A COMPARISON OF BLACK-BOX MODELS FOR SOFTWARE EVOLUTION

by

EDUARDO FUENTETAJA, B.S.

A THESIS

IN

SOFTWARE ENGINEERING

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

IN

SOFTWARE ENGINEERING

Approved

August, 2002

ACKNOWLEDGMENTS

This author would like to thank Dr. Donald J. Bagert, who has been my advisor since I came to Texas Tech and now is the chair of my thesis' committee. He proposed the thesis topic, which I never suspected would be as exciting as it has been, and guided me during the whole journey. I wish him the best of luck on his future endeavors.

I would like to thank Dr. Susan A. Mengel for accepting being part of my thesis committee and the kindness and good disposition she always showed me.

I do not want to forget the Graduate School and the Computer Science Department at Texas Tech for giving me the opportunity of being part of a learning experience that I never suspected one day I would reach.

Finally, thanks to my parents that sacrificed everything for my future and the future of my sisters. I wish them a future of harvest.

TABLE OF CONTENTS

ABSTRACT

Last thirty years have seen the birth and maturity of software evolution as a specialized field inside the vast area covered by software engineering. Software evolution focuses on the study of how software systems change during their lifetime, trying to shed some light on the nature of that change and the relationships between system's attributes from one release to the next.

Early empirical studies of software evolution discovered some patterns common to the evolution of every software system that was analyzed. These invariants were formalized in the "laws of software evolution." The election of the term law is intentional and expresses their universality. However, as this study intends to prove, the laws have undergone multiple revisions, and some early conclusions were contradicted by late discoveries, as new empirical studies of the evolution of software systems became available.

One of the practical derivations of the laws is the inverse square model. The model explains the growth of evolutionary software systems during their lifetime and it can be used to forecast accurately a system's size many releases in the future, which allows management to plan and allocate resources well in advance. Although many papers have presented examples of software systems whose growth is successfully modeled by the inverse square equations, some recent studies present prove of just the opposite.

The QUICK application, a web-based system developed locally at Texas Tech is one example of a system whose growth cannot be explained by the inverse square model. This motivated the analysis of the underlying reasons, finally found in the intrinsic limitations of the model. In addition, the present study proposes an alternative model, already suggested in some recent studies of software evolution, as a better alternative. The model is referred here as the "constant work rate" model and is based on a simple exponential relation. The present study includes a comparison of both inverse square and constant work rate models, presenting proof of the superiority of the constant work rate model.

LIST OF TABLES

LIST OF FIGURES

CHAPTER I
INTRODUCTION

After more than thirty years, software evolution is still a mostly unknown area of research, struggling to find its own place inside the vast field of software engineering. Studies on software evolution have contributed to the advancement of software engineering addressing questions such as how software systems change during their lifetime, what is the nature of that change, how different versions of a software product are related, or what are the connections between the evolution of a software system and its development organization. Software evolution advocates claim for the independence of the field, in particular stressing its differences with software maintenance, a well-established area inside the huge body of software engineering.

From a practical point of view, numerous benefits could be derived from progress in the field of software evolution:

1. Knowing how internal attributes change from version to version on a software system, theoretical models can be defined and used later to forecast future values of such attributes. Attributes include, for instance, software size, complexity, amount of work delivered per version, or productivity of the development organization. Forecasting models are valuable tools for a better planning and management of development and maintenance processes.

2. Understanding the relationships among a software system, its development organization, and the operational environment helps to foresee the results of related changes in any of them. Knowing beforehand the repercussions of changes in the staffing of the development organization, a hasty delivery of extra-functionality demanded by the customer, or the introduction of a new technology, is again precious information for an effective management. The internal dynamics of software processes offers an explanation to why software process improvement practices have failed to deliver, in some cases, the desired results.

However, software evolution is still to produce major breakthroughs in the practical arena. Today, only very large software organizations have the resources to

devote time and personnel to the study of their projects from an evolutionary perspective. Results are considered interesting but mostly of academic value. Academia does not fling itself into the software evolution field; on the contrary, the field only seems able to attract a niche of researchers.

One of the most important practical results derived from years of research in software evolution is the development of empirical models that forecast the expected trend of software systems' attributes during future releases. The best-known model at this point is the inverse square model, able to forecast the general trend of the size of some software systems, using only information gathered from a small number of early versions of the system. The utility of such model as a planning and management tool is undoubted, but its generality is put into question, since the model has failed to fit data coming from a number of software systems documented in the literature. Moreover, since the model's authors claim that the model is based on some of the laws of software evolution, the universality of such principles could be questioned.

To include those systems that failed to conform to the established model, a new model has been introduced recently, based on the organizational conservation of the work rate. This new model has delivered improved results, but the coexistence of multiple models may generate a certain degree of confusion. Practitioners who want to forecast the future state of their software applications will need to choose from one of the two models, and no guidance has been published yet to this respect on the literature.

The present study will try to make a modest contribution to the field of software evolution, by providing evidence of the weaknesses of the inverse square model and the superiority of the constant work rate model. The evidence suggests that the inverse square model could be discarded, promoting the constant work rate as the dominant focus of futures efforts in the seek for a final theory of software evolution.

Statement of the Problem

The introduction of the inverse square model represented a major accomplishment in the field of software evolution. The major virtues of this model are its simplicity,

based one some intuitive principles extracted from the laws of software evolution; and its ability to forecast successfully future trends of some software systems, even though the amount of data used to feed the model is extremely limited. After a period of fame, a number of counter-examples from the literature showed the limitations of the model. In addition, this study includes the analysis of the evolution of the QUICK, a web tool developed at Texas Tech University. The analysis includes six versions of the system. Results of the inverse square model manifests in this case the same pattern of problems that other authors have found before, fact that could provide additional evidence of the limitations of the model.

Such limitations have not been acknowledged explicitly by its authors, but in recent publications, the inverse square model appears together with a complementary model, the constant work-rate model, outlined before. This new model manages to successfully fit the evolutionary trend of software systems that the inverse square is unable to model. The coexistence of both models could indicate an implicit acknowledgement of the limitations of the inverse square model by its own authors.

The coexistence of multiple models may generate a certain degree of confusion in the field of software evolution. Practitioners who want to forecast the future state of their software applications will need to choose from one of the two models, and no guidance has been published yet on the literature. Since the trends forecasted by both models could produce very different results, it is clear than one of the two predictions has to be wrong, although there is no way to know which one will be the misleading prediction beforehand. An inaccurate forecast could discourage practitioners and made them spread the word of software evolution theories as of little practical interest, undermining future studies in this area.

The present study aims to present evidence of the superiority of the constant work-rate over the inverse square model, encouraging the usage of the former model in almost any situation. With only one model, practitioners would not need to make a choice and efforts in the field of software evolution would not be diverted to multiple fronts.

Original Contribution of this Study

The original contribution presented in this study aims to strengthen the position of the constant work-rate model as the dominant model in a future theory of software evolution:

1. A study of the different versions of the laws of software evolution and the reshaping they have undergone since their first formulation. There is evidence on the literature of smooth changes in the formulation of the laws as new data, which did not conform to previous results, were available. Moreover, late results from the authors contradict some of their early statements, putting into question the universality of the laws that they defend.

2. An evolutionary study of the QUICK scheduler, a web application developed at Texas Tech University. Results of the inverse square model applied to metrics of this application over its first six versions show a poor fit and inadequate forecasting ability. Such results confirm very precisely what other authors have found before, providing another evidence of possible limitations in the inverse square model.

3. An exploration of the limitations of the inverse square model. Proof is presented of the inability of the model to fit software systems with a lineal or super-lineal growth. This pattern of grow is observed in the systems where the inverse square model failed to work, including the QUICK application.

A comparison of the inverse square model and the more recent constant work-rate model. The new model is applied to evolutionary datasets of some software systems that were specifically chosen in the past to show the strengths of the inverse square as a predictive model. The comparison shows that the virtues that were highlighted for the inverse square model are also present in the constant work-rate model. The constant work-rate model is also applied to software systems that the inverse square failed to model. Again, the constant work-rate model delivers good estimations, manifesting its superiority over the inverse square model.

Organization

Chapter I has presented and introduction to the software evolution and time series topics and the main goals that this study aims to. Chapter II covers an in-depth literature review of the current research in software evolution and time series techniques that are used in this study. Chapter III covers the methodology used in this study. Chapter IV presents results derived from this study, and Chapter V summarizes the research and provides directions for future work.

CHAPTER II
REVIEW OF LITERATURE

The literature review of this study starts with the origins of software evolution with Lehman's seminal work on the field: the study of the evolution of the IBM's OS/360 (Lehman and Belady 1985c), first published in 1969 as an IBM's internal report. The evidence from that report led to the first formulation of the laws of software evolution (Lehman and Belady 1985d) on 1974. The original formulation included three basic laws, resembling not by chance some of the laws of physics. After new evidence were accumulated and new data from different software systems arrived, the laws were reshaped and new laws were incorporated, to a total amount of eight laws as of 2001. The evolution of the laws and the reasons underlying those changes will be covered in this study mostly using material from (Lehman and Belady 1985b; Lehman et al. 1997; Lehman and Ramil 2001).

The Laws of Software Evolution

Similarly to the laws of physics that formalize some basic and universal principles that govern the physical world, the laws of software evolution define the fundamentals that govern the evolution of any software system during its lifetime. Only three laws were defined on the first publication of the laws of software evolution, but after multiple revisions, the latest publication includes a total of eight laws. The next sections review every major revision, presenting how the definition of the laws has changed, as new evidences have been available. The study of the evolution of the laws of software evolution presents a process still uncompleted, with exceptions and contradictions still unresolved that question the scientific severity that the term "law" suggests.

First Formulation (1974)

The first formulation of the laws of software evolution appears on 1974 (Lehman and Belady 1985d). During the early 1970s, there was a great concern about the state of software engineering (although the term "software engineering" was not yet popular,

studies from that time use the longer term "design, implementation and maintenance of computer software systems," and any equivalent combination). Before structured programming (formulated in late 1960s), contractors could not understand the poor performance, cost overruns, and frequent delays of software projects. Lehman pondered about the slow advances in the field and humble improvements achieved during previous years of efforts. Based on his personal experience and starting with a study from 1968 that he leaded about the IBM OS/360 evolution (Lehman and Belady 1985c), Lehman made explicit a number of principles found during a detailed analysis of the collected data. Those principles suggest an explanation for the irregular successes that were hindering the early years of software engineering, explanation that, at a large extent, is still valid today.

The study analyzes the evolution of attributes such as the number of modules, statements, instructions, and modules handled (modified, added or removed) per version. The data was collected on each of the 20 releases of the IBM flagship operative system, available at that time. The OS/360 was developed entirely in assembler and its 20th version totalized over two million lines of instructions.

Table 2.1 summarizes the first formulation of the laws of software evolution, and a detailed description of each of them follows.

Table 2.1. Laws of Software Evolution as of 1974.

No.	Brief Name	Law
I	Continuous Change	A system that is used undergoes continuing change until it becomes more economical to replace it by a new or restructured system.
II	Increasing Entropy	The entropy of a system increases with time unless specific work is executed to maintain or reduce it.
III	The 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.

Source: Lehman and Belady (1985d)

The first law of software evolution is the fundamental principle of software evolution: the never-ending state of change of software systems operated in a real environment. The universality of this principle seemed so evident that no further proof was presented at this point.

The second law, whose definition resembles the second law of thermodynamics, states that entropy irreversibly increases for an isolated system. Entropy is a concept still under intense debate by philosophers and physics, but from a practical point of view, it can be considered as equivalent to the amount of information or complexity of the system. As physical systems do, a software system does not spontaneously reduce its complexity. Its nature tends to convert it in an increasingly complex system as new versions are released. The law is based again in the empirical analysis of the OS/360 system and the intuition that the same principle can be applied universally to any other software system (Lehman and Belady 1985d).

The presentation of the third law was the most significant contribution, since its rational is not evident and mostly counter intuitive. The law suggests that a new view of the software development and maintenance activities should be adopted, before a real advancement in the area could be achieved. The new view connects such activities with organizational issues and the environment where of software system operates, constituting a systemic approach to software engineering.

A Systemic Approach of Software Engineering

The measured attributes showed apparent erratic fluctuations from version to version, but once the data were conveniently averaged, very clear trends started to manifest on the plots. The plot of the number of modules per versions against the release sequence number was particularly striking for Lehman, since it showed an almost perfect linear fit. In the words of Lehman, the linear average growth of the OS/360 is that the system, considered globally (including the development organization, people and resources) behaves as a "self-regulating organism subject to apparently random shock, but overall obeying its own specific conservation laws and internal dynamics" (Lehman

and Belady 1985d). In other words, although the evolution of a software system behaves randomly at short time-scales, its regulatory internal processes lead to a smooth trend when larger time-scales are considered. While some processes tend to increment the size of the software system, others try to keep its size small. A client's demand for additional functionality or a decision of the marketing department of an opportunity to reach new markets by extending the system are arguments that pushes for an increment of the software system's size. However, an increment of functionality increases the complexity of the system and therefore the probability of defects and undesired side effects to happen, lingering the future ability of the system to grow. When the complexity of the system exceeds what the developing organization can handle, the delivery of new functionality requires first undertaking activities intended to reduce the system's complexity. Such activities include for instance: improvement in the system documentation, redesign of the system's architecture to reduce module's dependencies, or complete re-programming of defect-prone modules. In general, these activities do not lead to a growth on the software system, on the contrary, they tend to keep its size constant or even lighten it. Overlooking the importance of such activities can put the future of the system at stake: the time will come when an overwhelming incoming rate of defects may collapse the development organization, just to cite one possible worst-case scenario.

The activities that strive for an increment of the system's functionality are called progressive, whereas the activities that do not add new functionality to the system but reduces system's complexity, allowing it to keep growing in the future are called anti-regressive. Since a growth in progressive activities increments the need for more anti-regressive work, the development organization should redirect more resources from the progressive to the anti-regressive area, decreasing the rhythm of functionality delivery. Lehman sustains that the compensation of both forces is what stabilizes the effective work rate delivered by the development organization, (Lehman and Belady 1985d).

Should the development organization fail to redirect resources towards anti-regressive activities, the future of the system may be put at risk. Another possible

scenario is that the organization decides to keep the rate of progressive work and staff the anti-regressive area with more personnel, but this may be a risky decision since the decrement of productivity can undermine the ability of the organization to compete, jeopardizing again the long-term survival of the system.

The suggestion of a constant work rate in evolutionary software systems will constitute its own law at the next revision of the laws of software evolution.

A Reason for Failures in Software Process Improvement Initiatives

Lehman envisions software processes as dominated by forces that have positive and negative effects over systems attributes and that dependent among them. Using control engineering terminology, software processes would constitute a complex system with multiple positive and negative feedback loops. Under the systemic nature of software processes lies an explanation for the difficult and slow improvements in the field of software engineering, despite the numerous efforts dedicated. Process improvement activities that lack to consider the software development and maintenance as a complex feedback system are destined to fail. Such activities, focused on improvements over a local aspect of software processes, may improve that particular aspect, but the global performance may suffer at the same time (Lehman 1996).

An Example of a Local Improvement but a Global Negative Impact

As a hypothetical example, consider a software company that acquires a sophisticated CASE tool, which will be used to elaborate formal design documents of new projects. Previously, the design documents where plain text documents following a common template, but analyst were free about content, terminology, or language. The introduction of the new tool brings important costs in the form of license purchase and training. After an early period of excitement, analysts get discouraged by the overhead introduced in their daily activity, the lose of flexibility, the need to comply with the particularities of the new tool, and mainly, by the evidence that the new tool does not

help them to produce better designs. Analysts finally recommend discontinuing the use of the CASE tool and go back to their previous practices.

A CASE tool does not necessarily imply an automatic improvement in the quality of the produced designs. The main advantage of such tool is to provide the basis for a common language, able to be understood by any member of the organization, even recently hired. A common language in design documents facilitates greatly the introduction of formal reviews, technique that could have derived in a direct improvement of the design's quality, but was not implemented.

The example proves that the introduction of a well-meaning practice, by itself, may not generate any direct benefit to the organization, but should it have been accompanied by another practice–the design artifacts review–it had produced a real advantage.

In Defense of the Universality of the Laws

The solid rational foundations of the principles that sustain the laws of software evolution and a great deal of personal intuition suggested to Lehman about their universality. Lehman was struck by the OS/360 evolutionary patterns, declared the universality of the principles found, there promoting them to the category of laws. Lehman received a lot of criticism for this (Lehman 1996). Its detractors complained that only one example (the OS/360) does not constitute a strong proof of the universality that the term "law" suggests. However, Lehman's first formulation of the "laws" of software evolution are not based on the particular OS/360 data, but on the underlying qualitative principles that particular example contributed to disclose. Moreover, numerous studies of software systems that came after the first publication of the laws confirmed, or at least not contradicted, their formulation.

First Revision of the Laws of Software Evolution (1978)

The conference "Why Software Projects Fail," held four years after the first publication of the laws of software evolution (Lehman and Belady 1985a), included a

lecture by Lehman called "Laws of Program Evolution–Rules and Tools for Programming Management." The lecture presented the laws of software evolution, and introduced a third law; the law of statistical smooth growth, to address the question rose by the conference's title. As it was discussed in the previous section, the third law implies a systemic approach to software engineering. Software projects where project related decisions are made without consideration of their global consequences, are risking the long-term survival of the project.

The most interesting point in this revision, summarized in Table 2.2, is how some of the implications derived from the third law have been split up in two additional and independent laws, making a total of five laws for this revision.

Preliminary Considerations

Before introducing the laws, a couple of points are addressed early in the lecture. First, a criticism of the current state of software engineering practice. Practitioners are censured because they usually consider only the current state of the project and the environment in decision-making processes. On the other hand, Lehman work is based on stressing the importance of the history of a software system. The invariants observed during the study of historical trends in software systems were the basis for the formulation of the laws of software evolution. Moreover, the laws are used here to explain the reasons for the chronic poor performance of software engineering.

Second, the consideration that the laws of software evolution only apply to large systems. The term "large" is defined not by any metric related with extension, but with variety. Variety in the code's logic, that relates the adopted solution to the changing operational environment and variety in the human interest and activities that the system reflects. Variety here could be understood as an indication of the number of intervening forces and the complexity of the relationships among those forces in the process of software development and maintenance. The higher the number of the forces and complexity of interactions, the stronger the system dynamics will be, and it has been

presented in the previous section how the laws of software evolution are an externalization of that system dynamics.

Table 2.2. Laws of Software Evolution as of 1978.

No.	Brief Name	Law
I	Continuing Change	A large 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 recreated version.
II	Increasing Complexity	As a large program is continuously changed, its complexity, which reflects deterioration structure, increases unless work is done to maintain or reduce it.
III	Statistically Regular Growth	Measures of global project and system attributes are cyclically self-regulating with statistically determinable trends and invariances.
IV	Invariant Work Rate	The global activity rate in a large programming project is invariant.
V	Law of Incremental 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 over-runs.

Source: Lehman and Belady (1985a).

The First Law: Continuing Change

This version offers further qualitative explanations about this law than the first publication did. Software systems represent models of the real world where they operate. Customer understanding of the problem whose solution is sought and requirements definition are two previous steps until the final model–the code itself–is reached (Lehman

1998d). A software system could be defined as satisfactory if it solves a real-world problem or problems on an effective manner. The problem definition and the meaning of what effectiveness represents to all stakeholders are given at a large extent by the environment that, in its turn, is conditioned by human, economic, and technological factors and its scale could be as large as the whole world. The environment is in a continuous state of evolution and therefore, continually redefining the problem and defying the development and maintenance organization to keep the system's perception by its users as useful, non-obsolete.

Should the development/maintenance organization fail to keep up with the environment, the system would be perceived as of declining quality, and customers will start to consider the possibility of replacing the system (the concept of perceived declining quality will deserve its own law in a future revision).

The Second Law: Increasing Complexity

The second law also is followed in this revision by an extended qualitative explanation.

First of all, the term complexity is defined in the context used here. Program complexity is considered as relative to a certain level of abstraction: functional, subsystem, component, module, and instruction. Two different kinds of complexity are considered, both connected to concepts of program comprehensibility.

- Internal or intrinsic complexity: reflecting only internal attributes of the code, it expresses the relative difficulty to understand a program, without any previous knowledge of it.
- External complexity: again, the relative difficulty to understand a program, but this time supposing that exact and accurate documentation of the program would be accessible.

The difference between external and internal complexity would reflect the improvement achieved by a–theoretically perfect–documentation, and it constitutes an

interesting measure of a profitable amount of resources devoted to documentation in a software development organization.

After these considerations about the meaning of complexity, the reasons for the increasing complexity of software systems are addressed. The main reason is found in economic factors of software development and maintenance activities. New releases of software systems are produced trying to maximize the business benefits of the development and maintenance organization. The goal is to produce a maximum of functionality at the minimum possible cost. This does not necessarily imply that the organization will always try to cut corners. However, whereas quality practices are proven profitable at a long-term scale, sometimes organizations are pushed to sacrifice the long-term due to short-term compromises, business opportunities, or financial issues. Therefore, unless a strict long-term plan was carefully planned and resources allocated beforehand, organizations will tend to embrace progressive activities rather than anti-regressive activities, as they were described on the previous section. Progressive activities include work intended to improve or add functionality to a system, something that users perceive directly and benefit from. However, anti-regressive activities, which include quality practices or preemptive maintenance, are intended to keep the system manageable and allow it to keep evolving in the future. Anti-regressive activities do not have a direct impact on the quality perceived by users of the system. This is the reason that explains why sometimes organizations neglect to invest in anti-regressive activities and divert most of the resources to the progressive area.

Version-upon-version of new or improved functionality built over an original system, whose original design and intentions could be barely distinguishable at this point, is the manifestation of increasing levels complexity. Sooner than later, the organization will face a collapse on its ability to evolve the system and satisfy its customers.

Empirical proof of the second law of software evolution is also presented in this lecture. Five out of six software systems analyzed present evidence of increasing complexity (one of the systems, system T, is very likely the OS/360). The systems are described as "of different types, treated and maintained by quite different organizations."

Complexity is introduced here as proportional to the number of modules handled (modified, added, or removed) per version, which is quite surprising at this point because the previous, extensive disquisition about complexity did not make any explicit reference to this definition. An explanation of why one of the systems analyzed failed to follow this pattern is also offered (its particular architecture is pointed out as guilty of this).

The Third Law: Statistically Regular Growth

An extensive discussion of the third law was presented in the previous section. This new revision reaffirms what was said before, presenting proof of six software systems analyzed (the same that were used in the previous law). All of them exhibit smooth general trends of different attributes, confirming the third law. In addition, cyclic variations over the average trend are observed in the datasets, suggesting the presence on well-defined cycles during the evolution of the systems.

Connecting the results of the third law with concepts like inertia, momentum, and feedbacks effects offers another deliberated association of the laws of software evolution with the laws of physics.

The Fourth Law: Invariant Work Rate

This version introduces the first formulation of the fourth law of software evolution: the law of invariant global activity rate. Evidence is presented of two software systems where plots of the cumulative number of modules handled per day of work show perfect linear trend. The number of modules handled (modified, added, or removed), which were used in the second law as a measure of system's complexity, are interpreted here as a measure of the global activity.

The explanation of this behavior is found in the principles of the third law: strong system dynamics that tend to stabilize the long-range trend of system's attributes (in this case the global activity rate), whereas short-term variance of the same attributes presents an irregular landscape: high values followed more likely by low values and vice versa. "High" and "low" mean in this context "higher" and "lower" than the long-term general

trend. An activity rate higher than the average is likely to produce a system's version that contains more defects, less documentation, and wears programmers more than a lower activity rate would do. Following versions will likely be devoted to all the anti-regressive activities presented before: defect fixing, preventive maintenance, or documentation improvement. Such activities do not increment the count of modules handled as much as the progressive activities do, and therefore the activity rate drops on further versions. After these periods the system is prepared to grow again in functionality and a new cycle starts. Thus, the average activity rate is kept constant when large temporal scales are observed.

In is important to notice that the law defines a "statistically invariant" activity rate, or, in other words, that the activity rate is normally distributed, with constant mean and variance, following a smooth long-term trend. However, it does not define in which way the trend is smooth neither its shape. Lehman seems to be the first surprised to observe that the activity rate of the systems analyzed follows an almost perfect linear trend.

The Fifth Law: Law of Incremental Growth Limit

This version also represented the first introduction of the fifth law: the law of incremental growth limit. The law is not as well defined at this point as the previous four laws are and its introduction seems to be more a conscious, bold assumption, mostly based on what Lehman's intuition seem to perceive as a singular pattern on three of the analyzed systems. The pattern seems to correlate versions that represent large increments in the system size with troubles derived from those versions: reliability problems, poor performance, late delivery, or cost overruns. A "large increment" is defined here as an increment two times above the average increment, measured over the whole lifetime of the system. The system's size at a given release is measured counting the total number of modules of the system at that release.

The smooth trend of the incremental growth observed is also mentioned here, but its importance will be stressed in the next revision of the laws. Another point with

important future repercussions is that a "normal" increment of a software system is defined as the average increment, calculated using data from every release of the system. A system that always grows at a normal, average pace is growing at a constant rate, and therefore, the evolution of its size must follow a perfect straight line. This implication is continued in the next revision of the laws.

The present formulation of the third law has some important practical implications, and if confirmed, it could represent a serious managerial tool. Software managers would be more than willing to count on a tool that allows calculating the right amount of "safe" functionality to include in the next release of a software system. Unfortunately, as it will be shown later, this point could not be proved and the fifth law underwent a serious overhaul on the next revision.

Second Revision of the Laws of Software Evolution (1980)

In 1980, Lehman published the second revision of the laws of software evolution (Lehman and Belady 1985b). The laws and their definitions are summarized in Table 2.3. This second revision confirms previous results over the first three laws, but redefines the fourth law, making it more generic, and changes drastically the fifth law's content.

Table 2.3. Laws of Software Evolution as of 1980.

No.	Brief Name	Law
I	Continuing Change	A program that is used and that–as an implementation of its specification–reflects some other reality, undergoes continuing change or becomes progressively less useful. The change or decay process continues until it is judged more cost effective to replace the program with a recreated version.
II	Increasing Complexity	As an evolving program is continuously changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain it or reduce it.
III	The Fundamental Law of Program Evolution	Program evolution is subject to a dynamics which makes the programming process, and hence measures of global project and system attributes, self-regulating with statistically determinable trends and invariances.
IV	Conservation of Organizational Stability (invariant work rate)	The average effective global activity rate in a project supporting an evolving program is statistically invariant.
V	Conservation of Familiarity (perceived complexity)	The release content (changes, additions, deletions) of the successive releases of an evolving program is statistically invariant.

Source: Lehman and Belady (1985b).

Lightening the Laws

Before introducing the laws, Lehman dedicates some paragraphs to lighten up the term "law," probably due for the criticism aroused among other investigators, who did not receive well such a strong term, as a later paper relates (Lehman et al. 1997). The laws are softened by stressing the following ideas:

1. Laws of software evolution only apply to large programs. Only large programs develop the dynamics that generate a continuous cycle of maintenance and evolution. A previous definition of what qualifies a program as "large" is criticized. A more accurate definition is promised for later.

2. Laws of software evolution are not as immutable as the laws of physics are. The reason is that laws of software evolution are, at a large extent, the manifestation human phenomena involved in software development and maintenance processes. Decisions are influenced by the personalities of managers, the level of pressure applied to developer and how they deal with it, customer preferences, just to cite some examples. This is an important clarification, but it must be remembered that it was Lehman himself who chose the second law of software evolution as an analogous to the second law of thermodynamics, establishing an explicit link between his laws and the laws of physics.
3. Laws can only explain large-scale behavior of the evolution of software systems. This is because the evolution of software systems is dominated by human factors, whose precise prediction is not possible at an act-by-act scale. However, the self-regulatory nature of the processes involved in the software system, organization, and environment regulate the trends and smooth large-scale behaviors, making its prediction possible.
4. The Eisenberg principle. Lehman ponders that a future acceptation of the laws of software evolution could affect the way that software projects are managed, affecting their behavior and effectively changing the laws. The alteration of the laws by their simple knowledge and their application for a better software engineering can be seen as equivalent to the Eisenberg principle, which refers to the paradox of experiments affected by a passive observer.

The Fourth Law: Conservation of Organizational Stability

The name of the law is changed in this version from the "law of invariant activity rate" to the "law of organizational stability." The new definition reflects the inner cause for the invariant activity rate observed in some software systems, whereas the invariant activity rate is included later in the law as a consequence, a practical implication of the

underlying principle. The change is not critical and contributes to make all the laws more consistent.

The principle of organizational stability refers to the fact that human organizations tend to smooth variations, rejecting sudden changes. Abrupt changes are avoided because they have the potential of destabilize the organization, bringing initial chaos and turmoil. Shareholders hate this situation. Changes in resource allocation, staffing, organization structure, or business plans are carefully planed to occur in a controlled way. However, not only smooth progress happens because organizations tend to reject hastiness, but also because sudden changes are impossible at all when the organization is over a certain size. A large organization cannot change its structure or business line from one day to the next; the same way a large ship needs plenty of time to change its course, even though when the ship is heading a large iceberg.

Conservation of the organizational stability implies smooth changes in the organization, and one of the consequences is the observed invariant activity rate observed in some software systems.

The Fifth Law: Conservation of Familiarity

The fifth law experiments a major revision, changing its name from the "law of incremental growth limit" to the "law of conservation of familiarity." Again, the change tries to improve the definition by naming the law using the fundamental principle, not an observed empirical result. The empirical result (suggestion of a safe increment rate of software systems), which was the fundamental point in the previous revision, is in this occasion shadowed, relegated to a secondary position. The focus is moved to the empirical evidence, observed in the evolution of some software systems, of smooth incremental growth trends at large temporal scales, and oscillatory trends at small temporal scales. This result was presented in previous revisions of the laws.

The conservation of familiarity is the principle that Lehman suggests as the cause of the incremental growth pattern. The concept is not easy to comprehend. Briefly, the cyclic ups and downs around the long-term trend observed in the plot of incremental

growth of software systems are correlated with the perceived complexity of the system. A version of a software system delivering a great deal of new or modified functionality, requires a major intellectual effort by users, developers and maintainers. Stakeholders are unfamiliar with the new version and a major learning process is necessary to recover the old degree of comfort. The work time that developers and maintainers would be dedicating to develop new functionality for next versions is spend instead in the mentioned learning process, experimenting with the new version and getting to know its peculiarities. After a period of decreased effective activity, the levels of familiarity with the version are recovered and the organization is ready to deliver again a large version of the system, which in its turn will start a new cycle again

Familiarity is consequently defined as the opposite of (or inversely proportional to) the system's perceived complexity. The stabilizing effect of variations in the level of familiarity of a software system is the cause of the cyclic variations and invariant long-term trend in the incremental growth of the system.

It is important to highlight the definition used in the law, in particular how the release content of evolutionary software systems is defined as "statistically invariant." Lehman has chosen those words carefully, defining the law with an intentional generality. The explanation of the law goes a little bit further. The pattern observed on the evolution of software systems' size is described as exhibiting a large variability from one version to another (as was presented above), but remarkably constant on the average. Thus, the statistic referred in the definition is simply the average, and the implications of a constant average rate of growth are important. During the previous revision of this law, it was highlighted how a system that grows at a constant pace is a system that grows at a linear rate. A linear trend to explain or predict a system's size is therefore implied here, but never explicitly stated. It must be mentioned that the OS/360 follows a linear trend when the system's size, measured as the count of modules, is plotted against the release number. This plot is almost a perfect linear fit.

Lehman was prudent not explicitly talking about software systems growing a linear rate. That would have been a very specific assertion and perhaps at this point not enough empirical evidence was available.

Third Revision of the Laws of Software Evolution (1996)

The next revision of the laws of software evolution considered here marks the official addition of three more laws, making a total amount of eight laws (Lehman 1996). This revision is accounted in Table 2.4 and it will be the last major modification to the laws of software evolution until today. Later revisions will just concentrate on the redefinition of some of the laws, mainly due to the discovery of new evidences connected to the principles formalized by the laws. The three new laws included in this revision do not represent major breakthroughs in the field; they simply formalize some ideas or implications that were implicit in the former five laws and other software evolution studies and conferences.

This revision uses the term E-type program to refer to software systems that experiment evolutionary processes, according to Lehman's classification of program according to how they relate to their operational environment. A compendium of this classification can be found at (Lehman and Belady 1985e), and a summary of that paper at (Pfleeger 1998). Before continuing with the laws of software evolution, a brief account of the classification follows:

> S-type programs: are based on a formal specification of the problem intended to solve. Examples are programs that multiply matrixes or find a solution for the dinning philosophers problem. S-type programs can be applied to solve real-world problems, but should the statement of the problem change, a new, different program must be developed. Programs of this type of do not evolve.
>
> P-type programs: are programs that solve a precisely defined real-world problem, but due to the particularity of the problem, the solution can only be an approximation. An example is a program that plays chess. Whereas the problem is precisely defined by the rules of the game, a precise solution in computational

terms in not available, at least not yet. Current chess programs try to optimize the best possible move found in the time allotted, where the definition of "best possible mode" is clearly subjective. P-type programs experiment evolutionary processes because, although the definition of the problem does not change, there is a continuous pressure to release improved solutions.

E-type programs: as a logical continuation of S and P-type programs, E-type programs try to solve a real-world problem, whose definition is not precise, neither is the attempted solution. Examples of E-type programs are everywhere: a word processor, for instance: the problem intended to solve is changing every season with consumer preferences and competence pressure: formatting, WISIWIG capabilities, connectivity with other applications, graphics, revisions, support for web documents, are only some examples. How the solutions are implemented is also changing from version to version. Evolution is inherent to this kind of program, driven by the difference between the model of the world that the program represents and the real world itself, that continuously redefines: the problem that the program intends to solve, the assumptions implicit in the program, and the perceived quality of the solution produced by the program.

Continuing now with the revision of the laws, it is important to point out that the first five laws just experimented minor cosmetic changes in this revision. Neither new evidence nor more conclusions are added here to support the principles behind the laws. Thus, the following paragraphs will focus only on the newly introduced laws.

Table 2.4. Laws of Software Evolution as of 1996.

No.	Brief Name	Law
I	Continuing Change	An E-type program that is used must be continually adapted else they become progressively less satisfactory.
II	Increasing Complexity	As a program is evolved its complexity increases unless work is done to maintain or reduce it.
III	Self Regulation	The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.
IV	Conservation of Organizational Stability (invariant work rate)	The average effective global activity rate on an evolving system is invariant over the product lifetime.
V	Conservation of Familiarity	During the active life of and evolving program, the content of successive releases is statistically invariant.
VI	Continuing Growth	Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.
VII	Declining Quality	E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.
VIII	Feedback System	E-type evolution processes constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.

Source: Lehman (1996).

The Sixth Law: Continuing Growth

The formulation of this law resembles greatly to the first law of software evolution. Both talk about the never-ending changes in software system, but the first and sixth laws make an emphasis in two different causes for this state of continuous change.

The first law states that the continuous change of software systems is due to the differences between the environment where the software system operates, and the system itself. This environment is a continuous state of change. First of all, it cannot be controlled by the development and maintenance organization, by definition the environment starts where the system and its organization ends. Second, the environment is driven by multiple forces, such us technological improvements, competence pressure, or even the global economy. Both factors contribute to produce the chaotic world where software systems live, at least from the organization point of view. On the other hand, changes in the environment redefine the problem; the assumptions implicit in the solution adopted; and how the system is perceived by their users, or in other words the definition of customer satisfaction. Software systems must be continuously adapted to keep up with the environment's pace. This is the principle stated by the first law.

The new sixth law also implies continuous change in software systems, but in this case, a second, complementary cause is described as responsible. The sixth law states that economic and market factors in software organizations force the early delivery of releases, before the planned functionality is fully completed or tested. The differences between the promised–explicitly or implicitly–and the finally delivered functionality produces customer complains and demands whose satisfaction will have to wait for further releases of the system. This gap in the expectations of software systems is present during their whole lifetime, from their very first version to the latest one, reflecting the business reality of software development and maintenance. Such reality is strongly driven by a ferocious competence for the market share, the concept of time-to-market, vapor-ware products, seasonal factors, or the inevitable shareholders that quarterly demand good financial numbers.

Another point of difference between the first and sixth laws it that while the fist describes the perpetual state of change in software systems, the sixth law makes an explicit mention not to change, but to growth of functional content. There is a straightforward logical connection between the previous description of the sixth law and how it implies increments in the functional content of the software system. On the one

hand, the delivery of missing or incomplete functionality implies that part of the work that could not be completed in a given version, will be delivered in future versions, adding an extra to the new functionality specifically planned for those versions. On the other hand, the gap between expected and released functionality affects to every release of the software system. Together, both facts represent a force that pushes for an increment of the amount of functional delivery during the whole life of the software system.

As a final thought about the sixth law, the opinion of this writer is that although the fact that support the law, that software systems fail to fulfill the promises they made, is a well-known fact of the industry, perhaps promoting this phenomenon to the category of law is a bit hasty at this point of time. No further proof is presented to this respect here or in later papers. In addition, the principle behind this law does not introduce striking new evidences. Both sixth and first law have similar conclusions and the second law already makes a connection between economic factors and their importance in the evolution of software systems.

A last consideration, again the opinion of this author, is the gloomy landscape this law pictures. It is gloomy that software evolution, a part of software engineering, contains a universal principle–a law–that states that software organizations never deliver the promised functionality at any given release of the software system or product. The broken promise is based on the inability of software organizations to deliver the planned job in the planned schedule and allocated resources. The evidence from years of experience of failed software project supports Lehman opinion, but this is something that software engineering researchers and practitioners are trying to correct. Should Lehman be proven right, every software organization would be condemned to produce incomplete, unsatisfactory products, without regards of advancement in the field. Efforts of researchers trying to improve the field of software engineering would be naïve and futile.

The Seventh Law: Declining Quality

How the changing environment affects the way a software system is perceived was described as a consequence of the first law of software evolution. Now, this effect is enunciated in a law of its own. The seventh law states that software systems are perceived as of declining quality unless work is spent to maintain and update the system. The reason for this changes in how the system is perceived lies again in the changing environment. Changes in the definition of the problem that a software system is intended to solve can invalidate assumptions embedded in the system during its development and maintenance. In addition, competence pressure and advances in the technology produce alternative solutions that raise the bar of customer satisfaction. Therefore, a system that is not maintained is perceived as of declining quality not because it changes wearing with time, but precisely because it does not change at all.

This phenomenon has been described in previous chapters, so no more emphasis will be done at this point. A final consideration is that no further proof, empirical or of any other type, is presented to sustain the principles the law is based on.

The Eighth Law: Feedback System

Previous laws, in particular the third law, exposed the systemic nature of software systems' evolution and presented some of the feedback loops that regulate the system. The eighth law comes in this revision to synthesize the whole concept in a law of its own, highlighting the importance of the result. The discovery of systemic behavior in software evolution processes is perhaps one of the most important results derived from thirty years of studies in the field. It must be remembered that the second revision of the laws enunciated the third law as "the fundamental law of program evolution," but the next version changed some of the law's names (mainly to give consistence to the names, naming the laws by their underlying principle and moving any further comments and thoughts to the explanation of the law). Thus, the third law was renamed as "self-regulation," and relegated to a poor-visible place in the middle of the set of laws. Now, the eighth law, the last of the laws of software evolution, closes the series summarizing

the principles presented in previous laws and stressing the importance of the systemic approach to software systems.

The major evidence presented to support the law is again the evolution of the OS/360 system, in particular how the system's size (measured by counting its number of modules) evolved during its lifetime, including 26 major releases. The smooth long-range trend and the ripple superimposed on the trend are indicators of a system regulated by negative feedback loops. The presence in the graph of large oscillations after a release introduced a large amount of functionality resembles a dynamic system that enters into an unstable state after a stimulus perturbed the system. This could be an indication a positive feedback loop. Altogether, both patterns observed in the graph qualify the software evolution processes as a complex system with multiple feedback loops. Later studies presented evidence of the same patterns on different software systems (Turski 1996; Lehman et al. 1997; Shepperd 2000).

Fourth Revision of the Laws of Software Evolution (2001)

This is the latest revision of the laws published so far. No major changes, such as a new law, a change in the name or in the definition of a law are presented. However, a fine alteration on the fifth law appears at this revision, and the derivations are significant enough to deserve further analysis. Table 2.5 presents a summary of the last definition of the laws of software evolution.

The inverse square model, introduced in 1996, proposed a trend for the growth of software systems that gets slower and the system grows in complexity, measuring the complexity of the system as proportional to the square of the system's size (Turski 1996). However, the fifth law described in previous revisions, understands the growth of software systems as statistically invariant. By statistically invariant, Lehman means that the incremental growth as a constant average over the system's lifetime. The incremental growth of a system cannot have both a declining average–as the inverse square model says–and a constant average, as the fifth law supported in previous releases, so one of the two theories must be wrong. The former definition of the fifth law was strongly

supported by the study of the IBM OS/360, which exhibits an almost perfect linear long-term growth during the more than twenty releases analyzed (it has been presented how an incremental growth with constant average implies a linear growth trend).

Table 2.5. Laws of Software Evolution as of 2001.

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

Source: Lehman and Ramil (2001).

The inverse square model is supported by empirical evidence of many software systems (Lehman et al. 1997, 2001), and it offers a more intuitive theoretical base. If the

linear model were the natural way a software system growths, then given a large enough number of releases, the system can be arbitrarily large. This is very counter-intuitive since any other known human activity presents limits to growth: intuition says that systems become increasingly difficult to work with, as they grow bigger. There should be some sort of upper boundary to the size of software systems that can be developed as a result of human activities, and the growth of any software system should converge asymptotically to that value. Since the linear model is not bounded, it must be wrong; although it is yet to be proven that the inverse square model is truly bounded.

Previous contradictory results over the IBM OS/360 system are revisited in Lehman and Ramil (2001) and Lehman et al. (2001). The first study qualifies the growth of the system as excessive; causing the chaotic patterns experimented during the late releases of the system. This explanation does not negate the linear growth model as valid. Software systems would be allowed to grow at linear rates, faster than an inverse square increment, but doing so, it would risk the future stability of the system. The second study focuses on a different fact: late releases of the IBM OS/360 system were more separated in time than the early versions were. This is extremely important, since the pattern of grow may change completely from one scale to the other. As Lehman supports, the OS/360 system growths at an almost linear rate in the release-sequence scale, but at a slower rate in the true-time scale, a rate that would be approximated by the inverse square model. Advantages of the release-sequence over the true-time scale were highlighted in (Turski 1996). It seems contradictory to the author of the present study to excuse the OS/360 explaining how the inverse square model could fit the OS/360 dataset in a true-time scale, since the inverse square model is precisely defined in a release-sequence time scale, as it will be explained in depth during the description of the model.

The bottom line is that superior empirical evidence and better foundations promoted the inverse square model as the dominant model of software system's growth and the fifth law of software evolution was modified in this revision to match the model's theoretical foundations. The new formulation of the law states that "in general," the incremental growth and long-term growth rate of evolutionary software systems tend to

decline. The definition is not excluding and ambiguous enough to accommodate "generally" declining growth rate–inverse square model–and "exceptionally" other patterns of growth, such as a linear or any other imaginable combination.

The FEAST/1 and FEAST/2 Projects

First results gather a great deal of interest and as a result the FEAST/1 and FEAST/2 projects (Lehman 1998a, 1998b, 1998c, 1998d, 2001; Ramil et al. 2000) were initiated to allow a further exploration of this emerging field of research. FEAST/1 and FEAST/2 helped to found Lehman's team during the nineties, and progress were made derived from these initiatives. The effort was directed to two main fronts, each benefiting from advancements in the other: white box and black box modeling of evolutionary software processes.

White Box and Black Box Approaches

White Box

The goal of the white box approach is the identification of the internal system dynamics involved in evolutionary software processes. Efforts on this area try to develop the simplest models that could produce simulation of software systems' attributes analogous to those of the real world (Lehman 1998b, 1998d). Similar studies from other authors are presented in (Humphey and Kellner 1989; Wernick and Lehman 1999; Chatters et al. 2000). The most comprehensive models in the literature and obliged reference (Abdel-Hamid and Madnick 1989; Abdel-Hamid and Madnick 1991) were discarded by Lehman because of their high degree of complexity.

Black Box

The black box approach seeks the mathematical equations that model the evolution of software systems' attributes over their lifetime and how they change from release to release of the system. Many efforts have been undertaken in this area, modeling different software metrics and correlations (Boehm 1987; Kemerer and

Slaughter 1999; Hall and Munson 2000), but researchers in the field seem to have concentrated on the study of the size and size-related metrics and how they evolve over the lifetime of the system. One of the major accomplishment is the development of the inverse square model (Turski 1996), delivering accurate predictions of future releases' size over some software systems analyzed to this effect (Lehman et al. 1997). However, later publications presented evidence that contradicted the universality of the model (Shepperd, Godfrey, and Tu 2001), showing some weaknesses of the model. As a result, Lehman introduced a complementary model based on a constant work rate assumption (Lehman et al. 2001) whose repercussions will also be analyzed. Ideas that led to the constant work rate can be find in (Shepperd, Godfrey, and Tu 2001).

Kemerer and Slaughter (1999) offer a detailed study of the evolution of a software system. Changes in the code at every release of the system are classified according to their type (corrective, adaptive or enhancement/perfective as the main categories) and a statistical approach is used to determined correlations among the type of changes. Although the results are promising, no other similar study has been performed yet, probably due to the amount of effort required to classify the thousands of code changes that can be found in the history of any large software system.

Hall and Munson (2000) present results of the evolution of an industrial software system. The evolution of system's complexity is measured using the "code delta" and "code churn" metric, which measure the difference of code between two consecutive releases and the amount of code modified during two releases, respectively. A release that adds and removes the same amount of code will measure zero in the code delta metric, but the code churn metric will not be zero. The approach is innovative, but it did not arouse further interest in the literature.

Software Evolution Smooth Growth Models

The Inverse Square Model

The inverse square model of smooth software system evolution (Turski 1996) represented a major breakthrough in the software evolution field. It was the first time that a non-trivial model could accurately fit the long-term trend of a software system's size during its evolution. Moreover, it was solidly founded in previous results from the field (laws of software evolution, empirical studies of software systems) and it exhibits excellent forecasting properties, as will be presented later.

Description

Since the 1980's revision of the laws of software evolution, the fifth law is defined as the law of conservation of familiarity. The law implies that the long-term trend of growth of evolutionary software systems is statistically invariant. Statistically invariant means that the average of the system size is remarkably constant during system's lifetime (Lehman and Belady 1985b), and this signifies that software systems should exhibit a linear trend when the number of modules from every release is plotted against the release sequence number. However, when Tursky tried a linear fit over data published by Lehman on a previous paper, the fit was not completely satisfactory. The result suggested to Tursky that the increment of the system was getting slower as the system's release advanced, in contradiction with an almost constant growth suggested by Lehman.

With this evidence, Tursky decided to develop a new model able to explain this dataset. Two pillars serve as foundations for the new model:

1. Software systems evolve with a constant effective work rate. This idea is derived from the third law of software evolution, which understands evolutionary software systems as self-regulatory, with statistically invariant attributes, such as the effective work rate.
2. As a software system becomes more complex, its growth slows proportionally. In other words, the incremental growth rate of a system is

inversely proportional to its complexity. This is based on the interpretation of the dataset mentioned above, which exhibits a declining rate of growth as the system becomes bigger.

It is straightforward to derive an equation based on both principles:

$$\hat{S}(i+1) - \hat{S}(i) = E/\hat{S}^2(i), \tag{2.1}$$

where $\hat{S}(i)$ is the estimated size of the software system at release number I (measured as the total count of modules of the system), and E is a constant. Then, $\hat{S}(i+1) - \hat{S}(i)$ is the estimated increment of the system's size from version i to the next version. $\hat{S}^2(i)$ is an indication of the system's complexity. As it is presented in the section " measuring a system's size," the complexity of a system is approximated by the number of relationships among its modules, and this number is a function of the square of the number of modules of the system. The constant E plays the role of a merit or productivity indicator of the software organization. The bigger its value, the greater the amount of functionality that an organization can deliver at a given release. E is considered constant during the whole lifetime of software systems due of the first principle, which defines the amount of effective work rate as statistically invariant, therefore, constant on the long-term.

Reading the equation again, it says: the incremental growth of a software system is inversely proportional to the system's complexity, which is the combination of the two principles presented above.

Eq. 2.1 is a recurrence relation and it would not be completed without the first term:

$$\hat{S}(1) = s(1), \tag{2.2}$$

where s(1) is the actual size of the system at it first considered release. A dataset composed of n versions will produce n equations:

$$\hat{S}(1) = s(1)$$
$$\hat{S}(2) = \hat{S}(1) + E/\hat{S}^2(1)$$
$$\hat{S}(3) = \hat{S}(2) + E/\hat{S}^2(2) = \hat{S}(1) + E/\hat{S}^2(1) + E/\hat{S}^2(2)$$
$$\ldots$$
$$\hat{S}(n) = \hat{S}(n-1) + E/\hat{S}^2(n-1) = \hat{S}(1) + E/\hat{S}^2(1) + E/\hat{S}^2(2) + \ldots + E/\hat{S}^2(n-1). \quad (2.3)$$

Each equation can be solved and E determined. It is cumbersome because every equation produces a different solution for E, therefore it is more precise to name E as a variable: E(2),…,E(n), depending of the release number. Now, replacing the estimated values $\hat{S}(i)$ by the actual values s(i) and minding that each equation produces a different value of E, Eqs. (2.3) are rewritten as:

$$s(2) = s(1) + E(2)/s^2(1)$$
$$s(3) = s(1) + E(3)/s^2(1) + E(3)/s^2(2)$$
$$\ldots$$
$$s(n) = s(1) + E(n)/s^2(1) + E(n)/s^2(2) + \ldots + E(n)/s^2(n-1), \quad (2.4)$$

which are easily solve in terms of E(2),…E(n)

$$E(2) = (s(2) - s(1))(1/s^2(1))$$
$$E(3) = (s(3) - s(1))(1/s^2(1) + 1/s^2(2))$$
$$\ldots$$
$$E(n) = (s(n) - s(1))(1/s^2(1) + 1/s^2(2) + \ldots + 1/s^2(n-1)), \quad (2.5)$$

or

$$E(i) = (s(i) - s(1))\sum_{j=1}^{i-1} 1/s^2(j), \qquad \{i=2,\ldots,n\} \quad (2.6)$$

Finally a constant $\underline{E}$ is calculated as the average of solutions E(2),…,E(n):

$$\underline{E} = \frac{1}{n-1}\sum_{i=2}^{n} E(i) \tag{2.7}$$

Thus, the final form of Eq. (2.1) and (2.2) is:

$$\hat{S}(i+1) - \hat{S}(i) = \underline{E}/\,\hat{S}^2(i), \tag{2.8}$$

$$\hat{S}(1) = s(1) \tag{2.9}$$

There is an alternative method consisting of choosing $\underline{E}$ as the value that produces the best possible fit of $\hat{S}(1),\ldots, \hat{S}(n)$ over the actual dataset s(1),…(n), using Eq. (2.8) and (2.9). The best possible fit is calculated as the fit that minimizes the mean absolute error between the original dataset and the output of the model. Any min-max algorithm should work to this effect. This method is preferred because produces an optimal fit, and therefore the best possible estimate the model can offer. It must be pointed out that although the analytical approximation using the average of the n solutions of E(i), described before, produces results close to optimal.

Advantages and Disadvantages

Further studies that applied the inverse square model over dataset obtained from different software systems, confirmed the success of the inverse square model (Lehman et al. 1997, 2001; Lehman 1998b; Lehman and Ramil 2001). Early results of the OS/360 system (remember that this system exhibited an almost linear growth, without symptoms of declining increments as the system grows) were reinterpreted in a different way, as it will be presented later.

It is also important to highlight how the inverse square model seems to be operating in a pseudo-temporal scale. Eq. (2.8) defines how size increments are related to sizes of previous releases; no specific mention is made to the period of time that

separates one version to the next. It seems obvious that a version whose production lingers over an extended period will deliver more functionality that a shorter version. Therefore, there seem to be an implicit relationship between the release's size (proportional to the amount of functionality included in a release) of a software system and the duration of the preparation of that release. Then, the inverse square model, by taken into account releases' size, it would be implicitly operating in a real temporal scale. The importance of the temporal scale used to calculate the growth patterns of software systems will be used later to explain how the original results of the OS/360 were reinterpreted in a recent paper (Lehman et al. 2001), as mentioned in the previous paragraph.

The strongest point of the inverse square model is found in its forecasting capabilities. Tursky shows how the results from the inverse square model using only information from the first six versions of the system under study produces an estimate almost as good as the results produced using the whole dataset (twenty releases). The mean error of the prediction is inside the band of the 10% of deviation over the minimum mean error. Moreover, predictions using half of the dataset are off the best prediction by just a few percent.

Since information coming from a few, early releases of a system can be used to accurately forecast system's attributes over a large number of releases, then, the implications are clear to Lehman: system dynamics are strong and established from the beginning of the system's lifetime. The evolution of a software system during its early stages strongly determines its future evolution, without regard of managerial, technical, or business decisions taken during the development and maintenance of the system. Characteristics of the system, organization, and environment solidify during the first releases of the system and determine dynamics that control how the system will evolve in its future. The result was striking and counter-intuitive to many, but not to Lehman. This result comes to confirm hypotheses reflected in his laws of software evolution and validating what was said since the late sixties about software systems:

Rather than the manager managing the (evolving software) system, the system manages the managers (Lehman et al. 1997, p. 27).

However, it is important to point out that some of the laws were contradicted by the results derived from the inverse square model. The fifth law, which implies a linear growth of software system, is undermined by these results, since the inverse square model definitely supports a declining growth rate during the evolution of software systems. This may be the reason why the fifth law changed its formulation in 2001 to what is now its latest definition, which states that, in general, the incremental growth of software systems tend to decline (Lehman and Ramil 2001).

Another point that deserves consideration is the meaning of the "E" constant in Eq. (2.1). The constant acts as the proportional term in the incremental grow formulate and therefore it may be considered as a productivity measure or merit factor of the organization. The bigger the value of E, the greater the functionality increment that the organization can put into the next version of the system. E is considered as a dimensionless parameter, but if dimensions were considered, it should be measured in terms of the cube of the number of modules, so both terms in Eq. (2.1) would match. It seems difficult to find a qualitative explanation for the meaning of such unusual dimension, and so far, this point has been disregarded by researches as a minor issue.

Comments

In the opinion of this author, the inverse square model represented a major breakthrough in the software evolution field, delivering a tool of practical importance. Its usefulness relies mainly as a forecasting tool, used in the planning and management of software system's evolution. Results mostly confirm the knowledge gained during the two previous decades of software evolution studies. However, there are a number of minor loose ends in the models that, altogether rest some charm to the theory: the way that constant E is calculated; contradictions with the fifth law, which, in its turn, was based on empirical evidence; and the unusual dimensions of the constant E, point so far unexplained in the literature.

The Constant Work Rate Model

The inverse square model has been used by Lehman and his collaborators on many different software systems, delivering, in general, good fits and excellent forecasts. As the model become more popular, other investigators have used it on their own studies of the evolution of software systems. In 2001, M. Godfrey and Q. Tu publish a study of the evolution of the Linux kernel over a period of eight years. The data shows that this system has growth at an exponential or super-linear rate, which means a rate faster that a linear rate would do. This would contradict the basic principle that supports the inverse square model. The principle says that as systems become bigger, their complexity increases exponentially and the growth rate of the system declines in the same proportion. A linear growth cannot be modeled by the inverse square equation and much less a super-linear model.

Different papers have presented alternative models based on exponential or power-law relations. Shepperd (2000) proposes an exponential recurrence relation, given by the equations:

$$S(i) = S(i-1) + R(i)$$
$$R(i) = R(i-1)^{e}, \qquad (2.10)$$

where $\hat{S}(i)$ is the estimated system's size at version i, R(i) is the release size again at version i, and e is a constant calculated to maximize the fit.

Godfrey and Tu (2000) are the first to present how the inverse square model can be approximated by the equation:

$$S(i) = (3\ E\ i)^{1/3}, \qquad (2.11)$$

where S(i) is again the estimated system's size at version i, and E is a constant. However, after an unsuccessful application of the inverse square, the model is neglected and a polynomial fit is presented as a better option.

Finally, in a recent paper, Lehman (2001) announced the extension of the inverse square model, complemented now by a new model based on the equation:

$$\hat{S}(i) = i^{1/g}, \tag{2.12}$$

where $\hat{S}$ is the estimate of the normalized system's size at version i and g is a constant ≥ 2. The distinction that g must be greater than two implies the assumption of systems the grow at declining rates, in agreement with Lehman's laws of software evolution.

All these ideas, plus the results presented in the previous sections, which show how an exponential equation such as

$$\hat{S}(i) = a\, i^{b}, \tag{2.13}$$

can produce good fits on most of the analyzed examples, plus the assumption that an exponent b in Eq. (2.13) without restrictions on its value can be used to model systems with sub-linear ($b < 1$), linear ($b \sim 1$), and super-linear ($b > 1$) growth trends, altogether seem enough reasons to explore a new model based on this equation.

It must be pointed out that Eq. (2.12) and (2.13) are almost equivalent. In a logarithmic scale, the multiplier "a" has the effect of just a displacement along the vertical axis.

Towards a Final Theory of Software Evolution

Finally, work from the FEAST/1 and FEAST/2 is compiled in Lehman and Ramil (2000), setting the roadmap for investigators seeking the final theory of software evolution.

CHAPTER III
METHODOLOGY

Measuring the Size of a System

There are many different possible ways of measuring a system's size. The most popular in software evolution are: counting lines of code (LOC), counting function points (FP) or counting the number of modules. Each of them has their own advantages and disadvantages, but for software evolution research purposes, the preferred metric is the last one, the number of modules in the system.

Lines of Code (LOC)

This metric was used originally in the study of the OS/360 (Lehman and Belady 1985c), the starting point of software evolution studies. Actually, the number of instructions was the metric used, but the OS/360 was developed using pure assembler; and for the assembler language, the count of instructions and LOC are identical. The study shows how the trend of the number of instructions and total count of modules follow a very similar trend. Further studies considered both metrics as similar, although the number of modules is preferred since, a module has functional integrity but lines of code does not (Lehman et al. 1997). The LOC metric is subjected to implementation details that have nothing to do with functionality. The interpretation could be that both metrics follow the same general trend, but LOC would be to some extent blurred, noisier than the count of modules. Concerning the inverse square model of software system's growth, it can be assumed that both metrics are interchangeable, since the model deals with the general trend of the system's growth and is resistant to any ripple superimposed, an inherent phenomenon in the evolution of software systems.

Lines of code also have some other virtues: is very straightforward to calculate and it is widely available, as long as the source code of the software system under study can be recovered.

Another problem is that software systems may be composed of modules developed using different languages, common practice in large systems. A large system

will be composed of different sub-systems arranged in some hierarchical fashion. Sub-systems aim for different goals, such as performance, extensibility, or usability; and therefore each of them could be better served by a different languages or technologies, taking the best of each world. Anyway, a mixture of languages should not represent a problem, since there are studies that publish language scale factors that can be used to normalize tallies and produce a total amount of LOC. For instance, (Jones 1995) presents the Backfiring method, that can be used to convert LOC metrics of a variety of languages to function points.

Function Points (FP)

The count of function points is probably the best indicator of the amount of functionality contained in a given release of a software system. Function points are counted by a careful analysis of the system, its documentation (requirements, design, architecture) and interviews with the development/maintenance members, should part of the documentation be unavailable. Individual functionality is counted following a well-defined standard. The IFPUG (International Function Points User Group) facilitates a common framework where raters can communicate, train, and unify criteria, so FP counts obtained by different raters could be reasonably compared (Garmus and Herron 2000).

There are important drawbacks that must be taken into account, though. First of all, the counting of function points of a software system is an intensive human-based process. Thus, results are subjective, depending on the judgment of the analyst. Even though the FP measurement process is standardized and well defined, the comparison of results cannot obviate deviations due to the particular characteristics of the system, the rating process, and raters. Results from similar systems developed or maintained by the same organization and prepared by the same team of raters, following well-defined guidelines, have an important value because relative differences among the FP counts can be trusted. On the other hand, results coming from different systems, at different organizations, or prepared by different raters, are difficult to compare. Absolute results have an unknown degree of accuracy and comparison can be questioned.

A function point measurement is costly, cumbersome, and for its very nature, difficult, if not impossible, to automate. The FP count cannot be considered a widespread practice in the industry, only a limited number of organizations have the resources or the interest to invest in such measurement. As a result, few software systems have collections of FP counts for every release of the system, or at least a number of releases large enough to be worthy in an empirical study. Software evolution investigators have the option to undertake the measurement process, but again, this is costly and sometimes even impossible, since release documentation may be totally lost or the whole design/ development team may be out of the organization. Moreover, the effort of counting FP of a given release must be multiplied by the number of releases that researchers want to consider in their study of the evolution of a software system. The presence of easier measures of a software system's size makes the FP metric of little practical interest, at least from the point of view of software evolution.

Number of Modules

The number of modules in a software system is considered a good estimate of the amount of system's functionality. Although the number of modules is used in the seminal study of the OS/360 (Lehman and Belady 1985c), further studies contributed later to sustain the consistency of this metric. A later paper (Lehman et al. 1997), analyzes some of the reasons that can support this fact and are presented in the following paragraphs.

Lehman's considers that the number of modules in a system is an attribute that emerges spontaneously, since it is controlled neither by any management decision nor by any programming practice. Modules have a different meaning depending on the specific language or technology used in the development of the system. A module can be, for example, a function in C, a class in C++ or Java, a procedure in Pascal or Basic, a component in Microsoft COM technology, or an ASP file in Web applications. However, without regard of the language or technology, the significant part is that a module

contains functional integrity, and therefore is a good estimate of the amount of system's functionality.

From a practical point of view, the number of modules is straightforward to calculate, can be fully automated and is available on every release of software systems where the source code is available.

There is another advantage of this metric for software evolution studies: the number of modules of a system can be considered as a good estimate of the system's complexity. The idea of complexity here is connected with the coupling concept, which has deserved studies since the early times of the structured programming (Parnas 2001).

Coupling refers to how well a module (source file, function, class, procedure) is self-contained and independent to the internals of other modules. A loose coupling is a desirable design quality of a system: the lower the coupling, the easier modules can be used and the lower the significance of side effects. Side effects are produced when local changes over a module (defect fixing, functional changes) lead to changes in other modules, which are not directly related to the cause of the original change. The maintenance and evolution of a system with a high coupling metric can be a real nightmare for the organization. Since complexity in a software system is understood as the factor that undermines the ability of the system to evolve, then, coupling can be considered as a good estimate of the system's complexity.

Briefly, a module's coupling is measured by counting the number of dependences of this module with other modules. A system, composed of n modules, has an upper limit for the number of connections of every module equal to n-1 (every other module). Since there are n modules, the total number of possible connections is n(n-1) (Figure 3.1).

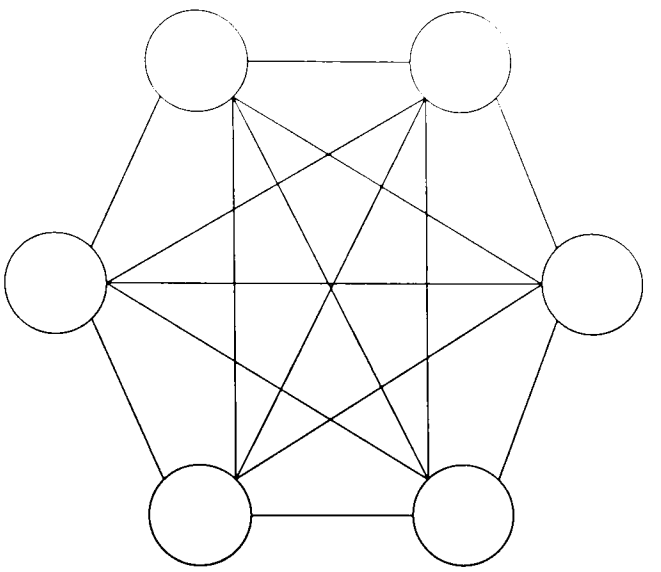

Figure 3.1. Number of modules and complexity.

CHAPTER IV
RESULTS

An Evolution Example: QUICK

One of the motivations for the present study was the exploration of the evolution of the QUICK application under the current theories of software evolution. QUICK presents an excellent opportunity for many reasons: It has been locally developed at Texas Tech and therefore the source code of every release is available, the documentation was available as well, many of the people primarily responsible for the project are still at the institution and available for consultancy (although the developers–former students– are mostly gone), and finally the application has been around for some years and has experienced eight releases to date, although only six will be used in the present study. The two releases discarded in the present study are the first release, because it is just a prototype developed in a different language (C++) than later releases (ASP), and the spring 2000 release, because its source code is not available in the configuration management system.

QUICK is a web-based system that facilitates students at Tech the course's section registration process before the semester starts. Students choose the courses they wish to take during the next semester and QUICK automatically retrieves the open sections for those courses and prepares an schedule so no two sections overlap during the same period of the same day, nor do any section conflict with work or other activities. After students agree with the schedule, they can proceed to register to the proposed sections. The first version of QUICK (actually a prototype) goes back to the fall of 1998. QUICK development has been supervised by Texas Tech software engineering instructors: Dr. Donald Bagert and Dr. Susan Mengel, as well as Dr. Cherry Owen a former PhD student. QUICK has been developed and maintained by successive generations of students from software engineering courses, therefore the personnel has been rotating almost completely every semester. The project is developed and maintained with an emphasis in the quality of the documentation and good software engineering practices. The project has been in production from some releases ago and is

currently used by a significant portion of the Texas Tech student population, especially in scheduling classes during their first semester of study.

QUICK has been developed as a web application, using a mixture of HTML and dynamic HTML technology implemented with Microsoft ASP (Active Server Pages). The project also includes an auxiliary tool that imports table files from the central Texas Tech servers to the QUICK server using FTP (File Transfer Protocol), and has been implemented in Microsoft Visual Basic. The technology and approach followed in the QUICK project are representative of what the industry is doing today for this kind of applications. QUICK can be considered as an excellent representative of a group of small web applications used today at corporate portals or as part of e-business systems. Thus, the exploration of the evolution of QUICK can contribute to a deeper understanding of this group of systems.

Analysis of QUICK Releases

Table 4.1 presents a summary of the modules (source files) included on the first known version of the QUICK application. It should be considered more as a working prototype developed in C++, and for these reason this version will be excluded in future analysis.

Table 4.1. QUICK: Fall 1998 release.

Module Name	Language	LOC	Size (bytes)
advise.cpp	C++	1157	36407

Table 4.2 and Table 4.3 present summaries of the next releases of the application, here recoded as a web system using the ASP technology. Version of fall 1999 includes the small utility to transfer tables' data from the corporate servers to the local QUICK server.

Table 4.2. QUICK: Spring 1999 release.

Module Name	Language	LOC	Size (bytes)
advise-t.asp	ASP	1162	42922

Table 4.3. QUICK: Fall 1999 release.

Module Name	Language	LOC	Size (bytes)
advise-t.asp	ASP	2342	78980
main.htm	HTML	1673	80580
copy.form1.frm	Visual Basic	156	4319

Although there is an entry in the QUICK's configuration management system for the spring 2000 release, no source code is available, so this version has to be excluded from further analysis.

Table 4.4 covers the summer II release, which represents a large increment in the system. Probably most of the functionality was developed during the previous term.

Table 4.4. QUICK: Summer II 2000 release.

Module Name	Language	LOC	Size (bytes)
advise-t.asp	ASP	2615	97041
activities.asp	ASP	481	13510
begin.asp	ASP	120	2527
classes.asp	ASP	435	16142
navigate.asp	ASP	114	3815
main.htm	HTML	945	33930
copy.form1.frm	Visual Basic	154	4319
loaddb.loaddb.bas	Visual Basic	170	4923

Table 4.5 and Table 4.6 covers releases of fall 2000 and spring 2001, the latest version analyzed in this study.

Table 4.5. QUICK: Fall 2000 release.

Module Name	Language	LOC	Size (bytes)
advise-t.asp	ASP	2615	97041
activities.asp	ASP	481	13510
begin.asp	ASP	120	2527
classes.asp	ASP	421	15765
displaysource.asp	ASP	50	1397
interactivedirectory.asp	ASP	69	1530
summer2main.htm	HTML	944	33874
copy.form1.frm	Visual Basic	154	4319
loaddb.loaddb.bas	Visual Basic	170	4923

Table 4.6. QUICK: Spring 2001 release.

Module Name	Language	LOC	Size (bytes)
advise-t.asp	ASP	2622	97435
activities.asp	ASP	485	13633
begin.asp	ASP	120	2536
classes.asp	ASP	458	17269
displaysource.asp	ASP	50	1397
interactivedirectory.asp	ASP	76	1661
appvartest.asp	ASP	58	1639
datetest.asp	ASP	10	94
global.asp	ASP	36	747
newnavigate.asp	ASP	35	1064
quicktoc.asp	ASP	38	945
work.asp	ASP	947	25165
summer2main.htm	HTML	944	33874
copy.form1.frm	Visual Basic	154	4319
loaddb.loaddb.bas	Visual Basic	170	4923

Table 4.7 presents a summary of the total number of modules per version, the total count of LOC separated by language and the total source code size, measured in bytes.

Table 4.7. QUICK: Summary I.

Release	Modules	C++ LOC	ASP LOC	HTML LOC	VB LOC	Total LOC	Total bytes
Fall 98	1	1157	0	0	0	1157	36407
Spring 99	1	0	1162	0	0	1162	42922
Summer II 99	1	0	2342	0	0	2342	78980
Fall 99	3	0	2343	1673	156	4172	164542
Summer II 00	8	0	3765	945	324	5034	176207
Fall 00	9	0	3756	944	324	5024	174886
Spring 01	15	0	4935	944	324	6203	206728

Finally, Table 4.8 offers an estimated release date for every release and the count of days from every release to the first release of fall 1998.

Table 4.8. QUICK: Summary II.

Release	Release date (aprox.)	Release day (aprox.)
Fall 98	12/01/1998	0
Spring 99	05/01/1999	151
Summer II 99	08/01/1999	243
Fall 99	12/01/1999	365
Summer II 00	08/01/2000	609
Fall 00	12/01/2000	731
Spring 01	05/01/2001	882

Figure 4.1 plots the total LOC, including every language, versus the release number. The first release (fall 1998) has been excluded from the plots. The trend is smooth and almost linear, presenting symptoms of decreasing growth rate, confirming some of the laws of software evolution (third law: self regulation, fifth: conservation of

familiarity and inverse square model for growth: decreasing growth rate). However, it must be pointed out that the total LOC mixes together counts of LOC from different languages without any weighting. Although ASP and HTML, and ASP and Visual Basic both share a great deal in common, they cannot be considered as completely equivalents for the purpose of this study.

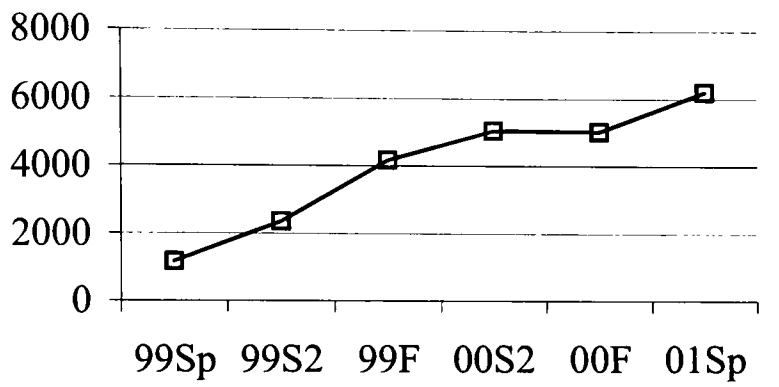

Figure 4.1. QUICK: total LOC per release.

Figure 4.2 presents the same data but this time the horizontal axis represents the point in time when the release was delivered, measured as the count of days passed since the QUICK's first release. This has the effect of moving the spring 99 and Summer II 99 releases closer while separating the fall 99 and Summer II 00 releases. The effect of decreasing growth rate is accentuated by this change of scale.

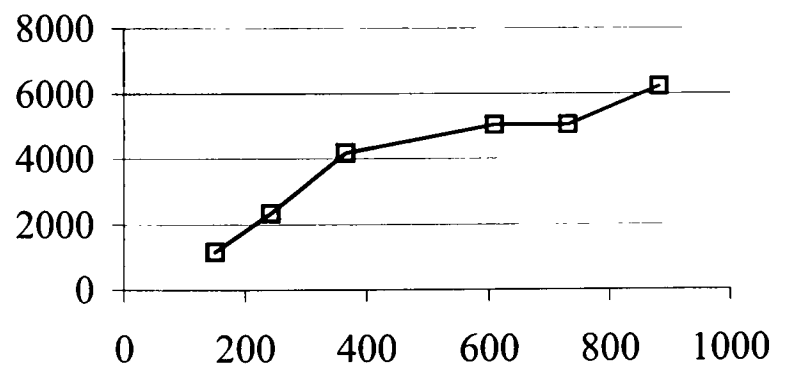

Figure 4.2. QUICK: total LOC per day.

Figure 4 and Figure 4.4 show the evolution of the bytes of source code per release and per day, respectively. The pattern is similar to what has been presented for the total LOC: smooth trend and decreasing growth.

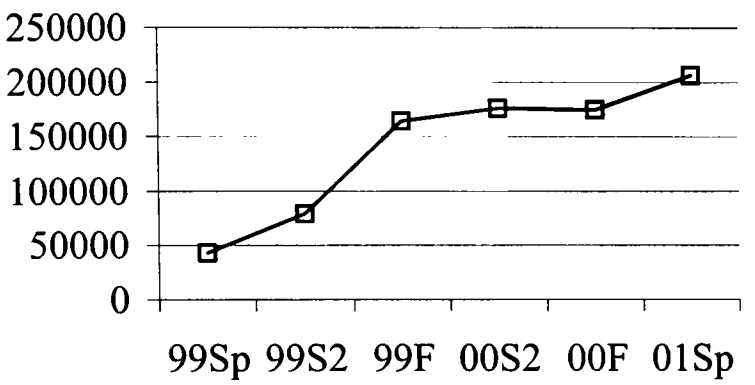

Figure 4.3. QUICK: total bytes per release.

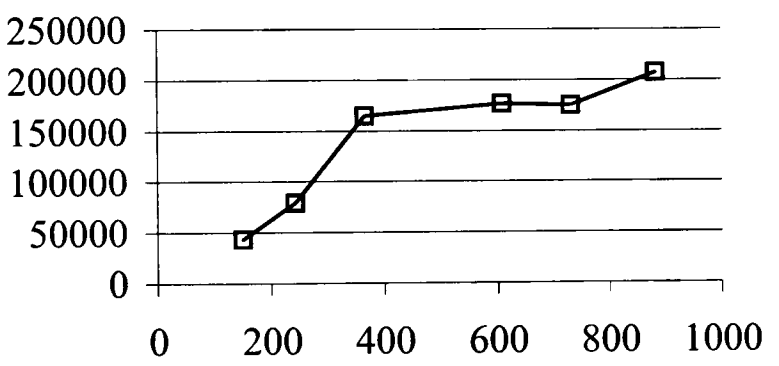

Figure 4.4. QUICK: total bytes per day.

Figure 4.5 and Figure 4.6 represents the total number of modules (source code files) per release and day, respectively. The growth seems to be increasing in later versions, although this result cannot be conclusive due to the small number of modules considered.

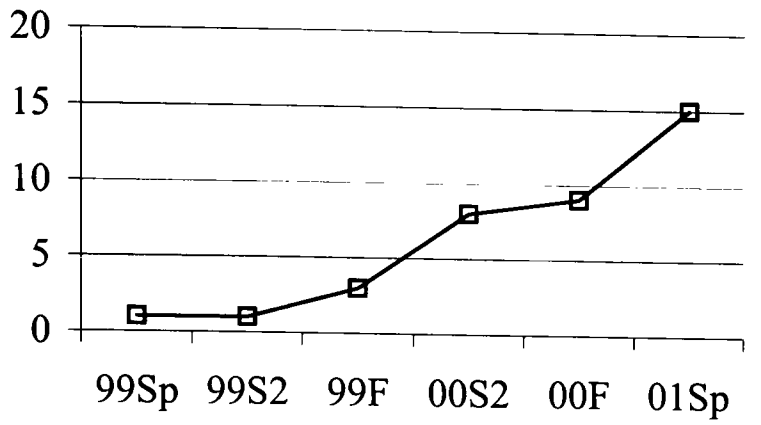

Figure 4.5. QUICK: total modules per release.

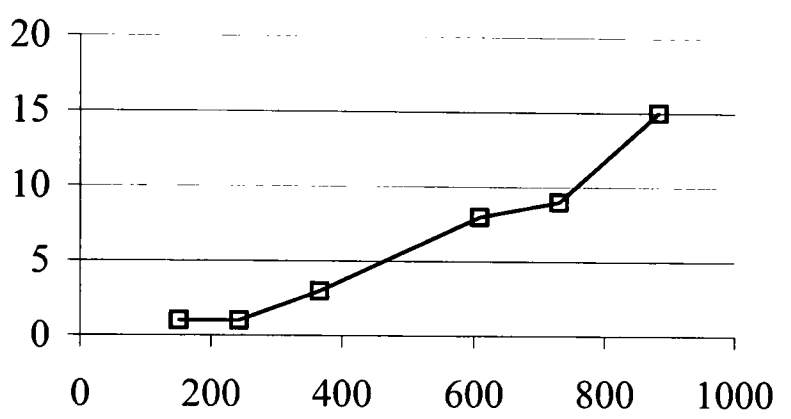

Figure 4.6. QUICK: total modules per day.

Finally, Figure 4.7 and Figure 4.8 represents the evolution of the number of ASP LOC at every version and day, respectively. This result can be considered more reliable than the total LOC presented before, since only one language is considered in the tally. The trend is again smooth and in this case, it resembles more a linear progression.

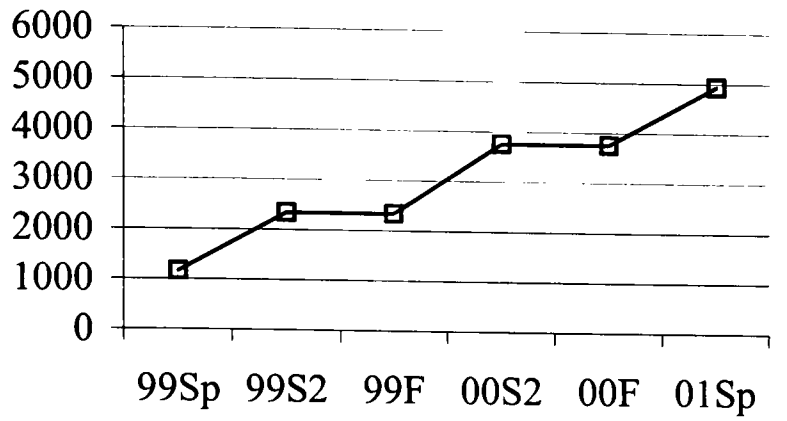

Figure 4.7. QUICK: total ASP LOC per release.

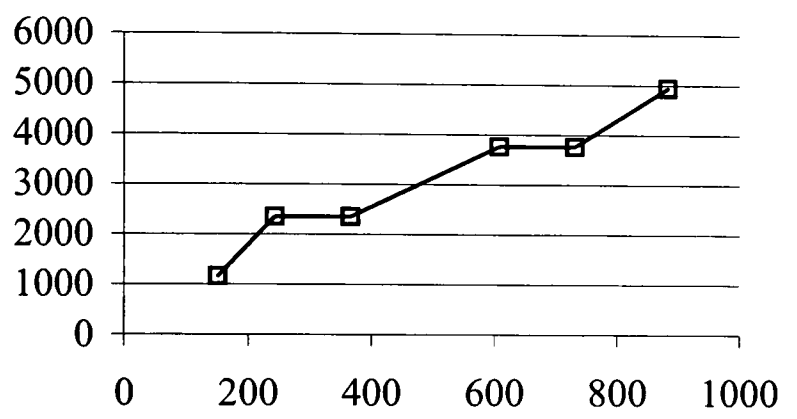

Figure 4.8. QUICK: total ASP LOC per day

The next step consists of the application of the inverse square model over all this data. The goal of developing this model is to (hopefully) produce good estimates for future QUICK releases, information that can be used for planning future maintenance team, resource allocations, or simply to gain a better understanding of the project.

Result of the Inverse Square Model

The inverse square model, described in previous chapters, and defined by equations (2.6) to (2.9) estimates the growth of software systems, using the system's size at as many releases as possible. The system's size can be measured using the count of modules or the count of LOC. In the particular case of the QUICK application, the

preferred metric is the count of LOC developed in the ASP language. The number of modules is also available, but due to the small size of the application, the representativeness of this metric may be questioned. In addition, the patterns shown by the evolution of total LOC and ASP LOC (Figure 4.1 and Figure 4.7) are similar, being the total LOC less preferred since it mixes counts coming from different languages.

Table 4.9 presents the results of equations (2.6) to (2.9), as well as the result of a lineal fit that minimizes the MMRE (Mean Magnitude Relative Error).

Table 4.9. QUICK: Inverse square and linear estimations.

i	s	E	$\hat{S}$	$\hat{S}$ MRE	Linear fit	Linear MRE
1	1162		1162	0	1162	0
2	2342	1.59E+09	3107.704607	32.6944751	2000.51	14.58112
3	2343	1.28E+09	2996.167158	27.8773901	2839.02	29.30726
4	3765	2.64E+09	3609.728714	4.12407133	3677.531	2.323224
5	3756	2.46E+09	3811.351697	1.47368735	4516.041	20.23538
6	4935	3.35E+09	3992.206943	19.1042159	5354.551	8.50154

The final value of E, calculated as the average of every E(i), is 2092342070. The MMRE of the inverse square estimation is 15.58%, and the MMRE of the linear fit, presented for the shake of comparison, is 10.53%. Results are plotted in Figure 4.9.

Figure 4.1 shows the MRE of the inverse square and linear fit estimations, at every version.

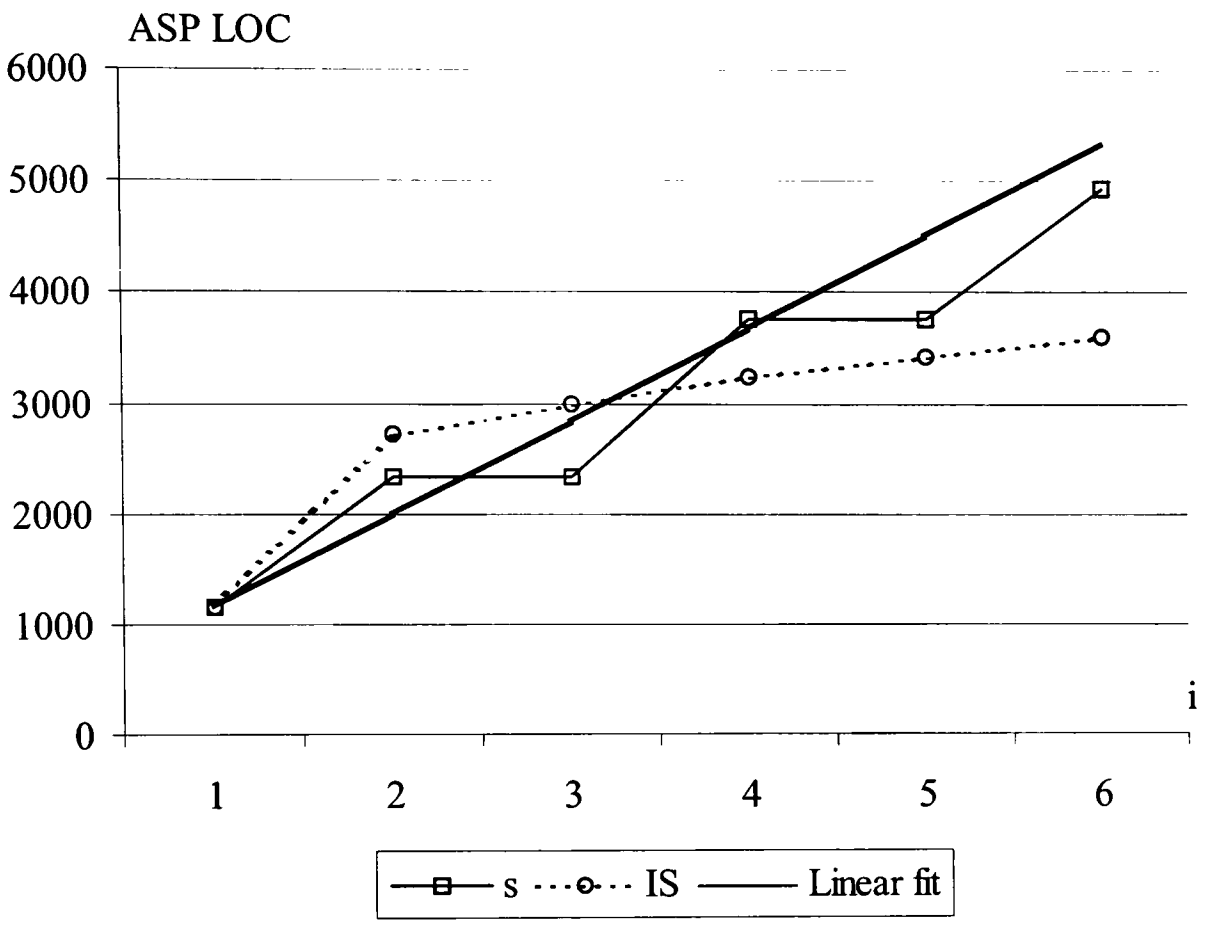

Figure 4.9. QUICK dataset and inverse square estimations.
The actual system's size (measured as the total number of LOC of ASP modules at any given release) is "s", "IS" is the estimates produced by the inverse square model, and "Linear fit" is the estimate produced by a linear fit of the data.

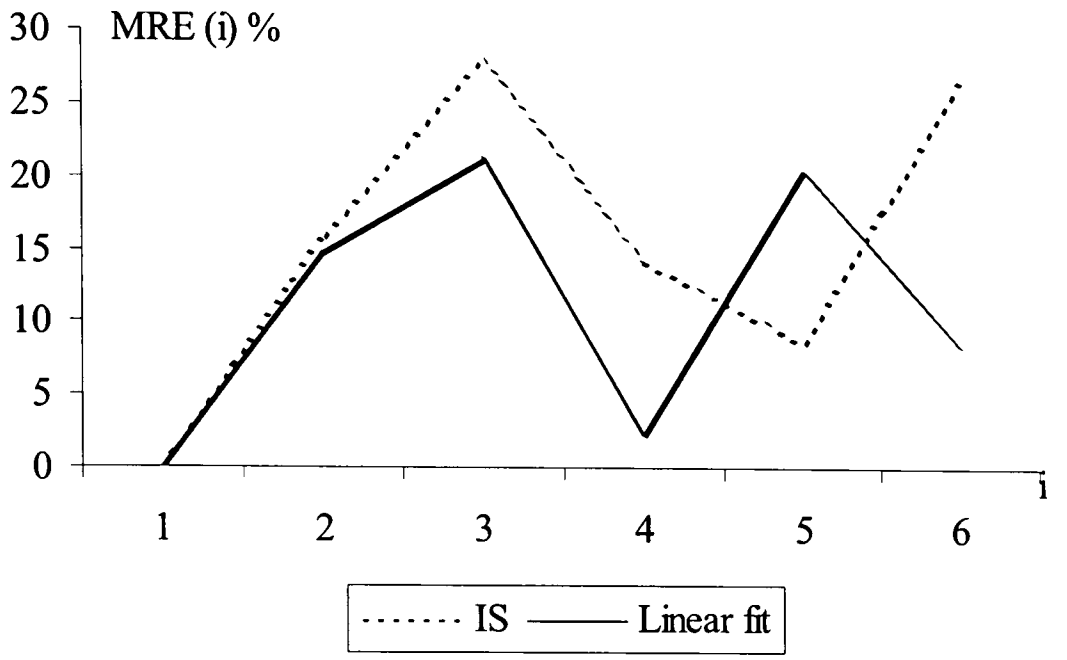

Figure 4.10. QUICK: estimations MRE.
Magnitude relative error of the inverse square (IS) and Linear fit estimates.

The inverse square model produces a fair estimation, it is not as accurate as the linear fit is (using the MMRE as the discriminator). However, the inverse square produces an unusual behavior: a large first step followed by smaller increments forming an almost straight line. The patterns observed in the literature of inverse square model estimations are much smoother that the pattern presented here (Turski 1996; Lehman et al. 1997, 1998b, 2001).

This suspicious effect motivated a closer look at the inverse square model and the mathematical properties of the equations that are part of the model and used to produce the estimations. Next section will present the result of the analysis.

Analysis of the Inverse Square Model

The inverse square model, which has been introduced in previous sections of this paper, defines the long-term growth of software systems by the equation (Turski 1996):

$$\hat{S}(i+1) - \hat{S}(i) = \underline{E}/\hat{S}^2(i), \tag{4.1}$$

$$\hat{S}(1) = s(1), \tag{4.2}$$

which says that the incremental growth of software systems is inversely proportional to the complexity of the system. $\hat{S}(i)$ is the estimated size of the software system at version i, measured usually as the count of modules at that version; $\hat{S}^2(i)$ is proportional to the complexity of the system at version i; s(1) is the actual size of the system at its first release; and $\underline{E}$ is a constant related to the amount of effective work that the organization is able to put into any given version. The third law of software evolution sustains that the effective work rate of a software organization is statistically invariant (in other words, the average is constant or almost constant) over the lifetime of the development and maintenance activities over the software system under consideration.

Eq. (4.1) and (4.2) can be solved in terms of E leading to the following equations:

$$E(i) = (s(i) - s(1)) \sum_{j=1}^{i-1} 1/s^2(j), \qquad \{i=2,\ldots,n\} \tag{4.2}$$

$$\underline{E} = \frac{1}{n-1} \sum_{i=2}^{n} E(i) \tag{4.3}$$

used to calculate the constant $\underline{E}$. $\underline{E}$ and s(1) are the only model's parameters and their value completely determines the model's estimates. Eq. (4.1) and (4.2) represents a recurrence relation whose mathematical properties are difficult to determine beforehand. The next section presents the approach that will be followed during the analysis.

Methodology

For the sake of simplicity, a theoretical software system with a perfect linear growth will be considered during the analysis. A linear growth greatly simplifies the equations of the inverse square model and it is not an estrange case in software systems. This is true in the case of QUICK, and some other examples that can be found in the literature (Lehman and Belady 1985c; Shepperd 2000; Godfrey and Tu 2001). This approach intends to prove how the behavior of the inverse square model changes

depending upon the relative growth rate of the software system under study. Whereas in "normal" cases the inverse square model is able to fit software systems growing at linear or under-linear rates, there are extreme cases where the inverse square model produces very inaccurate estimations of the growth trend. Next, it will be defined what qualifies a case as "extreme" or "normal," as well as proof of the QUICK application as an example of an extreme case, out of the boundaries where the inverse square model can produce good estimations. Additional proof will be presented that demonstrates that examples of good estimates in the literature are inside the mentioned boundaries.

This is an example of a software system growing at a perfect linear rate:

$$s(i) = s(1) + k\,(i-1), \qquad \{i=1,\ldots,n\} \tag{4.4}$$

where s(i) is the system's size at version i, i is the release sequence number, and k is the growth rate, or in other words, the slope of the graph that represents s(i) on the vertical axis and the release sequence i on the horizontal axis.

Before continuing equation (4.4) will be normalized in terms of s(1), so further operation will be greatly simplified. Some papers (Lehman 2001; Lehman et al. 2001) have started presenting results in a normalized scale and, as it will be shown later, the inverse square model's estimates are not affected by this change of scale:

$$\underline{s}(i) = (s(1) + k\,(i-1))/s(1). \tag{4.5}$$

Defining $\underline{k} = k/s(1)$, then

$$\begin{aligned} &\underline{s}(1) = 1 \\ &\underline{s}(2) = 1 + \underline{k} \\ &\ldots \\ &\underline{s}(n) = 1 + (n-1)\underline{k}, \end{aligned} \tag{4.6}$$

or

$$\underline{s}(i) = 1 + (i-1)\underline{k} \qquad \{i=1,\ldots n\} \tag{4.7}$$

Now, applying the system's normalized sizes over Eq. (4.2):

$$\begin{aligned}
&E(2) = (\underline{s}(2) - \underline{s}(1))\underline{s}^2(1) = \underline{k} \\
&E(3) = (\underline{s}(3) - \underline{s}(1)) \,/\, (1/\underline{s}^2(1) + 1/\underline{s}^2(2)) = 2\underline{k} \,/\, (1 + 1/(1+\underline{k})^2) \\
&E(4) = 3\underline{k} \,/\, (1 + 1/(1+\underline{k})^2 + 1/(1+2\underline{k})^2) \\
&E(5) = 4\underline{k} \,/\, (1 + 1/(1+\underline{k})^2 + 1/(1+2\underline{k})^2 + 1/(1+3\underline{k})^2) \\
&\ldots \\
&E(n) = (n-1)\underline{k} \,/\, (1 + 1/(1+\underline{k})^2 + \ldots + 1/(1+(n-2)\underline{k})^2)
\end{aligned} \tag{4.8}$$

Eq. (4.8) is still too complicated to allow any general conclusion, but some particularly interesting cases deserve to be analyzed in detail. Eq. (4.8) is a function of n and $\underline{k}$, where $\underline{k}$ is a positive real number greater than zero (remember that $\underline{k}$ is the normalized growth rate, a system that does not grow at all or even shrinks is of little interest in software evolution). The most interesting cases that in addition present opportunities to simplify the equations are:

- $\underline{k}$ and $n\underline{k} \ll 1$
- $\underline{k} = 1$
- $\underline{k} \gg 1$.

These three cases will be explored in the next sections.

Case 1: $k \ll 1$ and $nk \ll 1$

In this case Eq. (4.8) can be approximated as follows:

$$\begin{aligned}
&E(2) = \underline{k} \\
&E(3) = 2\underline{k} \,/\, (1 + 1/(1+\underline{k})^2) \sim 2\underline{k} \,/\, (1 + 1) = \underline{k} \\
&E(4) = 3\underline{k} \,/\, (1 + 1/(1+\underline{k})^2 + 1/(1+2\underline{k})^2) \sim 3\underline{k} \,/\, (1 + 1 + 1) = \underline{k} \\
&\ldots \\
&E(n) \sim \underline{k}.
\end{aligned} \tag{4.9}$$

Consequently, $\underline{E}$, which is calculated as the average of E(2),…,E(n), could be approximated by:

$$\underline{E} \sim \underline{k}, \tag{4.10}$$

which is very simple to work with. Now, using this result back to Eq. (4.1) and (4.2) leads to

$$\begin{aligned}
&\hat{S}(1) = \underline{s}(1) = 1 \\
&\hat{S}(2) = \hat{S}(1) + \underline{E}/\hat{S}^2(1) \sim 1 + \underline{k} \\
&\hat{S}(3) = \hat{S}(2) + \underline{E}/\hat{S}^2(2) \sim 1 + \underline{k} + \underline{k}/(1+\underline{k})^2 \\
&\hat{S}(4) = \hat{S}(3) + \underline{E}/\hat{S}^2(3) \sim 1 + \underline{k} + \underline{k}/(1+\underline{k})^2 + \underline{k}/(1 + \underline{k} + \underline{k}/(1+\underline{k})^2)^2 \\
&\cdots
\end{aligned} \tag{4.11}$$

that can be simplified again because $\underline{k} << 1$:

$$\begin{aligned}
&\hat{S}(1) = 1 \\
&\hat{S}(2) \sim 1 + \underline{k} \\
&\hat{S}(3) \sim 1 + \underline{k} + \underline{k}/(1+\underline{k})^2 \sim 1 + \underline{k} + \underline{k} = 1 + 2\underline{k} \\
&\hat{S}(4) \sim 1 + \underline{k} + \underline{k} + \underline{k} = 1 + 3\underline{k} \\
&\cdots \\
&\hat{S}(n) \sim 1 + (n-1)\underline{k}.
\end{aligned} \tag{4.12}$$

In general:

$$\hat{S}(i) \sim 1 + (i-1)\underline{k}, \tag{4.13}$$

which implies that the estimation follows perfectly the original data (Eq. 4.4).

An example of a dataset with $\underline{k} = 0.01$ and $n = 17$ is presented in Table 4.10.

Table 4.10. Example of a system with $\underline{k} << 1$ and $n\underline{k} << 1$.

i	$\underline{S}$	E	$\hat{S}$	MRE
1	1		1	0
2	1.01	0.01	1.010746251	0.000738862
3	1.02	0.0100995	1.021265208	0.0012404
4	1.03	0.010199	1.031568592	0.001522905
5	1.04	0.0102985	1.041667182	0.00160306
6	1.05	0.010398	1.051570917	0.001496112
7	1.06	0.0104975	1.061288983	0.001216022
8	1.07	0.010597	1.07082989	0.000775598
9	1.08	0.0106965	1.080201538	0.00018661
10	1.09	0.010796001	1.089411279	0.000540111
11	1.1	0.010895501	1.098465961	0.001394581
12	1.11	0.010995001	1.107371983	0.002367583
13	1.12	0.011094501	1.116135328	0.0034506
14	1.13	0.011194001	1.124761602	0.00463575
15	1.14	0.011293501	1.133256067	0.005915731
16	1.15	0.011393002	1.141623666	0.007283769
17	1.16	0.011492502	1.149869053	0.008733575

E(i) are calculated using Eq. (4.2). It is shown how these values are very close to the value of $\underline{k}$, as the Eq. (4.9) predicts for small values of $\underline{k}$. $\underline{E}$ is the average of E(i) and equal to 0.010746251, verifying the result of Eq. (4.10). Moreover, the estimated values $\hat{S}(i)$ almost follow a linear progression, as Eq. (4.13) predicts. All the simplified equations (4.9), (4.10), and (4.13), become less accurate in the final releases, as i reaches the value of n. This is an expected result, since in this particular example the product $n\underline{k}$ cannot be considered strictly $<< 1$.

Figure 4.11 shows a plot of the normalized original data, $\underline{s}(i)$, and the estimation $\hat{S}(i)$ produced by the inverse square model.

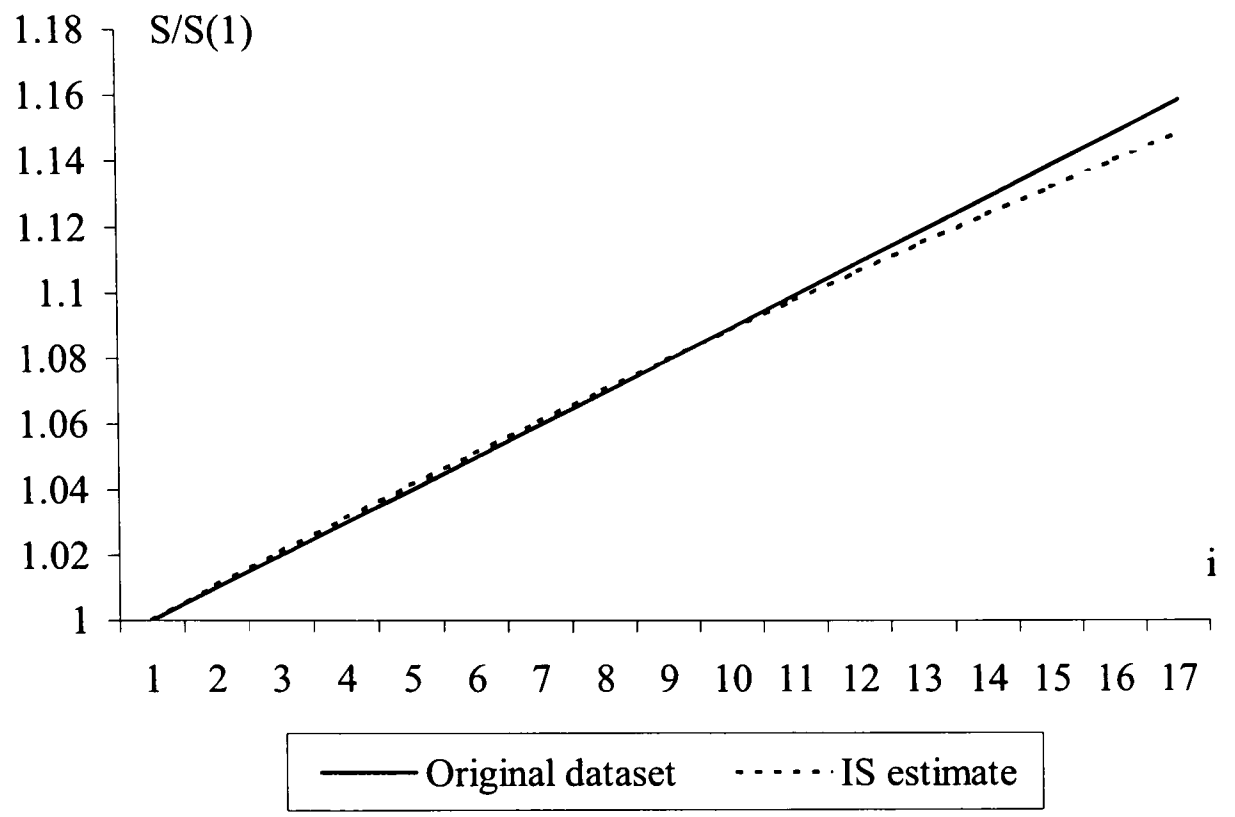

Figure 4.11. Case 1: inverse square estimations. $\underline{k} = 0.01$, $n = 17$.

The MMRE (Mean Magnitude of Relative Error) of the estimations is 0.25%, which is an overall excellent fit, proving that the inverse square model is able to produce good estimates of systems with a linear growth trend as long as the increment rate is kept small. A small increment rate is needed to keep both the normalized increment rate, $\underline{k}$, and the product $n\underline{k}$ much lower that 1, where n is the number of releases considered.

Case 2: $\underline{k} = 1$

Eq. (4.8) offers another opportunity of simplification when $\underline{k} = 1$. In this particular case:

$$E(2) = \underline{k} = 1$$
$$E(3) = 2\underline{k} / (1 + 1/4)$$
$$E(4) = 3\underline{k} / (1 + 1/4 + 1/9)$$
$$E(5) = 4\underline{k} / (1 + 1/4 + 1/9 + 1/16)$$
$$\ldots$$
$$E(n) = (n-1)\underline{k} / (1 + 1/4 + \ldots + 1/(n-1)^2), \qquad (4.14)$$

or:

$$E(i) = (i-1)\,\underline{k} \,/\, \sum_{j=1}^{i-1} \frac{1}{j^2} \qquad \{i=2,\ldots n\} \tag{4.15}$$

Remember that in this case $\underline{k} = 1$, although the constant is leaved in the equations for a reason that is presented later.

The analytical equivalent of this equation can be calculated with some difficulties:

$$E(i) = (i-1)\,\underline{k} \,/\, (\pi^2/6 + \psi(i)) \qquad \{i=2,\ldots n\} \tag{4.16}$$

where $\psi(i)$ is the logarithmic derivative of the Euler's gamma function $\Gamma(i)$ (Mathematica 1996). That is an ugly expression, but fortunately there is a shortcut since (4.16) follows an almost linear progression, at least for $i < 1000$. Since all the software systems studied count their number of releases in an order of magnitude under that limit, the replacement will not lose significant precision. Table 4.11 presents the numeric value of expression (4.16) for some values of i, as well as the results of a linear fit and the MRE (magnitude of relative error) of the fit.

Table 4.11. Approximation of Eq. (4.16).

i	E(i)	Linear fit	MRE
2	1	0.98957123	0.01042877
3	1.6	1.596795675	0.002002703
4	2.204081633	2.204020119	2.79088E-05
5	2.809756098	2.811244564	0.000529749
6	3.416208009	3.418469008	0.000661845
7	4.023095549	4.025693453	0.000645748
8	4.630251124	4.632917898	0.000575946
9	5.237583148	5.240142342	0.000488621
10	5.845037416	5.847366787	0.000398521
15	8.883272185	8.88348901	2.44082E-05
25	14.961442	14.95573346	0.00038155
100	60.554759	60.4975668	0.000944471

The linear fit corresponds to the equation:

$$\hat{E}(i) = 0.607224\, i - 0.22488 \tag{4.18}$$

and its MMRE is 0.14%, an almost perfect result. Figure 4.12 shows the plot of the values of E(i) and its linear approximation. It is difficult to perceive the differences between the two.

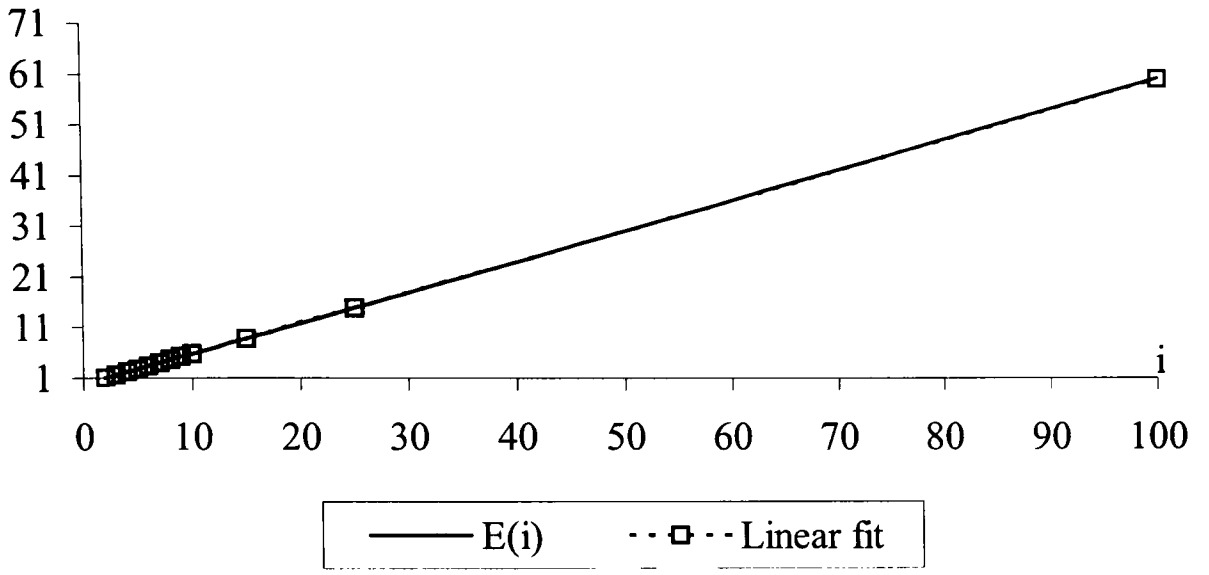

Figure 4.12. Approximation of Eq. (4.16) and linear fit.

Now, it is possible to calculate the results of the inverse square model in this particular case, since the equations for E(i) have been greatly simplified:

$$E(i) \sim \hat{E}(i) = (0.607224\, i - 0.22488)\, \underline{k} \quad \{i=2,\ldots n\} \tag{4.19}$$

Using (4.3)

$$\underline{E} \sim \frac{\underline{k}}{n-1} \sum_{i=2}^{n} (0.60722\, i - 0.22488), \tag{4.20}$$

and the summation can be simplified, yielding to

$$\underline{E} \sim \underline{k}\, (0.38234 + 0.30361\, n). \tag{4.21}$$

Using this result in Eq. (4.1) and (4.2):

$$\hat{S}(1) = \underline{s}(1) = 1$$
$$\hat{S}(2) = \hat{S}(1) + \underline{E}/\hat{S}^2(1) \sim 1 + \underline{E} = 1 + \underline{k}\, (0.38234 + 0.30361\, n)$$
$$\hat{S}(3) \sim \hat{S}(2) + \underline{E} / (1 + \underline{E})^2 \sim \hat{S}(2) + 1 / \underline{E}$$

$$\hat{S}(4) \sim \hat{S}(3) + \underline{E} / (1 + \underline{E} + \underline{E}/(1 + \underline{E})^2)^2$$
$$\ldots \tag{4.22}$$

This proves how the estimations depend upon the number of releases of the system under study, n, and the normalized growth rate $\underline{k}$. Since the total normalized increment experimented by the system, $\underline{s}(n) - \underline{s}(1)$, is equal to $(n-1)\,\underline{k}$, then

$$n\underline{k} = \underline{s}(n) - \underline{s}(1) + \underline{k} \tag{4.23}$$

and

$$\hat{S}(2) \sim 1 + 0.38234\,\underline{k} + 0.30361\,(\underline{s}(n) - \underline{s}(1)) \tag{4.24}$$

Therefore, it could be said that the first estimation produced by the inverse square model, when $\underline{k} = 1$, depends upon the total increment experimented by the system under study, and its normalized incremental growth rate, $\underline{k}$.

For estimations $\hat{S}(3),\ldots,\hat{S}(4)$, Eq. (4.22) shows how the increments are decreasing in value. In case of $n >> 1$, then $\underline{E}$ should be $>> 1$ too and the equations can be approximated by

$$\hat{S}(3) \sim \hat{S}(2) + 1 / \underline{E}$$
$$\hat{S}(4) \sim \hat{S}(3) + \underline{E} / (1 + \underline{E} + \underline{E}/(1 + \underline{E})^2)^2 \sim \hat{S}(3) + 1 / \underline{E}$$
$$\ldots$$
$$\hat{S}(n) \sim \hat{S}(n-1) + 1 / \underline{E} \tag{4.25}$$

The increment between two consecutive versions, $\hat{S}(i) - \hat{S}(i-1)$, for $i=3,\ldots,n$; is approximately constant and equal to $1 / \underline{E}$.

Table 4.12 presents a dataset that would correspond to a system with $\underline{k} = 1$, and again $n = 17$.

Table 4.12. Example of a system with $\underline{k} = 1$.

i	$\underline{s}$	E	$\hat{S}$	MRE
1	1		1	0
2	2	1	6.543454335	2.271727168
3	3	1.6	6.672923493	1.224307831
4	4	2.204081633	6.797417425	0.699354356
5	5	2.809756098	6.917392932	0.383478586
6	6	3.416208009	7.03324281	0.172207135
7	7	4.023095549	7.145307618	0.020758231
8	8	4.630251124	7.253884812	0.093264398
9	9	5.237583148	7.359235934	0.182307118
10	10	5.845037416	7.461592338	0.253840766
11	11	6.452579828	7.561159801	0.312621836
12	12	7.060187872	7.658122264	0.361823145
13	13	7.667846091	7.752644913	0.403642699
14	14	8.275543525	7.844876714	0.439651663
15	15	8.883272185	7.934952535	0.471003164
16	16	9.491026113	8.022994922	0.498562817
17	17	10.09880078	8.1091156	0.5229932

In this case, the estimations of the inverse square model are very poor, the MMRE is 49%. The resulting $\underline{E}$ is equal to 5.5434, very close to the approximation produced by Eq. (4.21): $\underline{E} \sim 0.38234 + 0.30361*17 = 5.5437$.

Eq. (4.22) approximates $\hat{S}(2) \sim 1 + \underline{E} = 6.5437$, which is verified in this example. Eq. (4.24) produces an identical result.

The increment between two estimates $\hat{S}(i)- \hat{S}(i-1)$, for i=3,…,n; which according to Eq. (4.25) should be approximately constant and equal to $1/\underline{E} = 0.18$. Table 4.12 shows how the average increment of the estimates is actually equal to 0.104. The inaccuracy comes from the fact that in this case $\underline{E}$ is not really >> 1.

Finally, Figure 4.13 shows a comparative plot of the data and the estimates.

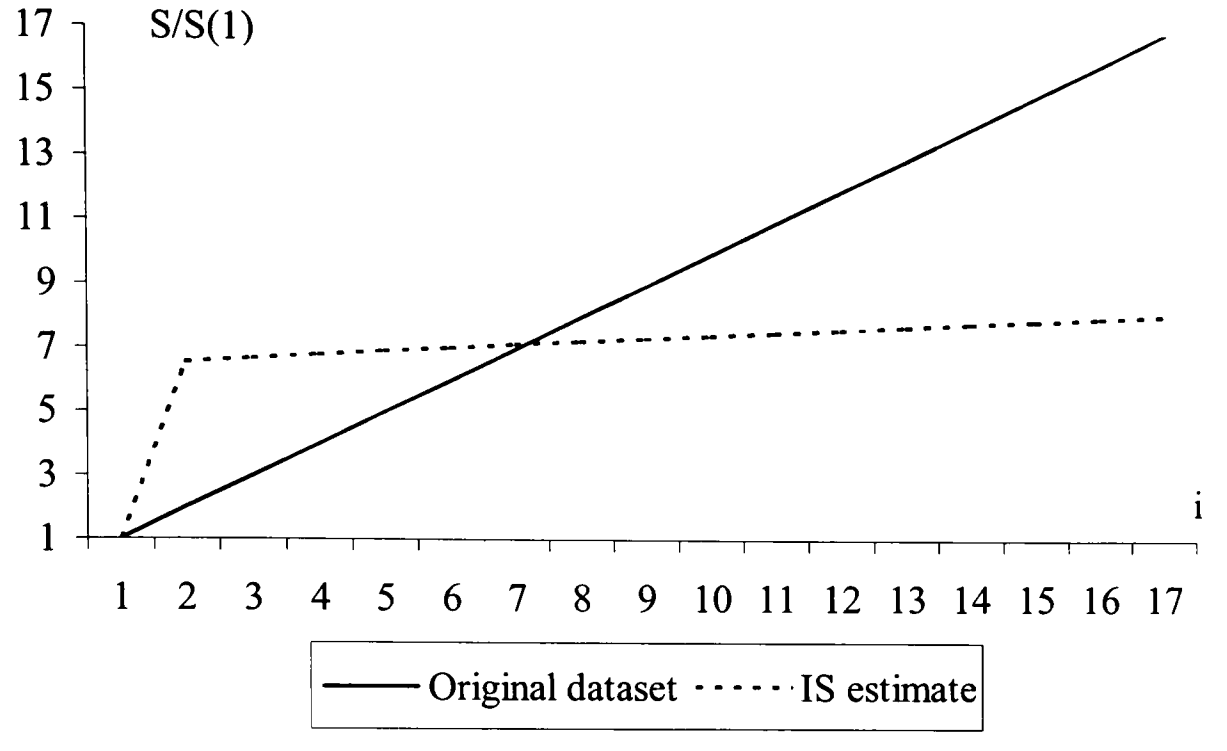

Figure 4.13. Case 2: inverse square estimations. $\underline{k} = 1$.

There is a great similarity between this result and the result of the inverse square model when it was applied to the QUICK evolution dataset: a large increment followed by small, almost constant increments.

Case 3: $k >> 1$

This case is considered to complete the demonstration of how the estimations produced by the inverse square model depend upon the values of $\underline{k}$. It is very unlikely to find an example in the real world of a software system growing at this frantic pace.

Eq. (4.8) is approximated as follows:

$$
\begin{aligned}
&E(2) = \underline{k} \\
&E(3) = 2\underline{k} / (1 + 1/(1+\underline{k})^2) \sim 2\underline{k} / (1 + 1/\underline{k}^2) \sim 2\underline{k} \\
&E(4) \sim 3\underline{k} / (1 + 1/\underline{k}^2 + 1/(2\underline{k})^2) \sim 3\underline{k} \\
&\ldots \\
&E(n) \sim (n-1)\underline{k}.
\end{aligned}
\tag{4.25}
$$

Therefore,

$$\underline{E} \sim \frac{\underline{k}}{n-1}\sum_{i=1}^{n-1} i \tag{4.26}$$

and

$$\underline{E} \sim \frac{n\underline{k}}{2}. \tag{4.27}$$

With this result, the estimations can be approximated by

$$\begin{aligned}
&\hat{S}(1) = 1\\
&\hat{S}(2) \sim 1 + n\underline{k}/2\\
&\hat{S}(3) \sim 1 + n\underline{k}/2 + 1/(1 + n\underline{k}/2)^2 \sim 1 + n\underline{k}/2\\
&\hat{S}(4) \sim 1 + n\underline{k}/2 + 1/(1 + n\underline{k}/2)^2 \sim 1 + n\underline{k}/2\\
&\cdots\\
&\hat{S}(n) \sim 1 + n\underline{k}/2.
\end{aligned} \tag{4.28}$$

Now, using Eq. (4.23) and (4.28):

$$\hat{S}(2) \sim 1 + n\underline{k}/2 = 1 + (\underline{s}(n) - \underline{s}(1) + n)/2, \tag{4.29}$$

predicting a large first increment of a value a bit over half of the total increment of the system and later increments approximately null.

Table 4.13 presents a dataset that would correspond to a system with $\underline{k} = 100$, and again $n = 17$.

Table 4.13. Example of a system with $\underline{k} >> 1$.

I	$\underline{s}$	E	$\hat{S}$	MRE
1	1		1	0
2	101	100	850.8733884	7.424488995
3	201	199.980396	850.8745623	3.233206778
4	301	299.9631701	850.8757362	1.826829688
5	401	399.9464796	850.8769101	1.121887556
6	501	499.9299909	850.878084	0.698359449
7	601	599.9135994	850.8792578	0.415772476
8	701	699.8972619	850.8804317	0.21380946
9	801	799.8809575	850.8816056	0.062274164
10	901	899.8646748	850.8827794	0.055623996
11	1001	999.8484072	850.8839533	0.149966081
12	1101	1099.83215	850.8851271	0.227170638
13	1201	1199.815902	850.886301	0.291518484
14	1301	1299.799659	850.8874748	0.34597427
15	1401	1399.783422	850.8886487	0.392656211
16	1501	1499.767188	850.8898225	0.43311804
17	1601	1599.750957	850.8909963	0.468525299

The MMRE is this case is 102%, extremely poor. $\underline{E}$ is equal to 849.87, verifying Eq. (4.27), which approximates $\underline{E}$ with $n\underline{k}/2 = 17*100/2 = 850$. $\hat{S}(2)$ is approximated very closely by Eq. (4.29) as well: $1 + n\underline{k}/2 = 1 + 17*100/2 = 851$. Later increments are negligible as Eq. (4.28) predicts.

Figure 4.14 shows the huge difference between the estimation and the original dataset.

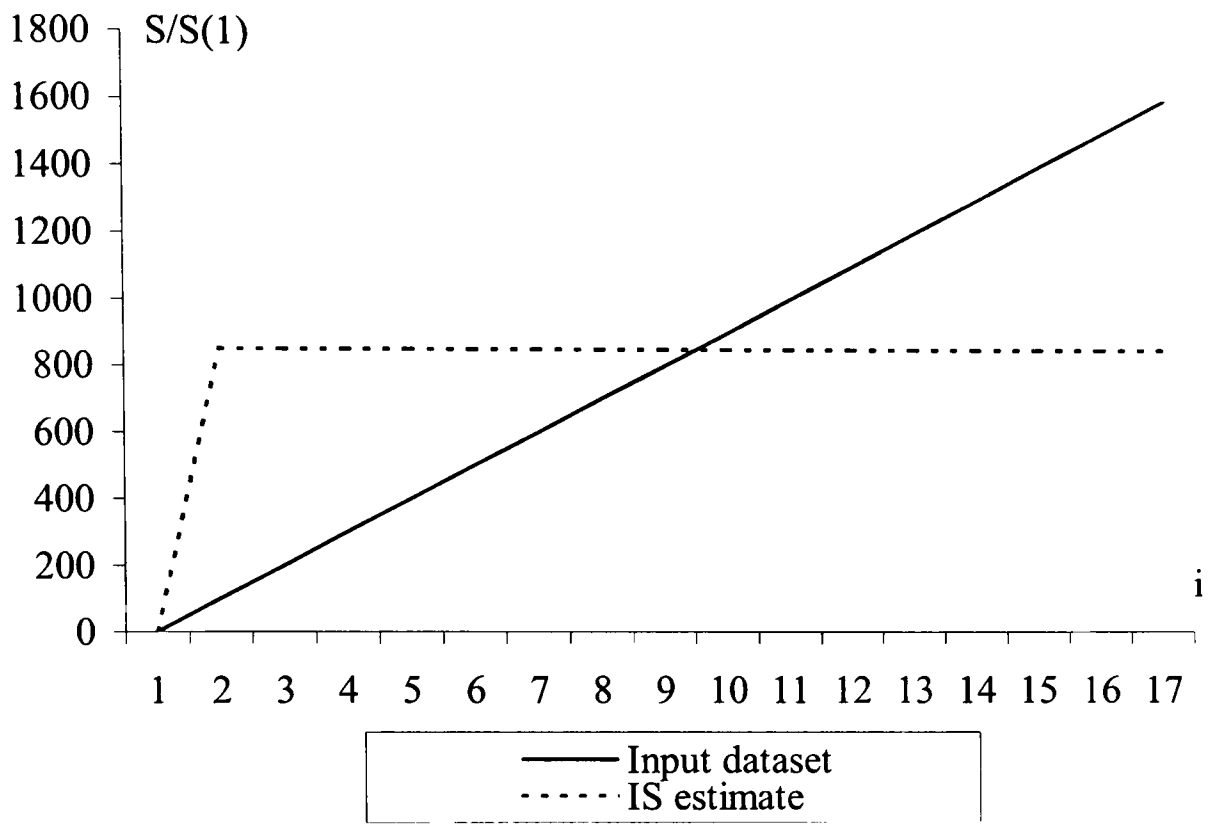

Figure 4.14. Case 3: inverse square estimations. $\underline{k} >> 1$.

Conclusions of the Analysis

Results for ad hoc datasets exhibiting a perfect linear growth has proven how the pattern followed by the inverse square estimations depends exclusively upon the growth rate of the system. When the normalized growth rate $\underline{k} = (s(n) - s(1))/s(1)$ is very small ($\underline{k} << 1$), the estimations follow a linear growth pattern, whereas for values near 1 or over this value, the estimations follows an peculiar pattern given by a first large step follow by small and almost constant increments. Between both values of $\underline{k}$, the pattern changes smoothly from one extreme to another. This area will be explored next, using examples of software systems published in the literature.

Revision of Literature Examples

So far, results have been confirmed by ad hoc linear datasets. The exploration of literature examples will bring the opportunity to study the inverse square model's behavior in real world evolutionary systems.

Logica FW

The "Logica FW" system's dataset (Turski 1996) is important because it led to the definition of the inverse square model. The paper presents how the growth of the Logica FW system can be approximated fairly well by a linear fit, although the inverse square model produces much more accurate estimations. The system does not follow a perfect linear trend because the trend is decaying and the data is affected by the ripple described as an effect of the third law of software evolution (Lehman and Ramil 2001). However, considering the system as approximately linear, and according to the previous analysis of the inverse square model, the behavior of the estimations would be given by the number of releases n, and the normalized incremental growth $\underline{k}$. The whole dataset of the system is not reproduced here, only s(1) and s(n) are needed at this point:

$$s(1) = 977$$
$$s(n=21) = 2315$$

therefore, the normalized sizes are

$$\underline{s}(1) = 1$$
$$\underline{s}(21) = 2315 / 977 = 2.3695$$

Using Eq. (4.23) for linear growth,

$$\underline{k} \sim (\underline{s}(n) - \underline{s}(1))/(n-1) = (2.3695 - 1)/20 = 0.0685,$$

this value is lower than 1, but cannot be considered extricly as << 1, in addition $n\underline{k}$ is equal to 1.4380, which definitely is not much lower than 1. If the experience from linear system can be applied here, then the inverse square model should produce a pattern between the patterns shown in Figure 4.11 and Figure 4.13.

Figure 4.15 shows the estimations of the inverse square model on this system, calculated using Eq. (4.1) though (4.4), the same way used in previous examples. The result is presented in a normalized scale.

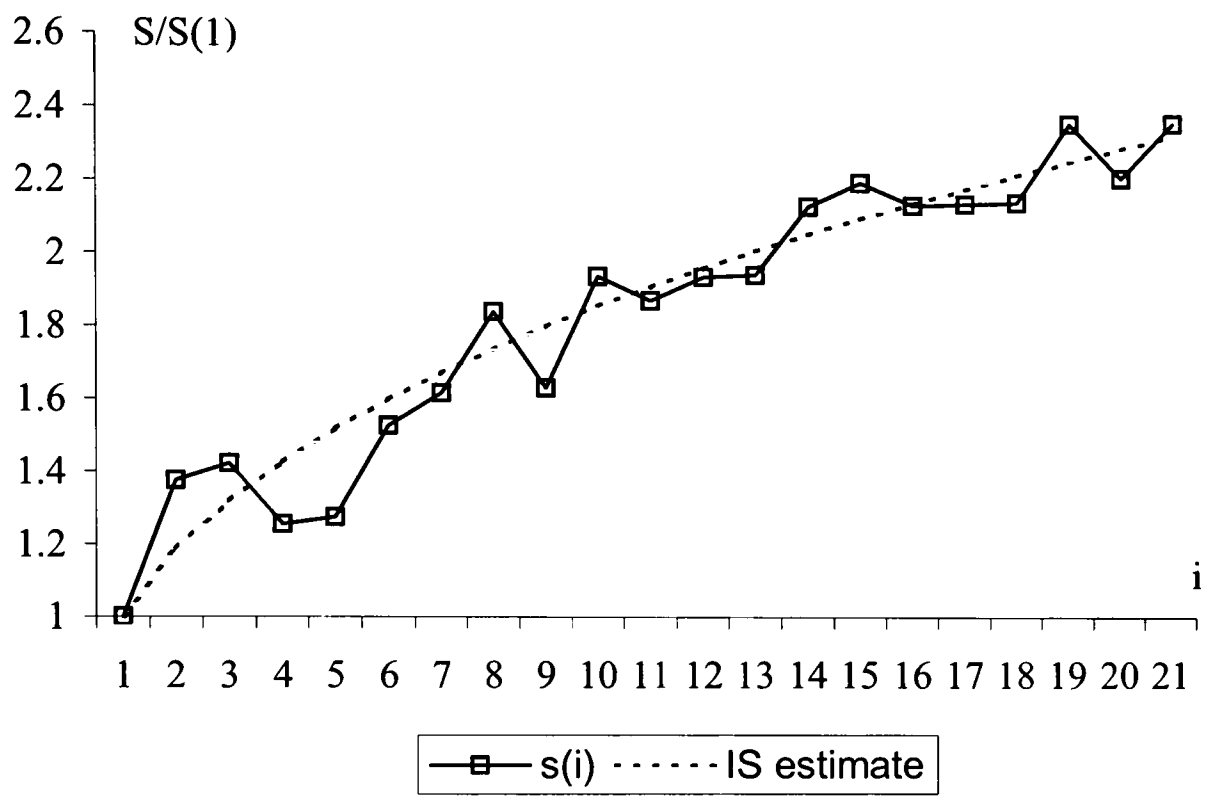

Figure 4.15. Inverse square model applied to the Logica FW dataset.
System's dataset published in Turski (1996).

The value of $\underline{E}$ is 0.188 and the MMRE is 5.38%, a good estimation. The assumption is that, since the value of $\underline{k}$ is far away from the cases $\underline{k} = 1$ or even $\underline{k} >> 1$, the model produces a good estimation.

Continuing working with this dataset it is revealing to display the graph of Figure 4.15, in a double logarithmical scale, as Figure 4.16 does:

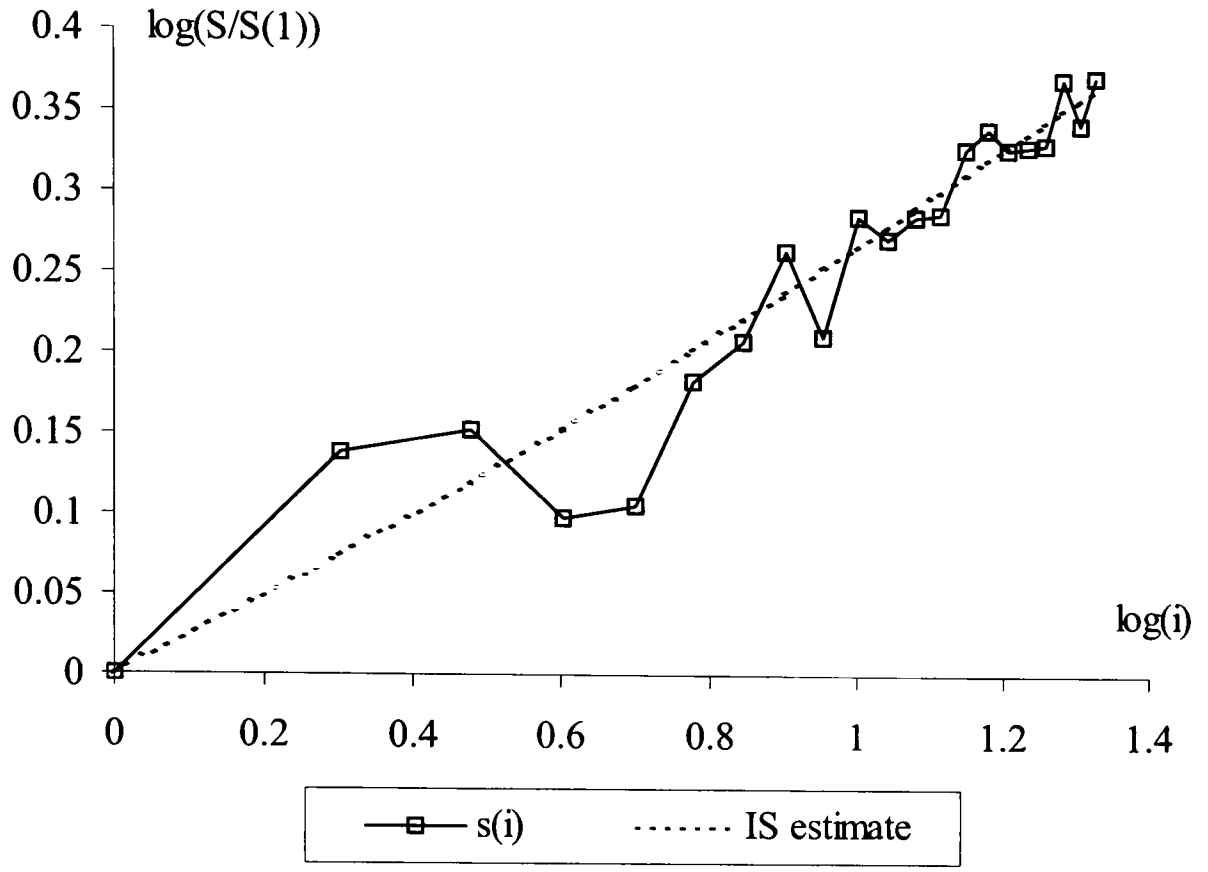

Figure 4.16. Logica FW dataset and IS estimations in a double-log scale.

The data from both Figure 4.15 and Figure 4.16 is the same, only the scale has changed. The striking evidence is how much the inverse square model estimates follow an almost straight line. To prove that, a linear fit of the estimates in the double-log scale is calculated. The result is that the estimates can be approximated by the equation:

$$\log(\hat{S}(i)) = 0.2931 \log(i) - 0.0510 \qquad (4.30)$$

with an MMRE equal to 1.07%, proof of an excellent approximation.

But a linear relation in the log domain corresponds to an exponential relation in the linear domain. From Eq. (4.30):

$$\hat{S}(i) = e^{-0.0221}\, i^{0.2931} = 0.9781\, i^{0.2931} \qquad (4.31)$$

This equation is connected with another result. Godfrey and Tu, in their study of the evolution of the Linux kernel present another analytical approximation of the inverse square model's equations (Godfrey and Tu 2001). This paper rewrites the equations of the inverse square model as the differential equation:

$$\hat{S}(i)' = \hat{S}(i) + E / \hat{S}(i)^2 \tag{4.32}$$

or

$$\hat{S}(i)' \, \hat{S}(i)^2 = \hat{S}(i)^3 + E. \tag{4.33}$$

Integrating both sides:

$$\int \hat{S}^2(i) \, \hat{S}'(i) = \int \hat{S}^3(i) + \int E \tag{4.34}$$

and

$$\frac{1}{3} \hat{S}^3(i) = \int \hat{S}^3(i) + E\, i \tag{4.35}$$

The mentioned paper seems to assume that the right side of Eq. (4.34) can be approximated by E i. If this were true, then the inverse square estimations could be approximated by:

$$\hat{S}(i) = (3\, E\, i)^{1/3} \tag{4.36}$$

Going back to the Logica FW system's results, if the actual value of the constant E is replaced in equation (4.36), then

$$\hat{S}(i) = (3 \cdot 0.187\, i)^{1/3} \sim 0.826\, i^{0.333} \tag{4.37}$$

The similarity between Eq. (4.31) and (4.37) is striking. This result could suggest that a system with a "normal" growth, or more precisely, a system with $k \sim 0.0685$, the

inverse square model behaves as an exponential equation would do, with an exponent close to 1/3, as Eq. (4.36) shows.

This is an important result that will be try to be verified in the next example.

Lucent

The Lucent system's dataset has been published in (Lehman 1998b) together with the inverse square model estimations. The dataset covers the evolution of the system's growth during its first 17 versions. Treating the system as a linear system, the normalized growth rate can be calculated using Eq. (4.23):

$$\underline{s}(1) = 1$$
$$\underline{s}(17) = 107000 / 44000 = 2.432$$
$$\underline{k} \sim (2.432 - 1)/(17 - 1) = 0.0895,$$
$$n\underline{k} \sim 1.521$$

which is a value lower than 1, but not << 1 and in the same order of magnitude than the $\underline{k}$ calculated for the Logica FW system. Figure 4.17 plots the original dataset and the estimations in a normalized scale.

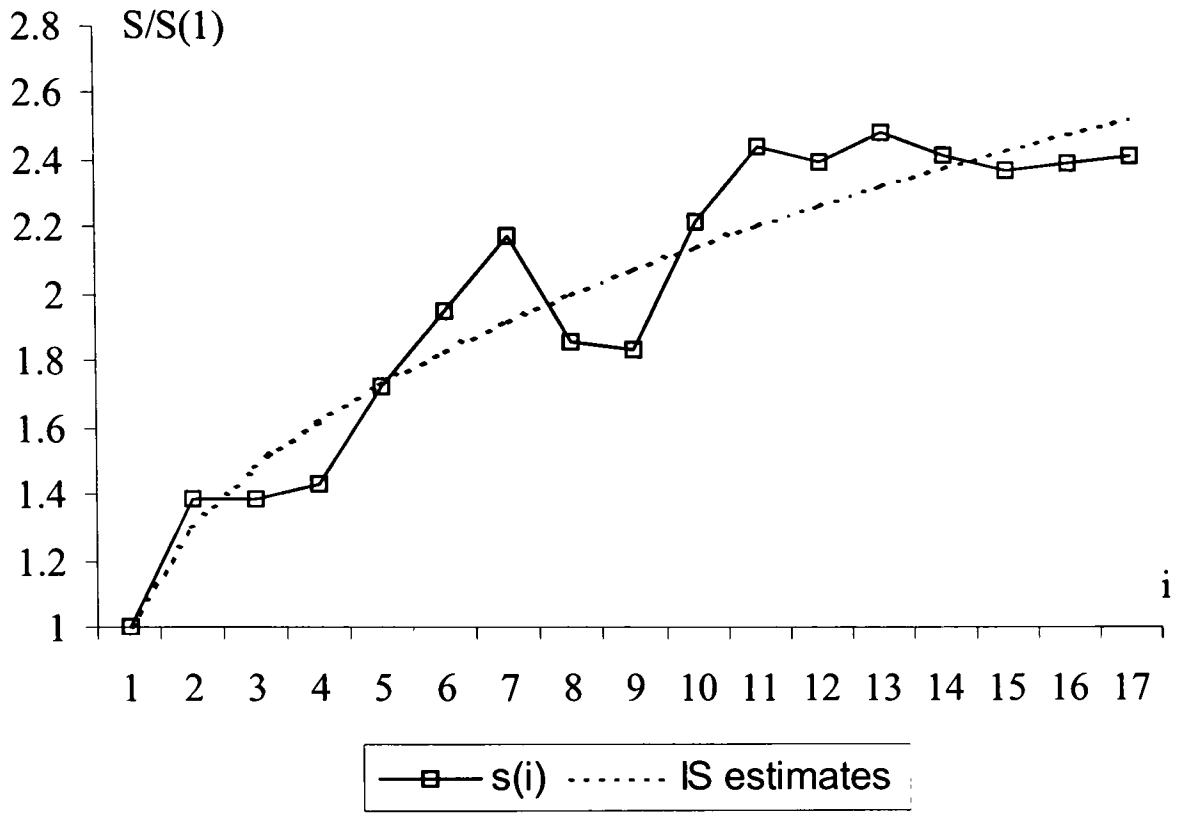

Figure 4.17. Lucent dataset and inverse square estimations.
Example published in (Lehman 1998b).

The value of E is 0.302 and the MMRE of the estimations is 6.02%, a good result although a bit higher than for the Logica FW example, perhaps due to the fact that this dataset is more irregular than the Logica FW's dataset.

Figure 4.18 plots the same results in a double logarithmic scale, as it was done on the previous example.

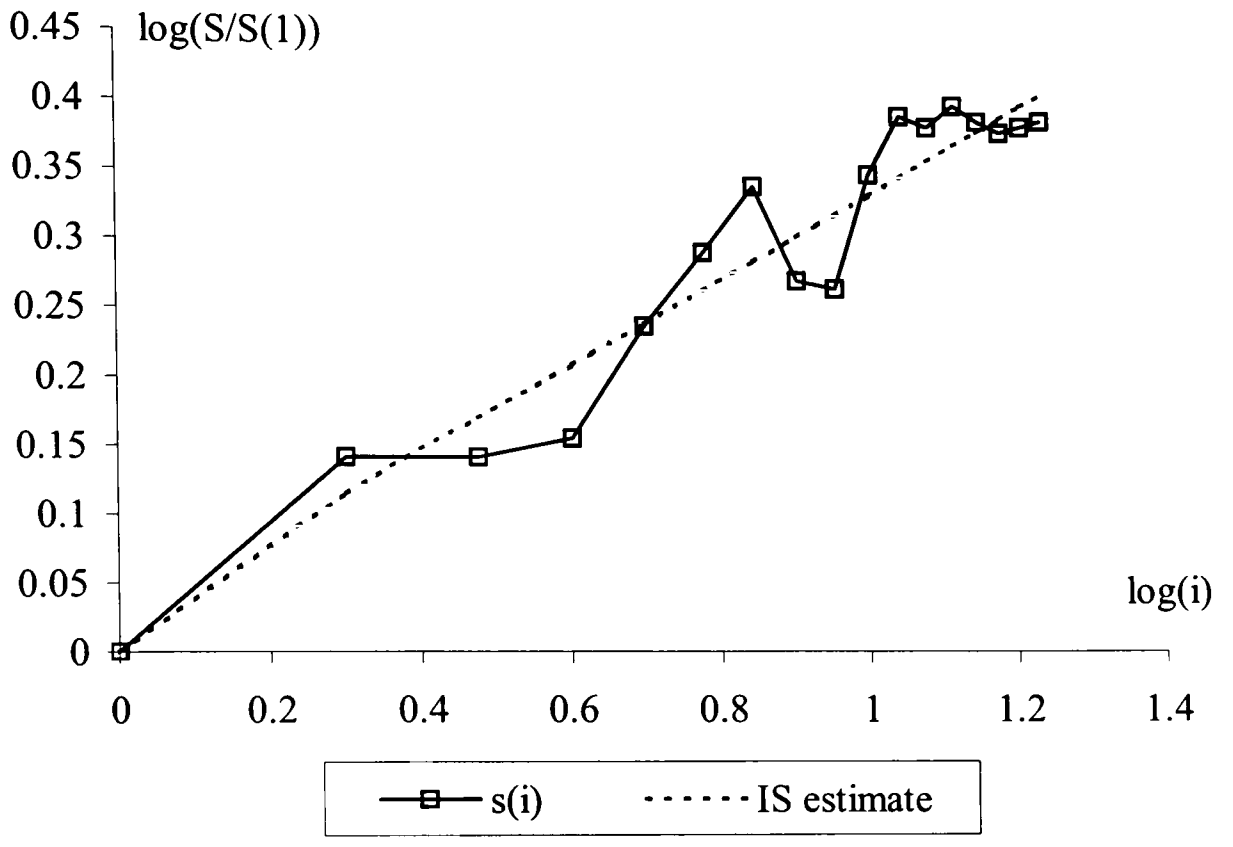

Figure 4.18. Lucent dataset and inverse square estimations in a log-log scale.

Again the inverse square estimations follow and almost perfect straight line that is approximated by the equation:

$$\log(\hat{S}(i)) = 0.3127 \log(i) + 0.0201, \tag{4.38}$$

which fits the estimations with an MMRE = 0.13%, an almost perfect fit. Eq. (4.38) can be rewritten as:

$$\hat{S}(i) = 1.0203\, i^{0.3127} \tag{4.39}$$

Using again Eq. (4.36):

$$\hat{S}(i) = (3 \cdot 0.302\, i)^{1/3} \sim 0.967\, i^{0.333} \tag{4.40}$$

Equations (4.39) and (4.40) represent almost the same function, confirming again the result obtained by the Logica FW study. This could indicate that in cases where $\underline{k}$ is in the order of magnitude of 1/100, the inverse square model is approximated by the exponential function:

$$\hat{S}(i) = i^{1/3}, \tag{4.41}$$

At this point two examples has been presented of systems that can be successfully fitted using the inverse square model. Three more examples will be presented next, but this time of systems where the inverse square model fails to explain their growth.

Shepperd

(Shepperd 2000) includes the study of the evolution of a large information system. Its author tried to apply the inverse square model, but the results where disappointing. These are his words:

> The first stage of our investigation was to apply the single feedback model suggested by Turski [12] based upon an inverse square law. Unfortunately, when this particular formulation of a self-limiting system was applied to our dataset we observed an extremely poor fit, essentially very rapid growth followed by no growth whatsoever. This may in part be explained by the fact that our time series commences from zero, whereas the systems they studied had quite considerable initial sizes (Shepperd 2000, p. 3)

They match very precisely the present study's observations over the inverse square model in QUICK, and linear cases $\underline{k} = 1$ and $\underline{k} >> 1$. Again, considering the dataset as linear, the normalized growth rate is in this case:

$$\underline{k} = 7.54$$

a very large value, compared to previous examples. $\underline{E}$ is 186.54 in this case. Figure 4.19 plots the dataset as well as the inverse square estimations.

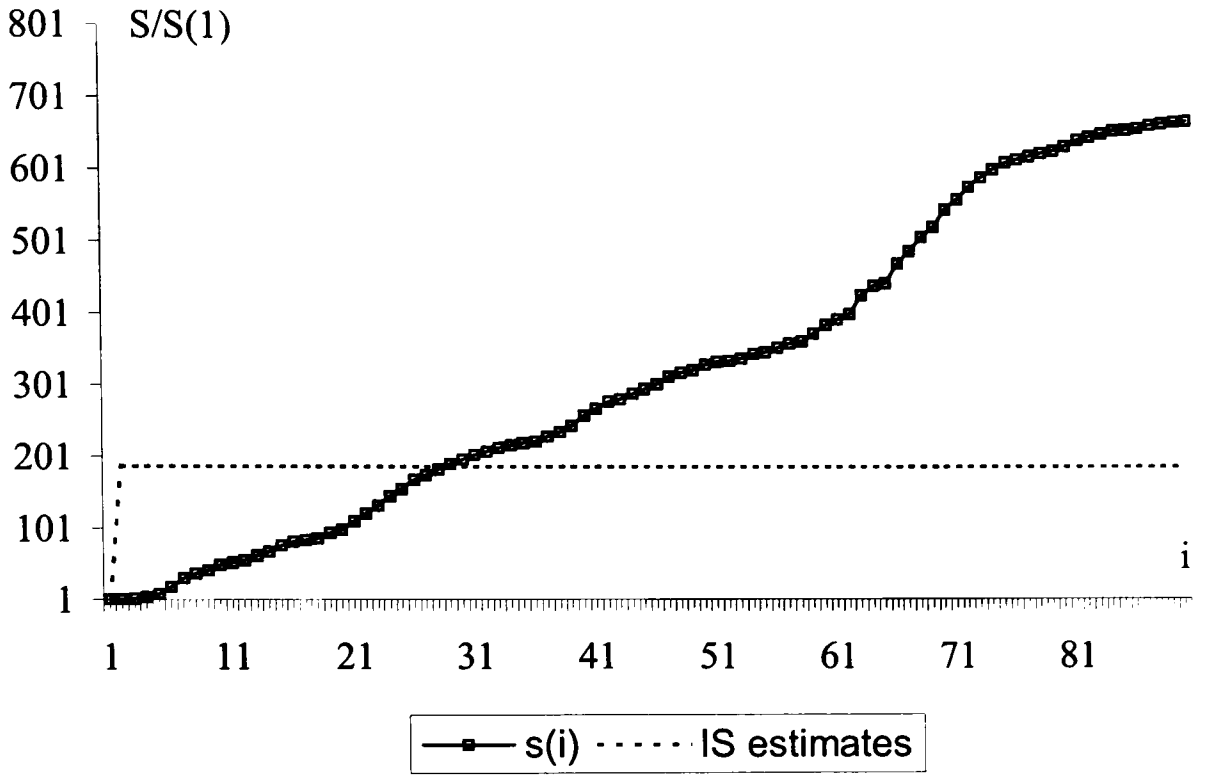

Figure 4.19. Shepperd's dataset and inverse square estimations.

The pattern confirms Shepperd's words. The value for the first increment is 187.54. Eq. (4.29) than can be applied to linear systems with $k >> 1$, predicts a first increment approximated by:

$$\hat{S}(2) \sim 1 + n\underline{k}/2 = 340,$$

which is not a very close approximation, indicating that a value of $\underline{k}$ around seven units is not enough to be considered as $>> 1$. Later growth of the estimates is negligible, as the linear case with $\underline{k} >> 1$ predicts though. The MMRE of the fit is 1449.3%, an extremely poor fit.

The observation made by Shepperd about the cause of the problem, that the dataset analyzed from zero whereas Lehman's datasets start well over that value, has been discarded by the present study, since every dataset is normalized before models are

applied. Thus, the initial value $\underline{s}(1)$ is always the same, one unit, for every dataset. The reason suggested by this study is that the inverse square model cannot fit dataset with a normalized growth rate $\underline{k}$ greater than one. In this particular case, a linear fit could explain much better the evolution of the system's growth. More about this point will be presented in the next sections.

Linux

The evolution of the Linux kernel (Godfrey and Tu 2000, 2001) is interesting because it presents evidence of a system that grows at pace faster than linear, therefore called super-linear. This system defies the laws of software evolution, since instead of growing at slower rates as the system becomes bigger and more complex, the system seems to be accelerating. It is clear that the inverse square model will have problems with this example.

Using the same methodology that in previous examples, the value of the normalized growth rate is calculated first:

$$\underline{k} = 0.14,$$

which is lower than one, but one order of magnitude higher than the $\underline{k}$ that was calculated for the Logica FW and Lucent examples.

The value of E in this case is 0.2599 and the inverse square estimations as well as the original dataset is presented in Figure 4.20.

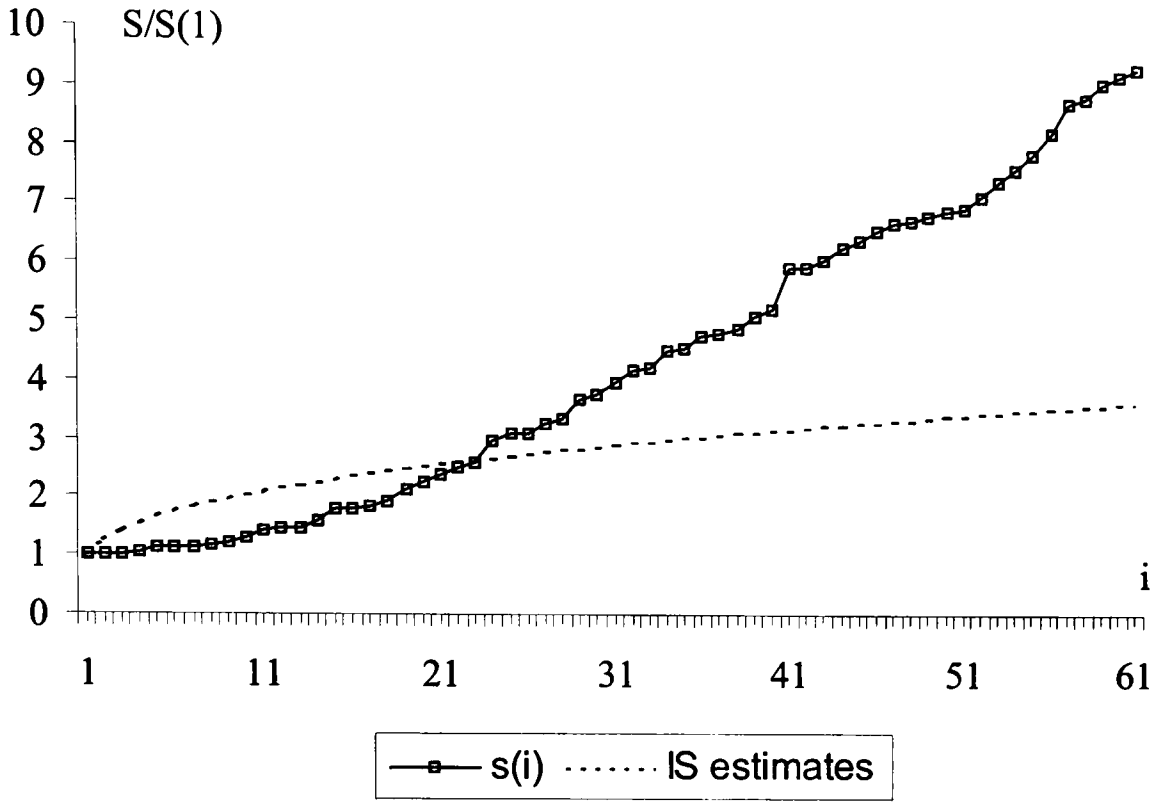

Figure 4.20. Linux kernel dataset and inverse square estimations.

The MMRE is 38.1%, proof of the poor fit that the plot suggests. The estimates follow a smooth trend, not as sharp as in the case of $\underline{k} = 1$, but since the system is growing at increasing speed, the inverse square model fails to produce a good fit.

IBM

The last example analyzed corresponds to the evolution of the IBM OS/360 system, a classical study that contributed to start the field of software evolution (Lehman and Belady 1985c). The evolution of the OS/360 growth was the basis for the third law of software evolution (law of self regulation) as well as the fifth law (law of conservation of familiarity). The lineal growth that is suggested by the data was relegated in later revisions of the law, in favor of the inverse square model which sustains a declining growth rate. One of the latest revisions (Lehman et al. 2001) of the original OS/360 results defends that a release sequence number scale is not completely appropriate because later releases of the system are much more separated in time than the earlier

releases are. The effect is that, in a real-time scale, the growth would look more declining, agreeing with the inverse square model assumptions. However, the inverse square model and the fifth law of software evolution are based on the release sequence number scale. They assume that if a release delivers a large amount of new functionality, without regard of the time consumed on the development of the release, future releases will deliver small increments, resulting in an average long-term increment almost constant, or at least statistically invariant.

The IBM OS/360 normalized growth rate, k, is 0.1565, very close to the value calculated on the previous example; E is 0.40. Figure 4.21 shows the inverse square estimates for this particular system.

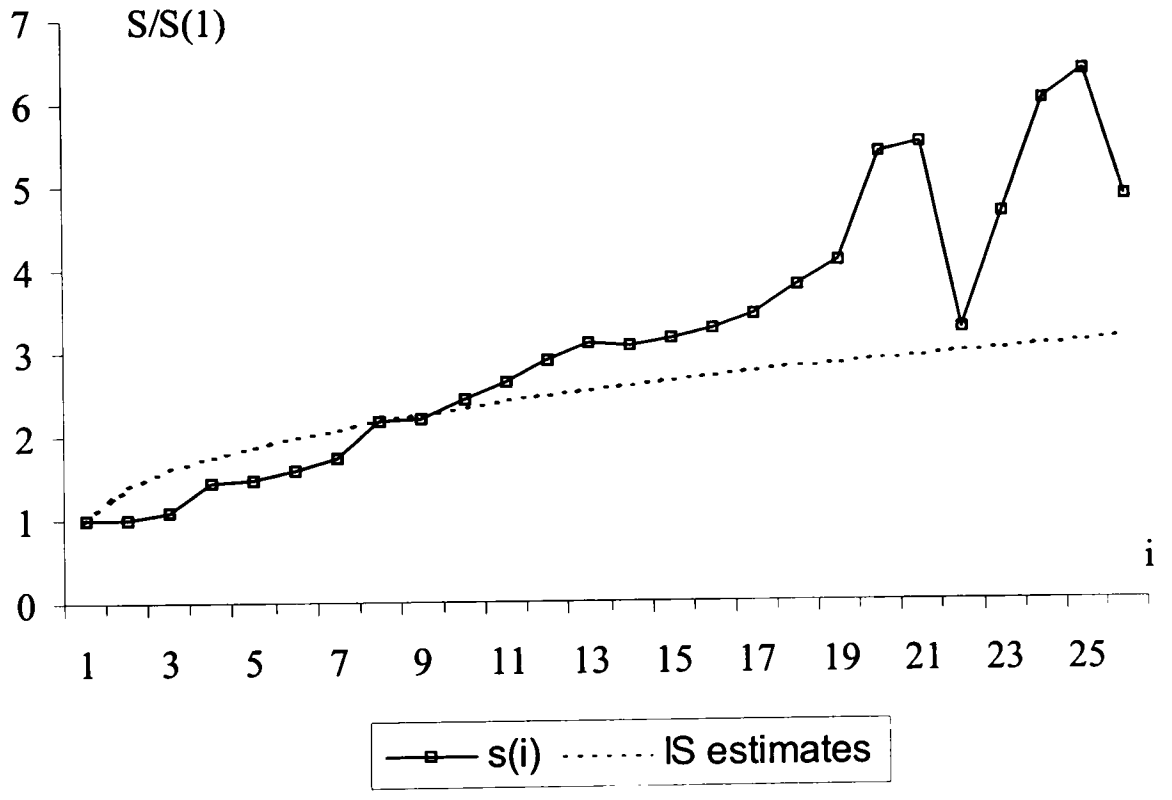

Figure 4.21. IBM OS/360 dataset and inverse square estimations.

The MMRE is 63.62%, a high value proof of a poor fit.

Plotting the results in a double logarithmic scale, as it was done for the Logica FW and Lucent examples, a exponential function can be calculated to fit the inverse square estimates. In this case the function is:

$$\hat{S}(i) = 1.032\, i^{0.3390},$$

which resembles again the equation $\hat{S}(i) = i^{1/3}$, presenting additional supporting evidence of what has been said before concerning the inverse square model: a value of k < 1 produces estimates with values that approximate the cubic root of the sequence number.

Revision of the QUICK Dataset

Finally, a revision of the QUICK evolution, that was used to start the current section, is presented. Table 4.14 contains the normalized growth, E(i), inverse square estimates, and MRE of every estimate.

Table 4.14. QUICK: Normalized inverse square estimations.

i	S	E	$\hat{S}$	MRE
1	1		1	0
2	2.015490534	1.015490534	2.674444584	0.65895405
3	3.456110155	1.970924042	2.908545747	0.547564408
4	3.24010327	1.68442615	3.1064791	0.13362417
5	3.232358003	1.566407794	3.279992854	0.047634851
6	4.246987952	2.134974401	3.435634203	0.811353749

Results from Table.14 are equivalent to those already presented in Table 4.9 and Figure 4.22, showing the inverse square estimates, has the same pattern that the pattern showed at Figure 4.9, only the vertical scale has changed.

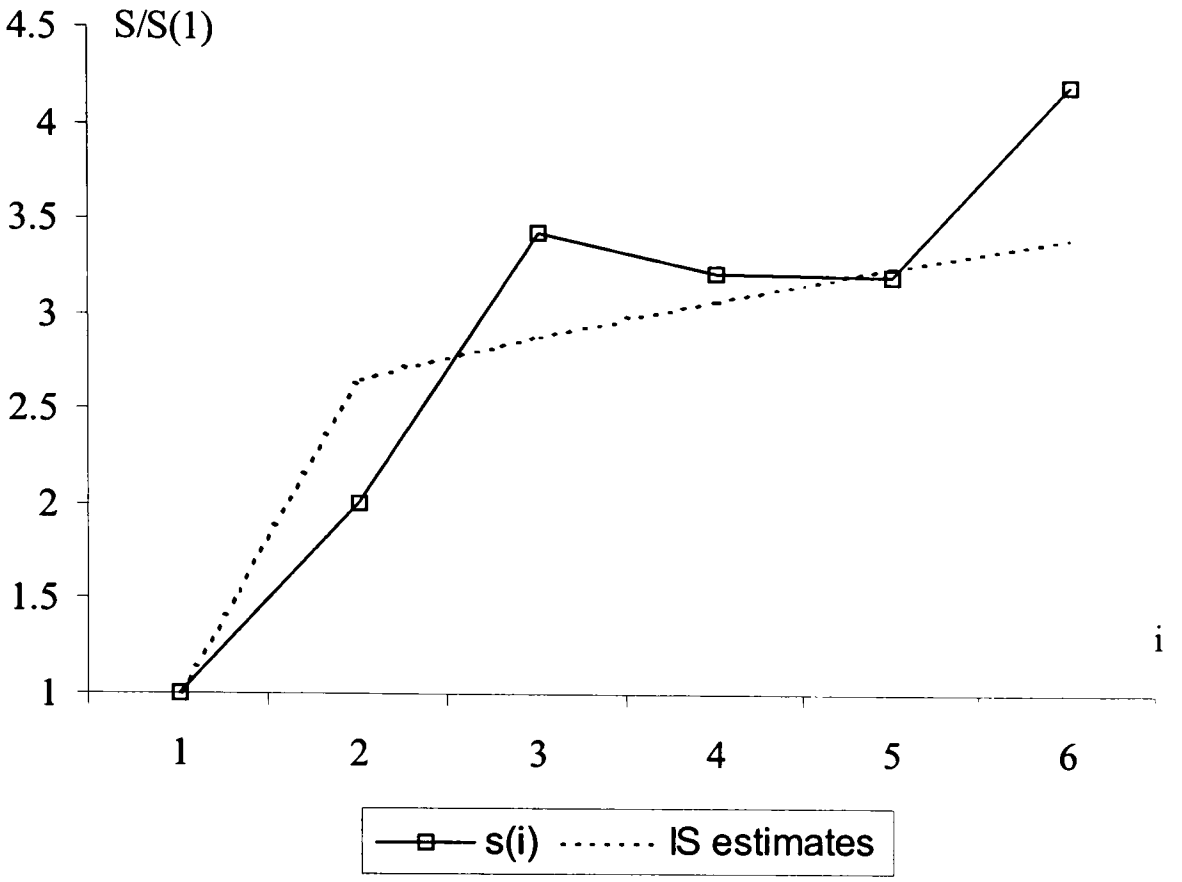

Figure 4.22. QUICK dataset and IS estimates, normalized scale.

$\underline{E}$ is 1.6744 and MMRE is again 12.2%. The QUICK's incremental growth rate is $\underline{k} = 0.6494$, which explains the pattern of the estimates. Since the value is close to one, the first estimate is large and is follow by small and almost constant increments. Using Eq. (4.22), the approximated value of the first estimate, $\hat{S}(2)$,when k = 1, would be:

$$\hat{S}(2) \sim 1 + \underline{E} = 1 + \underline{k}\,(0.38234 + 0.30361\ n).$$

In this particular case,

$$\hat{S}(2) \sim 1 + 0.6494*(0.38234 + 0.30361*6) = 3.0699,$$

similar to the actual value 2.674. Later increments are small and decreasing, although the estimation of Eq. (4.25) does not work since in this case $\underline{E}$ is not >> 1

Figure 4.23 plots the results in a double logarithmic scale.

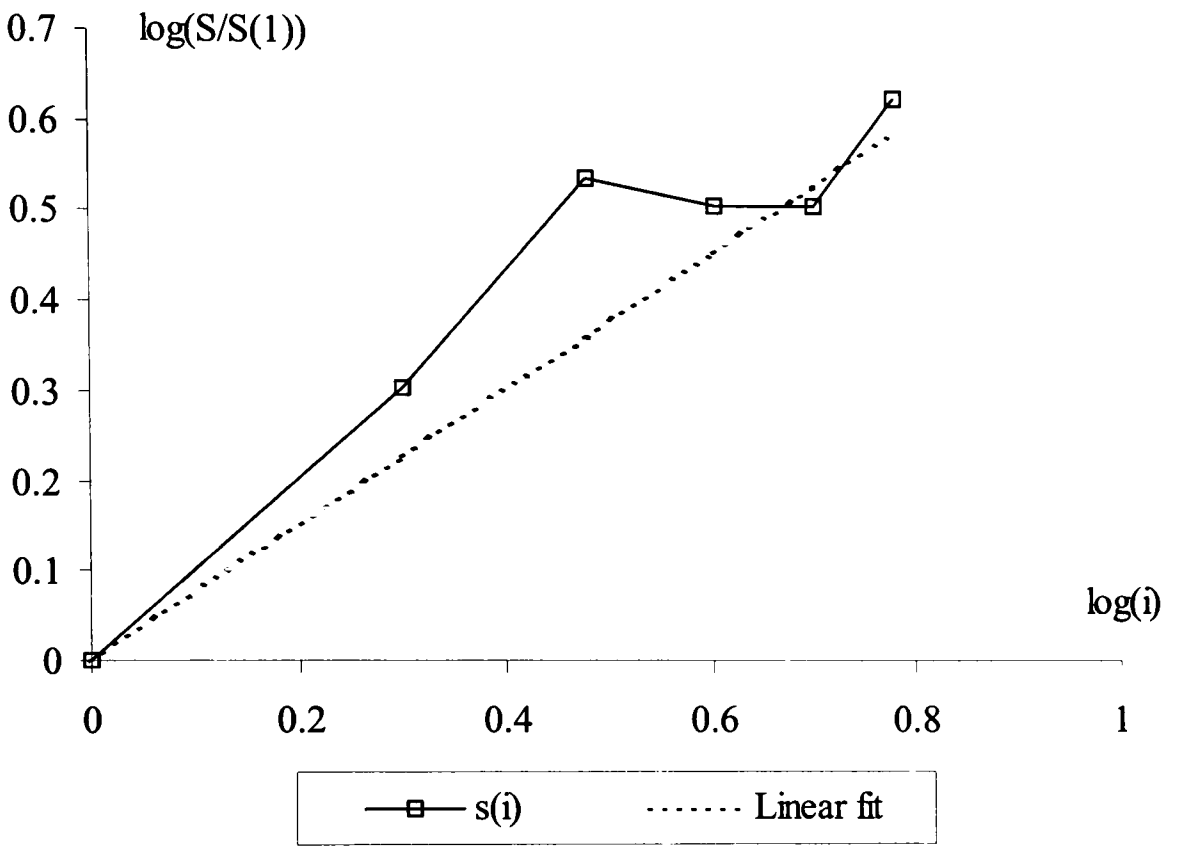

Figure 4.23. QUICK dataset and IS estimates, logarithmic scale.

A linear fit in the log. scale is given by the equation:

$$\log(\hat{S}(i)) = 0.7596 \log(i) + 0.1231,$$

which is equivalent to:

$$\hat{S}(i) = 1.131\, i^{0.7596},$$

a much larger growth than the growth found on previous examples. The MMRE of the exponential fit is 10.9%, slightly better and the inverse square fit.

Summary

The previous analysis of the inverse square model suggests that the pattern followed by the estimates depends upon the normalized growth rate of the system, $\underline{k}$. The constant $\underline{k}$ is defined as:

$$\underline{k} = (s(n)/s(1) - 1)/(n-1),$$

where s(n) is the system's size at version n, s(1) is the system's size at the first considered version and n is the number of versions of the system.

According to the value of $\underline{k}$, there are three possible cases:

1) $\underline{k} << 1$. In this case the estimates follow an almost straight line. The estimations are approximately:

$$\hat{S}(i) = 1 + (i-1)\,\underline{k}.$$

If the system follows an almost linear growth trend, the estimates will be accurate otherwise the model will fail.

2) $\underline{k}$ is in range [0.05, 0.2]. The estimates can be approximated by the equation:

$$\hat{S}(i) = i^{1/3}.$$

If the system follows a growth that is declining with the release version number, the inverse square model will produce a good fit otherwise the fit will be poor.

3) $\underline{k} > 0.5$. In this case the first estimate scores a large value and further estimates follow a linear trend. In an extreme case, when $\underline{k} >> 1$, all the estimates have a value approximately equal to $1 + n\underline{k}/2$.

Since the inverse square equations are continuous, the transition from one pattern to another is smooth and it is difficult to establish a defined border. The values of $\underline{k}$ that are pointed out as representative of each pattern come from the empirical results of the previous analyzed examples.

The examples that the inverse square model has been able to fit successfully correspond, all of them, to the second case. In this case, the inverse square model estimates can be approximated very closely by an exponential or power-law relation. This evidence is the basis for the next model: the constant work rate model.

The Constant Work Rate Model

Different papers have presented alternative models based on exponential or power-law relations. Shepperd (2000) proposes an exponential recurrence relation, given by the equations:

$$S(i) = S(i\text{-}1) + R(i)$$
$$R(i) = R(i\text{-}1)^{e}, \qquad (4.42)$$

where $\hat{S}(i)$ is the estimated system's size at version i, R(i) is the release size again at version i, and e is a constant calculated to maximize the fit.

Godfrey and Tu (2000) are the first to present how the inverse square model can be approximated by the equation:

$$S(i) = (3\ E\ i)^{1/3}, \qquad (4.43)$$

where S(i) is again the estimated system's size at version i, and E is a constant. However, after an unsuccessful application of the inverse square, the model is neglected and a polynomial fit is presented as a better option.

Finally, Lehman announces in a recent paper (Lehman 2001) the extension of the inverse square model, complemented now by a new model based on the equation:

$$\hat{S}(i) = i^{1/g}, \qquad (4.44)$$

where $\hat{S}$ is the estimate of the normalized system's size at version i and g is a constant ≥ 2. The distinction that g must be greater than two implies the assumption of systems the grow at declining rates, in agreement with Lehman's laws of software evolution.

All these ideas, plus the results presented in the previous sections, which show how an exponential equation such as

$$\hat{S}(i) = a\, i^{b}, \tag{4.45}$$

can produce good fits on most of the analyzed examples, plus the assumption that an exponent b in Eq. (4.45) without restrictions on its value can be used to model systems with sub-linear ($b < 1$), linear ($b \sim 1$), and super-linear ($b > 1$) growth trends, altogether seem enough reasons to explore a new model based on Eq. (4.45).

It must be pointed out that Eq. (4.45) and (4.44) are almost equivalent. In a logarithmic scale, the multiplier a has the effect of just a displacement along the vertical axis.

The name chosen for this model is the "constant work rate" model. In systems where b is equal to one, the Eq. (4.45) is just a linear progression:

$$\hat{S}(i) = a\, i,$$

exactly the effect that results of the third and fifth laws of software evolution, which predicts a linear long-term growth of software systems. The meaning of the parameter b is a merit or productivity factor for the dynamic system composed of the organization, the environment, and the software system. Systems where there is a lack of anti-regressive activities, or any other factor undermines the ability of the system to growth at a linear rate are modeled by a $b < 1$. Systems especially well tuned, based on an architecture prepared to grow, or efficient organizations (such as open-source projects, the case of the Linux kernel) score a value of b equal or higher than one. The fact that

both parameters a and b are constant during the whole lifetime of the software system is based on the third and eight laws of software evolution, which sustain that system dynamics are strong in evolutionary software systems and established from the early releases of the system.

Parameters a and b are calculated very easily as a linear regression fit in the double logarithmic scale of system's size versus release sequence number. A linear progression in a double logarithmic scale is expressed as:

$$\log(\hat{S}(i)) = c \log (i) + d, \tag{4.46}$$

c and d are calculated as the coefficients of the linear regression fit. Applying the exponential to both sides of the equation:

$$\hat{S}(i) = (e^{d})\, i^{c}, \tag{4.46}$$

Therefore,

$$\begin{aligned} a &= c \\ b &= e^{d}, \end{aligned} \tag{4.47}$$

where e is the base of the natural logarithms.

Optimal Algorithm

There is another alternative that can be used to calculate the model's parameters a and b. Instead of using a linear regression fit in the double logarithmic scale, the new approach consists of the application of a simplex search algorithm that calculates the values of those parameters that minimizes the error between the fit and the actual data. Since the fit's error is the lowest possible, the calculated parameters are optimal. Errors will be continued to be measured using the MMRE (Mean Magnitude of Relative Error).

Any simplex search algorithm is adequate. The present study makes use of Matlab (1997), which comes with powerful high-level constructors that calculate easily the optimal values for the a and b parameters.

Finally, it is important to highlight again that this is not a new model but just a different algorithm to calculate the model's parameters a and b.

Review of Literature Examples

The examples analyzed during the study of the inverse square model will be revisited again, applying the constant work rate model and using the regular (CWR) and the optimal (oCWR) algorithms to calculate the model's parameters.

Logica FW

The Logica FW was presented in the study of the inverse square model as an example of a system whose inverse square estimations follow an exponential function.

Figure 4.16 presented the Logica FW's dataset in a double-logarithmic scale, linear fit of the dataset in that scale is defined by the following expression:

$$\log(\hat{S}(i)) = 0.2779 \log(i) - 0.02306,$$

which is equal to:

$$\hat{S}(i) = 0.9772\, i^{0.2779}. \tag{4.48}$$

Figure 4.24 plots the original dataset as well as the results of Eq. (4.48).

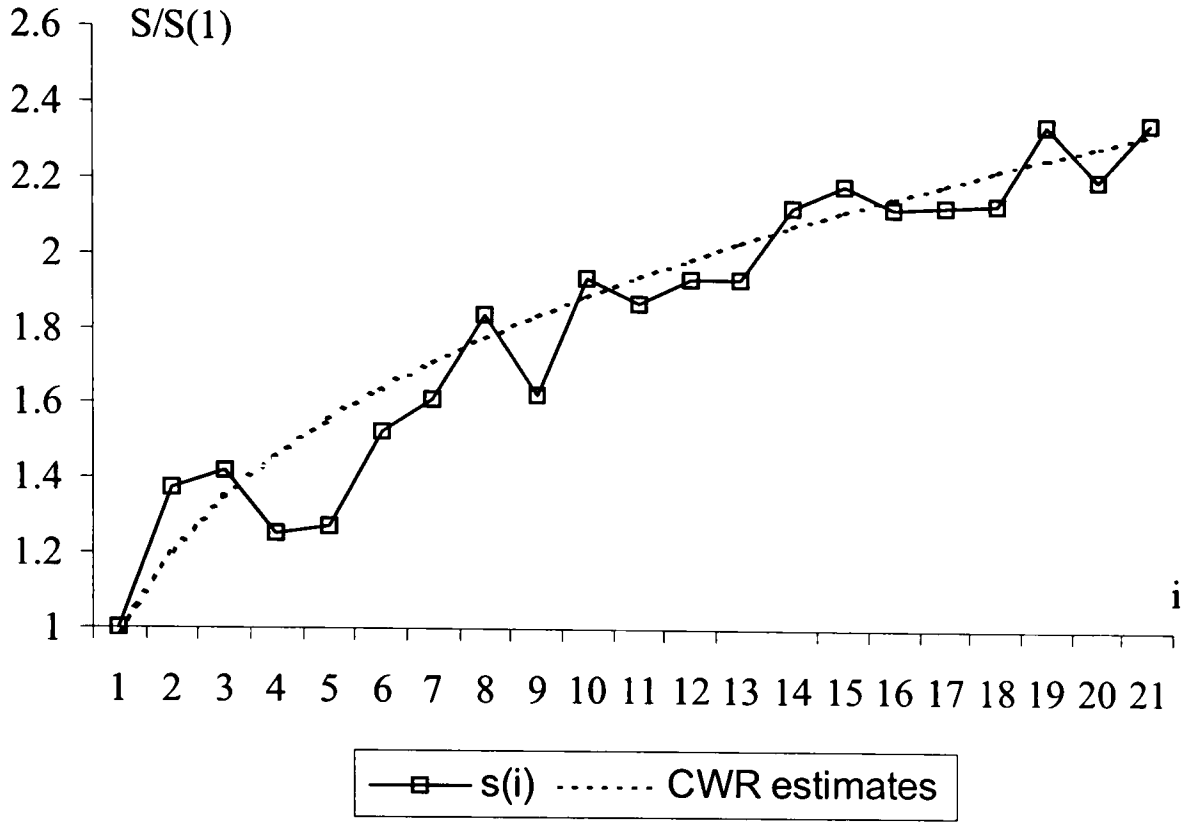

Figure 4.24. Logica FW dataset and CWR estimates.

The MMRE of the fit is 5.51%, slightly over the MMRE obtained by the inverse square fit that was 5.38%.

The optimal algorithm for this particular dataset produces the expression:

$$\hat{S}(i) = 0.9999\ i^{0.2693}$$

In this case the MMRE is 5.47%, almost the same that the obtained by the regular algorithm, proving, at least for this particular case, that the regular algorithm produces an exponential fit that is almost optimal.

Lucent

Using the same methodology, the Lucent dataset can be approximated by the expression:

$$\hat{S}(i) = 1.0192\ i^{0.3265},$$

which scores a MMRE = 6.105%, almost identical to the inverse square fit's MMRE (6.017%). Figure 4.25 shows the original and estimated dataset.

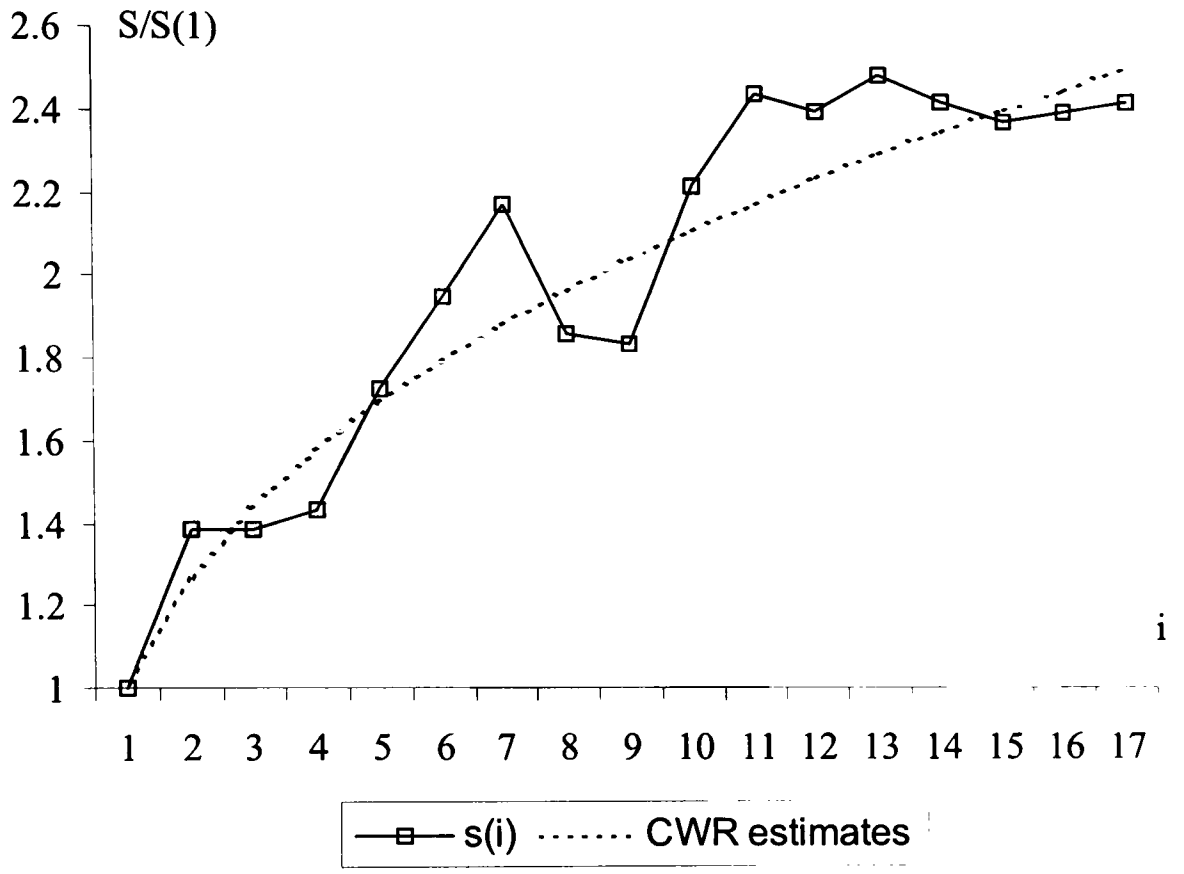

Figure 4.25. Lucent dataset and CWR estimates.

The optimal algorithm produces the expression:

$$S(i) = 1.0\ i^{0.3367},$$

which obtains an MMRE = 6.02%. Again, the fit error of both, optimal and regular algorithms, are almost the same.

The ability of the constant work rate model to fit both Logica FW and Lucent datasets is not really a great surprise. This result was anticipated during the previous section. More interesting is to see whether the new model is able to fit those examples that presented poor inverse square estimations.

Shepperd

This system was presented in the previous section as representative of a poor inverse square fit. The growth pattern seems to follow a linear progression with areas of different slopes. The inverse square model was not able to fit the dataset and, in the case of the constant work rate, this is the expression produced by the model:

$$\hat{S}(i) = 1.2111\ i^{1.4437},$$

the MMRE of the fit is 15.26% and Figure 4.26 plots the resulting estimates. The fit can be considered as correct and implies that the system would be growing at a super-linear rate, since the parameter b = 1.44 is higher than one.

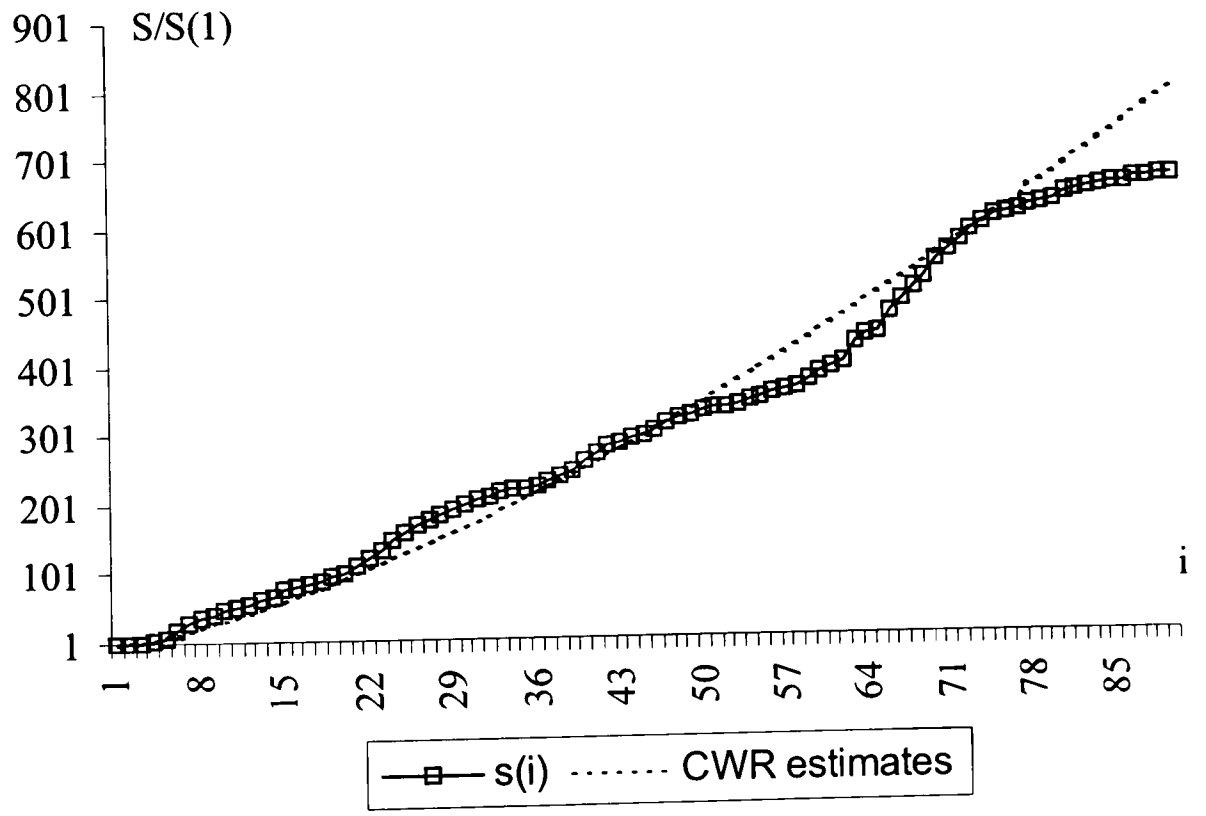

Figure 4.26. Shepperd's dataset and CWR estimates.

The optimal algorithm produces the expression:

$$\hat{S}(i) = 1.6018\, i^{1.3685},$$

and a MMRE = 14.6%, result similar again to the previously obtained by the regular algorithm.

Linux

The evolution of the Linux kernel presented one of the first examples in the literature of a model whose growth pattern cannot be explained by the inverse square model equations. According to the Godfrey and Tu, the authors of the study, the system exhibits a super-linear growth, point that could contradict some of the laws of software evolution.

In this particular case, the constant work rate model produces the expression:

$$\hat{S}(i) = 0.3293\, i^{0.7409}$$

with a fit error (MMRE) equal to 20.90%, which is an average fit, but not as good as the fits of previous examples. Figure 4.27 shows the estimates and the original dataset.

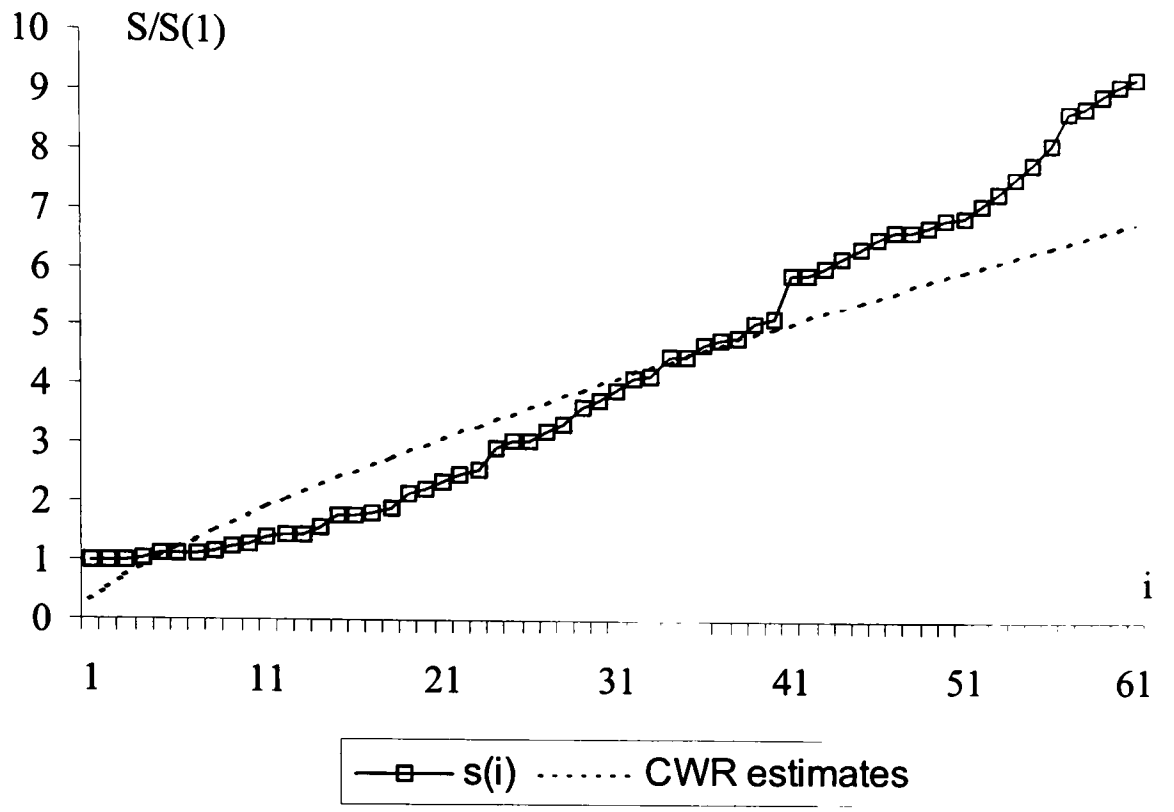

Figure 4.27. Linux kernel dataset and CWR estimates.

The plot confirms that the fit is not very accurate and only follows the evolution of the system during a short number of intermediate releases. In addition, the exponent b is lower than one, which indicates a estimated growth under-linear, not super-linear as the authors sustain.

However, the optimal algorithm produces a different estimate:

$$\hat{S}(i) = 0.0767\, i^{1.1518}.$$

The MMRE is in this case 11.80%, meaning a much better fit. The optimal estimate is plotted in Figure 4.28.

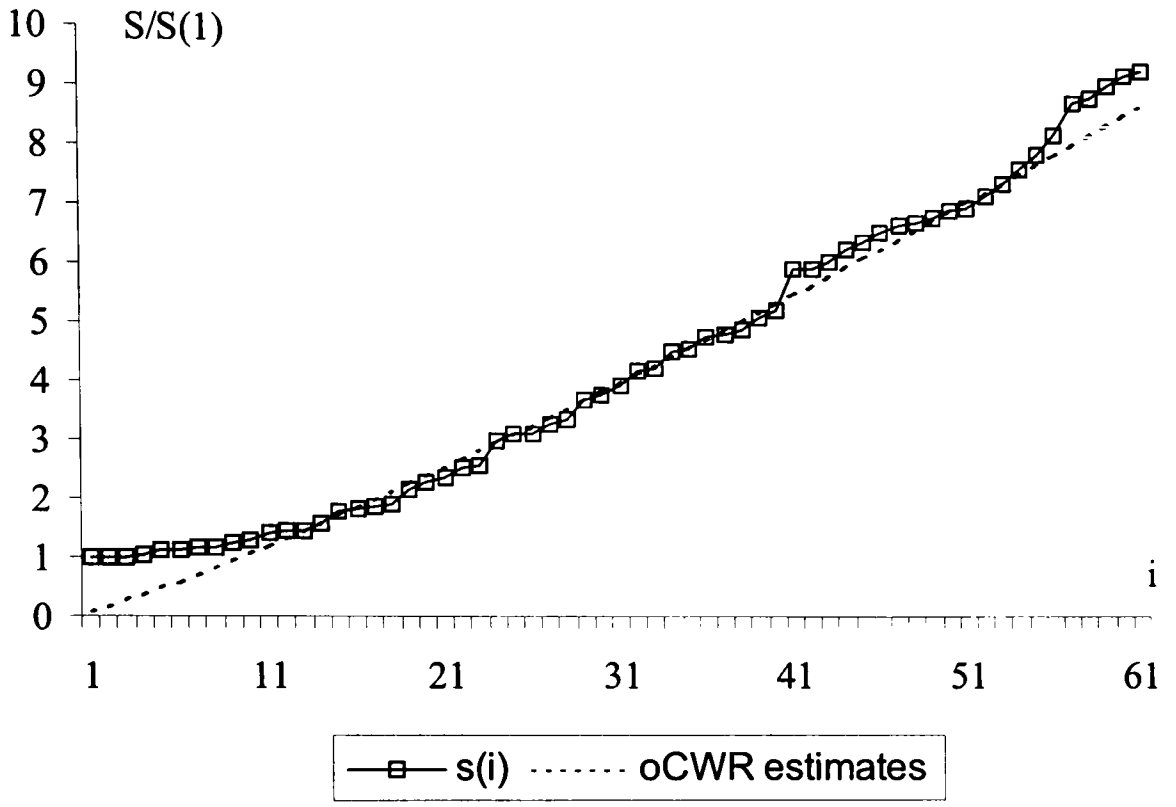

Figure 4.28. Linux kernel dataset and oCWR estimates.

In this particular case, the different estimates produced by the regular and optimal algorithm lays on the system growth pattern. Figure 4.29 shows the Linux kernel growth in a double logarithmic scale.

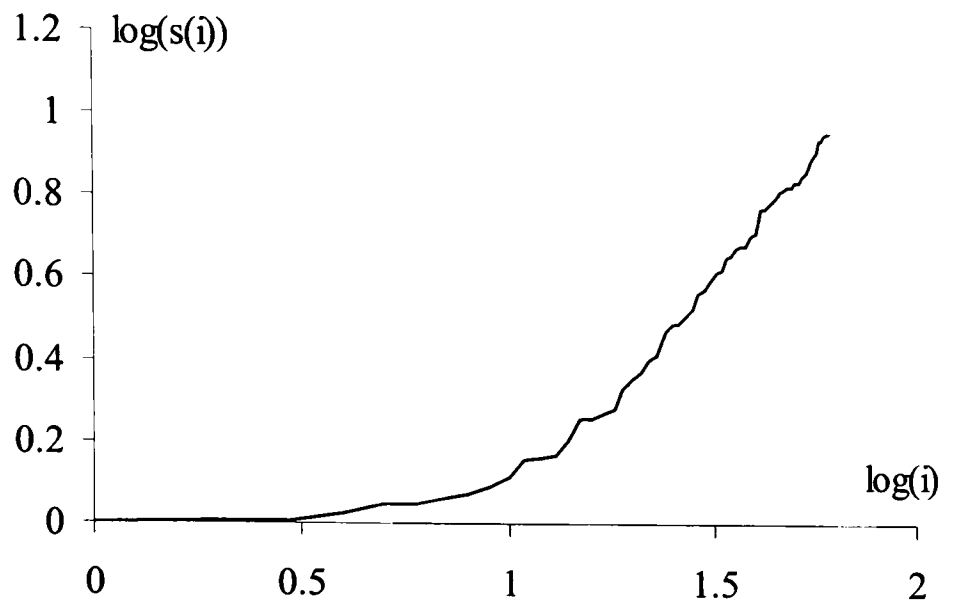

Figure 4.29. Linux kernel dataset in a double-log. scale.

Instead of following a linear progression, as happens with the Logica FW and Lucent examples, in this case the trend has two well defined slopes, with a crossover point in the neighborhood of log(i) = 1. A linear fit of this curve produces an estimated slope that is an average of both slopes, and therefore only fits well the intermediate area. The optimal algorithm produces a much better result because it tries to minimize the global MMRE, without regard of slopes.

IBM

Finally, the IBM OS/360 growth is estimated using the constant work rate model. The anticipated result is that the model will produce a good fit, with an exponent b around close to one, since the long-term growth trend of this system is almost linear.

The estimates follow the expression:

$$\hat{S}(i) = 0.6281\ i^{0.6281},$$

and the MMRE is 12.81%, much better than the inverse square fit that scored an MMRE = 63.62%. Results are presented on Figure 4.30.

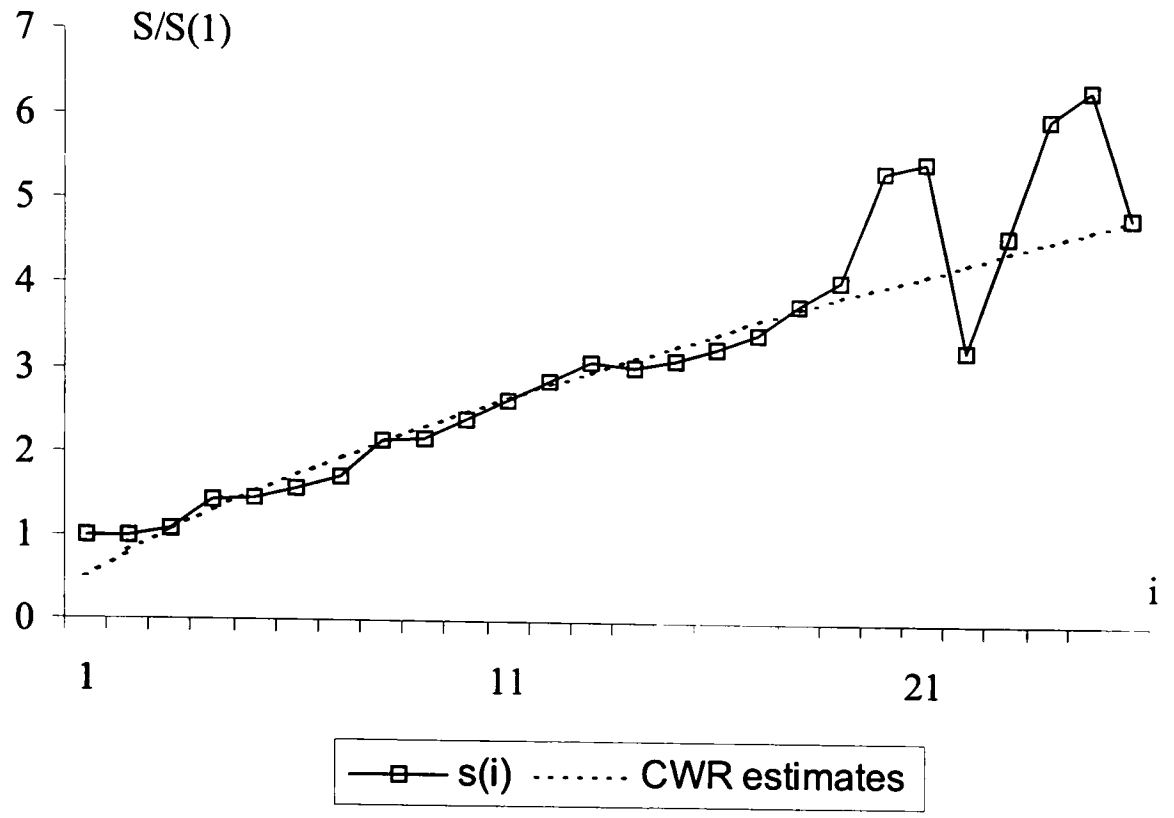

Figure 4.30. IBM OS/360 dataset and CWR estimates.

The optimal algorithm produces similar results:

$$\hat{S}(i) = 0.5045\, i^{0.6986},$$

and MMRE = 10.85%.

Summary of the Results

All the previous examples constitute evidence of the ability of the constant work rate model to produce good estimates on a variety of software systems. Applied to examples that the inverse square model is able to fit successfully, the constant work rate model produces similar fits, although not superior. Applied to systems whose growth cannot be explained by the inverse square model, the constant work rate model is able to produce good estimates. Model's parameters calculated using both, the regular and optimal algorithms, are similar in all but in one case that was affected by a sudden change

of the productivity or merit exponent (b). The optimal algorithm has consistently delivered good fits on all the analyzed examples.

Constant Work Rate Forecasting

One of the strongest arguments in favor of the inverse square model is the ability of the model of producing good estimates even when the number of releases available is very small (Turski 1996; Lehman et al. 1997). To prove this, an example is presented that shows how the value MMRE changes with the different number of releases used as an input to the model. The plots shows high MMRE values when the number of releases considered is 2, 3, 4, or 5, but at 6 the MMRE drops dramatically and keeps its low value as more releases are considered in the estimations. This result proves the excellent forecasting capability of the inverse square model and supports the laws of software evolution. The laws are reinforced since the model produces excellent forecasts even though only data coming from the first six releases is considered. This implies that the system dynamics are established firmly during those six versions, controlling the future evolution of the system even in releases far away in time.

Figure 4.31 shows a plot of the MMRE produced by constant work rate estimates using different number of releases. The dataset corresponds to the Lucent system, presented in a previous section of this study.

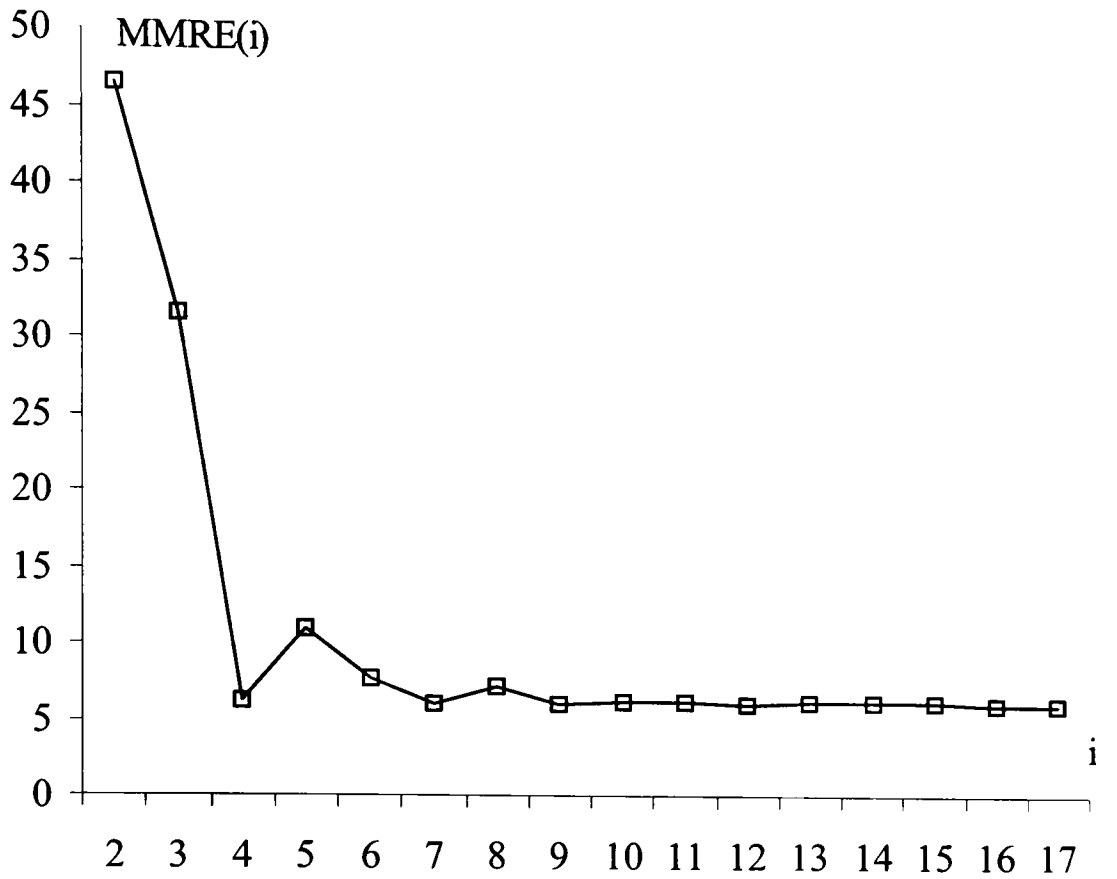

Figure 4.31. Lucent: MMRE as a function of the number of data points used in the CWR model.

The plot shows that the constant work rate has the same forecasting properties that were described for the inverse square model. When six or more system's releases are considered the MMRE of the estimates drops to values below 10% and very close to the MMRE achieved when all the releases are considered, which should be the best estimate.

This result confirms the assumption of strong system dynamics in software evolution systems, but negates that the inverse square model is the only one with this property.

Summary

A comparison of the inverse square and the constant work rate model reveals that the estimates that constant work rate model produces are at least as good as the estimates obtained by the inverse square model. This is not a surprise, since it has been

demonstrated that the inverse square model can be approximated by the same equation defined in the constant work rate model, at least in those cases where the inverse square's estimates follow the same pattern as the original data. In addition, the constant work rate model is able to estimate systems with a linear or super-linear growth trend, while the inverse square model consistently fails in such cases.

The optimal algorithm to calculate the constant work rate model's parameters has proven to consistently deliver excellent fits.

Finally, it must be highlighted again that both inverse square and constant work rate models estimates the long-term trend of the data, not every individual data-point. Since the data is affected by ripple, an indicator of the internal system dynamics in software evolution, the actual dataset oscillates around the long-term trend. In this context an estimate with MMRE lower than 10% can be considered as "excellent."

Table 4.15 compiles the estimates' MMRE of every example analyzed during the present study.

Table 4.15. Summary

Dataset	k	IS MMRE (%)	CWR MMRE (%)	oCWR MMRE (%)
Logica	0.068	5.38	5.51	5.47
Lucent	0.089	6.017	6.10	6.023
Shepperd	4.42	1449.3	15.26	14.61
Linux Kernel	0.14	38.1	20.90	11.80
IBM	0.16	63.62	12.81	10.85
QUICK	0.65	12.21	10.90	10.31

And finally, Figure 4.32 presents the results of Table 4.15 in a graphical format.

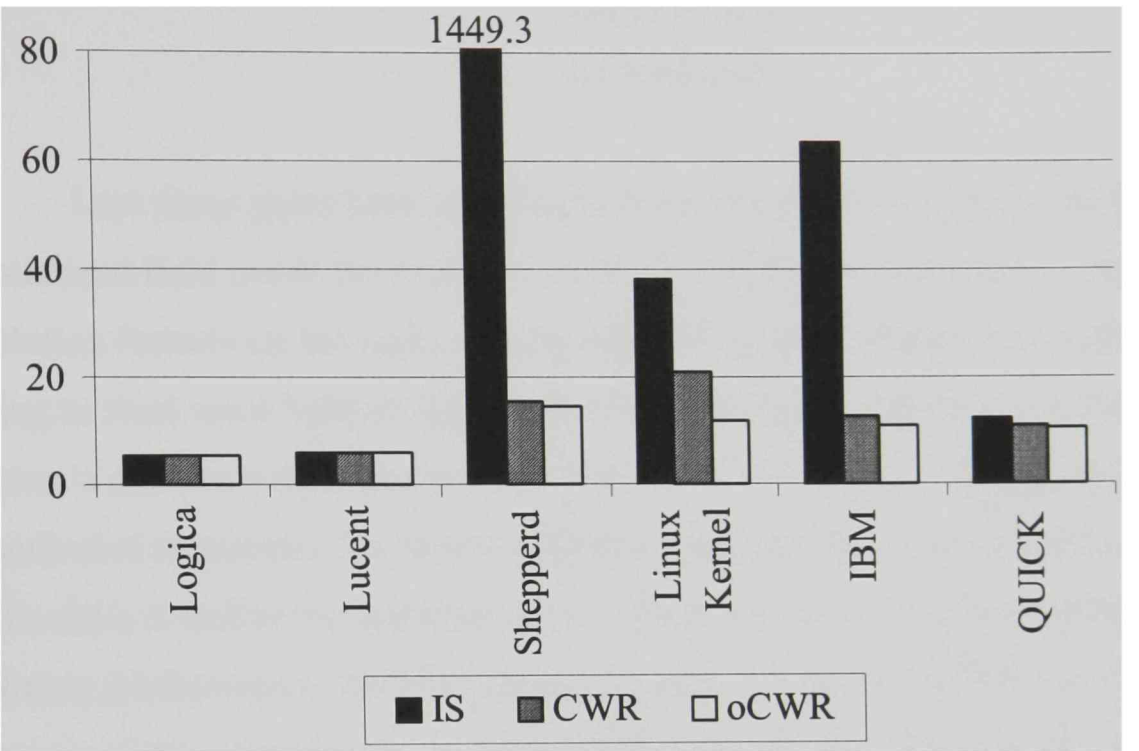

Figure 4.32. Summary of results: Models fit's MMRE.
IS: Inverse square model.
CWR: Constant work rate model, regular algorithm.
oCWR: Constant work rate model, optimal algorithm.

CHAPTER V
SUMMARY

Last thirty years have seen the birth and maturity of software evolution as a specialized field inside the vast area covered by software engineering. Software evolution focuses on the study of how software systems change during their lifetime, trying to shed some light on the nature of that change and the relationships between system's attributes from one release to the next. Studies in software evolution have contributed to advance the field of software engineering by, for instance, explaining the difficulties found in the implementation of software process improvement methods, defining mathematical models than can be used to forecast software system's attributes in future version, or clarifying the relationships among software systems, their development and maintenance organizations, and their operational environments.

Early empirical studies of software evolution discovered some patterns common to the evolution of every software system that was analyzed. These invariants were formalized in the "laws of software evolution," and include concepts such as: the perpetual state of change of software systems, the systemic nature of system's evolutionary processes, or the organizational inertia that smoothes the long-term system attributes variation. The election of the term "law" is intentional and expresses the universality of the invariants. However, as this study has intended to prove, the laws have undergone multiple revisions, and some early conclusions were contradicted by late discoveries, as new empirical studies of the evolution of software systems became available.

The most influential mathematical model of evolutionary software systems is the inverse square model, based on some of the laws of software evolution. The model explains the growth of software systems during their lifetime. This is of great practical importance, since the model can be used to forecast accurately a system's size many releases in the future, which allows management to plan and allocate resources well in advance. Although many papers have presented examples of software systems whose

growth is successfully modeled by the inverse square equations, some recent studies present prove of the opposite.

The QUICK application, a web-based system developed locally at Texas Tech is one example of a system whose growth cannot be explained by the inverse square model. This motivated the analysis of the underlying reasons, which is included in this study. The reasons were discovered in the some of the mathematical properties of the model. It has been demonstrated how the model is unable to explain software systems whose average growth is over a certain limit. That is the case of the QUICK application, and additionally, the literature presents examples of systems that growth well over that limit.

As a result, an alternative model, suggested in some recent studies of software evolution, has been analyzed as a plausible option. The model is referred here as the "constant work rate" model and is based on a simple exponential relation. The present study has compared both models and presented proof of the superiority of the constant work rate model, able to consistently produce accurate estimates on every software system analyzed.

Future Work

The presence of recent examples of software system whose growth seems to contradict some of the well-established principles of software evolution, opens the question whether modern software engineering practices are making a difference with respect to how software systems were development and maintained twenty years ago, during the maturation of the software evolution principles. One of the counter examples analyzed in this study is a small web-based application (QUICK), other is an open-source system (Linux kernel). The analysis of how the evolution of software systems developed and maintained using open-source approaches or agile methodologies seems to be a path that is worth exploration.

Continuing with black-box models, all the existent models of growth evolution focus on long-term trends, none considers the ripple superimposed on the trend, which is understood as a symptom of the systemic nature of evolutionary processes. To this

effect, the ripple is considered as random noise and neglected by the models. However, there are some techniques from the field of time-series analysis that are prepared to work with noisy data and in the past have been able to shed some light on the properties and nature of complex dynamic systems. Since software evolution processes constitute a dynamic system, whose nature is not yet well understood, the application of time-series analysis to datasets obtained from the evolution of software system's attributes seems to be another path worth to explore.

REFERENCES

Abdel-Hamid, Tarek and Madnick, Stuart E. Software Projects Dynamics, An Integrated Approach, Prentice Hall, Englewood Cliffs, NJ (1991).

Abdel-Hamid, Tarek and Madnick, Stuart E. "Lessons Learned from Modeling the Dynamics of Software Development." Communications of the ACM, 32. (1989): 1426-1438.

Boehm, Barry. W. "Industrial Software Metrics Top 10 List." IEEE Software, 4. (1987): 84-85.

Chatters, B. W., Lehman, M. M., Ramil, J. F. and Wernick, P. "Modeling A Software Evolution Process: A Long-term Case Study." Software Process Improvement and Practice, 2000.5 (2000): 91-102.

Garmus, David and Herron, David. Function Point Analysis. Addison-Wesley, Boston (2000).

Godfrey, Michael W. and Tu, Qiang. Evolution in Open Source Software: A Case Study. ICSM, San Jose, CA (2000).

Godfrey, Michael W. and Tu, Qiang. Growth, Evolution, and Structural Change in Open Source Software. to appear in Proc. of 2001 Intl. Workshop on Principles of Software Evolution, Vienna (2001).

Hall, Gregory A. and Munson, John C. "Software evolution: code delta and code churn." The Journal of Systems and Software, 54. (2000): 111-118.

Humphey, Watts and Kellner, Marc. Software Process Modeling: Principles of Entity Process Models, Technical Report CMU/SEI-89-TR-002 (1989).

Jones, Capers. "Backfiring: Converting Lines of Code to Function Points." Computer, 28.11 (1995): 87-88.

Kemerer, Chris F. and Slaughter, Sandra. "An Empirical Approach to Studying Software Evolution." IEEE Transactions on Software Engineering, 25.4 (1999): 493-509.

Lehman, M. M. Laws of Software Evolution Revisited. Proceedings of EWSPT'96, LNCS 1149., Springer Verlag, New York (1996).

Lehman, M. M. "FEAST - Feedback, evolution and software technology." Software Process Technology. 1487 (1998a): 150-150.

Lehman, M. M. FEAST/1 Final Report - Grant Number GR/K86008. London, Dept. of Comp., Imp. Col. (1998b)

Lehman, M. M. FEAST/2: Case for Support, ICSTM, DoC, EPSRC Proposal (1998c): 11.

Lehman, M. M. Feedback, Evolution and Software Technology - The Human Dimension. ICSE 20 Workshop, Human Dimensions in Successful Software Development, Kyoto, Japan (1998d).

Lehman, M. M. FEAST/2 Final Report - Grant Number GR/M44101. London, Dept. of Comp., Imp. Col. (2001)

Lehman, M. M. and Belady, L. A. "Laws of Program Evolution - Rules and Tools for Programming Management." Program Evolution - Processes of Software Change. London, Academic Press(1985a): 247-274. Why Software Projects Fail?, Infotech State of the Art Conference, 1978, Pergamon Press.

Lehman, M. M. and Belady, L. A. "On Understanding Laws, Evolution and Conservation in the Large-Program Life Cycle." Program Evolution - Processes of Software Change. London, Academic Press(1985b): 375-392. Original publication: Journal of Systems and Software, Vol. 1, No. 3, 1980, Elsevier, New York.

Lehman, M. M. and Belady, L. A. "The Programming Process." Program Evolution - Processes of Software Change. London, Academic Press(1985c): 39-83. Original publication: IBM Res. Rep. RC 2722. IBM Research Center, Yorktown Heights, New York, September 1969.

Lehman, M. M. and Belady, L. A. "Programs, Cities, Students - Limits to Growth?" Program Evolution - Processes of Software Change. London, Academic Press(1985d): 133-163. Original publication: Imperial College of Science and Technology, Inaugural Lecture Series, Vol. 9, 1974, pp. 211-229. Also in Programming Methodology, (D. Gries. ed.), Springer Verlag, New York,1978, pp. 42-62.

Lehman, M. M. and Belady, L. A. "Programs, Life Cycles and Laws of Software Evolution." Program Evolution - Processes of Software Change. London, Academic Press(1985e): 393-449. Proc. IEEE Spec. Issue on Software Engineering, Vol. 68, No. 9, Sept. 1980, pp. 1060-1076.

Lehman, M. M., Perry, D. E., Ramil, J. F., Turski, W. M. and Wernick, P. Metrics and Laws of Software Evolution - The Nineties View. Proceedings of the International Symposium on Software Metrics, Metrics 97, Albuquerque, New Mexico (1997).

Lehman, M. M. and Ramil, J. F. Towards a Theory of Software Evolution and its Practical Impact. International Symposium on Principles of Software Evolution - ISPSE 2000, IEEE Computer Society, Los Alamitos, CA (2000).

Lehman, M. M. and Ramil, J. F. "Rules and tools for software evolution planning and management." Annals of Software Engineering, 11. (2001): 15-44.

Lehman, M. M., Ramil, J. F. and Sandler, U. An Approach to Modelling Long-Term Growth Trends in Software Systems. Proc. ICSM 2001, Florence, Italy (2001).

Mathematica. Mathematica, v3.0, Wolfram Research (1996)

Matlab. Matlab, v5.1, The Math Works Inc. (1997)

Parnas, David L. Software Fundamentals: Collected Papers by David L. Parnas. Addison Wesley, Boston (2001).

Pfleeger, Shari Lawrence. "The Nature of System Change." IEEE SoftwareMay/June 1998 (1998): 87-90.

Ramil, J. F., Lehman, M. M. and Kahen, G. "The FEAST approach to quantitative process modelling of software evolution processes." Product Focused Software Process Improvement. 1840 (2000): 311-325.

Shepperd, Martin. Dynamic Models of Maintenance Behaviour. Intl. Workshop on Empirical Studies of Software Maintenance, WESS 2000, San Jose, CA (2000).

Turski, W. M. "Reference model for smooth growth of software systems." IEEE Transactions on Software Engineering, 22.8 (1996): 599-600.

Wernick, P. and Lehman, M. M. "Software process white box modelling for FEAST/1." Journal of Systems and Software, 46.2-3 (1999): 193-201.

APPENDIX

Matlab code for the CWR model

```
%Function CWR
function CWR()

%Dataset for the QUICK evolution:
Sizes = [1162,2342,4016,3765,3756,4935];

%Dataset is normalized first:
n = length (Sizes);
S1 = Sizes(1);
Sizes = Sizes/S1;

y = Sizes(1:n);
x = (1:n);

%Log transformation:
ly = log(y);
lx = log(x);

%Simple linear fit:
fit = polyfit(lx,ly,1);
a = exp(fit(2))
b = fit(1)

%Optimal CWR
%Initial approximations of a and b model's parameters:
a1 = a;
b1 = b;
```

```
[X,options] = fmins('ocwr',[a1,b1],[],[],y);
a = X(1)
b = X(2)
x = (1:n);
estimate = a * (x .^ b);
plot(estimate,'k-')

%Function ocwr
function [cost_] = ocwr(a_X, a_Sizes)

n = length(a_Sizes);
a = a_X(1);
b = a_X(2);
y = zeros(1,n);
error_ = zeros(1,n);

%MMRE calculation
for i=1:N
  y(i) = a*(i^b);
  error_(i) = abs(a_Sizes(i) - y(i))/a_Sizes(i);
end

mmre = 100*sum(error_)/n;

%Return the MMRE as the cost factor
cost_ = mmre;
```

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

_____ _____

Student Signature Date

Disagree (Permission is not granted.)

_____ _____

Student Signature Date