultural forces [Elliott and Scacchi 2002], narrative strategies [Scacchi 2002a] and economic conditions [Scacchi 2002c] within the productive units or work settings that affect how these systems will be developed and maintained, such that these forces can show similar systems in similar settings evolving along different trajectories [Bendifallah and Scacchi 1987]. This suggests that software systems are developed and maintained within particular organizational and informational ecologies [cf. Nardi and O'Day 1999], as well as situated within a technical system of production and larger overall technological regime.

Overall, this brief review of evolutionary theory across a sample of disciplines raises an awareness of the following issues: First, in studying software "evolution" it is necessary to clarify whether in fact attention is directed at matters more closely aligned to development of a given system throughout its life, or with the evolution of software technologies across generations that are diffused across multiple populations. It appears that much of what are labeled as studies of "software evolution" are more typically studies of patterns of development of specific systems, rather than patterns of evolution across different systems within one or multiple product lines (or species), at least as compared to work in biological evolution. Resolving this possible confusion between the sciences of biology and software is not the purpose of this chapter, but merely an observation reported in it.

Second, when considering the subject of software evolution at a macro level, it appears that there are no easily found or widely cited studies of that examine issues of memes, competition, resource scarcity, population concentration/density, legitimacy, and organizational ecology as forces that shape or impinge on software systems or software technology. In general, existing theory of the development or evolution of software does not yet have a substantial cultural, language, or economic basis, nor a basis in (the legacy of) software systematics, so studies, analyses, and insights from these arenas are yet to appear.

Last, conventional closed source software systems developed within centralized corporate locales and open source software systems developed within globally decentralized settings without corporate locale represent two alternative technological regimes, each representing a different technical system of production, distribution/retailing, and

consumption, and with differentiated product lines and dominant product designs. Thus, the concepts from theories of technological evolution and observations on patterns of software development and maintenance can be used to help shape an understanding of how open source software evolves.

Software Evolution and the Laws of Software Evolution

In order to understand the current state of the art in the development of a theory of software evolution, and whether and how it applies to open source software evolution, it is necessary to identify and describe (a) what are the entities for software evolution, (b) what the laws of software evolution do and do not explain, (c) what data or evidence supports these laws, and (d) what other empirical studies of software evolution have reported. Each of these items is presented in this section.

Entities for Software Evolution

Relying on a contemporary description of the laws and theory of software evolution [Lehman 2002], five types of entities are identified as the appropriate subjects for examining and explaining software evolution. These five entities are:

Evolution over Releases: A sequence of stable versions or releases of an application program or software system implements changes in system quality, performance, function, etc., and makes them available to users. Only stable released product versions are addressed, not intermediate or unstable pre-release product versions. No alpha or beta releases are addressed [cf. Cusumano and Yoffie 1997]. The first five laws of software evolution discussed below focus attention exclusively to evolution of a software system over its releases.

System or program: A system or program evolves from first statement of an application concept or a change required to an existing system to the final, release, installed, and operational program text with its documentation. There is no identification of constraint on the size of systems or programs, thus small (e.g., <5K SLOC[1]), medium (5K-50K SLOC), large (50K-500K SLOC) or very large (>500K SLOC) systems may evolve according to the same laws. It is unclear whether the system or program may exist in related but distinct versions or releases intended for different computer platforms. It is also unclear whether distributed systems (e.g., multiple clients, single shared server), systems configured using scripts (e.g., Cshell) or middleware, or dynamically linked systems configured at run-time are accommodated by the laws of software evolution.

(E-Type) application: An application comprises both the source concept and its implementation as a computer supported activity in some operational domain. Such a software application might be said to be embedded in its setting of use. E-type applications can be distinguished from S-Type programs for which the sole criterion for successful implementation is that the program satisfies a prior specification. There is no information or data presented on the evolution of S-Type programs, nor are any examples

[1] Source lines of code (SLOC) is used as a general indicator of program size, and as a surrogate for program complexity, though their reliability for use in theoretical studies does have its risks [Scacchi 1991].

of such systems identified or cited. So it is unclear what information is added with the E-Type designation, though it may be retained for historical purposes.

Process: The interrelated activities that constitute the development and maintenance of a software system or program are called a process. It is however unclear whether this process is intended only to be viewed as either a monolithic process, just the top-level of a decomposable process, or whether specific software engineering activities (e.g., software architecture design, module vs. system testing) can have distinct processes which may also evolve, either independently or jointly.

Models of Process: A software process for developing and maintaining a software system or program can be modeled in either a prescriptive, proscriptive or descriptive manner, and different process models typically focus on one of these perspectives [Scacchi 2002b]. Prescriptive models indicate what should be done, proscriptive models indicate what might be done, and descriptive models indicate what was done (historically) in the development and maintenance of software systems. Prescriptive and proscriptive models may be used to aid in the planning or management of software development and maintenance processes, whereas descriptive models may be used to measure and improve these processes, based on historical experiences. Any of these types of process models may be stated informally in narrative form (perhaps augmented with reporting forms and documents) or formally as computer enactable representations.

In moving from these entities of software evolution in the laws and theory of software evolution, it can be observed that the laws primarily draw attention to just the first three entities, and do not present data, evidence, or results indicating what software engineering activity processes, or which software process models, facilitate or constrain software evolution. This is not to say that processes or process models don't matter; rather that the laws and theory do address them in an explicit manner.

Laws of Software Evolution

Initially, Lehman and colleagues formulated five laws of software evolution, though ongoing investigation has produced three more. These laws of software evolution can be summarized as follows in Figure 1, and explained in detail elsewhere [e.g., Lehman 2000, 2002].

It is highly recommended that the reader unfamiliar with these laws of software evolution should consult one or more of the cited sources to gain a more complete understanding of these laws, how they are described, and better appreciate some of the challenges that must be faced in investigating and (re)formulating these laws and emerging theory, based on empirical study. Such action is a necessary precursor for determining whether and how these laws may or may not apply to the evolution of F/OSS systems. Developing a theory of software evolution is a difficult scholarly undertaking, so the purpose of critiquing and assessing it is to find ways and means through which the laws and theory can be improved and refined, rather than simply criticized, rejected or ignored.

No.	Brief Name	Law
I 1974	Continuing Change	*E*-type systems must be continually adapted else they become progressively less satisfactory in use
II 1974	Increasing Complexity	As an *E*-type system is evolved its complexity increases unless work is done to maintain or reduce it
III 1974	Self Regulation	Global *E*-type system evolution processes are self-regulating
IV 1978	Conservation of Organisational Stability	Unless feedback mechanisms are appropriately adjusted, average effective global activity rate in an evolving *E*-type system tends to remain constant over product lifetime
V 1978	Conservation of Familiarity	In general, the incremental growth and long term growth rate of *E*-type systems tend to decline
VI 1991	Continuing Growth	The functional capability of *E*-type systems must be continually increased to maintain user satisfaction over the system lifetime
VII 1996	Declining Quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of *E*-type systems will appear to be declining
VIII 1996	Feedback System (Recognised 1971, formulated 1996)	*E*-type evolution processes are multi-level, multi-loop, multi-agent feedback systems

Figure 1. An Overview of the Laws of Software Evolution
[Source: Lehman, Ramil, *et al.,* 1997]

These laws of software evolution rely primarily on understanding how software system releases and application change over time. In these laws, attention is directed to certain phenomena such as continual adaptation, satisfaction in use (or user satisfaction), self-regulating capabilities, feedback and feedback loops, global activity rates, growth, operational environment, and agents, as well as how these things vary whether by increasing, decreasing, remaining constant, or by being organized in multiples. These phenomena not only point to a need to comprehend how they can be observed, monitored, or measured, but also to an underlying ontology[2] or meta-model [Mi and Scacchi 1996] that either implicitly or explicitly accounts for observed phenomena.

For the laws and theory of software evolution by Lehman and colleagues, their ontology seems to come from the domain of *feedback control systems* [e.g., Bateson 1993, Dyle, Francis and Tannenbaum 1992], rather than from biology, cultural studies, language, technological evolution, or socio-technical systems. Feedback control systems are often a subject that students of electrical and mechanical engineering as well as cybernetics learn, rather than those for computer science or software engineering. Students of feedback control learn about linear vs. non-linear feedback, properties of flow elements (liquids, gases, electrical, thermal, chemical, biological, etc.), feedback signal conditioning, feedback loop shaping/contouring, control actuators (servomechanisms, valves,

[2] An ontology is a description (like a formal specification of a program) of the concepts and relationships that can exist for an agent or a community of agents. The concepts are part of a domain vocabulary, relationships between concepts represent a logical system of reasoning and accomplishment, and together they allow descriptive expressions of how things work to be formed. Thus, every domain can have one or more ontologies to organize and explain how things work or change over time within the domain. However, it is unclear what happens when an ontology from one domain is used to characterize another domain—creative insights can emerged, much like nonsense or confusion can emerge.

attenuators, etc.), process control state-spaces, transfer functions, and more all within a closed (bounded) system framework [Bateson 1993, Dyle, Francis and Tannenbaum 1992]. None of these concepts have been articulated for software evolution, yet they are all elements or entities of feedback control systems. In contrast, feedback control systems, are generally an essential element in the design and control of robots or robotic systems, rather than of organizational or social systems. Thinking back to biology, individual organisms like insects may exhibit behavior in their articulated movements, consumption of food, and reproduction that may be modeled as a feedback control system. However, when insects coalesce into colonies or open habitat ecologies, then their emergent collective behavior will be much more difficult to model or regulate as a closed-loop feedback control system.

Data Supporting the Laws of Software Evolution

The eight laws of software evolution formulated and refined by Lehman and colleagues rely on an empirical foundation. The laws are not speculative. Instead, the laws seek to account for observed phenomena regarding the evolution of software releases, systems, and E-Type application, and all within a manner consistent with their ontology of feedback (control) systems. The laws and emerging theory [Lehman 2000, 2002, Lehman and Ramil 2001] are conceived to be empirically grounded and formulated in a manner suitable for independent test and validation, or refutation [Lakatos 1976, Popper 1961]. However, it is not clear how such empirical testing should be performed (e.g., how many or what kinds of software systems constitute an adequate or theoretically motivated sample space for comparative study), what the consequences for refutation may be (rejection or reformulation of the laws/theory), and whether or how the laws and theory might be refined and improved if new or contradictory phenomena appear [cf. Glaser and Strauss 1976, Yin 1994].

The studies by Lehman and colleagues provide data from evolution of releases primarily from up to five software systems: two operating systems (IBM OS 360, ICL VME Kernel), one financial system (Logica FW), one telecommunications system (Lucent), and one defense system (Matra BAE Dynamics). The data is summarized in Figure 2 (Matra system not shown). In all the graphs, the X-axis denotes the number of software releases, while the Y-axis denotes the percentage of growth of the size of the system (e.g., measured in SLOC) after the first release. These graphs suggest that during its evolution (or maintenance process), a system tracks a growth curve that can be approximated either as linear or inverse-square model [Turski 1996]. Thus, these data/curves explicate conformity to the first six laws, in that they suggest continual adaptation via incremental growth, system complexity is being controlled or self-regulated in a constant/bounded (linear or inverse-square) manner. The last two laws addressing quality and feedback systems cannot be directly observed within the data, but may conform to observations made by Lehman and colleagues about these systems. Therefore, this data set conforms to the data presented, and the diversity of data seems to confirm the laws.

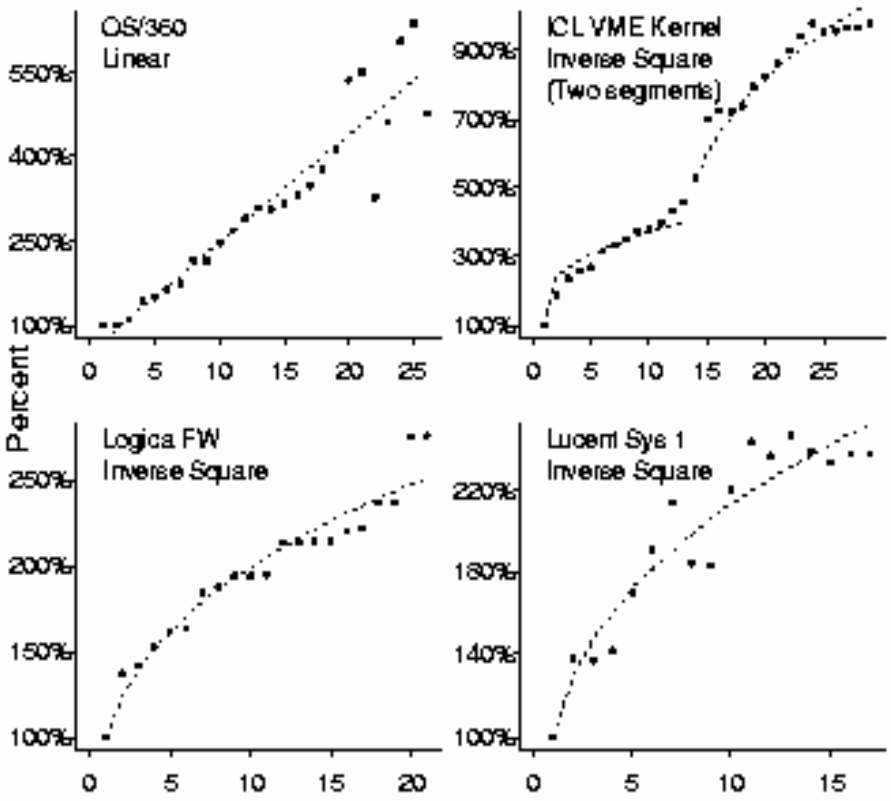

Figure 2. Data on the growth of different large software systems across releases supporting the Laws of Software Evolution [Sources: Lehman et al., 1980-2000].

As already noted, it is unclear whether such a data set is a representative sample of different kinds/types of software systems, or whether the laws can be interpreted as providing theoretical guidance for what kinds/types of software systems to study. In contrast, it may be apparent that these systems are all large or very large software systems, that they are developed and maintained in large corporate settings, that the customers for such systems are also likely to be large enterprises (i.e., they are not intended as software for a personal or hand-held computer), and that none is designed as a packaged software system for retail distribution [cf. Carmel and Becker 1995, Carmel and Sawyer 1998]. Furthermore, none of these systems has a timeline in months/years, and none is open for public inspection, nor is their data publicly available, so students and other scholars cannot readily access these systems or data for further study.[3]

Other Empirical Studies of Software Evolution

Beyond the studies by Lehman and colleagues of software evolution, many other empirical studies have been conducted and published. Here attention is directed to a sample of these studies where non-open source software systems were being investigated. This is mainly intended to see if other studies of software evolution conform to, refute, or

[3] Unfortunately, the *unavailability* of empirical data from software measurements studies is all too common of an occurrence, though studies of F/OSS may indicate a different future lies ahead regarding public data availability [cf. Koch and Schneider 2000, Robles-Martinez, Gonzalez-Barahona, *et al*., 2003].

otherwise extend and refine the laws and theory of software evolution. Furthermore, all of the systems involved in these studies appear to be E-Type applications.

Bendifallah and Scacchi [1987] present qualitative data and analysis of two comparative case studies revealing that similar kinds of software systems in similar kinds of organizational settings have different evolutionary trajectories. They report the differences can be explained by how system maintainers and end-users deal with local contingencies in their workplace and career opportunities in the course of maintaining their software systems. This notion of similar systems being maintained and evolved in similar setting but having different, non-parallel maintenance trajectories would not directly support the laws of software evolution, unless previously unidentified socio-technical variables are taken into account by the feedback control system that could account for overall evolutionary variability or circumstances.

Chong Hok Yuen [1987,1988] presents data and analysis of the evolution of large software systems in an attempt to confirm the first five laws of software evolution, given access to a new data set for different software system releases. He finds that his data and analysis reaffirm the first and second laws, but do not conform to the third, fourth, and fifth laws of software evolution. He attributes this result to human and organizational factors not addressed in the laws.

Tamai and Torimitsu [1992] present data and observations from a survey study of mainframe software system applications across product generations. Among other things, they find that software lifetime in their survey is on average about 10 years, the variance in application lifetime is 6.2, and that small software applications tend to have a shorter live on average. They also report that applications that constitute what they call "administration systems' live longer than "business supporting" systems, and that application systems that replace previous generation systems grow by more than a factor of 2. Last, they report that some companies follow policies that set the predicted lifetime of an application system at the time of initial release, and use this information in scheduling migration to next generation systems. These results are confounding in terms of the laws of software evolution in that they (a) introduce an upper-bound on system life based on product life policy rather than internal factors like feedback mechanisms, (b) introduce the concept that different types of applications will have predictably different life spans, and (c) that business product development schedules or marketing factors constrain the life and allocation of resources to maintain a system. While none of these observations is earth-shattering, they do not conform to the laws of software evolution, in that they point to exogenous factors, beyond the software system itself and not directly related to quality, user satisfaction, or growth, that intervene in the operationalization of the laws.

Cusumano and Yoffie [1999] present results from case studies at Microsoft and Netscape indicating strong reliance on incremental release of alpha and beta versions to customers as business strategy for improving evolution of system features that meet evolving user requirements. They show that user satisfaction can improve and be driven by the shortening the time interval between releases, as well as revealing that unstable releases

(e.g., alpha and beta versions) will be released to users as a way to enable them to participate in the decentralized testing and remote quality assurance, and thus affecting software evolution. Their study does not confirm or refute the laws of software evolution, but they introduce a new dynamic into software evolution by making the release activity an independent variable rather than a control variable.

Gall, Jayazeri, *et al*., [1997] provide data and observation based on software product release histories from study of a large telecommunications switching system. The growth of this system over twenty releases conforms to the general trends found in the data of Lehman and colleagues. However, they report that though global system evolution follows the trend and thus conforms to the laws, individual subsystems and modules do not. Instead, they sometimes exhibit significant upward or downward fluctuation in their size across almost all releases. Eick, Graves, *et al.*, [2001] also provide data demonstrating that source code decays unless effort and resources are allocated to prevent and maintain the system throughout the later stages of its deployment, and that the decay can be observed to rise and fall in different subsystems and modules across releases.

Kemerer and Slaughter [1999] provide a systematic set of data, analyses, and comparison with prior studies revealing that problems in software maintenance can be attributed to a lack of knowledge of the maintenance process, and of the cause and effect relationships between software maintenance practices and outcomes. Their study does not specifically confirm or refute the laws of software evolution, nor do they provide data easily associated with the laws. However, they do observe that their data may be associated with the growth of system entropy and other outcomes over time, which they attribute to the laws observed by Lehman and colleagues.

Perry, Siy, and Votta [2001] report findings from an observational case study of the development of large telecommunications systems that indicates extensive parallel changes being made between software system releases. This notion of parallel changes that may interact and thus confound software maintenance activities is not accounted for in an explicit way by the laws of software evolution. Thus, it does not directly conform to or refute these laws, though it does introduce yet another organizational factor that may affect software evolution.

Overall, these studies either conform to, refute in part, or suggest extensions to the laws and theory of software evolution. The extensions generally appear to be outside of the scope of the feedback control systems ontology that underlies these laws. Thus these conditions may point to the need for either new/revised laws, or alternative theories of software evolution that may or may not depend on such laws, or on feedback control systems.

Evolutionary Patterns in Open Source Software

OSS development has appeared and diffused throughout the world of software technology, mostly in the last ten years. This coincides with the spread, adoption, and routine use of the Internet and World Wide Web as a global technical system. Their infrastructure supports widespread access to previously remote information and software

assets, as well as the ability for decentralized communities of like-minded people to find and communicate with one another. This is a world that differs in many ways from that traditional to software engineering, where it is common to assume centralized software development locales, development work, and administrative authority that controls and manages the resources and schedules for software development and maintenance. Thus to better understand whether or how patterns of software evolution in the technical regime of F/OSS conform to or differ from the laws of software evolution, then it is appropriate to start with an identification of the types of entities for F/OSS evolution, then follow with an examination of empirical studies, data and analyses of F/OSS evolution patterns.

F/OSS Releases -- Large F/OSS system releases continue to grow over time. This suggests some consistency with the laws of software evolution. Both stable and unstable F/OSS release product versions are being globally distributed in practice. Periodic alpha, beta, candidate, and stable releases are made available to users at their discretion, as are unstable nightly F/OSS build versions released for those so inclined (generally contributing developers). F/OSS releases for multiple platforms are generally synchronized and distributed at the same time, though may vary when new platforms are added (in parallel). F/OSS releases thus evolve within a non-traditional process cycle between full stable releases. F/OSS releases are also named with hierarchical release numbering schemes, sometimes with three or four levels of numbering to connote stable versus unstable releases for different audiences. However, the vast majority of F/OSS systems, primarily those for small and medium size F/OSS systems, do not continue to grow or thrive.

F/OSS Systems – A F/OSS system or program in its evolution from first statement of an application concept or a change required to an existing system to the released, installed, and operational program text with its documentation. F/OSS systems may be small, medium, large or very large systems, with large and very large systems being the fewest in number, but the most widely known. Most large or very large F/OSS systems or programs may exist in related but distinct versions/releases intended for different application platforms (e.g., MS Windows, Solaris, GNU/Linux, Mac OS X). Many F/OSS are structured as distributed systems, systems configured using scripts (e.g., using Perl, Python, Tcl) or middleware, modules that plug-in to hosts/servers (e.g., Apache and Mozilla both support independently developed plug-in modules) or dynamically linked systems configured at run-time, when the F/OSS in developed in a programming language like Java or others enabling remote service/method invocation.

F/OSS E-Type Applications – A much greater diversity and population of F/OSS applications are being investigated for the software evolution patterns. Those examined in-depth so far include the Linux Kernel, Debian Linux distribution[4], Mono, Apache Web server, Mozilla Web browser, Berkeley DB, GNOME user interface desktop, PostgreSQL

[4] A GNU/Linux distribution includes not only the Kernel, but also hundreds/thousands of utilities and end-user applications. Distributions are typically the unit of installation when one acquires GNU/Linux, while the Linux Kernel is considered the core of the distribution. Many F/OSS applications are developed for non-Linux Kernel operating systems (e.g., MS Windows), thus assuming little/no coupling to the Linux Kernel.

DBMS, and about a dozen others. Studies of F/OSS application populations, taxonomy, and population demographics for hundreds to upwards of 40K F/OSS systems have appeared.

F/OSS Process – F/OSS are developed, deployed, and maintained according to some software process. It is however unclear whether F/OSS processes, as portrayed in popular literature, are intended only to be viewed as a monolithic process, just the top-level of a decomposable process, or whether specific software engineering activities have distinct processes which may also evolve, either independently or jointly. As noted above, F/OSS activities surrounding software releases may have their own distinct process [Jensen and Scacchi 2003] that may not reflect the activities involved in the release of closed-source systems examined in the preceding section.

Models of F/OSS Process – Existing models of software development processes [Scacchi 2002b] do not explicitly account for F/OSS development activities or work practices [cf. Scacchi 2002a, 2002c, Jensen and Scacchi 2003]. Thus it is unclear whether models of software evolution processes that characterize closed-source software systems developed within a centralized administrative authority can account the decentralized, community-oriented evolution of F/OSS.

Overall, evolving software systems may be packaged and released in either open source or closed source forms. The packaging and release processes and technical system infrastructure may at times differ or be the same, depending on the software system application and development host (e.g., a Web site for open source, a corporate portal for closed source). But the decentralized community-oriented technological regime and infrastructure of F/OSS appears different than the world of the centralized corporate-centered regime and infrastructure of the closed source systems that have been examined as the basis of the laws of software evolution.

Patterns in Open Source Software Evolution Studies

In contrast to the studies by Lehman and colleagues, as well as others examining closed-source software evolution, attention is now directed to a new sample of studies where F/OSS systems are being investigated.

Godfrey and Tu [2000] provide data on the size and growth of the Linux Kernel (2M+ SLOC) from 1994-1999, and find the growth rate to be super-linear (i.e., greater than linear), as portrayed in Figures 3 through 5. They also find similar patterns in F/OSS for the Vim text editor. Schach, Jin, *et al*., [2002] report on the result of an in-depth study of the evolution of the Linux Kernel across 96 releases [cf. Godfrey and Tu 2000] indicating that the common coupling across modules has been growing at an exponential (super-linear) rate. Their data are displayed in Figure 6. They predict that unless effort to alter this situation is mobilized, the Linux Kernel will become unmaintainable over time. Koch and Schneider [2000] report in their study of the GNOME user interface desktop (2M+ SLOC) provide data that shows growth in the size of the source code base across releases increases in a super-linear manner as the number of software developers contributing code to the GNOME code base grows. Data from their study appears in Figure 7.

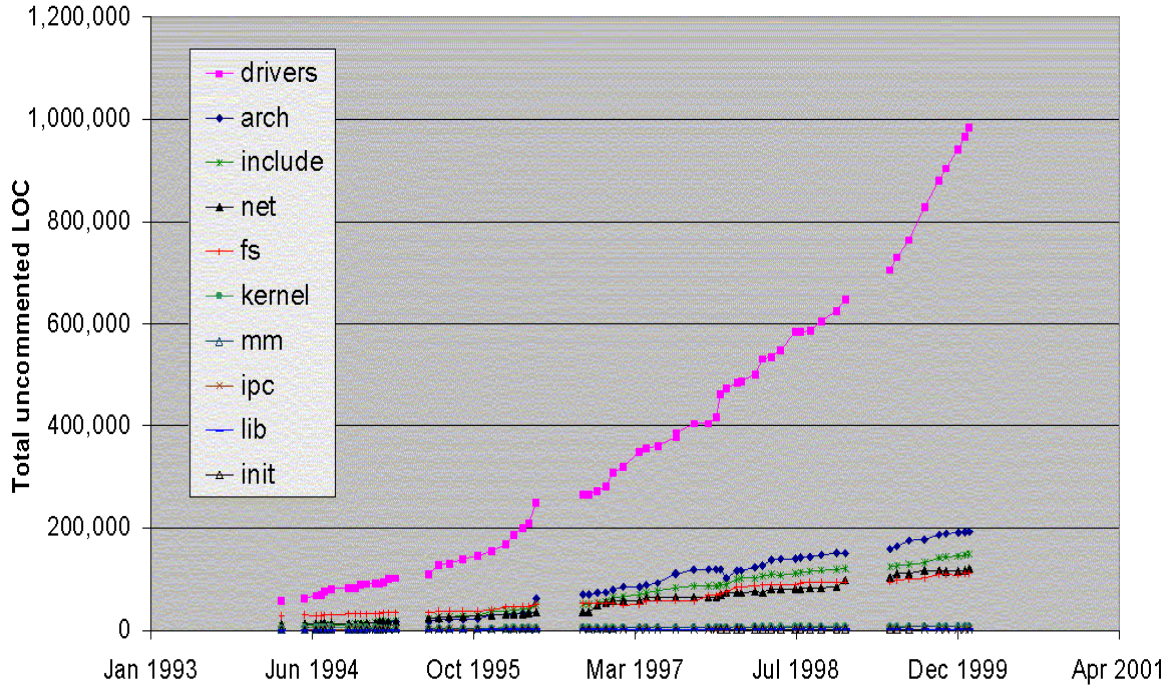

Figure 3. Data revealing the size and growth of major sub-systems in the Linux Kernel during 1994-1999 [Source: Godfrey and Tu 2000].

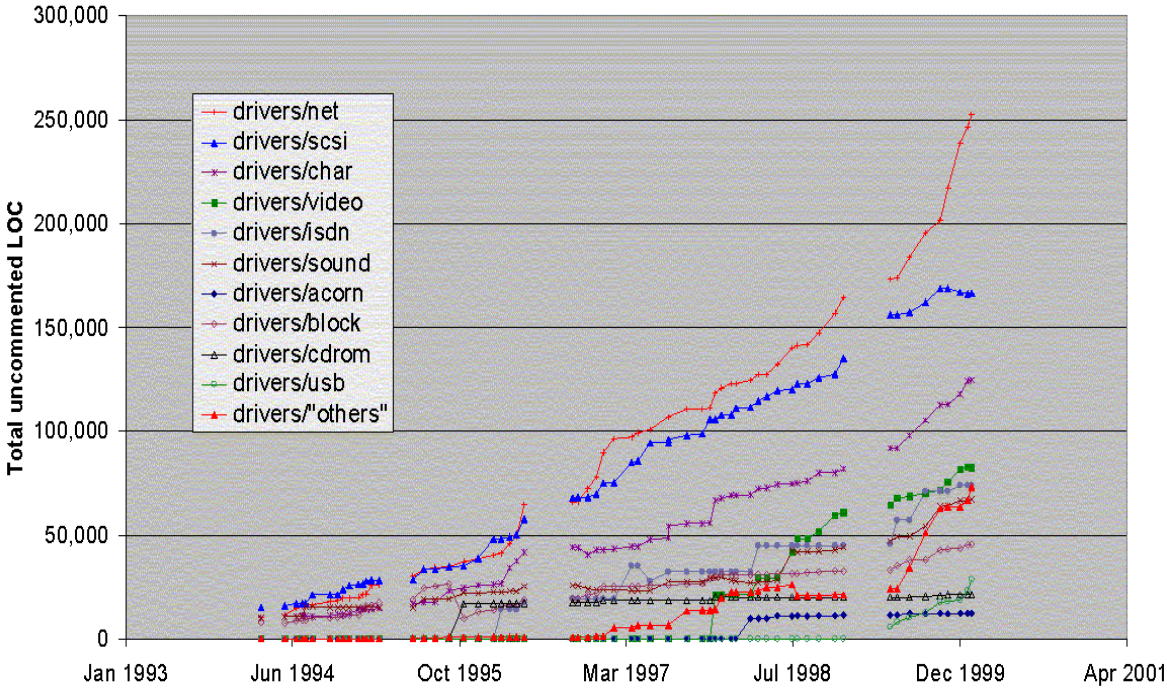

Figure 4. Data revealing the size and growth of device drivers in the Linux Kernel during 1994-1999 [Source: Godfrey and Tu 2000].

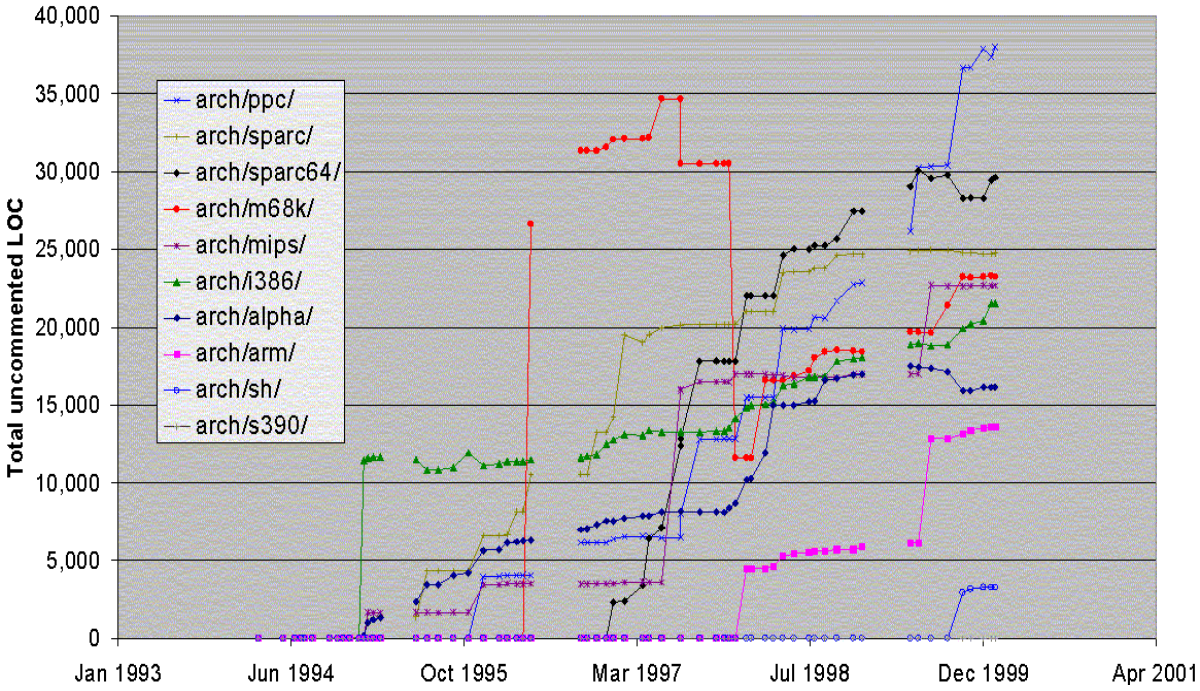

Figure 5. Data revealing the size and growth of the Linux Kernel for different computer platform architectures during 1994-1999 [Source: Godfrey and Tu 2000].

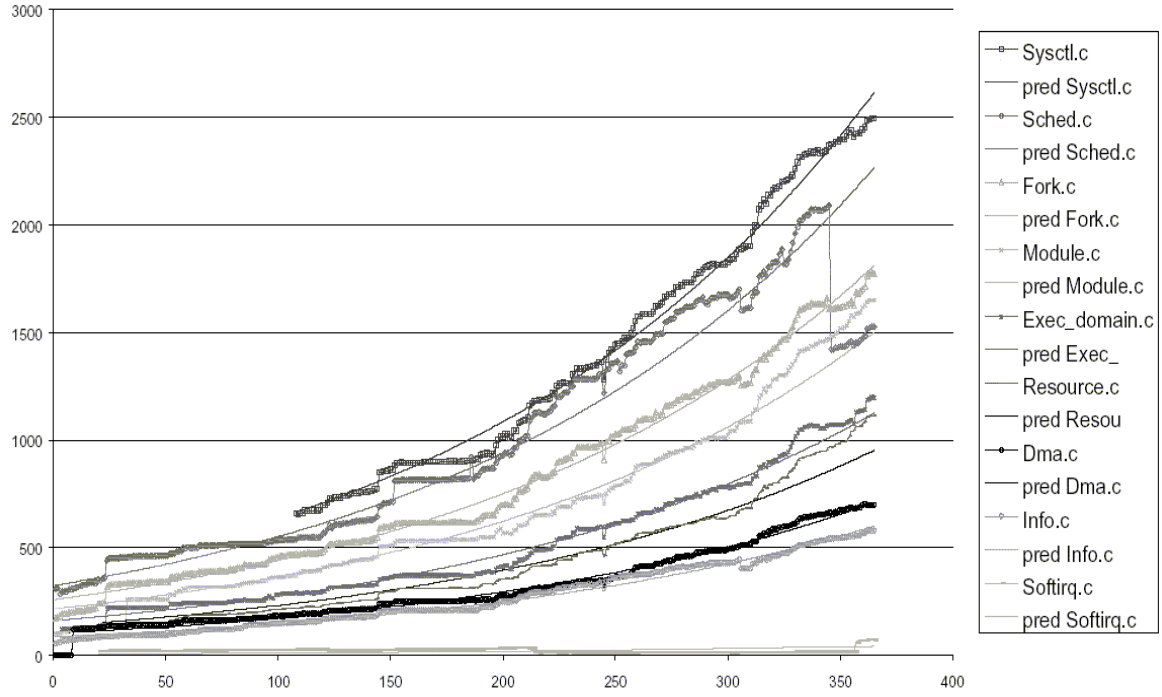

Figure 6. Measured (discrete points) versus predicted (smooth curves) of common coupling of source code modules in the Linux Kernel across releases [Source: Schach, Jin, *et al*., 2002].

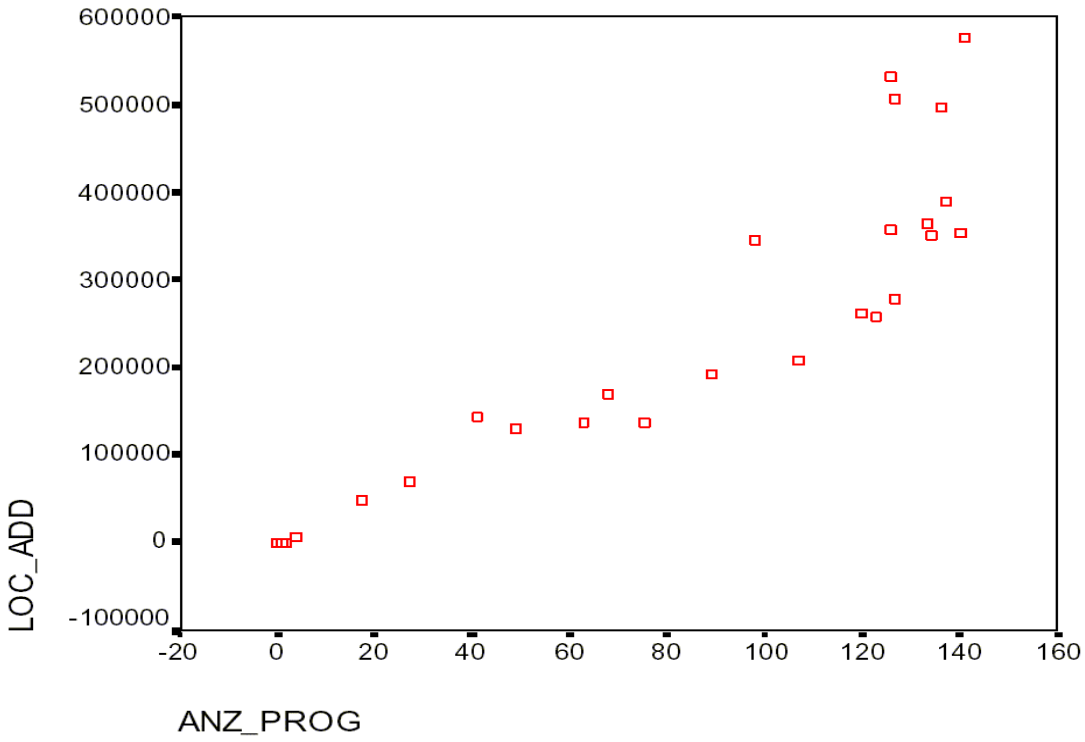

Figure 7. Data revealing the growth of the lines of source code added as the number of software developers contributing code to the GNOME user interface grows [Source: Koch and Schneider 2000].

Robles-Martinez, Gonzalez-Barahona, *et al.,* [2003] report in their study of Mono (a F/OSS implementation of Microsoft's .NET services, libraries, and interfaces), their measurements indicate super-linear growth rate in the code size and the number of code updates that are committed within the code base. They also report a similar though lesser growth pattern in the number of people contributing source code to the emerging Mono system over a 2-3 year period. According to Gonzalez-Barahona, Ortuno Perez, *et al*, [2001] their measurements indicate that as of mid-2001, the Debian GNU/Linux 2.2 distribution had grown to more than 55M SLOC. Last, O'Mahony [2003] presents data from her study of the Debian Gnu/Linux distribution from releases spanning 0.01 in 1993 through 3.0 in late 2002 that show growth of the size of the distribution rises at a super-linear rate over the past five years, while the number of its contributors grows incrementally.

In contrast, Godfrey and Tu [2000] find linear growth in Fetchmail, X-Windows, and Gcc (the GNU compiler collection), and sub-linear growth in Pine (email client). Such trends are clearly different from the previous set of F/OSS systems.

Why is there such a high growth rate for some F/OSS systems like the Linux Kernel, Vim, GNOME, Mono and the Debian GNU/Linux distribution, but not for other F/OSS? Godfrey and Tu [2000] report in the case of the Linux Kernel that (a) much of the source code is device drivers, as seen in Figure 3, (b) much of the code is orthogonal and

intended for different platforms, as suggested in Figure 5, (c) contributions to the code base are open to anyone who makes the requisite effort, and (d) typical Linux Kernel configurations, including those found in the Debian GNU/Linux distribution, only use as little of 15% of the total source code base. It is possible but uncertain whether these conditions also apply to GNOME, Vim, and Mono. However, it is unclear why they would or would not apply to Fetchmail, X-Windows, Gcc and Pine. Perhaps it might be because these latter systems are generally older and may have originally been developed in an earlier (pre-Web) technological regime. Elsewhere, Cook, Ji, and Harrison [2000] in their comparison study of the closed-source Logica FW system examined by Lehman and colleagues, and the F/OSS Berkeley DB system, find that growth across releases is not uniformly distributed, but concentrated in different system modules across releases.

Nakakoji, Yamamoto, *et al*., [2002] report findings from a comparative case study of four F/OSS systems, the Linux Kernel, Postgres DBMS, GNUWingnut, and Jun a 3D graphics library. They provide data indicating that these systems exhibit different evolutionary patterns of splitting and merging their overall system architectures across releases, as shown in Figure 8. Thus it appears that it is necessary to understand both the *age* and *architectural patterns* of sub-systems and modules within and across software releases, whether in closed source or open source systems, in order to better understand how a system is evolving [Godfrey and Lee 2000]. This observation is also implicated by earlier studies [Tamai and Torimitsu 1992, Gall, Jayazeri, *et al*. 1997, Eick, Graves, *et al.* 2001, Perry, Siy, and Votta 2001].

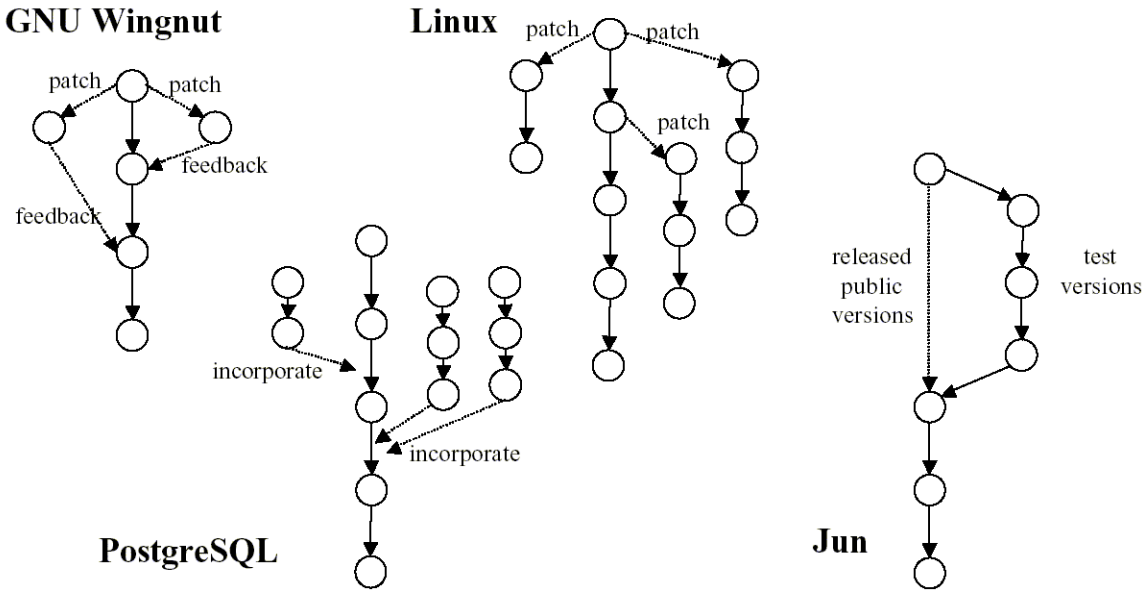

Figure 8. Patterns of software system release evolution for four different F/OSS systems [Source: Nakakoji, Yamamoto, *et al*., 2002]

Hunt and Johnson [2002] report discovery of a Pareto distribution in the size of the number of developers participating in F/OSS projects, from a sample population of >30K

projects found on the SourceForge Web portal.[5] Their results indicate that vast majority of F/OSS projects have only one developer, while a small percentage have larger, ongoing team membership. Madey, Freeh, and Tynan [2002] in an independent study similar to Hunt and Johnson, find a power law distribution characterizes the size of F/OSSD projects across a population of some 40K F/OSS projects at SourceForge. Independently, Hars and Ou [2002] report a similar trend, finding that more than 60% of F/OSS developers in their survey sample reported participating in 2-10 other F/OSS development projects. Capilupppi, Lago, and Morisio [2003] also draw from a sample of 400 F/OSSD projects posted on SourceForge. They find that the vast majority of systems in their sample are either small or medium size systems, and only a minor fraction are large. Only the large F/OSS systems tend to have development teams with more than a single developer. Their results might also be compared to those of Tamai and Torimitsu [1992], thereby substantiating that small F/OSS systems have a much shorter life, compared to large F/OSS systems. Overall, this suggests that results from studies that characterize large F/OSS efforts are not representative of the majority of F/OSS projects.

Di Penta, Neteler, *et al.,* [2002] provide results from a case study focused on the refactoring of a large F/OSS application, a geographical information system called GRASS, to operate on a small hand-held computer. Their effort was aimed at software miniaturization, reducing code duplications, eliminating unused files, and restructuring system libraries and reorganizing them into shared (i.e., dynamically linked) libraries. This form of software evolution and architectural refactoring has not been reported in, or accounted for by, the laws of software evolution. For example, miniaturization and refactoring will reduce the size of the software application, as well as potentially reducing redundancies and code decay, thereby improving software quality. Elsewhere, Scacchi [2002c] reports results from a case study of the GNUenterprise project that find that the emerging F/OSS E-Commerce application system being developed is growing through merger with other independently developed F/OSS systems, none of which was designed or envisioned as a target for merger or component sub-system. He labels this discontinuous growth of F/OSS system size and functionality, *architectural bricolage.* Such capabilities do not agree with data trends or the laws of software evolution, but may account for the discontinuities that can be seen in the growth trends displayed in Figure 5.

Mockus, Fielding, Herbsleb [2002] in a comparative case study of Apache Web server (<100K SLOC) and Mozilla Web browser (2M+ SLOC), find that it appears easier to maintain the quality of system features for a F/OSS across releases compared to closed-source commercial telecommunications systems of similar proportions. They also find evidence suggesting large F/OSS development projects must attain a core developer team size of 10-15 developers. This might thus be recognized as an indicator for *a critical mass in the number of developers for a given system type* that once achieved enables a high rate of growth and sustained viability.

[5] The SourceForge Web portal can be found at www.sourceforge.net. As of this writing, there are 60K F/OSS projects now registered at this specific F/OSS project portal. Other F/OSS Web portals like www.freshmeat.org and www.savannah.org have other projects, though there is some overlap across these three portals.

Scacchi and colleagues [2002a, 2002c, Elliott and Scacchi 2002, Jensen and Scacchi 2003] provide results from comparative case studies of F/OSSD projects within different communities. They find and explicitly model how F/OSS requirements and release processes differ from those expected in conventional software engineering practices. They also find that evolving F/OSS depends on co-evolution of developer community, community support software, and software informalisms as documentation and communication media. Nakakoji, Yamamoto, *et al*., [2002] also report that the four F/OSS systems they investigated co-evolve with the communities of developers who maintain them.

Von Hippel and Katz [2002] report results of studies that reveal some end-users in F/OSS projects become developers, and most F/OSS developers are end-users of the systems they develop, thereby enabling the co-evolution of the system and user-developer community. This observation of developers as users as developers is also independently reported in other studies as well [Mockus, Fielding, Herbsleb 2002, Scacchi 2002a, and Nakakoji, Yamamoto, *et al*., 2002]. Last, much like Hars and Ou [2002], Madey, Freeh, and Tynan [2002] report finding that some F/OSS developers participate in multiple projects, thereby creating social networks that interlink F/OSSD projects, and enable the systems interlinked in these social networks to also share source code or sub-systems. A sample from their data appears in Figure 9.

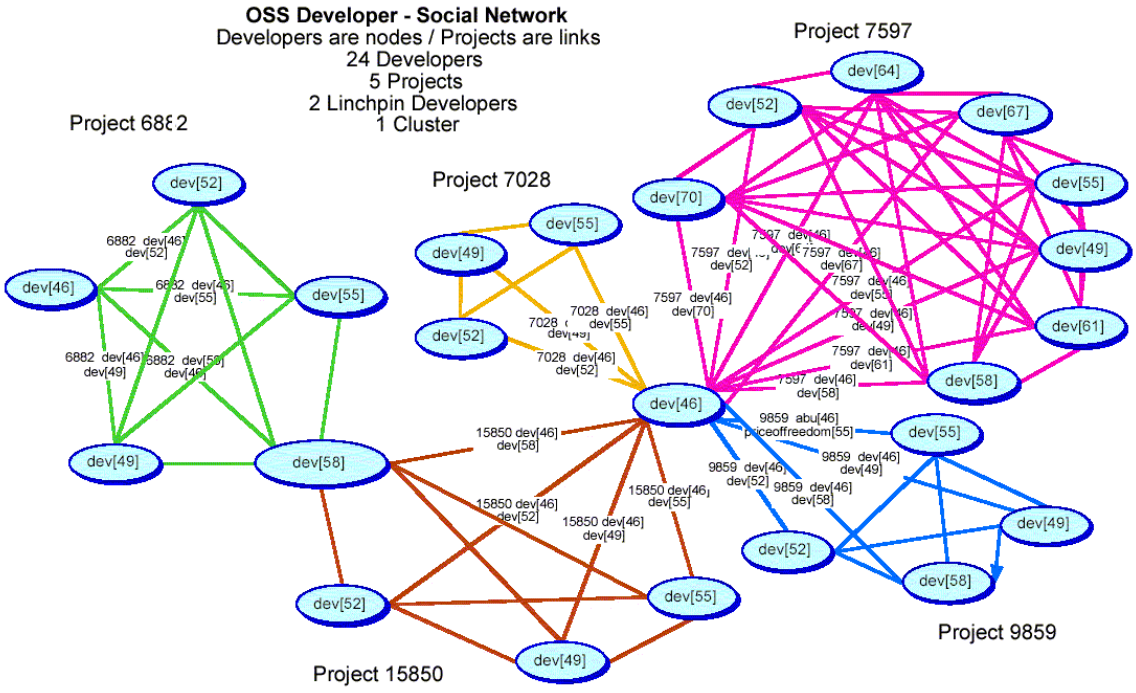

Figure 9. A social network of F/OSS developers that interlinks five different projects through two developers [Source: Madey, Freeh, and Tynan 2002].

However, across the set of studies starting above with Mockus, Fielding and Herbsleb [2002], there are no equivalent observations or laws reported in prior studies of closed

source software evolution that account for these data and evolutionary patterns addressing team and community structures. Clearly, such a result does not imply that the observed conditions or patterns do not occur in the evolution of closed source software. Instead, it reveals that other variables and patterns previously not addressed in prior empirical studies may be significant factors contributing to software evolution.

Overall, in evolving large F/OSS, it seems that it may be necessary for a critical mass of developers to come together to anchor the broader community of users, developers, and user-developers to their shared system. This critical mass and community will co-evolve with the architectural patterns that are manifest across unstable and stable F/OSS system releases as they age over time. Subsequently, older F/OSS systems that may have emerged before the F/OSS gained widespread recognition as a social movement and cultural meme, may have a lower rate of architectural and system release co-evolution. Furthermore, it may well be the situation that for large F/OSS systems/releases to evolve at a super-linear rate, that this may be possible only when their development community has critical mass, is open to ongoing growth, and that the focal F/OSS systems entail internal architectures with orthogonal features, sub-systems, or modules, as well as external system release architectures that span multiple deployment platforms.

Last, it appears that the evolutionary patterns of F/OSS systems reveal that overall system size and architecture can increase or decrease in a dis-continuous manner, due to bricolage-style system mergers, or to miniaturization and refactoring. Clearly, the laws of software evolution based primarily on the study of large closed source systems do not account for, nor anticipate, the potential for super-linear growth of software system size that can be sustained in the presence of satisfied developers-users-developers communities who collectively assure the quality of these systems over time.

Analyzing, Breaking and Rethinking the Laws of Software Evolution

There are now two kinds of empirical studies, data, and observations reported for closed source and open source software evolution. These studies present data and analyses that are either quantitative or qualitative. Examples of each have been presented, and they are available for further examination, assessment and comparison. So, it is now possible to reexamine and analyze the laws and theory of software evolution to see how well they can account for the data now at hand. Furthermore, if the laws and theory appear to be breaking down through non-conforming data and findings, then it seems reasonable to address how the laws and theory might be revised or rethought to account for the new data and findings.

The Laws of Software Evolution Revisited for F/OSS

Each of the eight laws of software evolution can be reexamined in turn.

The first law, *continuing change,* states an E-type program must be continually adapted, else it becomes progressively less satisfactory. There is data from the growth of both closed source and open source software that conforms to this law. However, it is unclear

whether or how "user satisfaction" is to be measured from data on the growth in the size of software system releases, whether closed or open source. Satisfaction is instead perhaps merely a conjectured inference or relationship that is assumed as long as the software system being released and therefore maintained. There is no direct accounting in the studies that reveals whether fluctuations apparent in the growth curves are in response to higher, lower, or dynamically shifting levels of satisfaction. Furthermore, it is unclear whether or how "continual change" accommodates the discontinuous changes due to system mergers, miniaturization, or refactoring, rather than just incremental changes in growth found in many systems. Finally, there is no accounting for the possibility of competing alternative offerings of a given type of system for a particular computing platform. Thus, there is no causality that accounts for how software systems change according to this law, and therefore little/no predictive power to assess whether or how long a system will grow and thrive in its settings of use or evolution.

The second law, *increasing complexity,* stipulates that as a program is evolved, its complexity increases unless work is done to maintain or reduce it. There is data from the growth of both closed source and open source software that conforms to this law. However, it is unclear whether or how "complexity" is to be measured from data on the growth in the size of software system releases, whether closed or open source. Is increasing complexity merely a conjectured inference or relationship that is assumed as long as the system is being released and therefore maintained? Does code decay represent an increase or decrease in complexity? In the case of the Linux Kernel, module coupling, as an indicator of software system complexity, appears to be increasing at a super-linear rate for more than five years. It is unclear whether in such a situation, that though complexity is increasing, that work is being done to maintain or reduce it, as might be inferred if module coupling grew only at a linear or inverse-square rate. Clearly, further study that operationalizes some measure of system complexity in other software systems is needed to help determine whether the Linux Kernel is anomalous on this dimension, or whether other systems exhibit a similar evolutionary pattern. Once again, there is no causality that accounts for how software systems complexity is manifest in this law, and therefore little/no predictive power to assess whether or how long a system will grow and thrive in its settings of use or evolution.

The third law, *self regulation,* states the program evolution process is self-regulating with close to normal distribution of measures of product and process attributes. But why or to what ends must the software evolution process be self-regulating? How does self-regulation work, on what input events or signals does it operate, and what transformation function takes these inputs to moderate what outcome variables or flows to realize self regulation? Self-regulation suggests a notion of internalized control and stabilization, which seems consistent with the ontology of feedback control systems, rather than an inherent property of software products or processes. Within the regime of F/OSS, the population trend across tens of thousands of F/OSS system projects reveals a distribution of self-organization, where the size of the core development team/community conforms to a Pareto or Power Law distribution, but not necessarily the size of the software release product (e.g., compare Linux Kernel vs. Apache Web server). Qualitative results suggest that F/OSS processes may also be self-organizing, since very few F/OSS projects have

any form of explicitly published process guidelines, yet they enact many software processes in a recurring and patterned manner that can be so modeled [Scacchi 2002a, Jensen and Scacchi 2003]. Self-organization as a configurational notion, does not imply self-regulation as a control notion, nor vice-versa.

The fourth law, *conservation of organizational stability,* indicates the average effective global activity rate on an evolving system is invariant over the product lifetime. First, in the case of closed source software systems, it is unclear what happens to the global activity rate when the corporation evolving its software system product releases undergoes a major decline in its market valuation, as has been the situation for most large telecommunications system firms during 2000-2003, and its executives lay off significant portions of its software staff to reduce internal costs. In such a situation, would we expect that global activity rate and system releases to decline or remain invariant, though the product is still being deployed and sustained? Second, it is unclear what is meant by, or what data is used as a measure of, "global activity rate." Does it indicate some form of software productivity, staff size, or software development budget and schedule? No data or results for software productivity, budget or schedule are provided, so staff size and software release size can only serve as weak surrogate measures [Scacchi 1991]. Studies of the GNOME user interface and Debian Linux distribution report a super-linear growth rate in system release size as staff size increases incrementally, which would seem to contradict the notion of a constant/invariant level of global activity.

The fifth law, *conservation of familiarity,* specifies that during the active life of an evolving program, the incremental growth size of successive releases tend to decline. In the case of F/OSS, there are examples of incremental growth of successive releases of different system applications that either increase, remain relatively flat, or decrease. When architectural bricolage or system mergers occur in F/OSS, they can introduce discontinuous increases in system growth. Whereas when software miniaturization and refactoring occur, they can significantly decrease the size of a system application. Furthermore, the size (small versus large systems), business or product marketing strategies, distribution of evolution effort across sub-systems or modules, and technological infrastructure and regime also seem to impinge on system growth in ways that either increase or decrease system growth rates across system releases.

The sixth law, *continuing growth,* indicates that the functional content of a program must be continually increased to maintain user satisfaction over its lifetime. Once again, there is no empirical grounding that provides data on how user satisfaction is assessed, or how it may change for a particular release versus over a system's lifetime. In most studies, there is evidence indicating growth in functional content of a program, as measured by the number of modules, source code files, object classes or methods appearing across a consecutive sequence of system releases. However, software miniaturization and refactoring bring awareness that users may also want their programs to operate on a smaller form factor in a manner that streamlines redundant code (for multiple platforms), simplifies the build and debugging process, improves execution performance on resource limited platforms, or eliminates decayed code. These conditions effectively remove

source code functionality and thus appear to do so in a way that maintains or improves user or developer-user satisfaction.[6]

The seventh law states that E-type programs will be perceived as of *declining quality* unless adapted to a changing operation environment. How is software quality to be assessed, and what data is to be measured to establish the trend over time? Are the number and type of software bug reports or modification requests direct indicators of software quality, or is it something else? If the functionality of a software release is held constant, as is the situation between releases/upgrades, then user and maintainer experience with the fixed system gives rise to declining perceptions of software system quality. So quality is a property or outcome of experience with a system, rather than solely a property of the system itself. Does an increase in software release size and complexity imply, correlate, or coincide with a decline in perceived software quality? System size, module coupling, and code decay grow in large software systems that have long useful lifetimes, sometimes following a super-linear trend for F/OSS systems. However, the population (user base and number) of deployed systems that employ F/OSS applications exhibiting super-linear trends, continues to grow and diversify. Subsequently, it is unclear what the rate and trend of this growth are. Does the duration and frequency of release cycles affect perceived system quality? Compared to closed source system applications from the mainframe regime, the duration of the evolutionary release cycles of large F/OSS systems has been effectively declining accompanying the transition to more frequent system releases (nightly build, alpha, beta, candidate, and full release versions), most of which are designated and perceived as unstable, hence of lower quality than expected of full stable releases. This evolved releasing cycle may therefore contribute to mitigating perceived quality declines, since there is a shorter duration of system constancy in which to acquire experience. At the same time, this may increase perceived quality through the rapid rate of change made by user-developers in response to defects or shortfalls in system functionality or features.

The eighth law, *feedback system,* specifies that E-type software evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to be successfully modified or improved. From the early days of software engineering, processes have been decomposable into sub-process levels (development entails design (functional or object-oriented), coding, testing, etc.), multi-loop (Waterfall model, iterative refinement model), and involving people and tools as agents performing software engineering activities [cf. Scacchi 2002b]. Thus, is this law merely a tautological prescription for what abstract properties a model of the software evolution process should represent? Alternatively, how are software evolution processes nested, webbed or otherwise arranged, or does any multi-level arrangement suffice? Are these levels of process abstraction, process decomposition, or both? What is the scope and content of the feedback loops? Are the loops nested or overlapping? What kinds of events or signals emitted by the process are used to determine whether a loop iterates or exits?

[6] Lehman, Ramil, and Kahen [2001] identify the notion of "anti-regressive" complexity control which seems related to refactoring. However, in their view, anti-regressive software development work consumes effort without any immediate, perceived stakeholder return in system value, reflected in system functional power or performance. Their view stands in contrast to the studies, results, and observations reported here.

Who are the agents, what roles do they play, and what process activities do they perform? What is the system of feedback; what is fed back to where, and why? Does feedback arise from level shifting, agent action, tool selection, tool usage, or something else, individually or in combination, at end of loop decision points? What kind of feedback control system is implied here – is it intended to merely serve as a metaphor to aid in planning and managing the allocation of corporate resources, or is their some physical or organizational embodiment being referenced, and if so, what is it? What is it about these software evolution products, processes or productive units that may be distinctive or unique to a feedback control system perspective? What empowerment or insight is provided, at what level of abstraction, and to what audience, from the identification of software evolution processes as a feedback system? None of the quantitative studies of software evolution, whether of closed or open source systems, provide data that specifies what the elements of the feedback system are or how they are arranged; what external/internal events, signals, or data streams should be monitored and controlled; or what the actuators are that control system behavior or outcome values. It is therefore unclear what empirically observed or theoretically derived situation could be investigated to determine whether such a law can be tested, refined, or refuted.

Overall, this analysis of the laws of software evolution in light of a substantial set of empirical studies of closed source and open source software development patterns suggests that the laws may not account for the data and studies at hand. Thus, it is appropriate to consider how the laws and theory of software evolution might be revised or rethought to provide a more adequate account that can link theory, practice, and empirical study.

Do We Need New Laws, Models or Theories for Open Source Software Evolution?

At this point it is reasonable to ask about the status of the laws of software evolution in general, and of F/OSS evolution in particular. Data trends and patterns accounting for the evolution of F/OSS is some cases conform, in other cases clearly do not conform and may refute the laws of software evolution. As such, simply refining or restating the laws is probably inadequate, while reformulating them to account for the data at hand is beyond the scope of this chapter. However, it is possible to reconsider the underlying ontologies for software evolution to rethink how the laws may be reformulated, as well as what kinds of theory or models of software evolution may further help in understanding, reasoning about, and explaining the evolution of both closed and open source software systems.

Embracing the Feedback Control Systems Ontology

Feedback and feedback systems are part of the conceptual foundation of the laws and theory of software evolution developed by Lehman and colleagues. So why not refine and revise the laws in a way that more fully embraces feedback control theory in articulating the laws so that they can address the evolution of F/OSS. This should be possible with an eye toward the interests of the audience for whom the laws are intended to serve. For example, executives and senior managers responsible for large software development

centers want to know where to make strategic investments and how to better allocate their software development staff, schedules, and related resources. Software developers may want to know what kinds of tools and techniques to use to make their software evolution efforts faster, better and cheaper. Scholars of software evolution theory want to know what kinds of data are collected, what types and sizes of systems are studied, what types of criteria are used in designing theoretically motivated samples of software systems under study, and what tests apply to verify, refine, or refute the laws or theory at hand. In terms of feedback control systems, there is need to identify where sensors should be placed in a software productive unit to collect different types of software change data, and to what or whom do they feed back. Similarly, there is need to identify where are the feedback control loops to be placed, where are their begin and end points, what functionality is within each loop, and what decision function determines whether a loop iterates or exits. It is also necessary to identify what roles people and software tools play in the regulation or control of the feedback system, or what feedback they produce, use, or consume along the way. Managers, developers, and scholars want to know how different types of feedback get employed to regulate and control the centralized corporate or decentralized open source productive unit that develops and maintains software systems of different size, type, age, and application setting.

Recent efforts by Lehman, Ramil and Kahen [2001], for example, employ system dynamics modeling techniques and simulation tools to demonstrate and iteratively refine an operational model of software evolution that embodies the existing laws. Their model seems able to reproduce via simulation the evolutionary data trends, similar to those appearing in Figure 2, that conform to the laws of software evolution. However, their models do not address or account for the kind of F/OSS evolution data and trends reported in the studies examined herein. But the stage is set for how to proceed in pursuing the ontological foundation of the laws and theory of software evolution.

On the other hand, if the theory of feedback control systems becomes too complicated or too rigid of an ontological framework for describing and explaining software evolution, then alternative ontological frameworks may be employed to further such study.

Alternative Ontologies for F/OSS Evolution

One observation from studying the evolution of technical systems is that the technologies and techniques for developing and maintaining F/OSS constitute a distinct technological regime. This regime for F/OSS is not anticipated or covered by the current laws of software evolution. The same may also be true of emerging technologies like component-based software systems and those with dynamically composed run-time architectures. Thus, it seems that any ontology for software evolution should account for the emergence, deployment, and consequences of use for new tools, techniques, and concepts for software development, as well as the productive units, technical system infrastructure, and technological regime in they are situated.

A second observation from the study of the evolution of F/OSS is that different types of software system evolve at substantially different rates--some super-linear, some constant, some sub-linear, some not at all. Small software systems may not evolve or thrive for

very long, nor will they be assimilated into larger systems, unless merged with other systems whose developers can form a critical mass sufficient to co-evolve with the composite system and productive unit. Drawing from biological evolutionary theory, it may be that software evolution theory requires or will benefit from *taxonomic analyses* to describe, classify and name different types of software systems or architectural morphologies, thus refactoring the conceptual space for software system evolution. Similarly, it may benefit from *phylogenetic analyses* that reconstruct the evolutionary histories of different types of software systems, whether as open source and closed source implementations. Last, it suggests that a science of *software systematics* is needed to encourage study of the kinds and diversity of software programs, components, systems, and application domains, as well as relationships among them, across populations of development projects within different technological regimes over time. This would enable comparative study of contemporary software systems with their ancestral lineage, as well as to those found within the software fossil record (i.e., those software systems developed starting in the 1940's onward for mainframe computers, and those developed starting in the 1970's for personal computers). Finally, this could all be done in ways that enable free/open source computational modeling of such a framework for software evolution.

A third observation from the emergence and evolution of F/OSS is that the beliefs, narratives, and memes play some role in facilitating the adoption, deployment, use and evolution of F/OSS. Their role may be more significant than the cultural and language constructs that accompanied the earlier technological regime of centralized, closed source software development that primarily developed systems for deployment in corporate settings. Similarly, relatively new software language constructs for scripting, plug-in modules, and extensible software architectures have been popularized in the regime of F/OSS. But these constructs may also have enabled new forms of architectural evolution and bricolage, thereby accelerating the growth rate of large F/OSS in a manner incommensurate to that seen in the world of mainframe software systems, an earlier technological regime. Finally, large and popular F/OSS systems are being extended and evolved to accommodate end-users and developers whose native language or ethnic legacy is not English based. The internationalization or localization of F/OSS systems, while neither necessarily adding nor subtracting functionality, does create value in the global community by making these systems more accessible to a larger audience of prospective end-users, developers, reviewers and debuggers. These software extensions add to the bulk of F/OSS code release size in probably orthogonal ways, but may or may not represent anti-regressive work [cf. Lehman, Jamil, and Kehan 2001].

A fourth observation from the evolution of F/OSS is that they have emerged within a technological regime where competitive market forces and organizational ecologies surrounding closed source software systems may have effectively served to stimulate the growth and diffusion of F/OSS project populations. Furthermore, it may be the case that these circumstances are co-evolving with the relative growth/demise of open versus closed source software product offerings, and the communities of developers who support them. The laws of software evolution make no statement about the effects of market forces, competition, organizational ecology, co-evolution, or the spread of software

project populations as contributing factors affecting how software systems may evolve. Yet many of the largest F/OSS systems are pitted directly against commercially available, closed source alternatives. These F/OSS systems typically compete against those developed within centrally controlled and resource managed software development centers. Thus, it seems appropriate to address how co-evolutionary market forces surround and situate the centralized or decentralized organizational ecologies that develop and maintain large software systems in order to better understand how they evolve.

A last observation from a view of F/OSS as a socio-technical world is that the evolution of F/OSS system is situated within distinct web of organizational, technological, historical and geographic contexts. The laws of software evolution do not, nor do feedback control systems, account for organizational productive units or their historical circumstances. Similarly, there is no accounting for the motivations, beliefs, or cultural values of software developers who may prefer software systems to be developed in a manner that is free and open, so as to enable subsequent study, learning, reinvention, modification, and subsequent distribution. But as seen above, these are plausible variables that can contribute to the evolution of F/OSS, and thus further study is required to understand when, where and how they might so influence how particular F/OSS systems may evolve.

Conclusions

The laws and theory of software evolution proposed by Lehman and colleagues are recognized as a major contribution to the field of software engineering and the discipline of computer science. These laws have been generally found to provide a plausible explanation for how software systems evolve throughout their life. They have been explored empirically over a period of more than 30 years, so their persistence is a noteworthy accomplishment. Developing laws and theory of software evolution relying on empirically grounded studies is a long-term endeavor that poses many challenges in research method, theoretical sampling of systems to study, theory construction, and ongoing theory testing, refutation, and refinement.

As the technology, process, and practice of software development and maintenance has evolved, particularly in the past ten years and with the advent of large numbers of free/open source software development projects, it has begun clear that the laws and theory may be breaking down, at least from results of the many empirical studies reviewed in this chapter. The laws of software evolution do not provide a rich or deep characterization of the evolution of F/OSS systems. Part of the reason may stem from the observation that the laws were formulated in the context of software development and maintenance processes and work practices that were based in centralized, corporate software development centers that built large closed source system applications with few competitive offerings for use by large enterprises. Large F/OSS systems, on the other hand, are developed and maintained in globally decentralized settings that collectively denote a loosely-coupled community of developers/users who generally lack the administrative authority, resource constraints, and schedules found in centrally controlled software centers. These F/OSS systems are typically competing alternatives to closed

source commercial software product offerings. Furthermore, there is data, evidence, and findings from multiple studies of F/OSS systems that indicate F/OSS systems co-evolve with their user-developer communities, so that growth and evolution of each depends on the other. Co-evolution results of this kind are not (yet) reported for closed source systems, and it is unclear that such results will be found. In short, the laws of software evolution were developed within and apply to systems maintained and used in a corporate world and technological regime that differs from the socio-technical communities, global information infrastructure, and technological regime which embeds open source software.

It appears that the laws of software evolution need a more articulate explication and refinement if they are to account for the evolution of F/OSS systems. One way this might be done is to embrace and extend reliance of the ontology of feedback control systems theory. This would entail identifying the types, operations, behaviors, and interconnection of mechanisms that embody and realize a complex, multi-level, multi-loop, and multi-agent feedback system. Building computational models and simulations of such a system (or family of systems) could be a significant contribution. Otherwise, alternative evolutionary ontologies might be adopted, individually or in some explicit hybrid combination form. The choice of which ontology to use will help determine what types of entities, flows, mechanisms, and controls for software evolution should be modeled, measured, improved, and refined according to some conceptual or theoretically motivated framework. Otherwise, use of alternative ontologies may accommodate new models of theories of software evolution that do not rely on high-level, abstract or over-generalized laws, but instead may result in theories or models of smaller and more precise scope that better account for the complex, socio-technical ecological niches where software systems evolve in practice, as well as for the type and history of the system in such context.

Theories of software evolution should be empirically grounded. They should be formulated or modeled in ways in which they can be subject to tests of refutation or refinement. The tests in turn should examine comparative data sets that are theoretically motivated, rather than motivated by the convenience of data at hand that may have been collected and conceived for other more modest purposes. There should be theories that address software evolution within, as well as, across generations of software technology or technological regimes. Laws and theories of software evolution should have a computational rendering so that their source code, internal representation, and external behavior can be observed, shared, studied, modified and redistributable. They should be free (as in *libre*) and open source. These models should then also be suitable for simulation, analysis, visualization, prototyping, and enactment [Scacchi 2002b, Scacchi and Mi 1997]. By doing this, software engineering and computer science can make a new contribution in the form of reusable assets that can be adopted and tailored for use in other domains of evolution theorizing.

The future of research in software evolution is more likely to lie within the technological regime of F/OSS. This will be an increasingly practical choice for empirical study of individual systems, groups of systems of common type, and of larger regional or global populations of systems. This is due in part to the public availability of the source code and related assets on the Web for individual versions/releases of hundreds of application

systems, as well as data about their development processes, community participants, tools in use, and settings of development work. Not that collecting or accessing this data is without its demands for time, skill, effort and therefore cost, but that useful and interesting data can be accessed and shared without the barriers to entry and corporate disclosure constraints of intellectual property claims or trade secrets. It seems unlikely that the software engineering community will get open access to the source code, bug report databases, release histories, or other "property or secrets" of closed source systems that are in widespread use (e.g., Microsoft Windows operating systems, Internet Explorer, Word, Outlook, Office, Oracle DBMS, or SAP R/3) in ways that can be shared and studied without corporate trade secret and publication constraints. In contrast, it is possible today to empirically study the ongoing evolution of the GNU/Linux operating systems (Kernel or alternative distributions), the Mozilla Web browser, Open Office, SAP DB, or GNUenterprise, which together with their respective and internetworked (technically and socially) communities, have publicly accessible Web portals and software assets that can be shared, studied, and redistributed to support research into the laws and theory of software evolution. The future of research in software evolution should be free and open, since it will likely take a community of investigators to help make substantial progress in developing, refining, sharing, and publishing models, laws, and theories of software evolution.

Acknowledgements

The research described in this report is supported by grants from the National Science Foundation #IIS-0083075, #ITR-0205679 and #ITR-0205724. No endorsement implied. Mark Ackerman at the University of Michigan Ann Arbor; Les Gasser at the University of Illinois, Urbana-Champaign; John Noll at Santa Clara University; Margaret Elliott, Mark Bergman, Chris Jensen and Xiaobin Li at the UCI Institute for Software Research; and Julia Watson at The Ohio State University are also collaborators on the research project from which this article was derived.

References

W.J. Abernathy, *The Productivity Dilemma: Roadblock to Innovation in the Automobile Industry*, John Hopkins University Press, 1978.

R.N Bateson, *Introduction to Control System Technology*, Prentice-Hall, Englewood Cliffs, NJ, 1993.

S. Beecham, T. Hall, and A. Rainer, Software Process Improvement Problems in Twelve Software Companies: An Empirical Analysis, *Empirical Software Engineering*, 8(1), 7-42, 2003.

S. Bendifallah and W. Scacchi, Understanding Software Maintenance Work, *IEEE Trans. Software Engineering*, 13(3), 311-323, March 1987. Reprinted in D. Longstreet (ed.), *Tutorial on Software Maintenance and Computers*, IEEE Computer Society, 1990.

P.J. Bowler, *Evolution: The History of an Idea* (Revised Edition), University of California Press, Berkeley, CA, 1989.

A. Capilupppi, P. Lago, and M. Morisio, Evidences in the Evolution of OS projects through Changelog Analyses, technical report, 2003.

E. Carmel and S. Becker, A Process Model for Packaged Software Development, *IEEE Trans. Engineering Management,* 41(5), 50-61, 1995.

E. Carmel and S. Sawyer, Packaged Software Development Teams: What makes them different, *Information, Technology, and People*, 11(1), 7-19, 1998.

M. Christiansen and S. Kirby (eds.), *Language Evolution: The States of the Art*, Oxford University Press, 2003.
C.S.K. Chong Hok Yuen, A Statistical Rationale for Evolution Dynamics Concepts, *Proc. Third Conf. Software Maintenance*, 1987.

C.S.K. Chong Hok Yuen, On Analyzing Maintenance Process Data at the Global and Detailed Levels, *Proc. Fourth Conf. Software Maintenance*, 1988.

R. Conradi and A. Fuggetta, Improving Software Process Improvement, *IEEE Software*, 92-99, July-August 2002.

S. Cook, H. Ji and R. Harrison, Software Evolution and Software Evolvability, unpublished manuscript, University of Reading, UK, 2000.

M.A. Cusumano and D.B. Yoffie, Software Development on Internet Time, *Computer*, 60-70, October 1999.

M. Di Penta, M. Neteler, G. Antoniol, and E. Merlo, Knowledge-Based Library Refactoring for an Open Source Project, *Proc. IEEE Working Conf. Reverse Engineering*, Richmond VA, October 2002.

J.C. Doyle, B.A Francis and A.R. Tannenbaum, *Feedback Control Theory*, Macmillan, New York, 1992.

S.G. Eick, T.L. Graves, A.F. Karr, J.S. Marron, and A. Mocku, Does Code Decay? Assessing the Evidence from Change Management Data, *IEEE Trans. Software Engineering*, 27(1), 1-12, January 2001.

M. Elliott and W. Scacchi, *Free Software Development: Cooperation and Conflict in a Virtual Organizational Culture*, technical report, Institute for Software Research. Revised version to appear in S. Koch (ed.), *Free/Open Source Software Development*, Idea Press, 2003

L. Gabora, The Origin and Evolution of Culture and Creativity, *Journal of Memetics - Evolutionary Models of Information Transmission*, 1, 1997.
http://jom-emit.cfpm.org/vol1/gabora_l.html

L. Gabora, The Beer Can Theory of Creativity, in P. Bentley and D. Corne (eds.) *Creative Evolutionary Systems*, Morgan Kaufman, 2000.

H. Gall, M. Jayazeri, R. Kloesch and G. Trausmuth, Software Evolution Observations Based on Product Release History, *Proc. 1997 Intern. Conf. Software Maintenance (ICSM'97),* Bari, IT, October 1997.

B. Glaser and A. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*, Aldine Publishing, Chicago, IL, 1976.

M.W. Godfrey and E.H.S. Lee, Secrets from the Monster: Extracting Mozilla's Software Architecture, *Proc.Second Intern. Symp. Constructing Software Engineering Tools* (CoSET-00), Limerick, Ireland, June 2000.

M.W. Godfrey and Q. Tu, Evolution in Open Source Software: A Case Study, *Proc. 2000 Intern. Conf. Software Maintenance* (ICSM-00), San Jose, California, October 2000.

J.M Gonzalez-Barahona, M.A. Ortuno Perez, P. de las Heras Quiros, J. Centeno Gonzalez, and V. Matellan Olivera, Counting Patotoes: The Size of Debian 2.2, *Upgrade Magazine*, II(6), 60-66, December 2001.

S.J. Gould, *The Structure of Evolutionary Theory*, Harvard University Press, Cambridge, MA, 2002.

A. Hars and S. Ou, Working for Free? Motivations for Participating in Open-Source Software Projects, *Intern. J. Electronic Commerce*, 6(3), 25-39, 2002.

T.J. Hughes, The Evolution of Large Technological Systems, in W. Bijker, T. Hughes, and T. Pinch (eds.), *The Social Construction of Technological Systems*, MIT Press, Cambridge, MA, 51-82, 1987.

F. Hunt and P. Johnson, On the Pareto Distribution of SourceForge Projects, in C. Gacek and B. Arief (eds.), *Proc. Open Source Software Development Workshop*, 122-129, Newcastle, UK, February 2002.

C. Jensen and W. Scacchi, Simulating an Automated Approach to Discovery and Modeling of Open Source Software Development Processes, *Proc. Software Process Simulation and Modeling Workshop (ProSim'03),* Portland, OR, May 2003.

M.T. Hannan and G.R. Carroll, *Dynamics of Organizational Populations: Density, Legitimation and Competition*, Oxford University Press, New York, 1992.

C.F. Kemerer and S. Slaughter, An Empirical Approach to Studying Software Evolution, *IEEE Trans. Software Engineering*, 25(4), 493-505, 1999.

G. Kniesel, J. Noppen, T. Mens, and J. Buckley, *WS 9. The First International Workshop on Unanticipated Software Evolution,* Workshop Report, Malaga, Spain, June 2002. http://joint.org/use2002/ecoopWsReportUSE2002.pdf

S. Koch and G. Schneider, Results from Software Engineering Research into Open Source Development Projects Using Public Data, *Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft*, Hans R. Hansen und Wolfgang H. Janko (Hrsg.), Nr. 22, Wirtschaftsuniversität Wien, 2000.

I. Lakatos, *Proofs and Refutations: The Logic of Mathematical Discovery*, Cambridge University Press, Cambridge, UK, 1976.

M.M. Lehman, Programs, Life Cycles, and Laws of Software Evolution, *Proc. IEEE*, 68, 1060-1078, 1980.

M.M. Lehman, Rules and Tools for Software Evolution Planning and Management, in J. Ramil (ed.), *Proc. FEAST 2000*, Imperial College of Science and Technology, London, 53-68, 2000.

M.M. Lehman, Software Evolution, in J. Marciniak (ed.), *Encyclopedia of Software Engineering,* 2nd Edition, John Wiley and Sons Inc., New York, 1507-1513, 2002.

M.M. Lehman and L.A. Belady, *Program Evolution – Processes of Software Change*, Academic Press, London, 1985.

M.M. Lehman, D.E. Perry and J.F. Ramil, Implications for Evolution Metrics on Software Maintenance, *Proc. 1998 Intern. Conf. Software Maintenance (ICSM'98),* Bethesda, MD, 1998.

M.M. Lehman and J.F. Ramil, An Approach to a Theory of Software Evolution, *Proc. 2001 Intern. Workshop on Principles of Software Evolution*, 2001.

M.M. Lehman, J.F. Ramil, and G. Kahen, *A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics*, technical report, Dept. of Comp., Imperial College, London, September 2001.

M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry, and W. Turski, Metrics and Laws of Software Evolution – The Nineties View, *Proc. 4th Intern. Symp. Software Metrics*, 20-32, Albuquerque, NM, November 1997.

G. Madey, V. Freeh, and R. Tynan, The Open Source Software Development Phenomenon: An Analysis Based on Social Network Theory. *Proc. Americas Conference on Information Systems (AMCIS2002)*. 1806-1813, Dallas, TX, 2002.

T. Mens, J. Buckley, M. Zenger, and A. Rashid, Towards a Taxonomy of Software Evolution, *Second Intern. Workshop on Unanticipated Software Evolution,* Warsaw, Poland, April 2003. http://joint.org/use2003/Papers/18500066.pdf

M.H. Meyer and J.M. Utterback, The Product Family and the Dynamics of Core Capability, *Sloan Management Review*, 34(3), 29-47, Spring 1993.

P. Mi and W. Scacchi, A Knowledge-Based Environment for Modeling and Simulating Software Engineering Processes, *IEEE Trans. Data and Knowledge Engineering*, 2(3), 283-294, September 1990. Reprinted in *Nikkei Artificial Intelligence*, 20(1), 176-191, January 1991 (in Japanese); also in *Process-Centered Software Engineering Environments*, P.K. Garg and M. Jazayeri (eds.), IEEE Computer Society, 119-130, 1996.

P. Mi and W. Scacchi, Process Integration in CASE Environments, *IEEE Software*, 9(2), 45-53, March 1992. Reprinted in Eliot Chikofsky (ed.), *Computer-Aided Software Engineering (CASE)*, Second Edition, IEEE Computer Society, 1993.

P. Mi and W. Scacchi, A Meta-Model for Formulating Knowledge-Based Models of Software Development, *Decision Support Systems*, 17(4), 313-330, 1996.

A. Mockus, R.T. Fielding, and J. Herbsleb, Two Case Studies of Open Source Software Development: Apache and Mozilla, *ACM Trans. Software Engineering and Methodology*, 11(3), 309-346, 2002.

K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y.Ye, Evolution Patterns of Open-Source Software Systems and Communities, *Proc. 2002 Intern. Workshop Principles of Software Evolution*, 76-85, 2002.

B. Nardi and V. O'Day, *Information Ecologies: Using Technology with Heart*, MIT Press, Cambridge, MA 1999.

R.R. Nelson and S.G. Winter, *An Evolutionary Theory of Economic Change*, Belknap Press, Cambridge, MA, 1982.

M.H. Nitecki, (ed.), *Evolutionary Progress*, University of Chicago Press, Chicago, IL, 1988.

S. O'Mahony, Developing Community Software in a Commodity World, in M. Fisher and G. Downey (eds.), *Frontiers of capital: Ethnographic Reflections on the New Economy*, Social Science Research Council, (to appear), 2003.

D.E. Perry, H.P. Siy, and L.G. Votta, Parallel Changes in Large-Scale Software Development: An Observational Case Study, *ACM Trans. Software Engineering and Methodology*, 10(3), 308-337, 2001.

K.R. Popper, *Conjectures and Refutations*, Routledge & Kagen, 1963.

G. Robles-Martinez, J.M. Gonzalez-Barahona, J. Centeno Gonzalez, V. Matellan Olivera, and L. Rodero Merino, Studying the Evolution of Libre Software Projects using Publicly Available Data, technical report, 2003.

P.P. Saviotti and G.S. Mani, Competition, Variety and Technological Evolution: A Replicator Dynamics Model, *J. Evolutionary Economics*, 5(4), 369-92, 1995.

W. Scacchi, Understanding Software Productivity: Towards a Knowledge-Based Approach, *Intern. J. Software Engineering and Knowledge Engineering*, 1(3), 293-321, 1991. Revised version in D. Hurley (ed.), *Advances in Software Engineering and Knowledge Engineering*, Volume 4, 37-70, 1995.

W. Scacchi, Experiences in Software Process Simulation and Modeling, *J. Systems and Software*, 46(2/3), 183-192, 1999

W. Scacchi, Understanding the Requirements for Developing Open Source Software Systems, *IEE Proceedings – Software*, 149(1), 24-39, February 2002a.

W. Scacchi, Process Models for Software Engineering, in J. Marciniak (ed.), *Encyclopedia of Software Engineering,* 2nd Edition, John Wiley and Sons Inc., New York 993-1005, 2002b.

W. Scacchi, *Open EC/B: A Case Study in Electronic Commerce and Open Source Software Development*, technical report, Institute for Software Research, July 2002c.

W. Scacchi and P. Mi, Process Life Cycle Engineering: Approach and Support Environment, *Intern. J. Intelligent Systems in. Accounting, Finance, and Management*, 6:83-107, 1997.

S.R. Schach, B. Jin, D.R. Wright, G.Z. Heller, and A.J. Offutt, Maintainability of the Linux Kernel, *IEE Proceedings – Software*, 149(1), 18-23, February 2002.

N. Smith and J.F. Ramil, Qualitative Simulation of Software Evolution Processes, *WESS'02 Eighth Workshop on Empirical Studies of Software Maintenance*, Montreal, October 2002.

M. Svahnberg and J. Bosch, Evolution in Software Product Lines, *J. Software Maintenance*, 11(6), 391-422, 1999.

T. Tamai and Y. Torimitsu, Software Lifetime and its Evolution Process over Generations, *Proc. Conf. Software Maintenance*, 63-69, November 1992.

W. Turski, Reference Model for Smooth Growth of Software Systems, *IEEE Trans. Software Engineering*, 22(8), 599-600, August 1996.

J.M. Utterback, *Mastering the Dynamics of Innovation: How Companies Can Seize Opportunities in the Face of Technological Change*, Harvard Business School Press, 1994.

N.M. Victor and J.H. Ausubel, DRAMs as Model Organisms for Study of Technological Evolution, *Technological Forecasting and Social Change*, 69(3), 243-262, April 2002.

J. van den Ende and R. Kemp, Technological transformations in history: how the computer regime grew out of existing computing regimes, *Research Policy*, 28, 833-851, 1999.

E. von Hippel and R. Katz, Shifting Innovation to Users via Toolkits, *Management Science*, 48(7), 821-833, July 2002.

R. Yin, *Case Study Research: Design and Methods (Second Edition)*, Sage Publications, Newbury Park, CA. 1994.