The Evolution of the Laws of Software Evolution: A Discussion Based on a Systematic Literature Review

ISRAEL HERRAIZ, Technical University of Madrid, Spain
DANIEL RODRIGUEZ, University of Alcala, Madrid, Spain
GREGORIO ROBLES and JESUS M. GONZALEZ-BARAHONA, GSyC/Libresoft, University Rey Juan Carlos, Madrid, Spain

After more than 40 years of life, software evolution should be considered as a mature field. However, despite such a long history, many research questions still remain open, and controversial studies about the validity of the laws of software evolution are common. During the first part of these 40 years, the laws themselves evolved to adapt to changes in both the research and the software industry environments. This process of adaption to new paradigms, standards, and practices stopped about 15 years ago, when the laws were revised for the last time. However, most controversial studies have been raised during this latter period. Based on a systematic and comprehensive literature review, in this article, we describe how and when the laws, and the software evolution field, evolved. We also address the current state of affairs about the validity of the laws, how they are perceived by the research community, and the developments and challenges that are likely to occur in the coming years.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance and Enhancement

General Terms: Management

Additional Key Words and Phrases: Laws of software evolution, software evolution

ACM Reference Format:
Herraiz, I., Rodriguez, D., Robles, G., and Gonzalez-Barahona, J. M. 2013. The evolution of the laws of software evolution: A discussion based on a systematic literature review. ACM Comput. Surv. 46, 2, Article 28 (November 2013), 28 pages.
DOI: http://dx.doi.org/10.1145/2543581.2543595

1. INTRODUCTION

In 1969, Meir M. Lehman did an empirical study (originally confidential, later published [Lehman 1985b]) within IBM, with the idea of improving the company's programming effectiveness. The study received little attention in the company and had no impact on its development practices. This study, however, started a new and prolific field of research: *software evolution*.

Software evolution deals with the process by which programs are modified and adapted to their changing environment. The aim of Lehman's research was to formulate a scientific theory of software evolution. As any sound theory, it was meant to be based on empirical results and aimed at finding invariant properties to be observed on entire classes of software development projects. As a result of his research, some invariants were found, which were first described in Lehman [1974] as the *laws of software evolution*.

Author's address: israel.herraiz@upm.es.

DOI: http://dx.doi.org/10.1145/2543581.2543595

After several years of intense activity and refinement, the last version of the laws was published in 1996 [Lehman 1996b], remaining unmodified since then. However, it is since 1996 that most studies questioning or confirming their validity have been published. We have framed our article based on the discussions and issues raised by those studies, leading us to the following *research questions*:

(RQ1) Why did the laws of software evolution undergo so many modifications during their early life, but have not changed in the last 15 years?
(RQ2) Which laws have been confirmed in the research literature? Which laws have been invalidated?
(RQ3) Which strategies, data, and techniques can be used to validate the laws of software evolution?
(RQ4) According to the reported results, which kind of software projects, and under which conditions, fulfill the laws of software evolution?

We will address these research questions in this article based on the literature published since the laws were first formulated. We have also explored the context of these questions by carefully reviewing how the laws have changed over time, the reasons for those changes, and the studies that have researched those changes.

The main contributions of our study can be summarized as follows:

—It is a comprehensive and systematic literature review[1] of the field of software evolution, since its inception to the most recent empirical studies, focusing on the empirical support for the laws.
—It is a detailed description of the process of progressive adaptation of the laws to a changing environment, a sort of evolution of the laws of software evolution, and how this process has been driven by the empirical results on the validity of the laws.
—It is an analysis of the current research environment and the impact of the availability of large software repositories for empirical software evolution research.
—It is a specific description of the state of the research of the laws of software evolution in the field of libre (free, open source) software development, where most cases of apparent invalidity have been reported.
—Finally, we highlight new challenges and criteria for future research, based on both the original recommendations by Lehman [1974] and the availability of new research repositories.

The amount of literature we have considered is vast, ranging over 40 years. For this period, we have performed a comprehensive analysis of the most relevant publications in the field. We start with the early studies that led to the formulation of the laws and finish with the most recent works questioning or confirming their validity.

We not only include the publications considering the laws as a whole but also summarize and classify the empirical studies aimed at validating aspects of specific laws. In fact, this *validity question* is perhaps the expression of a crisis in the field, resulting in the laws being unrevised for more than 15 years.

Lehman's seminal book published in 1985 [Lehman and Belady 1985], which reprinted most of the studies of the 1960s and 1970s, is an ideal background introduction to the early period of software evolution. A more modern, general overview of the area can be found in two recent books: Madhavji et al. [2006] and Mens and Demeyer [2008].

The rest of this article is organized as follows. Section 2 describes the initial formulation of the laws of software evolution. Then, the results of the systematic literature

[1]The details of the methodology followed for the systematic literature review are described in Appendix A.

Table I. Laws of Software Evolution in 1974

I	*Law of continuing change* A system that is used undergoes continuing change until it becomes more economical to replace it by a new or restructured system.
II	*Law of increasing entropy* The entropy of a system increases with time unless specific work is executed to maintain or reduce it.
III	*Law of statistically smooth growth* Growth trend measures of global system attributes may appear stochastic locally in time and space but are self-regulating and statistically smooth.

review are presented in chronological order, organized into four periods: before 1980, 1980–1989, 1990–1999, and 2000–2011. This organization into four periods fits well within the different stages in the history of software evolution as a research field and the different versions of the laws that have been proposed. Section 3 analyzes how they changed during the 1970s. In Section 4, we study the different works published during the 1980s, with special emphasis on the 1985 book [Lehman and Belady 1985] and the first studies on the validity of the laws. Section 5 moves forward to the 1990s, reviewing the works within the scope of the FEAST project. In Section 6, we describe the studies that questioned the validity of the laws in the case of libre software development. We then summarize the main findings in Section 7, providing answers to the research questions. After that, in Section 8, we propose some challenges for software evolution research in the future. In Section 9, we include a brief list of recommended readings for researchers coming to the field. Finally, Section 10 concludes this article. The details and methodology about the systematic literature review are given in Appendix A, included at the end of the article.

2. THE ORIGINAL LAWS

The laws of software evolution were presented for the first time in Lehman's inaugural professorship lecture at the Imperial College London [Lehman 1974] (reprinted as Lehman [1985c]). Initially, Lehman proposed three laws, shown in Table I, stating three basic principles for the evolution of software systems:

—Software systems must be continuously changed to adapt to the environment.
—Changes increase the complexity of software.
—Software evolution can be studied using statistical methods.

The first two laws show the fundamental idea behind software evolution. Software must change, but current changes make future ones more difficult to do, because the former increase complexity. This only happens for *large* software projects, not considering the size of the product but the size and structure of the project team. Thus, the laws of software evolution should be applied only to projects with several managerial levels.

Interestingly, if we compare the previously mentioned principles with Table I, we observe that Lehman did not use the term *complexity*, and referred instead to *entropy*. Lehman chose this term to highlight the fact that changes introduce disorder, and therefore the structure of programs *degrades*, making systems more difficult to comprehend, manage, and change. However, in the rest of the 1974 paper, he used the term *complexity* instead.

Lehman was referring to what today would be labeled as architecture complexity of the system. He distinguished between three levels of complexity: *internal*, *intrinsic,* and *external*. Internal complexity has to do with the structural attributes of the code. Intrinsic complexity is related to the dependencies between different parts of the

Table II. Laws of Software Evolution in 1978

I	*Law of continuing change* A program that is used undergoes continuing change or becomes progressively less useful. The change process continues until it is judged more cost effective to replace the system with a re-created version.
II	*Law of increasing complexity* As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.
III	*Law of statistically regular growth* Measures of global project and system attributes are cyclically self-regulating with statistically determinable trends and variances.
IV	*Law of invariant work rate* The global activity rate in a large programming project is invariant.
V	*Law of incremented growth limit* For reliable, planned evolution, a large program undergoing change must be made available for regular user execution (released) at maximum intervals determined by its net growth. That is, the system develops a characteristic average increment of safe growth, which, if exceeded, causes quality and usage problems, with time and cost overruns.

program.[2] Finally, external complexity is a measure of the understandability of the code provided documentation for it is available.

The third law was the first proposal for a statistical approach to the study of software evolution. The literal text of the law (see Table I) refers to stochastic growth trends, pointing out that these trends are self-regulating and statistically smooth. Because the trends are statistically smooth, they can be studied and forecasted using statistical methods. Lehman suggested the use of regression techniques, autocorrelation plots, and time series analysis for the study of software evolution, with a rationale based on the idea of feedback trends. However, the research environment of the time was very constrained: Access to software projects was difficult and scarce, and such statistical approaches were difficult to implement in practice. Current research environments are much richer, making this statistical approach possible (see Section 8.1).

3. THE LAWS IN THE 1970S

The first version of the laws of software evolution was based on a single case study, the OS/360 operating system [Belady and Lehman 1976]. Soon, Lehman started to search for further empirical support [Lehman and Parr 1976], following the statistical approach introduced in his 1974 lecture [Lehman 1974]. The new results led to the modification of the original laws as well as to the addition of two new ones [Lehman 1978] (see Table II).

Although the first three laws kept their essential meaning, their formulation changed. The first law, the basic principle of software evolution, now includes a clarification: Software must change or it becomes less useful.

In the second law, *entropy* was changed to *complexity*, the term used in the research context of the time. This context can be better understood thanks to the analysis by Belady of the research on software complexity during the 1970s [Belady 1979] (reprinted as Belady [1985]).

The third law was modified to include a mention of "statistically determinable trends and variances." While working at IBM, Lehman gained access to a subset of metrics regarding the evolution of OS/360: size of the system (in number of modules); number of modules added, removed, and changed; release dates; amount of manpower and

[2]A large program has a defined task that is divided into many subtasks. The intrinsic complexity is related to the relationships and the extent of those subtasks.

machine time used; and costs involved for each release. When he plotted these variables over time, plots resulted to be apparently stochastic. However, when averaged, variables could be classified into two groups: Some of them grew smoothly, while others showed some kind of conservation (either remained constant or with a repeating sequence). Lehman started to think of software evolution as a *feedback-driven* process, self-regulated, whose properties (trends, variances) could be estimated by empirical studies based on statistical methods. He did not explicitly mention the term *feedback* at this time, but it became a fundamental concept of the software evolution field during the 1990s.

The fourth law states that the work rate remains invariant over the lifetime of the project. In other words, using the terms of the third law: There is a statistically invariant trend in the activity rate of the project. That invariance is probably due to the feedback process governing the evolution of the system. This law can be understood as a subcase or corollary of Brooks' law [Brooks 1978]: Regardless of the variations in manpower in a software project, work rate remains invariant.

The fifth law states that there is a *safe* growth rate, and that the interval between releases should be kept as wide as possible to maintain the growth rate under control. This reflects the release practices of the time, when software was shipped through mail and courier and the distribution costs were much higher than today. Under those constraints, it is better to make sure that products are released only when they are ready, rather than releasing early and continuously sending updates to users (an extended practice nowadays).

4. THE LAWS IN THE 1980S

The 1980s saw the birth of the seminal book on software evolution [Lehman and Belady 1985], still called *Program Evolution* at the time. It reprinted many of the original works in the field, which maybe would have been lost otherwise.[3] It contains not only the laws of software evolution as formulated at the time but also all previous works, which makes it an invaluable resource to reconstruct the history of software evolution. It helps to understand the environment where the laws of software evolution were conceived. Finally, it also helps to realize how the laws are not immutable, and that they have undergone significant change since their original conception.

The book reprinted the first mathematical model of software evolution [Woodside 1985], which was proposed by Woodside [1980]. This was the first attempt to simulate the evolution of software using the concepts behind the laws of software evolution. Woodside's model was based on a balance between *progressive* and *antiregressive* work in the software process. As Lehman had shown in the previous decade, progressive work introduces new features in the system, while antiregressive work attempts to maintain the program well structured, documented, and evolvable. This balance between *progressive* and *antiregressive* work will be the base of software evolution simulation models in the next decades, which will be used to validate the laws, as we will show in Section 5.2.

This decade also saw the stabilization of the evolution framework, with new and more elaborated concepts such as the SPE scheme (described in Section 4.1), that refined the notion of *large* programs of the 1970s.

The second chapter of the 1985 book [Lehman 1985a] is a good summary of the state of the art of the decade, including the laws of program evolution, the SPE classification scheme, the notion of feedback and its influence on software evolution, and a discussion

[3]The book is out of print, and is kindly redistributed with permission of the author and the publisher by the ERCIM Working Group of Software Evolution at http://wiki.ercim.eu/wg/SoftwareEvolution/index.php/Publications (consulted October 2012).

on the need for a theory of software evolution. The second chapter reprints a paper that appeared a year before [Lehman 1984].

Finally, during these years the laws faced the first studies questioning their validity, including the PhD dissertation by Pirzada [1988], who proposed some changes to the laws such as making the third law specific only to commercial software.

Let us discuss these points further in the following subsections.

4.1. The SPE Scheme

In the early 1980s, Lehman abandoned the notion of *large programs*, modifying the laws to adapt to a new scheme, introduced with their new formulation [Lehman 1980]. Lehman narrowed the application of the laws to large programs because the evolutionary behavior of software was different in small ones.

Large programs are usually developed by large teams with more than one managerial level and have at their disposal a large user base, which could provide feedback to the programming processes. In addition, these programs usually produce some development and maintenance logs, which could be analyzed to test the laws.

However, this definition of *large* could not be applied without problems and uncertainty: there was no clear distinction, and two programs of similar size could behave differently. Because of this, the domain of applicability of the laws was changed from program size to the so-called *SPE scheme* classification. Under this scheme we can find three classes of programs:

—*S*-type (specified) programs are derivable from a static specification and can be formally proven as correct or not.
—*P*-type (problem-solving) programs attempt to solve problems that can be formulated formally, but which are not computationally affordable. Therefore, the program must be based on heuristics or approximations to the theoretical problem.
—*E*-type (evolutionary) programs are reflections of human processes or of a part of the real world. These kinds of programs try to solve an activity that somehow involves people or the real world.

The laws of software evolution were said to be referring only to *E*-type software. Because the world irremediably changes, this kind of software must be changed to keep it synchronized with its environment. Software requirements also change, because human processes are hard to define and state, which also leads to modifications in the software. Therefore, it is likely that a program, once implemented, released, and installed, will still need further changes requested by its users. Also, the very introduction of the system in the world will cause further demands for changes and new features. This last source of changes causes a feedback loop in the evolution of the program.

Unlike *E*-type, *S*-type software does not show an evolutionary behavior. Once the program is written, it is either correct or not with respect to a specification. If it is not correct, it will not be released, and there is no chance for evolution (by definition, it happens after the release of the program). However, if the program is correct, it is finished and there is no chance for further refinement or adaptation. This means that it will not reach the evolution stage either. In summary, because the specification of this type of programs is static, the program will remain the same regardless of any change in the environment.

The case of *P*-type software falls between the *E*- and *S*-type software. For *P*-type software, the specification cannot be completely defined before the implementation of the software. Thus, after releasing the system, the specification may still be subject to change, and it follows *P*-type software evolution. However, the nature of the specifications of *P*-type software is different from that of *E*-type software. The problem

Table III. Laws III, IV, and V Were Changed in 1980

III	*The Fundamental Law of Program Evolution* Program evolution is subject to dynamics that make the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and variances.
IV	*Conservation of Organizational Stability (Invariant Work Rate)* During the active life of a program the global activity rate in the associated programming project is statistically invariant.
V	*Conservation of Familiarity (Perceived Complexity)* During the active life of a program the release content (changes, additions, deletions) of the successive releases of an evolving program is statically invariant.

to be specified does not change, only our understanding of it. In *E*-type software, the environment around the system changes. Therefore, the evolution of *P*-type software is different from that of *E*-type. Lehman did not pay special attention to *P*-type software and focused on explaining the evolution of *E*-type software.

4.2. Changes to the Laws in the 1980s

In the early 1980s, Lehman published a new version of Laws III, IV, and V [Lehman 1979, 1980], as shown in Table III. The name of the third law was changed to *Fundamental Law of Program Evolution*, highlighting again the fact that software evolution is "self-regulating with statistically determinable trends and variances." Lehman also included a mention of the measures of process and product metrics that can be determined using statistical methods. In essence, the law is the same as in previous formulations. The change in the name reflects the importance that Lehman gave to this law, converting it in an essential property of evolution, and hence into a cornerstone of a theory of software evolution.

The fourth law remains the same in its essence too, referring to the stability within the organization that develops and maintains the system. Sudden and substantial changes are not possible in an organization that develops an *E*-type system. This causes a conservation of work rate in the programming project. In previous formulations, the law explicitly mentioned *large projects*, which are the kind of projects usually developed by organizations with several managerial levels (e.g., stable organizations). The term *large project* was substituted by the "associated programming project," thus discarding the notion of large programs and adapting the law to the SPE scheme, introduced soon after this law was formulated.[4]

The fifth law had major changes made to it. The *safe growth rate* notion of the 1970s was discarded, and the law name was changed to "conservation of familiarity." In summary, it states that the activity rate of a project remains constant during its active life and is closely related to the fourth law (constant work rate). Lehman formulated this law after analyzing the amount of change between releases. If a release varied widely from a previous release, it was followed by releases with fewer changes, making the average amount of change constant over a large number of releases. The amount of change is also related to the familiarity of the stakeholders (not only users) with the system. With new releases, stakeholders need some effort to learn how the behavior of the system has changed. They need to recover their familiarity with the system, and for that they have to invest some time. Once they are familiar again with the system, the perceived complexity of the system will again be close to that of the previous release. Thus, the average perceived complexity over all the releases should be constant.

[4]The discussed modification in the law appeared for the first time toward the end of 1979 [Lehman 1979], some months before the SPE scheme was published [Lehman 1980].

There is no mention to the influence of the growth rate on the quality of the system, suggesting that Lehman decided to drop the idea of a *safe* release rate. A similar idea of a constrained growth rate was reintroduced in the '90s, as will be shown in Section 5.1.

4.3. First Validating Research

One of the first works to address the issue of the validity of the laws of software evolution using a comprehensive statistical approach was the PhD thesis by Pirzada [1988]. In his thesis, he studied the evolution of several *flavors* of UNIX. At the time of the study, the evolvability and maintainability of those UNIX versions was a subject of concern. Applying the approach suggested by Lehman and Belady [1985], he not only studied the evolution of UNIX but also evaluated the validity of the predictions of the laws.

The versions of UNIX under study were divided into three branches (or *streams*, in the terminology used by Pirzada): research, academic, and supported and commercial. The versions of UNIX in each *stream* were the following:

—**Research stream**
The original UNIX version created by Ken Thompson and Dennis Ritchie at Bell Labs. Pirzada studied nine versions of this flavor of UNIX, released between 1971 and 1987.
—**Academic stream**
Versions of UNIX developed by the Computer Systems Research Group (CSRG) at the University of California at Berkeley, the Berkeley Software Distribution (BSD). The last release considered by Pirzada, 4.3 BSD, appeared in 1986.
—**Supported and commercial stream**
Versions developed by the UNIX Support Group (USG), a small team created by the Switching Control Center System at Bell Systems, between 1983 and 1986.

The conclusions arrived at by Pirzada were as follows: All the streams verified the first law (continuing growth), but the situation was different for the others. Only the supported and commercial streams evolved according to the laws of software evolution. This stream showed a slowdown pattern, because of its increasing complexity. This is the typical pattern predicted by the laws of software evolution: If we measure size over time, the growth rate decreases over time. More interestingly, the academic and research streams were growing rapidly, and the academic stream was even accelerating (the growth rate increased over time). These growth patterns are not compatible with the laws of software evolution.

According to Pirzada, processes in pure commercial environments were more constrained, and therefore commercial software was much more likely to exhibit structural deterioration (second law). Only software developed in pure commercial environments evolved according to the formulation of the laws. This also has been verified more recently [Siebel et al. 2003] with a system that was developed in an academic environment and was subsequently adapted to a commercial environment. The diversity of the processes and the change in the evolution of the system affect the quality of products and processes.

The second law of software evolution (as formulated at the time, see Table III) states that continuous changes cause the structural deterioration of the system. This deterioration increases the complexity of future changes, and the net result is a pattern of decreasing growth. Pirzada found evidence conforming to this law only for the commercial stream, but the increase in complexity is validated only *after* the commercialization of the products. Pirzada concluded that commercial pressures enforce constraints to the growth of software and foster the deterioration of the system.

Law by law, supported and commercial UNIX conformed with the third, fourth, and fifth laws (as formulated at the time, see Section 4.2). The third law could not be validated for the academic and research streams.

Pirzada cited some earlier works highlighting the controversy regarding the universality of the laws. From all those works [Lawrence 1982; Chong Hok Yuen 1980; Benyon-Tinker 1979; Kitchenham 1982], we could only gain access to a paper by Lawrence [1982], which concluded that the third, fourth, and fifth laws were not confirmed in different case studies. However, Lawrence found that the first and second laws were validated.

5. THE LAWS IN THE 1990S

In 1989, the *Journal of Software: Evolution and Process*[5] published its first issue, which included an article by Lehman about the "Principle of Uncertainty" in software technology [Lehman 1989]. This principle of software uncertainty was already scattered across the text of the laws of software evolution in previous publications. Lehman distilled and condensed the principle, which became another essential property of software evolution. The principle, in Lehman's own words [Aspray 1993], states:

> No *E*-type program can ever be relied upon to be correct.

In other words, *E*-type software is never finished and keeps evolving in order to (i) fix defects introduced in the previous programming activities and (ii) take into account new demands from users.

In the case of the theory of software evolution, the principle of software uncertainty refers not only to defects but also to change requests in general, which can be also about missing or desired functionality. Because *E*-type software is a model of the world, and the world is continuously changing, the only fate of software is to change or die. This is the main driving force explaining the laws of software evolution. In Lehman's opinion, the mission of software engineering is precisely to mitigate uncertainty through the introduction and control of processes, methods, and tools.

The principle of uncertainty was further developed during the following years [Lehman 1990, 1991] and finally included as part of a broader theory of software evolution in the following decade [Lehman and Ramil 2002b].

Software uncertainty led Lehman to develop additional laws of software evolution. In a footnote in one of his papers published during this decade [Lehman 1991], he discusses the relationship of uncertainty with the notion of *domain*. Lehman mentions the need for a new sixth law that would reflect the inevitability of software evolution because the scope of the system will irremediably increase over time. The increase of the scope is due to increasing user demands, which is, in other words, the formulation of the principle of software uncertainty. However, this law was not formulated in that publication. It appeared several years later, as we will discuss in the Section 5.1.

5.1. Dimensions of Software Evolution: Adapting and Expanding the Laws to a Changing Environment

In 1994, the invited keynote of the International Conference on Software Maintenance was entitled *Dimensions of Software Evolution* [Perry 1994] (republished as Perry [2006]). In his keynote, Perry argued that software evolution depended not only on the age, size, or stage of a project but also on the nature of its environment. This fact

[5]The name between 1989 and 2000 was *Journal of Software Maintenance: Research and Practice*, and between 2000 and 2011 it was *Journal of Software Maintenance and Evolution Research and Practice*. In 2011, it was merged with the journal *Software Process: Improvement and Practice*, and the name changed to its current form.

Table IV. Laws of Software Evolution in the 1990s. This Can Be Considered the Current Formulation of the Laws of Software Evolution.

I	*Law of Continuing Change* An *E*-type system must be continually adapted, or else it becomes progressively less satisfactory in use.
II	*Law of Increasing Complexity* As an *E*-type is changed, its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.
III	*Law of Self Regulation* Global *E*-type system evolution is feedback regulated.
IV	*Law of Conservation of Organizational Stability* The work rate of an organization evolving an *E*-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime.
V	*Law of Conservation of Familiarity* In general, the incremental growth (growth rate trend) of *E*-type systems is constrained by the need to maintain familiarity.
VI	*Law of Continuing Growth* The functional capability of *E*-type systems must be continually enhanced to maintain user satisfaction over system lifetime.
VII	*Law of Declining Quality* Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an *E*-type system will appear to be declining.
VIII	*Law of Feedback System* *E*-type evolution processes are multilevel, multiloop, multiagent feedback systems.

is interesting, because all the research on the validity of the laws had focused on the nature of the environment where each study took place (e.g., the different streams used by Pirzada [1988]).

These works led to the reformulation of the laws of software evolution in 1996 [Lehman 1996b]. Lehman added three new laws and adapted the rest, as shown in Table IV. This was the last reformulation of the laws.

Lehman et al. [1997] found empirical support for all the reformulated laws except the fifth (conservation of familiarity) and the seventh (declining quality). Some other studies found similar empirical support as well [Gall et al. 1997].

The changes in the laws were caused by results published in previous decades, rather than by studies performed during those years. One of the main changes consisted of the inclusion of the term "*E*-type," to highlight that the laws are only valid for that kind of software. The **third law** changed its name again, now to *Law of Self Regulation*. It was also briefer, just stating that software evolution is feedback regulated. There is an interesting change in the **fifth law**, *conservation of familiarity*, now highlighting that the growth rate is constrained by the need to maintain familiarity. This is similar to the *safe growth rate* of this law during the 1970s: any project has a limit in how fast it can grow without suffering from problems due to the growth rate.

The **sixth law**, derived from the *Principle of Software Uncertainty*, was called "*of continuing growth*." It was already mentioned in 1991 [Lehman 1991], although it was first formulated in 1996. It states that not only will software undergo changes but also that some of those changes will be performed to increase the functionality of the program. Lehman attributed this to the need for increasing functionality due to missing requirements in the original specification of the system, and to feedback-generated demand, rather than to defects.

The **seventh law**, of *declining quality*, states that unless work is carried out to avoid it, quality (and hence user satisfaction) will decline as time passes. We can consider the seventh law a corollary of the others, as the decline of quality is due to the degradation of the system under continuous changes.

The **eighth law** was derived by Lehman after observing the organizations that produced the software that he used as case studies for his empirical work. These projects often involved several managerial levels: they were multilevel, multiloop, and multiagent feedback systems. This, and the apparent conservation of some quantities (like the fourth and fifth laws show), led to the formulation.

Feedback in the software evolution process became the cornerstone of Lehman's work during the '90s, with the *Feedback, Evolution and Software Technology* (FEAST) research project, focused on the so-called *FEAST hypothesis* [Lehman 1996a]. This hypothesis states that software evolution is a complex feedback learning system, which is hard to plan, control, and improve.

5.2. The FEAST Project

The software evolution field was focused on the validity of the laws during the 1990s, with the FEAST project being one of the main sources of empirical works during that time. This effort resulted in the validation of the laws both by simulation [Chatters et al. 2000; Wernick and Lehman 1999] and by empirical methods [Lehman et al. 1997]. In the scope of the FEAST project, Turski produced the *reference model for smooth growth of software systems* [Turski 1996], which was generalized some years later [Turski 2002]. The FEAST project led to the recognition of software evolution as a subject of research on its own. It also started to systematize and formalize the field.

The 1990s also saw the birth of the *International Workshop on Principles of Software Evolution*. In the first edition, Lehman and Wernick [1998] presented a simulation model of the impact of feedback on software evolution with some of the results obtained in the FEAST project.

Simulation was in fact one of the main techniques used in FEAST to validate the laws, finding confirmation for seven of them. Only the second law, of increasing complexity, could not be confirmed due to the lack of empirical data [Lehman and Ramil 1999]. The project also found empirical evidence confirming the same seven laws, showing that Turski's model [Turski 1996] could be accurately fitted to empirical data extracted from industrial projects [Lehman et al. 1998b].

The results of the FEAST project were summarized in a paper by Lehman and Ramil that appeared in 2002 [Lehman and Ramil 2002a]. Lehman also published another good summary of the field during the 1990s, with a focus on the implications for the software industry [Lehman 1998].

But besides those carried out at FEAST, the field still lacked further empirical studies [Kemerer and Slaughter 1999]. Above all, more consistent studies were needed that could allow for the meta-analysis of results [Bennett et al. 1999].

6. SOFTWARE EVOLUTION IN THE 2000S

In the decade of the 2000s the field reached its maturity with the publication of two books [Madhavji et al. 2006; Mens and Demeyer 2008].

Since its inception in the late 1960s and early 1970s of the previous century, the field of software evolution kept evolving, adapting the laws or stating new ones (see Figure 1). But the laws of software evolution have remained invariant since 1996. The only publication written by Lehman that explicitly formulates the laws in the last decade is a chapter [Lehman and Fernández-Ramil 2006] in the book published in 2006 [Madhavji et al. 2006].[6] That book included a slight clarification of the SPE scheme

[6]In that book, and in a chapter in another software evolution book [Mens and Demeyer 2008], Ramil signs his works as Juan Fernández-Ramil, while in all his earlier works he appears as Juan F. Ramil. Although obviously both names correspond to the same author, we have cited his works using the name as it appears in each paper.

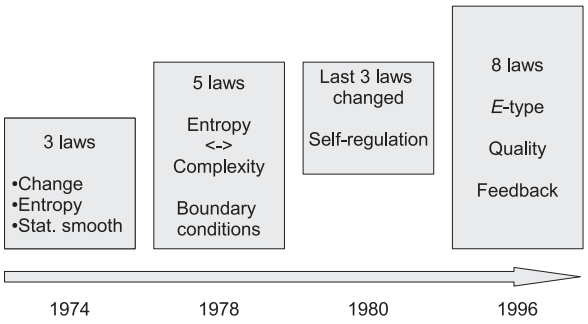

Fig. 1. Diagram of the evolution of the laws of software evolution.

[Cook et al. 2006a] (also published as Cook et al. [2006b]), but the modification did not impact the formulation of the laws.

This decade also saw the publication of some of the main results of the FEAST project. One of them was a quantitative model for the simulation of the software process [Ramil et al. 2000]. Simulation could be applied to manage long-term software evolution and control complexity in the software process [Kahen et al. 2001; Lehman et al. 2002]. These results led to a paper proposing a road map for a theory of software evolution [Lehman and Ramil 2001a].

The FEAST project also produced a guide to control, plan, and manage the software process [Lehman and Ramil 2001b]. The guide addressed some of the criticisms about the lack of formalization and absence of precise definitions of the laws of software evolution.

In fact, this lack of formalization and precise definitions are two of the causes that explain the proliferation of studies about the validity of the laws. The FEAST project mentioned some of these validity studies, with conflicting results: First, the case of the OS 360/370 defense system (mentioned in Lehman and Ramil [2002a]), and second, the Linux kernel, by Godfrey and Tu [2000]. The latter, which found some deviations from the laws, was labeled as an *anomaly* by Lehman et al. [2001].

The availability of data thanks to the open nature of libre software development, and the attention given by Lehman to the case of Linux and to the question of the validity of the laws, fostered a number of empirical studies of libre software evolution during this decade, leading to a growth by an order of magnitude in the number of works in the field (see Figure 2 and Section A.3 in the Appendix).

The popularity of the studies of the validity of the laws in libre software finally allowed for meta-studies comparing results for different projects and with different research methodologies [Fernández-Ramil et al. 2008].

6.1. Are the Laws Valid for Libre Software?

The first studies trying to verify the laws of software evolution for libre software were based on a relatively small set of case studies. Initially, Godfrey and Tu studied only the case of the Linux kernel [Godfrey and Tu 2000, 2001], finding that it was evolving at an accelerating pace, contrary to the predictions of the laws. These results were confirmed by Robles et al. [2005] 5 years later, extending the results to a set of 19 projects.

More recently, Israeli and Feitelson [2010] also studied the Linux kernel, trying to find out whether it fulfilled the laws or not. They found that the superlinear growth pattern that had been found in 2000 by Godfrey and Tu, and confirmed in 2005 by

Robles et al., stopped with release 2.5 of Linux. From that release on, growth has been linear.

This change in the growth pattern coincides in time with the changes in the release policies of Linux. Up to the 2.5 release, Linux development occurred in two parallel branches: stable and unstable. New functionality was added to the unstable branch, which eventually became the stable branch. The stable branch only accepted bug fixes, and its growth pattern was completely flat (i.e., zero growth). However, with the 2.6 release, the procedure changed, and both stable and unstable development occurs in the same branch.

Contrary to the previous studies, Israeli and Feitelson concluded that Linux confirmed most of the laws. In particular, it fulfills those related to the growth and the stability of the process. The laws of increasing complexity, conservation of familiarity, and declining quality were contradicted though:

—The second law (increasing complexity)
The average cyclomatic complexity is decreasing in Linux. This is due to the high rate of aggregation of small functions that dilute the average value of the complexity.
—The fifth law (conservation of familiarity)
This law is only validated for minor releases. Major releases lead to a sudden change in the familiarity with the system, because they introduce severe changes.
—The seventh law (declining quality)
The quality of the system is increasing over time, as measured by the maintainability index.

Moreover, some of the laws are only partially validated:

—The third law (self-regulation) was validated using indirect empirical support, analyzing the shape of the incremental change in files over time. This plot oscillates around an average value, in what seems to be a signal of self-regulation.
—The fifth law (conservation of familiarity) is only valid for minor releases. From time to time, Linux creates major releases with substantial changes. This leads to discontinuous familiarity: It is only conserved through the minor releases between major releases.
—The eighth law (feedback system) was not empirically validated, but it was assumed to be true considering how the Linux project works (i.e., driven by a community of users and developers). However, Linus Torvalds, the leader of the project, has a strong influence in the project, so it is not clear whether the project is completely feedback driven.

The main conclusion of Israeli and Feitelson's study is that the *perpetual development* model[7] of libre software is compatible with the laws of software evolution. In a typical textbook software life cycle model, there is a clear division in two main stages: development until first release, and maintenance thereafter. This is not the usual pattern found in Linux, which evolves continuously with frequent releases. The system is developed in collaboration with users, with the community driving the evolution of the project. This model fits the feedback-driven paradigm, and so authors conclude that the perpetual development model is a good description of the life cycle of *E*-type software.

Koch [2005, 2007] addressed the validity question with a large sample of software projects. He extracted a sample of 8,621 projects from `SourceForge.net`, a hosting

[7]The *perpetual development* model is a life cycle model for *E*-type software where most of the changes occur in a continuous manner. When new features accumulate, the project is frozen to prepare for a new major release. After its publication, development continues in the project toward the next major release. However, bug fixes are still accepted for the past releases, which are distributed in the form of new minor releases.

site for libre software projects. His methodology is similar to that found in the works of Godfrey and Tu, and Robles et al. Koch measured the size in lines of code for all the projects and fitted a regression model to each curve. He found most of the projects to grow either linearly or with a decreasing rate, as predicted by the laws. However, around 40% of the projects showed a superlinear pattern, incompatible with the laws. Those projects were usually large, in terms of code size, with a high number of participants and a high inequality in the number of contributions [Mockus et al. 2002]. Koch speculated that the cause of the superlinear growth might be a certain organizational model, which he calls the *chief programmer team*, a term originated in IBM in the 1970s [Baker 1972]. This organizational model seems to affect the growth rate, allowing for these patterns of increasing growth.

Therefore, at first glance, the studies by Koch showed that a majority of projects evolve according to the behavior predicted by the laws. However, `SourceForge.net` is known to contain many small and pet projects, many of which are abandoned [Rainer and Gale 2005]. That means that, for the kind of regression analysis used by Koch, projects that stopped evolving would appear as conforming to the laws of software evolution. But nonevolving projects cannot be considered *E*-type software. In addition, Koch reports that 67.5% of the sample are projects with only one developer, which would make them fall under the category of pet projects. If we focus on projects that fit the perpetual development model described in Israeli and Feitelson [2010], we can only consider the results obtained for large projects, with several developers and a large user base. That implies that the consequences of Koch's study are that most of the permanent-evolving projects grow with accelerating paces, without constraints, which is contrary to the predictions of the laws of software evolution.

A more recent study explores different effort models for libre software development, comparing them with effort models used traditionally in closed-source software [Fernández-Ramil et al. 2009]. One of the conclusions of the study is that complexity does not slow down the growth of the analyzed libre software projects, contrary to what is stated by the laws. The authors suggest that libre software is *more effective* than closed source, and therefore less effort is required to develop projects of similar sizes.

In 2008, Fernández-Ramil et al. [2008] reviewed and summarized the main empirical studies about the evolution of libre software and how they helped to confirm (or not) the laws of software evolution. In their opinion, libre software development follows a process more chaotic than in-house development, which may be an explanation for the divergence.

Some libre software projects evidence discontinuities in their evolution. Similar patterns were found as well in the 1990s in the FEAST project, this being the reason the fourth law included the clarification "or phases of that lifetime" (see Table IV). A software project goes through different stages during its lifetime. There are different models for the stages of a software project. The simplest one includes two stages: development and maintenance. Other models divide the software lifetime into five stages [Rajlich and Bennett 2000]. In any case, studies reviewed in Fernández-Ramil et al. [2008] did not take into account this possibility. Discontinuity means that growth rates may suddenly change when there is a phase transition, and therefore growth rates measured with low-level parameters (like lines of code) cannot be trusted to judge the validity of the laws. Interestingly, the complete study about Linux by Israeli and Feitelson [2010] found that the growth of Linux changed from superlinear to linear when the project entered a new phase, that is, the change from having unstable and stable branches in parallel to a single development branch.

Another source of controversy in the validation of the laws has been their lack of formalization. According to Fernández-Ramil et al., each law can be formalized in more than one way. In the original empirical works that led to the formulation of

the laws, the studied aspects and properties were of a high-level nature; that is, they were defined at a high level of abstraction. However, the empirical studies about libre software use lower-level parameters, which suffer more variability and are harder to predict.

After analyzing the common and diverging points of the reviewed papers (some of them already mentioned in this article [Godfrey and Tu 2000; Robles et al. 2005; Herraiz et al. 2006]), Fernández-Ramil et al. [2008] summarized the state of the validation question for each one of the laws:

—Continuing change (I)
This law applies well to libre software, because successful projects are continuously developed over time. However, in some occasions even successful projects can experience periods without activity.
—Continuing growth (VI)
Although some projects may experience periods with flat growth, successful libre software projects continuously increase their functionality, and therefore grow continuously.
—Declining quality (VII)
In this case, the confirmation is difficult to test in practice, because it depends on how quality is measured. However, the number of defects found in libre software projects tends to grow, meaning that quality might decline over time, and therefore this law is possibly confirmed.
—Feedback system (VIII)
Libre software projects are feedback driven, as predicted by the laws, but the feedback process is more chaotic than in the case of in-house software development. This is probably because libre software projects are open systems. This means the development team is far from constant over time, with contributors joining and leaving, and with code being sometimes duplicated or imported from other projects.

The rest of the laws could not be confirmed with the studies included in their review [Fernández-Ramil et al. 2008]. However, this does not mean that the laws have been invalidated, and the authors recommend further work in the matter.

6.2. Status of the Validity Question for Libre Software

Table V shows some selected references that have attempted to validate the laws of software evolution for the case of libre software.

Bauer and Pizka [2003] confirmed the validity of all the laws using a sample of selected libre software projects. Their review is qualitative: They focus on the software process of the selected projects, on their high-level architecture, and on the involvement of companies in the projects. They conclude that all the laws are verified in all the case studies. However, there is no further empirical evaluation.

The next study [Herraiz et al. 2006] is similar to the study by Robles et al. [2005], although it only measured size at the overall level, using different metrics. The same regression approach was used, but this time also fitting a quadratic model. If the quadratic correlation coefficient was positive, the growth was assumed to be superlinear, sublinear if it was negative, and linear if it was close to zero. Some cases presented a superlinear pattern, as in the three previous studies. Some others were either linear or sublinear. Therefore, the invalidation or validation of the second and fourth laws were only partial.

In a study of 705 releases of nine open-source software projects, Neamtiu et al. [2013] reported that only the laws of continuing change and continuing growth are confirmed for all programs.

Table V. Main Studies about the Validity of the Laws for Libre Software
The laws marked with "(p.)" are only partially validated.

Ref.	#	Approach	Validated laws	Invalidated laws
[Godfrey and Tu 2000]	1 (Linux)	Size metrics (overall, submodules). Growth rate.	I, VI	II, III, IV, V
[Godfrey and Tu 2001]	4	Size metrics (overall, submodules). Growth rate.	I, VI	II, III, IV, V
[Bauer and Pizka 2003]	8	Qualitative review of some selected projects.	I–VII (all laws)	None
[Robles et al. 2005]	19	Size metrics (overall, submodules). Growth rate.	I, VI	II, III, IV, V
[Herraiz et al. 2006]	13	Size metrics (only overall, different metrics, quadratic regression model). Growth rate.	I, VI	II, III, IV, V
[Koch 2005] & [Koch 2007]	8621	Size metrics (overall level, quadratic regression model). Growth rate.	I, VI	II, IV (p.).
[Israeli and Feitelson 2010]	1 (Linux)	Deep analysis (submodules). Size, complexity, quality metrics.	I, III (p.), IV, V (p.), VI, VIII (p.)	II, V (p.), VII
[Neamtiu et al. 2013]	40	Size and defects density. Overall level.	I, VI	II, III, IV, VII, VIII
[Vasa 2010]	40	Size, object-oriented, and complexity metrics. Overall level.	I, III, V, VI	II, IV, VII

Vasa [2010] studied 40 large open source systems, finding support for Laws I (continuing change), III (self-regulation), V (conservation of familiarity), and VI (continuing growth). His methodology was based on size measurements and object-oriented and complexity metrics.

If we examine Table V, it seems clear that Laws I (continuous change) and VI (continuing growth) have been validated by all the studies. Laws II (increasing complexity), IV (conservation of organizational stability), and V (conservation of familiarity) have been invalidated for most of the cases. These three laws are related to the growth pattern of the project. Software projects whose growth accelerates over time do not fulfill these three laws.

All the mentioned studies are based either in qualitative considerations or in empirical evaluation of size and complexity over time. However, Fernandez-Ramil et al. [2008] think that the laws should be validated through simulation as well as through empirical studies. The laws are coupled, so the effects of one law are not independent of the effects of the rest of laws, but empirical studies cannot isolate each law to verify it. Simulation can take into account this coupling between the laws. Simulation was already attempted in the 1990s [Chatters et al. 2000; Wernick and Lehman 1999] in the FEAST project. More recently, Smith et al. [2005] have simulated the evolution of libre software. According to them, the laws of software evolution are a natural language expression of empirical data and can be mapped to many different lower-level empirical scenarios. The diversity of scenarios is the root of the discrepancy between different empirical studies.

6.3. The Debate on Metrics in Confirming Studies

The study by Israeli and Feitelson is the first one providing an extensive quantification of the laws of software evolution. The initial studies in the decade of the 2000s [Godfrey and Tu 2000, 2001; Robles et al. 2005] measured only size in lines of code. Israeli and Feitelson studied the size of Linux using lines of code, number of system calls, number of configuration options, and number of functions. They also measured some other

complexity metrics and the maintainability index for estimating the quality of the system [Coleman et al. 1994].

The formulation of the laws does not include any mention of how to quantify them, even though Lehman proposed the laws based on empirical results that used a particular set of metrics, and therefore they had an implicit quantification. This lack of an explicit quantification of the laws is the source of the *anomaly* incident described at the beginning of this section. Lehman questioned the studies on Linux because they were using different metrics than the original software evolution studies. However, neither the laws themselves nor any publication on Lehman's theory of software evolution specifies metrics, measurements, or processes to evaluate or verify the laws.

In the early studies on software evolution, Lehman, Belady, and others did not have direct access to the source code and the change records. They could gain access only to some datasets provided by companies, owners of the systems, with a small set of metrics.

Having only a single data point per release, Lehman decided to use releases as a pseudo-unit of time, formalized as *Release Sequence Number* (RSN). He decided to use the number of modules as the base unit for size as well.

These units have persisted in later works by Lehman and others [Lehman 1996b; Lehman et al. 1997; Ramil and Lehman 2000; Lehman et al. 2001], although some of the simulations in the FEAST project used calendar time rather than RSN [Chatters et al. 2000; Wernick and Lehman 1999]. Other studies have used Source Lines of Code (SLOC) as size metric [Godfrey and Tu 2001; Robles et al. 2005]. Therefore, the metrics used in some of the nonconfirming studies were different from those of the original studies.

These differences were the base of Lehman's argument [2001], when he labeled the case of Linux [Godfrey and Tu 2000] as an anomaly, raising concerns about the comparability of studies because they were using different metrics. In a previous work, Lehman et al. [1998a] had referred to the suitability of lines of code as a size metric for evolution studies:

> A lower level metric, lines of code (locs, or equivalently klocs) much beloved by industry, does not have semantic integrity in the context of system functionality. Moreover, when programming style directives are laid down in an organisation, the number of locs produced to implement a given function are very much a matter of individual programmer taste and habit.

Some studies indicate that the conflicting studies questioning the validity of the laws of software evolution are methodologically comparable to the original studies by Lehman, regardless of the metrics chosen. Nonetheless, both size metrics have been shown to be comparable in several studies, including some cases with very large datasets containing thousands of software systems [Herraiz et al. 2006, 2007a; Herraiz 2009]. The topic of time units in software evolution studies has attracted less research than the case of size metrics. To the extent of our knowledge, only two studies recommend the use of calendar time rather than RSN when calendar time data are available [Barry et al. 2007; Vasa 2010].

7. WHAT HAS THE RESEARCH COMMUNITY FOUND ABOUT THE LAWS OF SOFTWARE EVOLUTION?

7.1. Research Question 1: Why did the laws of software evolution undergo so many modifications during their early life, but have not changed in the last 15 years?

The last modification of the laws of software evolution dates from 1996 [Lehman 1996b], while the main works about the validity of the laws were published later

(see Section 6.2). There were some early empirical studies about the validity though [Lawrence 1982; Pirzada 1988].

However, Lehman did not change the laws because of the early empirical findings. He adapted the laws to the new development practices that evolved during the 1980s and the 1990s. He also took into account new concepts, such as feedback, to adapt the formulation of the laws [Lehman and Ramil 2003].

To address the problem of different software development, maintenance, and releasing practices, Lehman coined the SPE scheme, modifying the laws, and making it explicit that they are only valid for a particular type of software, that is, *E*-type.

Perry proposed the notion of domains in software evolution [Perry 1994]. Evolutionary behavior could differ for software in different domains. The need for domains is very similar to the reasons that led Lehman to propose the SPE classification scheme: not all software evolves in the same way. As a matter of fact, the SPE scheme was recently updated [Cook et al. 2006b] to clarify the differences between software evolution categories (*domains*).

The case of libre software is a paradigmatic example of a new domain that has been ignored in the definition of the laws and the SPE scheme. Prior to the appearance of the Internet, no software was developed by teams distributed across the globe without regularly meeting in person, and communicating just through electronic channels. This environment is very different from the software engineering practices common during the times of the first versions of the laws.

However, although Lehman updated the SPE classification scheme [Cook et al. 2006b], he did not incorporate the notion of domain into the laws or try to adapt them to different software engineering practices. The laws kept exclusively referring to *E*-type software. Thus, as a summary, we can conclude that:

—During the first decades, the laws did not evolve as an answer to new empirical findings questioning their validity.
—The main modifications in the laws were due to changes in software development and maintenance practices and standards. The only source of modifications are publications from Lehman and collaborators. Proposals of modifications by other authors (such as Pirzada [1988]) were explicitly rejected [Lehman and Ramil 2003].
—In modern publications by Lehman and collaborators, they have not taken into account recent studies about the validity for libre software. The most recent publication about the laws [Lehman and Fernández-Ramil 2006] still contains the same formulation that was published 10 years before [Lehman 1996b].

7.2. Research Question 2: Which laws have been confirmed in the research literature? Which laws have been invalidated?

In the beginnings of the software evolution research field, the laws could only be verified with very few case studies. Lehman extracted his conclusions mainly from the IBM OS/360 operating system project [Lehman 1980]. In addition to Lehman's empirical findings, the laws were also validated using mathematical models [Woodside 1980].

When the laws started to confront new empirical studies, the conclusions were that not all the laws were universally valid. The first to raise the issue of validity was Lawrence [1982]. For the laws, as formulated at the time (see Tables II and III), his data could only validate Laws I (continuing change) and II (increasing complexity). Laws III (statistically smooth evolution), IV (invariant work rate), and V (conservation of familiarity) could not be validated.

Similar results were obtained by Pirzada [1988]. He validated all the laws, but only for software that underwent a commercialization process. This kind of software evolves

as predicted by the laws after it has been marketed. However, software developed in research and academic communities was evolving in a different way, and the laws could not explain its evolution.

During the '90s, the results were promising. After the last version of the laws was published (see Table IV), Lehman et al. [1997] found empirical validation of all of them except V (conservation of familiarity) and VII (declining quality). These two laws could not be validated because of the lack of empirical data. However, this does not mean that those laws were invalidated, either. Other works could validate all the laws using simulation techniques [Chatters et al. 2000; Wernick and Lehman 1999]. Also, Turski's model [Turski 1996] was validated using industrial software as case study [Lehman et al. 1998b].

The case of libre software, already discussed in detail in Section 6, supports the validity of Laws I (continuing change) and VI (continuous growth). It also shows that the second law (increasing complexity) is not valid. For other laws, the results are not consistent. A recent meta-study by Fernández-Ramil et al. [2008] agrees with with these results. Laws I and VI are valid, while Laws II to V could not be validated using empirical studies.

In summary, Lehman observed in the 1960s and 1970s that software needed to be changed and updated (continuous growth). These observations have remained valid over the years. Therefore, Laws I and VI seem to remain valid. The rest of the laws cannot be verified, although it is not clear that they are invalid, either. In any case, it is interesting to highlight the case of Law II (increasing complexity), which has been invalidated in many libre software empirical studies. According to Lehman, software changes degrade the structure of the system. This degradation increases the complexity of further changes. This is similar to Parnas' view of code decay and software aging [Parnas 1994]. However, empirical data show that further changes are not more difficult, and thus, the complexity of software does not seem to increase with continuous changes.

7.3. Research Question 3: Which strategies, data, and techniques can be used to validate the laws of software evolution?

The laws of software evolution have been studied at different levels of granularity (micro, macro) and using two main techniques: simulation and empirical studies.

The original works by Lehman were mainly empirical studies at a macro level [Lehman and Belady 1985]. Several works claiming the invalidity of the laws were also empirical studies at a macro level [Pirzada 1988; Godfrey and Tu 2000].

These invalidating studies measured size over time and found that the curve of size over time was growing with superlinear profiles [Godfrey and Tu 2000; Robles et al. 2005]. That is, growth was accelerating. As it was said, this behavior is incompatible with the laws of software evolution. The most notable case of this kind of growth was the Linux kernel. However, when this very same case was analyzed in detail at a micro level, the superlinear growth curve was found to be a set of linear segments [Israeli and Feitelson 2010]. The aggregation of these segments produced a superlinear growth curve. However, individual modules were not growing with acceleration. The results of this micro-level study about Linux [Israeli and Feitelson 2010] were in conflict with the previous macro studies.

This difference among different levels of granularity was also observed by Gall et al. [1997], who were the first ones to suggest the study of software evolution at different levels of granularity. These differences in the level of the study are known to have caused similar discrepancies in other research areas and in some works in empirical software engineering [Posnett et al. 2011].

Another dimension of the validation of the laws is the approach followed for the validation process. We have presented two main classes of studies: simulation and empirical studies.

The first simulation work can be dated back to 1980, when Woodside published the first model based on the laws of software evolution [Woodside 1980] (later republished as Woodside [1985]). More recently, the model by Turski [1996] was used to validate the laws using simulation [Lehman et al. 1998b]. In general, the FEAST project was the main source of simulation studies [Chatters et al. 2000; Lehman and Wernick 1998; Wernick and Lehman 1999]. Most of these simulation studies have been done at a macro level, considering overall properties of the software projects over time. However, some models are also of a micro nature, studying the software development and maintenance processes [Smith et al. 2005].

The laws of software evolution are tightly coupled. It is difficult to isolate the effects of one law while discarding the rest. This makes it more difficult to validate the laws using empirical studies. In fact, all empirical studies reviewed here tried to validate (or invalidate) the laws one by one, without considering any interaction between them. However, with simulation studies, it is possible to study how the laws change all at the same time and to take into account the influence of one law over the others.

On the other hand, the advantage of empirical studies is the scale of the study. It is possible to study large amounts of software projects using an empirical approach (e.g., Koch studied more than 8,500 software projects [Koch 2007]). To our knowledge, there are no large-scale simulation studies.

Summarizing, what we have found is that:

—The validation process is done through simulation and empirical studies.
—Empirical studies yield different (and conflicting) results if they analyze evolution at different levels of granularity (using the same case studies).
—Simulation studies are usually of small scale; that is, they are empirically validated with only a few software projects. There is a lack of large-scale simulation studies.
—Empirical studies can be of large scale, but at a macro level. There are no micro-level large-scale empirical studies.

7.4. Research Question 4: Which kind of software projects, and under what conditions, fulfill the laws of software evolution?

Initially, Lehman [1974] conceived the laws of software evolution as applicable to large software systems. Those systems are developed by teams organized in several managerial levels, and they have a large user base, which provides feedback about the quality of the system and demands new features.

Lehman [1980] later improved the definition of "large software system," proposing the SPE scheme. This scheme classifies software in three categories. Only one of those categories, *E*-type, is described by the laws.

Other empirical studies found that software evolution is different for different kinds of software projects [Pirzada 1988]. In particular, only software that undergoes a commercialization process was found to follow the laws.

Koch [2007] found differences in the evolution of libre software projects of different sizes. Small libre software projects that are developed by a single person or a small group of people fulfill the laws. However, large software projects do not follow them. These projects have a large number of participants and a high asymmetry in the distribution of work among participants.

Other studies have analyzed the quality of the evolution of different kinds of software projects [Mockus et al. 2002] but not explicitly addressed the applicability of the laws.

The main conclusion of Mockus et al. [2002] is that there are significant differences in the quality and evolution of libre and nonlibre software.

To summarize, we can only conclude that software projects of different kinds evolve differently. Despite the intense research, especially in the case of libre software in recent years, we have not found any other concluding remark.

8. WHAT'S NEXT? CHALLENGES AND DIRECTIONS FOR SOFTWARE EVOLUTION RESEARCH

In his 1974 lecture, Lehman suggested the study of software evolution with an empirical approach based on statistical methodologies [Lehman 1974]. At that time, the approaches suggested by Lehman were very difficult to implement in practice. However, the current research environment makes that much easier. Based on the findings of the previous sections and on over 40 years of software evolution research literature, we suggest some challenges and directions for the next years of software evolution, which would help to achieve Lehman's original goal.

8.1. The Importance of Replication for Empirical Studies of Software Evolution

Public availability of the data used for empirical studies is crucial. A theory of software evolution must be based on empirical results, verifiable and repeatable, and made on a large scale, so that conclusions with statistical significance can be achieved [Sjøberg et al. 2008]. If software evolution is analyzed with data that is not available to third parties, it cannot be verified, repeated, and replicated. It is dangerous to build a theory on top of empirical studies that do not fulfill those requirements.

Empirical studies of software evolution should conform to the guidelines suggested for empirical software engineering [Kitchenham et al. 2002, 2008]. The entry barrier for repeatable empirical studies can be lowered using reusable datasets such as FLOSSMetrics [Herraiz et al. 2009], FLOSSMole [Howison et al. 2006], the Qualitas Corpus [Tempero et al. 2010], or the UCI Sourcerer Dataset [Lopes et al. 2010]. These datasets aim to achieve replicability of empirical studies of software development and evolution. Replicability is a concern recently raised in the empirical software engineering research community [González-Barahona and Robles 2012], with many authors highlighting its potential benefits [Robles and German 2010; Shull et al. 2008; Brooks et al. 2008; Barr et al. 2010].

However, Kitchenham [2008] warns about some of the problems that may arise with replication and sharing, in particular when reusing the so-called *laboratory packages*. If these packages are only gathered once and reused many times, that would amplify the probable errors that occur during the gathering phase. The availability of datasets does not remove the need for new datasets that can be used to test the correctness of previous results. In any case, this does not invalidate the previous argument: Software evolution studies must be replicable, either by reusing third-party datasets and tools or by making their data and/or tools publicly available.

8.2. Statistical Methodologies for Software Evolution

In the 1970s, the methodology proposed by Lehman to study software evolution was primarily based on a statistical study of different product and process metrics. He explicitly suggested the use of regression techniques, autocorrelation plots, and time series analysis for the study of software evolution, based on the idea of feedback-driven evolution [Lehman 1974].

Although time series analysis has not been a popular technique in the research community, some initial results show that it is a promising technique that should be explored [Herraiz et al. 2007, 2008]. One possible cause that could explain the lack of research using this approach is that this kind of analysis requires historical

information. If we focus on the case of libre software, it is relatively easy to retrieve the source code releases of a project, or even of a large sample of projects, thanks to the availability of corpus such as *Qualitas* [Tempero et al. 2010] and *Sourcerer* [Lopes et al. 2010].

However, historical information other than source code is harder to obtain. For instance, version control systems are very heterogeneous, from a semantic point of view. It is difficult to compare historical information extracted from different kinds of version control repositories. For large samples of projects, the variability in the different repositories used by each project makes it difficult to compare and aggregate the information. These problems have been recently addressed by some research projects such as SQO-OSS [Gousios and Spinellis 2009], FLOSSMetrics [Herraiz et al. 2009], and FLOSSMole [Howison et al. 2006]. The tools and datasets published by these projects should lower the entry barrier to apply statistical techniques using richer and more complex historical information. A recent survey [Kagdi et al. 2007] reviewed the works that study software evolution from a mining software repositories perspective.

8.3. Quantification and Formalization of the Laws

The recent research activity around the validity of the laws for libre software is an example of the ambiguity of applying and measuring the laws of software evolution. According to Fernández-Ramil et al. [2008], the laws cannot be quantified in a unique manner. Each law accepts different mappings to lower-level metrics. In the opinion of Israeli and Feitelson [2010], this lack of formalization of the laws is the cause of discrepancies in the validation studies using Linux as a case study [Godfrey and Tu 2000; Robles et al. 2005].

The SPE scheme is insufficient for the formalization of the laws. The next version of the laws of software evolution should incorporate Perry's dimensions, or some other classification scheme adapted to the richer current software environment. It is doubtful that the laws can be universal, except perhaps for Laws I and VI. The rest of them should be specialized for different kinds of software and different software development and maintenance practices and standards.

Mens et al. [2005] also recommend a formalization of the theory of software evolution. An area that is lacking more contact with software evolution is formal methods, where specifications are typically not allowed to change. Mens et al. defend an integration of formal methods and software evolution research. This would foster software evolution and formal methods being better accepted by software developers, who deal on a daily basis with the software evolution phenomenon.

9. RECOMMENDED READINGS

Table VI shows a classification of a reduced selection of papers on software evolution, since the early papers by Lehman to the most recent empirical studies leading to future research. The theory of software evolution was mainly addressed by Lehman and close collaborators up to the 1990s. During the last years, several papers questioned the future of software evolution as a field, proposing challenges and directions for research [Mens et al. 2005; Godfrey and German 2008; Antoniol 2007].

The first paper questioning the validity of the laws for libre software [Godfrey and Tu 2000] has brought several other studies on the topic, with contradictory results [Robles et al. 2005; Koch 2007; Israeli and Feitelson 2010]. The lack of formalization of the laws and their ambiguity makes it harder to perform meta-analyses of these studies to extract common conclusions [Fernández-Ramil et al. 2008].

Simulation and modeling research of software evolution is another approach fostered by Lehman and collaborators. It was one of the main areas of the FEAST project in the 1990s. Although there are several publications on simulating software evolution

Table VI. Classification of the Main Research Literature on Software Evolution of the Last Decades

		<1980	1980-89	1990-99	>1999
Theory of software evolution	Lehman	[Belady and Lehman 1971] [Lehman 1974] [Lehman 1978]	[Lehman 1980] [Lehman 1989]	[Lehman 1996a] [Lehman 1996b] [Lehman and Ramil 1999]	[Cook et al. 2006a] [Lehman and Ramil 2001a]
	Others			[Perry 1994]	[Rajlich 2000] [Mens et al. 2005] [Antoniol 2007] [Godfrey and German 2008]
Validation or invalidation	Empirical	[Kitchenham 1982] [Lawrence 1982]	[Pirzada 1988]	[Gall et al. 1997] [Lehman et al. 1998b] [Bennett et al. 1999] [Kemerer and Slaughter 1999]	[Ramil et al. 2000] [Fernández-Ramil et al. 2008] [Godfrey and Tu 2000] [Robles et al. 2005] [Koch 2007] [Israeli and Feitelson 2010]
	Simulation	Belady and [Lehman 1976]	[Woodside 1980]	[Wernick and Lehman 1999] [Turski 1996]	[Chatters et al. 2000] [Lehman et al. 2001] [Kahen et al. 2001] [Lehman et al. 2002] [Smith et al. 2005]

[Kahen et al. 2001; Wernick and Lehman 1999; Chatters et al. 2000], the number of studies addressing the simulation of the evolution of libre software is so far insufficient [Smith et al. 2005], and to the extent of our knowledge no study has tried to validate (or invalidate) the laws of software evolution for the case of libre software using simulation.

Finally, as stated previously, the software evolution literature is vast. Table VI may help to focus on the main readings to those starting research in the field. These papers are important to understand the historical evolution of the laws and are must-reads for any researcher new to the field. In addition to the works already mentioned in the earlier paragraphs, Table VI also includes Lehman's seminal papers: the different works presenting the laws of software evolution [Lehman 1974, 1978, 1980, 1996b], the SPE classification scheme (and its modification [Lehman 1980; Cook et al. 2006b]), the principle of software uncertainty [Lehman 1989], and the notion of feedback-driven software evolution [Lehman 1996a].

10. CONCLUSIONS

Software engineering still lacks a completely solid scientific basis. The laws of software evolution were one of the first approaches to define a theory of software systems that provides such a scientific basis. However, they have faced many controversial analyses since their inception, and now it seems clear that their validity, although proven in many cases, is not universal.

The laws were designed with *change* in mind. Software cannot be immutable (or at least, *E*-type software), and the laws have been shown to need changes themselves as time passed. Their formulation had to be adapted several times since they were first defined, to take into account new software development and maintenance practices and standards, so that they did not become obsolete. However, these changes to the laws are not enough. The overall software evolution theory must be adapted to face another fundamental principle: Not only is software not immutable, but also the way it is produced is not immutable either.

With respect to how the validation studies are conducted, the original statistical approach suggested by Lehman is now becoming possible because of the myriad publicly available software repositories. Precisely because of this, the laws have to confront more studies, some of them questioning their validity.

However, there is one important aspect that current research is perhaps missing: a more general approach to invariants of software evolution. From this point of view, the important question is not whether the laws are universal, since their history has already shown how the laws themselves needed several changes. What is needed is a revamping of the laws, a quest for invariants in the evolution of software, a formal classification of invariants for the different kinds of software projects that can be currently found, and the expression of those invariants in precise ways that can be shown valid, or refuted, by empirical analysis.

Had Lehman started his research today, with the rich environment that we enjoy, he probably would not have tried to validate the old laws, but would have looked for invariant properties using new statistical and empirical approaches, classifying projects by applicability of the laws as he did with the SPE scheme, and obtaining models that would help in the development and maintenance of software. Sadly, Lehman is no longer among us to lead this research stream [Canfora et al. 2011]. The best tribute we can do to honor his work is to study modern programming processes as he did in the 1960s and 1970s to help developers, managers, and users in order to produce better software.

REFERENCES

Antoniol, G. 2007. Requiem for software evolution research: A few steps toward the creative age. In *Proceedings of the International Workshop on Principles of Software Evolution*. ACM, 1–3.

Aspray, W. 1993. Meir M. Lehman, Electrical Engineer, an oral history. IEEE History Center. Rutgers University, New Brunswick, NJ.

Baker, F. T. 1972. Chief programmer team management of production programming. *IBM Syst. J. 11,* 1, 56–73.

Barr, E., Bird, C., Hyatt, E., Menzies, T., and Robles, G. 2010. On the shoulders of giants. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER'10)*. ACM, New York, 23–28.

Barry, E., Kemerer, C., and Slaughter, S. 2007. How software process automation affects software evolution: A longitudinal empirical analysis. *J Software Maintenance Evol. Res. Pract. 19,* 1, 1–31.

Bauer, A. and Pizka, M. 2003. The contribution of free software to software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*. IEEE, Los Alamitos, CA

Belady, L. A. 1979. On software complexity. In *Proceedings of the Workshop on Quantitative Software Models for Reliability*. IEEE Computer Society, Los Alamitos, CA, 90–94.

Belady, L. A. 1985. On software complexity. In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady, Eds., Academic Press, San Diego, CA, 331–338.

Belady, L. A. and Lehman, M. M. 1971. Programming system dynamics or the metadynamics of systems in maintenance and growth. Res. rep. RC3546, IBM.

Belady, L. A. and Lehman, M. M. 1976. A model of large program development. *IBM Syst. J. 15,* 3, 225–252.

Bennett, K., Burd, E., Kemerer, C., Lehman, M. M., Lee, M., Madachy, R., Mair, C., Sjoberg, D., and Slaughter, S. 1999. Empirical studies of evolving systems. *Empirical Software Eng. 4,* 4, 370–380.

Benyon-Tinker, G. 1979. Complexity measures in an evolving large system. In *Proceedings of the Workshop on Quantitative Software Models for Reliability*. IEEE Computer Society, Washington, DC, 117–127.

Brooks, A., Roper, M., Wood, M., Daly, J., and Miller, J. 2008. Replication's role in software engineering. In *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjberg, Eds., Springer, London, 365–379.

Brooks, F. P. 1978. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman, Boston, MA.

Canfora, G., Dalcher, D., Raffo, D., Basili, V. R., Fernández-Ramil, J., Rajlich, V., Bennett, K., Burd, L., et al. 2011. In memory of Manny Lehman, "father of software evolution." *J. Software Maintenance Evol. Res. Pract. 23,* 3, 137–144.

CHATTERS, B. W., LEHMAN, M. M., RAMIL, J. F., AND WERNICK, P. 2000. Modelling a software evolution process: A long-term case study. *Software Process: Improv. Pract. 5,* 2–3, 91–102.

CHONG HOK YUEN, C. 1980. *A Phenomenology of Program Maintenance and Evolution.* Ph.D. thesis, Imperial College, London.

COLEMAN, D., ASH, D., LOWTHER, B., AND OMAN, P. 1994. Using metrics to evaluate software system maintainability. *IEEE Computer 27,* 8, 44–49.

COOK, S., HARRISON, R., LEHMAN, M. M., AND WERNICK, P. 2006a. Evolution in software systems: Foundations of the SPE classification. In *Software Evolution and Feedback. Theory and Practice*, N. H. Madhavji, J. Fernández-Ramil, and D. E. Perry, Eds., Wiley, 95–130.

COOK, S., HARRISON, R., LEHMAN, M. M., AND WERNICK, P. 2006b. Evolution in software systems: Foundations of the SPE classification scheme. *J. Software Maintenance Evol. Res. Pract. 18,* 1, 1–35.

EBSE. 2010. Template for a systematic literature review protocol. Retrieved November 13, 2013 from http://www.dur.ac.uk/ebse/resources/templates/SLRTemplate.pdf.

FERNÁNDEZ-RAMIL, J., IZQUIERDO-CORTAZAR, D., AND MENS, T. 2009. What does it take to develop a million lines of open source code? In *Open Source Ecosystems: Diverse Communities Interacting*, C. Boldyreff, K. Crowston, B. Lundell, and A. Wasserman, Eds., IFIP Advances in Information and Communication Technology Series, vol. 299. Springer Berlin Heidelberg, 170–184.

FERNÁNDEZ-RAMIL, J., LOZANO, A., WERMELINGER, M., AND CAPILUPPI, A. 2008. Empirical studies of open source evolution. In *Software Evolution*, T. Mens and S. Demeyer, Eds., Springer, 263–288.

GALL, H., JAZAYERI, M., KLÖSCH, R., AND TRAUSMUTH, G. 1997. Software evolution observations based on product release history. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 160–170.

GODFREY, M. W. AND GERMAN, D. M. 2008. The Past, Present, and Future of Software Evolution. In *Proceedings of the International Conference in Software Maintenance (ICSM) Frontiers of Software Maintenance*.

GODFREY, M. W. AND TU, Q. 2000. Evolution in open source software: A case study. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, 131–142.

GODFREY, M. W. AND TU, Q. 2001. Growth, evolution, and structural change in open source software. In *Proceedings of the International Workshop on Principles of Software Evolution*. 103–106.

GONZÁLEZ-BARAHONA, J. AND ROBLES, G. 2012. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Eng. 17,* 1, 75–89.

GOUSIOS, G. AND SPINELLIS, D. 2009. A platform for software engineering research. In *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories (MSR'09)*. 31–40.

HERRAIZ, I. 2009. A statistical examination of the evolution and properties of libre software. In *Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM'09)*. IEEE Computer Society, 439–442.

HERRAIZ, I., GONZALEZ-BARAHONA, J. M., AND ROBLES, G. 2007a. Towards a theoretical model for software growth. In *Proceedings of the International Workshop on Mining Software Repositories*. IEEE Computer Society, 21–30.

HERRAIZ, I., GONZALEZ-BARAHONA, J. M., AND ROBLES, G. 2008. Determinism and evolution. In *Proceedings of the International Working Conference on Mining Software Repositories*. ACM, Leipzig, Germany, 1–10.

HERRAIZ, I., GONZALEZ-BARAHONA, J. M., ROBLES, G., AND GERMAN, D. M. 2007b. On the prediction of the evolution of libre software projects. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 405–414.

HERRAIZ, I., IZQUIERDO-CORTAZAR, D., RIVAS-HERNANDEZ, F., GONZALEZ-BARAHONA, J. M., ROBLES, G., DUEÑAS-DOMINGUEZ, S., GARCIA-CAMPOS, C., GATO, J. F., AND TOVAR, L. 2009. FLOSSMetrics: Free/libre/open source software metrics. In *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*. IEEE Computer Society.

HERRAIZ, I., ROBLES, G., GONZALEZ-BARAHONA, J. M., CAPILUPPI, A., AND RAMIL, J. F. 2006. Comparison between SLOCs and number of files as size metrics for software evolution analysis. In *Proceedings of the European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 203–210.

HOWISON, J., CONKLIN, M., AND CROWSTON, K. 2006. FLOSSMole: a collaborative repository for FLOSS research data and analyses. *Int.l J. Inf. Technol. Web Eng. 1,* 3, 17–26.

ISRAELI, A. AND FEITELSON, D. G. 2010. The linux kernel as a case study in software evolution. *J. Syst. Software 83,* 3, 485–501.

KAGDI, H., COLLARD, M., AND MALETIC, J. 2007. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *J. Software Maintenance Evol. Res. Pract. 19,* 2, 77–131.

KAHEN, G., LEHMAN, M. M., RAMIL, J. F., AND WERNICK, P. 2001. System dynamics modelling of software evolution processes for policy investigation: Approach and example. *J. Syst. Software 59,* 3, 271–281.

KEMERER, C. F. AND SLAUGHTER, S. 1999. An empirical approach to studying software evolution. *IEEE Trans. Software Eng. 25,* 4, 493–509.

KITCHENHAM, B. A. 1982. System evolution dynamics of VME/B. *ICL Tech. J. 3*, 43–57.

KITCHENHAM, B. A. 2008. The role of replications in empirical software engineering: a word of warning. *Empirical Software Eng. 13,* 2, 219–221.

KITCHENHAM, B. A., AL-KHILIDAR, H., BABAR, M., BERRY, M., COX, K., KEUNG, J., KURNIAWATI, F., STAPLES, M., ZHANG, H., AND ZHU, L. 2008. Evaluating guidelines for reporting empirical software engineering studies. *Empirical Software Eng. 13,* 1, 97–121.

KITCHENHAM, B. A. AND CHARTERS, S. 2007. Guidelines for performing systematic literature reviews in software engineering. Tech. rep. EBSE-2007-01, Keele University.

KITCHENHAM, B. A., PFLEEGER, S. L., PICKARD, L. M., JONES, P. W., HOAGLIN, D. C., EL EMAM, K., AND ROSENBERG, J. 2002. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Software Eng. 28,* 8, 721–734.

KOCH, S. 2005. Evolution of Open Source Software systems—a large-scale investigation. In *Proceedings of the International Conference on Open Source Systems*.

KOCH, S. 2007. Software evolution in open source projects—a large-scale investigation. *J. Software Maintenance Evol. Res. Pract. 19,* 6, 361–382.

LAWRENCE, M. J. 1982. An examination of evolution dynamics. In *Proceedings of the International Conference on Software Engineering*. IEEE, 188–196.

LEHMAN, M. M. 1974. *Programs, Cities, Students: Limits to Growth?* Inaugural lecture, Imperial College of Science and Technology, University of London.

LEHMAN, M. M. 1978. Laws of program evolution—rules and tools for programming management. In *Proceedings of Infotech State of the Art Conference, Why Software Projects Fail*.

LEHMAN, M. M. 1979. On understanding laws, evolution, and conservation in the large-program life cycle. *J. Syst. Software 1*, 213–221.

LEHMAN, M. M. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE 68,* 9, 1060–1076.

LEHMAN, M. M. 1984. Program evolution. *Inf. Process. Manage. 20,* 1–2, 19–36.

LEHMAN, M. M. 1985a. Program evolution. In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady, Eds., Academic Press, San Diego, CA, 9–38.

LEHMAN, M. M. 1985b. The programming process. In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady, Eds., Academic Press, San Diego, CA, 39–84.

LEHMAN, M. M. 1985c. Programs, Cities, Students: Limits to Growth? In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady, Eds., Academic Press, San Diego, CA, 133–164.

LEHMAN, M. M. 1989. Uncertainty in computer application and its control through the engineering of software. *J. Software Maintenance: Res. Pract. 1,* 1, 3–27.

LEHMAN, M. M. 1990. Uncertainty in computer application (technical letter). *Commun. ACM 33,* 5, 584–586.

LEHMAN, M. M. 1991. Software engineering, the software process and their support. *Software Eng. J. 6,* 5, 243–258.

LEHMAN, M. M. 1996a. Feedback in the software evolution process. *Inf. Software Technol. 38,* 11, 681–686.

LEHMAN, M. M. 1996b. Laws of software evolution revisited. In *Proceedings of the European Workshop on Software Process Technology*. Springer-Verlag, London, 108–124.

LEHMAN, M. M. 1998. Software's future: Managing evolution. *IEEE Software 15,* 1, 40–44.

LEHMAN, M. M. AND BELADY, L. A. 1985. *Program evolution. Processes of software change*. Academic Press, San Diego, CA.

LEHMAN, M. M. AND FERNÁNDEZ-RAMIL, J. 2006. Software evolution. In *Software Evolution and Feedback. Theory and Practice*, N. H. Madhavji, J. Fernández-Ramil, and D. E. Perry, Eds., Wiley, 7–40.

LEHMAN, M. M., KAHEN, G., AND RAMIL, J. F. 2002. Behavioural modelling of long-lived evolution processes: some issues and an example. *J. Software Maintenance Evol. Res. Pract.e 14,* 5, 335–351.

LEHMAN, M. M. AND PARR, F. N. 1976. Program Evolution and its impact on Software Engineering. In *Proceedings of the International Conference on Software Engineering*. IEEE, Los Alamitos, CA, 350–357.

LEHMAN, M. M., PERRY, D. E., AND RAMIL, J. F. 1998a. Implications of evolution metrics on software maintenance. In *Proceedings of International Conference on Software Maintenance*. IEEE Computer Society, 208–217.

LEHMAN, M. M., PERRY, D. E., AND RAMIL, J. F. 1998b. On evidence supporting the FEAST hypothesis and the laws of software evolution. In *Proceedings of the International Software Metrics Symposium*. 84–88.

LEHMAN, M. M. AND RAMIL, J. F. 1999. The impact of feedback in the global software process. *J. Syst. Software 46,* 2–3, 123–134.

LEHMAN, M. M. AND RAMIL, J. F. 2001a. An approach to a theory of software evolution. In *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE'01)*. ACM, New York, 70–74.

LEHMAN, M. M. AND RAMIL, J. F. 2001b. Rules and tools for software evolution planning and management. *Ann. Software Eng. 11,* 1, 15–44.

LEHMAN, M. M. AND RAMIL, J. F. 2002a. An overview of some lessons learnt in FEAST. In *Proceedings of the Workshop on Empirical Studies of Software Maintenance*.

LEHMAN, M. M. AND RAMIL, J. F. 2002b. Software uncertainty. In *Proceedings of Soft-Ware 2002: Computing in an Imperfect World*. Vol. 2311. 174–190.

LEHMAN, M. M. AND RAMIL, J. F. 2003. Software evolution: Background, theory, practice. *Inf. Process. Lett. 88,* 12, 33–44.

LEHMAN, M. M., RAMIL, J. F., AND SANDLER, U. 2001. An approach to modelling long-term growth trends in software systems. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 219–228.

LEHMAN, M. M., RAMIL, J. F., WERNICK, P. D., PERRY, D. E., AND TURSKI, W. M. 1997. Metrics and laws of software evolution—the nineties view. In *Proceedings of the International Symposium on Software Metrics*.

LEHMAN, M. M. AND WERNICK, P. 1998. System dynamics models of software evolution processes. In *Proceedings of International Workshop on the Principles of Software Evolution (IWPSE'98)*. 20–24.

LOPES, C., BAJRACHARYA, S., OSSHER, J., AND BALDI, P. 2010. UCI source code data sets.

MADHAVJI, N. H., FERNÁNDEZ-RAMIL, J., AND PERRY, D. E., EDS. 2006. *Software Evolution and Feedback. Theory and Practice*. Wiley.

MENS, T. AND DEMEYER, S. 2008. *Software Evolution*. Springer, Berlin.

MENS, T., WERMELINGER, M., DUCASSE, S., DEMEYER, S., HIRSCHFELD, R., AND JAZAYERI, M. 2005. Challenges in software evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*. 13–22.

MOCKUS, A., FIELDING, R. T., AND HERBSLEB, J. D. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Software Eng. Methodol. 11,* 3, 309–346.

NEAMTIU, I., XIE, G., AND CHEN, J. 2013. Towards a better understanding of software evolution: an empirical study on open-source software. *J. Software: Evol. Process 25,* 3, 193–218.

PARNAS, D. L. 1994. Software aging. In *Proceedings of the International Conference on Software Engineering*. 279–287.

PERRY, D. E. 1994. Dimensions of software evolution. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, Washington, DC, 296–303.

PERRY, D. E. 2006. A nontraditional view of the dimensions of software evolution. In *Software Evolution and Feedback. Theory and Practice*, N. H. Madhavji, J. Fernández-Ramil, and D. E. Perry, Eds., Wiley, 41–51.

PIRZADA, S. S. 1988. *A Statistical Examination of the Evolution of the Unix System*. Ph.D. thesis, Imperial College, University of London.

POSNETT, D., FILKOV, V., AND DEVANBU, P. 2011. Ecological inference in empirical software engineering. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 362–371.

RAINER, A. AND GALE, S. 2005. Sampling open source projects from portals: Some preliminary investigations. In *Proceedings of the 11th IEEE International Symposium on Software Metrics*. 10–27.

RAJLICH, V. 2000. Modeling software evolution by evolving interoperation graphs. *Ann. Software Eng. 9,* 1–2, 235–248.

RAJLICH, V. AND BENNETT, K. Jul 2000. A staged model for the software life cycle. *IEEE Computer 33,* 7, 66–71.

RAMIL, J. F. AND LEHMAN, M. M. 2000. Metrics of software evolution as effort predictors—a case study. In *Proceedings of the International Conference on Software Maintenance*. IEEE Computer Society, 163–172.

RAMIL, J. F., LEHMAN, M. M., AND KAHEN, G. 2000. The FEAST approach to quantitative process modelling of software evolution processes. In *Product Focused Software Process Improvement*, F. Bomarius and M. Oivo, Eds., Lecture Notes in Computer Science Series, vol. 1840. Springer, Berlin, 149–186.

ROBLES, G., AMOR, J. J., GONZALEZ-BARAHONA, J. M., AND HERRAIZ, I. 2005. Evolution and growth in large libre software projects. In *Proceedings of the International Workshop on Principles in Software Evolution*. 165–174.

ROBLES, G. AND GERMAN, D. M. 2010. Beyond replication: An example of the potential benefits of replicability in the mining of software repositories community. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER'10)*.

SHULL, F., CARVER, J., VEGAS, S., AND JURISTO, N. 2008. The role of replications in empirical software engineering. *Empirical Software Eng. 13,* 2, 211–218.

SIEBEL, N. T., COOK, S., SATPATHY, M., AND RODRGUEZ, D. 2003. Latitudinal and longitudinal process diversity. *J. Software Maintenance Evol. Res. Pract. 15,* 1, 9–25.

SJØBERG, D. I. K., DYBÅ, T., ANDA, B. C. D., AND HANNAY, J. E. 2008. Building theories in software engineering. In *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjberg, Eds., Springer, London, 312–336.

SMITH, N., CAPILUPPI, A., AND RAMIL, J. F. 2005. A study of open source software evolution data using qualitative simulation. *Software Process: Improve. Pract. 10,* 3, 287–300.

TEMPERO, E., ANSLOW, C., DIETRICH, J., HAN, T., LI, J., LUMPE, M., MELTON, H., AND NOBLE, J. 2010. Qualitas corpus: A curated collection of Java code for empirical studies. In *Proceedings of the Asia Pacific Software Engineering Conference*.

TURSKI, W. M. 1996. Reference model for smooth growth of software systems. *IEEE Trans. Software Eng. 22,* 8, 599–600.

TURSKI, W. M. 2002. The reference model for smooth growth of software systems revisited. *IEEE Trans. Software Eng. 28,* 8, 814–815.

VASA, R. 2010. *Growth and Change Dynamics in Open Source Software Systems*. Ph.D. thesis, Swinburne University of Technology, Melbourne, Australia.

WERNICK, P. AND LEHMAN, M. M. 1999. Software process white box modelling for FEAST/1. *J. Syst. Software 46,* 2–3, 193–201.

WOODSIDE, C. M. 1980. A mathematical model for the evolution of software. *J. Syst. Software 1,* 4, 337–345.

WOODSIDE, C. M. 1985. A mathematical model for the evolution of software. In *Program Evolution. Processes of Software Change*, M. M. Lehman and L. A. Belady, Eds., Academic Press, San Diego, CA, 339–354.

Received April 2012; revised October 2012; accepted June 2013