Laws of Software Evolution Revisited

M. M. Lehman

Department of Computing, Imperial College, London SW7 2BZ, UK

Abstract. Data obtained during a 1968 study of the software process [8] led to an investigation of the evolution of OS/360 [13] and and, over a period of twenty years, to formulation of eight *Laws of Software Evolution.* The FEAST project recently initiated (see sections 4 - 6 below) is expected to throw additional light on the phenomenology underlying these laws, to increase understanding of them, to explore their finer detail, to expose their wider relevance and implications and to develop means for their beneficial exploitation. This paper is intended to trigger wider interest in the laws and in the FEAST study of feedback and feedback control in the context of the software process and its improvement to ensure beneficial exploitation of their potential.

1 Historical Background

The first three of a total of now eight laws of software evolution[1] were formulated in the mid seventies [9] arising from analysis of data first acquired during a study of the IBM programming process [8]. These three were discussed in somewhat greater detail in 1978 [10]. Two further laws were introduced in a 1980 paper [11] with the sixth introduced in a footnote [15]. The remaining two have been discussed in presentations but are published here for the first time. All relate specifically to *E*-type systems [12] that is, broadly speaking, to software systems that solve a problem or implement a computer application in the *real world.*

Section 2 restates and briefly discusses the laws stressing, in particular, the role of process feedback in generating the phenomenology they reflect. This is followed in section 3 by an equally brief discussion of the use of the term *laws* in describing the observations from which they were inferred. Section 4 introduces the FEAST project, the concepts and observations on which it is based and outlines the planned, now funded FEAST/1 investigation relating it to a broader long term, multi-disciplinary collaborative investigation which must follow.

2 The Laws

2.1 I — Continuing Change

An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.

[1] Numbered in order of formulation and publication. Over the years the names and detailed wording of some of the laws have been modified but the underlying understanding they reflect has remained essentially the same

This law is in accord with universal experience. It suggests that the growth of an *E*-type software system is in some ways akin to that of an organism. It result, however, from *feedback* pressures caused by mismatch between the software and its operational domain, whereas that of biological organisms results primarily from pressures within the organism itself.

This need for continuing adaptation and evolution is intrinsic to *E*-type applications and software. It is, in part, due to the fact that development, installation and operation of the software changes the application and its operational domain so creating mismatch between the two. Evolution is achieved in a feedback driven and controlled maintenance process. If the consequent pressure for evolution to adapt to the new situation is resisted the degree of satisfaction provided by the system in execution declines with time.

2.2 II — Increasing Complexity

As a program is evolved its complexity increases unless work is done to maintain or reduce it.

This law may be an analogue of the second law of thermodynamics or an instance of it [20]. It results from the imposition of change upon change upon change as the system is adapted to a changing operational environment. As the need for adaptation arises (first and seventh laws) and changes are successively implemented, interactions and dependencies between the system elements increase in an unstructured pattern and lead to an increase in system entropy.

If the growth in complexity is not constrained, the progressive effort [1] needed to maintain the system satisfactory becomes increasingly difficult. If anti regressive effort [9] is invested to combat the growth in complexity, less effort is available for system growth. Given that resources are always limited the rate of system growth declines as the system ages WHICHEVER STRATEGY IS FOLLOWED. In practice the balance between progressive and anti-regressive activity is determined by feedback.

2.3 III — Self Regulation

The program evolution process is self regulating with close to normal distribution of measures of product and process attributes.

The evolution of industrially produced *E*-type software is implemented by a technical team operating within a larger organisation. The interests and goals of the latter extend far beyond completion of the system in question. Checks and balances will have been established by corporate and local management to ensure that operational rules are followed and organisational goals at all levels are met. The positive and negative feedback controls that implement these checks and balances provide one example of feedback driven growth and stabilisation mechanisms. There will be many others. Together they establish a disciplined dynamics whose parameters are, at least in part, normally distributed [5] being the consequence of a large number of pseudo independent managerial and

implementation decisions. After a while this dynamics determines many of the growth and other development characteristics of the evolving product.

2.4 IV — Conservation of Organisational Stability (invariant work rate)

The average effective global activity rate on an evolving system is invariant over the product life time.

Of the eight laws this fourth law was and remains the most counter intuitive. By and large it is still generally believed that the effort expended on system growth and evolution is determined by managerial decision. To some degree corporate and local management certainly do control activity targets and resource allocation to a project, system or activity. Their ability to do this is, however, constrained by external forces, trade unions or the availability of personnel with appropriate skills for example. But as suggested by the third law the effort usefully expended, that is to achieve satisfactory results, is also determined by system attributes, complexity for example, that are analogous to attributes such as inertia and momentum in mechanical systems. As indicated by Brooks [4] circumstances may even arise where, for example, providing additional resources may actually reduce the effective rate of productive output as a result of increased communication and other overheads or decreases in process quality. In any practical situation the level of activity is not decided exclusively by management edict but by a host of feedback inputs and controls. Project data so far analysed suggests that in practice this leads to stabilisation at a fairly constant level.

2.5 V — Conservation of Familiarity

During the active life of an evolving program, the content of successive releases is statistically invariant.

One of the factors that determines the progress of a software development is the familiarity of all involved with its goals. The more changes and additions are associated with, say, a particular release, the more difficult it is to for all concerned to be be aware, to understand and to appreciate what is required of them. The rate and quality of progress and other parameters are influenced, even limited, by the rate of acquisition of the necessary information by the participants collectively and individually. The larger work package the more challenging mastery of the matter to be acquired. Data to date suggests that this is not a nice linear relationship but one in which there are one or more critical size levels which if exceed trigger behavioural change. The rapidity of the change suggests that here too feedback mechanisms play an important role.

2.6 VI — Continuing Growth

Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

At first sight the sixth law, Continuing Growth, appears little different to the first law, Continuing Change. In fact, however, the two laws reflect distinct, though not unrelated, phenomena. The first relates closely to the Software Uncertainty Principle [14, 15] which recognises, *inter alia*, that, for example, changes in the operational domain will invalidate assumptions embedded in *E*-type software and so cause unexpected behaviour in execution. As users become aware of consequent imprecision, unsatisfactory or incorrect operation they will demand a fix. The law reflects the feedback impact of system usage on the application, on its domain, on users and on assumptions made during development and maintenance of the software. Change is inevitable in software as it is in any active system. The rate at which pressure for change develops in software relative to human perception and the intolerance for mismatch if changes are not implemented is, however, greater than for other real world systems. Hence the common perception of continuing maintenance and for the view that *E*-type software must be seen and treated as organisms.

The sixth law addresses change deriving from a different source. When a new system is to be developed (whether from scratch or from of-the-shelf (OTS) components) or an existing one is to be upgraded the first input required, though often not provided, is a detailed description (model) of the application in its actual or desired operational domain. This application domain model is the foundation and definitional source of the requirements and specification for the required system. Because of limitations arising from constraints such as budget, delivery dates, technology and understanding of the application in its domain[2] the domain model and the definitions of requirements and specifications have to be bounded. Items relating to candidate functional, behavioural and other attributes that cannot be accommodated, for whatever the reason, within the imposed constraints will be explicitly or implicitly excluded. Sooner or later omitted attributes will become the bottlenecks and irritants in usage as the user has to replace automated operation with human intervention. Hence they also lead to demand for change, in this case growth in capability to provide attributes that could not be accommodated in the original development. Behaviour and functionality associated with or arising from system execution and implemented by humans at the interface, by ancillary systems or by applications software is integrated into the system to remove bottlenecks and/or sources of imprecision or error. The *E*-type system inevitably grows with time driven by feedback from user and marketeers.

2.7 VII — Declining Quality

E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment.

Discussion of the sixth law made brief mention of the Principle of Software Uncertainty. In one of its alternative formulations this states that the *real world*

[2] The constraints are here listed on the order of normal organisational concerns not of technical importance or operational significance

outcome of E-type software execution is inherently uncertain with precise area of uncertainty also not knowable. A system that has performed satisfactorily for some period of time may suddenly exhibit unexpected, previously unobserved, behaviour. Several causes to explain this phenomenon, all valid and complementary, may be identified [14, 15] and encapsulated in the Principle of Uncertainty. The simplest is associated with the fact that there is a gap between the potentially unbounded *E*-type application and its real world operational domain and the finite system developed with finite resources in finite time to address a constrained application in a constrained domain. The constraints represent assumptions about the application, the theory that underlines its component parts, the domain, human behaviour, the execution system and so on together with the passive or active reaction of all these to each other and to system operation. They are required because an unbounded system cannot be constructed. They are adopted, explicitly or implicitly, according to the perceptions and understanding of the application and its domain at the time of implementation and are embedded in the system to bridge the gap. But the real world is always changing. In fact, such change is, in part, driven or accelerated by the process of computerisation. Hence, however, justified or valid the assumptions will have been at the time of adoption the full set embedded in the system will contain progressively more that are not (no longer) valid or justified. What the consequence of encountering the embodiment of such an assumption during execution will be is unpredictable. Hence there must be a degree of uncertainty, unpredictability, about *E*-type system behaviour.

The seventh law states that such uncertainty increases with time unless successful attempts to detect and rectify the embodiment are taken as part of the maintenance activity. It is also a consequence of the fact that familiarity breeds contempt. As time elapses and the user community become more perceptive and expectant; alternative products become available, the criteria of acceptability and satisfaction change. Ultimately quality of a product must relate to user satisfaction. Hence the quality of a software system declines with time and once again information feedback plays a key role.

2.8 VIII — Feedback System

E-type Programming Processes constitute Multi-loop, Multi-level Feedback systems and must be treated as such to be successfully modified or improved.

This brief outline of the Laws of Software Evolution has included references to the role of feedback in the process. These remarks may be generalised with the observation that global *E*-type software system evolution processes constitute complex multi-loop, multi-level, multi-agency feedback systems. The role of feedback in the process was, in fact, recognised almost from the start of the detailed study of the software process [8]; a study that led to the wider exploration of software evolution. A 1972 paper [3], for example, discussing an earlier version of the full OS/360 IBM operating system growth curve reproduced in figure 1, observed that, "the ripple is typical of a self stabilising process with positive and negative feedback loops. From a long-range point of view the rate

of system grow this self-regulatory, despite the fact that many different causes control the selection of work implemented in each release, with varying budgets, increasing numbers of users reporting faults or desiring new capability, varying management attitudes towards system enhancement, changing release intervals and improving methods".

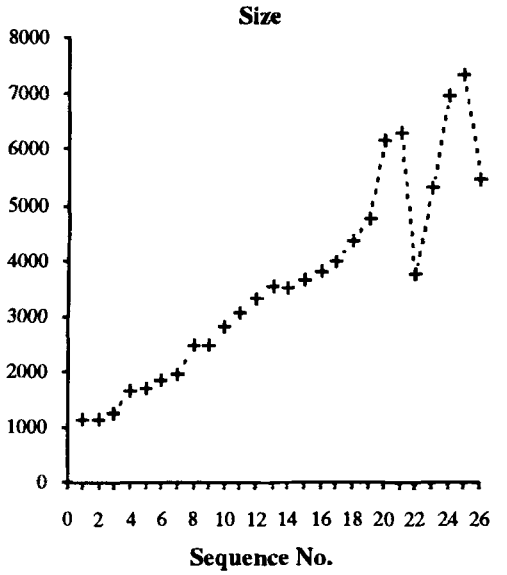

Fig. 1. The growth of OS/360

This plot, and others like it, provides many clues to the properties of the overall OS/360 evolution process [12]. Here the principle interest is in the indicators of feedback control. The preceding paragraph referred to the ripple effect which was, and is, believed to reflect process self stabilisation through negative feedback. The *chaotic* (in a strictly technical sense) behaviour exhibited over the final releases is similarly interpreted as *instability* due to excessive positive feedback evidenced by the excessive feedback driven growth in evolving from release 19 to release 20.

The conclusions drawn from the 1970s studies recognised that the presence of feedback in the software development process and organisation required one to "regard the organisation developing and maintaining a large program as a system in the system theoretic sense. Observation has shown that the system behaves as a self stabilising feedback system. The process leads to an organisation and a process dominated by feedback with long range trends and invariances" [10]. All these observations and many more are encapsulated in the eighth law.

The eighth law is, therefore, not new though it was not originally formulated as a *law*. Its formulation illustrates how one may obtain and interpret clues as to the nature of a phenomenon, which, as they accumulate, provide a growing observation base, behavioural and factual, from which one builds a theory of behaviour. Such theory, when established in outline, may then be refuted or refined, tested and improved by further observation and experimentation. What is surprising is that it has taken more than fifteen years for the full implications of the observations of the 70s to be realised; that only now has it become apparent that statement of a incontrovertible fact as a law may have important benefits. The full impact of this law on the formulation of the other seven laws, on their impact or on their practical implications has not yet been systematically explored. It is intended that this be one of the outputs of the FEAST project.

3 Why Laws?

Before introducing the FEAST programme it seems appropriate to briefly examine the use of the term *laws* rather than words such as *observations* or *hypotheses*.

When the laws were first presented in the literature [9, 10, 11, 12] widespread criticism of the use of the term *laws* was voiced. It was suggested, for example, that the observed phenomena reflected the behaviour of human designers, implementors, managers and users. Thus they could not be laws in the normal sense of the word. Others felt that the phenomenology they reflect could be altered at will, by education for example. Still others observed that the behaviour described was intimately bound up with a particular organisation (IBM) and/or a particular system (OS/360) and/or the software system development technology of the 70s. As such the observed phenomena lacked the generality that use of the term *law* implies. The refutation of the first view was based on the facts that the laws reflect the cooperative activity of many individual and organisational behaviour. Their analysis requires, therefore, experience in disciplines removed from computer science and software engineering, psychology, organisation theory and management science, for example. Moreover since, in part at least, the characteristics identified by the laws stem from the feedback system nature of the software process, discipline such as control theory or system dynamics the phenomenology associated with such systems also plays a role. The observed behaviour reflects the environment within which software engineering operates and the laws governing that behaviour are not part of that discipline. From the point of view of software engineering they must be accepted as laws [10, 12]. The study was, however, not limited to the examination of one system. Data from several other sources was also examined and upheld, or at least did not contradict, the earlier conclusions [12]. With hindsight it must however be admitted that a strong intuition played a part in the decision to use the term *law*. Only now is that intuition beginning to be supported [21].

A second criticism is exemplified in Lawrence's ICSE 6 paper [7]. His view, first expressed by Chong [5], was that the analysis of the data from which the laws were inferred was not statistically significant. As a criticism of the laws,

however, this too is unfortunate. It stems from a misunderstanding. At no time was any claim made that the laws were derived from statistical models of the observed behaviour of the five or so systems studied. How could they have been, after all, even in the best case there were only some twenty data points? Models were derived by curve fitting. These led to exploratory, statistical probes to help and guide interpretation of the observed behaviour in terms of ones knowledge and understanding of the observed process and of related activities. The laws represent an emerging theory of software process and software evolution based on many inputs including the reality of software development. Observation and modelling, numerical, statistical or otherwise, provide guides, circumstantial evidence and inspiration which must then be tested against the real thing. As such evidence accumulates and to the extent that the models support and extend each other they may eventually provide statistically significant support. But that time is not yet here.

4 FEAST

4.1 The FEAST Project

This project is rooted in hypotheses outlined in section 4.4. It seeks to investigate the role and influence of feedback in the evolution of *E*-type software systems and on the improvement of the software process. Hence the name FEAST: *F*eedback, *E*volution *A*nd *S*oftware *T*echnology.

4.2 Process Improvement

The first recorded mention of a need for improvement in the process of programming is believed to be in remarks by Wilkes, Wheeler and Gill in connection with their invention of the concept of subroutines [22]. In the preface to their book *The Preparation of Programs for an Electronic Digital Computer*, they write "The methods of preparing programs for the EDSAC described in this book were developed with a view to *reducing to a minimum* the amount of *labour* required and hence of making it *feasible* to use the machine for problems that require only a few hours of computing time as well as for those which require many hours"[3]. The search for improvement has been in the forefront of programming research and development ever since. Some two years ago the question was asked why, despite the many innovations in programming methodology, in process organisation and in project management over the past 45 years industrial software and systems development still suffers major problems?

Why has improvement of the process proved so difficult and slow despite a massive research investment by government, industry and academia? The conventional approach to answering this question associates lack of global progress with problems of individual innovations. It is suggested, for example, that high level languages have not solved the overall problem because their impact is local

[3] The present author's italics

not global. Their use may produce a three to five fold speed up in coding. It certainly makes the resultant text more intelligible, hence less error prone. But the effort that goes into coding represents only a small proportion of the total system development effort. The local gain is only minimally reflected in the global process. Formal methods have not made a major impact on industrial development effectiveness because people do not have or do not like the mathematical skill required to use the methods effectively. Thus they have not found widespread application in industry. CASE has not delivered the expected promise because organisations adopt methods and acquire tools one at a time. Only when they have several tools, is it discovered that they cannot be used together effectively. As often as not as much is lost in progressing work from one method/tool to the next as was gained by using the first tool in the first place; truly discouraging. And so on.

This technique by technique approach to explain the source of difficulty in major process improvement is unsatisfactory as, indeed, is the search for improvement through the introduction of innovations one by one. For complex systems - and the software development process is that - the latter approach which is akin to local optimisation normally leads to global sub optimisation. Where so many innovations have failed to deliver their promise it would surely be appropriate, in the first instance at least, to look for a common cause. One must search for an intrinsic constraint on the improvement of the global process of transforming an application concept into an operational system and on maintaining the resultant system satisfactory over its working life.

4.3 The Process as a Feedback System

Given that formulation of the issue, an immediate solution to the conundrum suggested itself. The *global industrial software process is a feedback system.* It involves not only technical development but process engineering, many levels of management, corporate executes, marketing, user support and so on. The direction, quality, effectiveness and output of the process is a complex function of the directive, control and information flow between many *agencies* and *agents.* that drive, guide, redirect and generally seek to control the process. In such a feedback systems positive feedback triggers or accelerates growth and may lead to instability (as seen in the final releases of OS/360 - fig. 1). Negative feedback, on the other hand, has a stabilising influence. When negative feedback is applied over some forward path element, be it a single mechanism or a subsystem, changes in the output of that element in response to changes in its characteristics are reduced by approximately the gain in the feedback loop. The precise impact depends, of course, also on the delay or phase shift in the loop. One achieves global stability in the face of changes in element characteristics.

With many feedback paths in a system, a complex relationship exists between internal changes, the characteristics of communication and control various paths between elements, internal interactions and observable external behaviour. In particular replacement of an element with one having different characteristics, even addition of a new element, may make no significant or even perceptible

difference to observed system behaviour. At best the observed changes will be much less than might have been anticipated from changes in element characteristics. To impact global system behaviour significantly by internal changes requires adjustment of feedback paths and attributes of their mechanisms in a way that is neither simple nor self evident. Changes to forward path elements alone will have a much smaller global impact than analysis of the local impact would suggest.

The insight that followed acknowledgment of the software process as a feedback system should now be self evident. The process must surely possess the same general feedback system property. The impact of internal changes to process mechanisms must surely be constrained by the many feedback paths in the process and the organisations within it is embedded and applied. The visible benefit derived from the introduction of improved languages, methods, procedures or tools in the forward, development path of the process can only have limited impact. Locally an innovation, whatever its nature, might prove to be most beneficial, yielding significant improvement in productivity, quality, responsiveness or whatever. But such gain is likely to be attenuated or even inverted by the feedback mechanisms that certainly modify, perhaps determine, overall process characteristics.

Process improvement efforts should not be concentrated on forward path technical development. Nor is it sufficient to extend consideration to system definition steps such as requirements engineering, system specification and design. Communication channels between technical development, the organisations within which it is embedded and the user community must be considered. These channels include feed forward and feedback mechanisms operating in the global process, which be improved and tuned to the state of the forward path at any point in time.

Improvement efforts must consider the influences stemming from all organisational levels. Marketing, sales and user support feed back information and requests that influence the process. Software engineering activity that defines and designs the process used, its support and the activities that progress the product through its various stages of development also exercises significant influence. All these agencies and groups must be considered when process changes are proposed. All impact product and process goals, product distribution, installation and introduction into usage, the product evolution process. Much of that influence is achieved via feedback, whether in the form of control or as information that influences local decision taking. Equally, participants in the process must recognise that for users the product as not an end in itself. It is a means to an end, producing benefit that accrues to the development organisation, the client organisation, the user community and a wider circle. This entire community is a source of feedback pressures.

4.4 The FEAST Hypothesis

However convincing the reasoning, the insight summarised by the brief phenomenological analysis above must, nevertheless, be regarded as a hypothesis. It

must either be proven false or evidence supporting, strengthening and extending it must be obtained. This can be achieved, for example, by studying real, industrial, processes, identifying loops and demonstrating how they constrain process improvement. If that can be done and if the software engineering community can learn to exploit the phenomenon, its implications and its interpretation the hypothesis will become a basis for a theory of the software process and of software product and process evolution. Its achievement would constitute significant progress.

The FEAST hypothesis is the formulation of such a hypothesis. It represents the encapsulation and formulation of the 1970s observation, their reflection in the laws outlined above and process insight developed more recently. It has been widely presented and discussed. As a result its precise formulation has changed somewhat since first proposed [6]. Versions of the hypothesis will be found in [6] and in [16]. We present below a recent formulation [18].

> **Hypothesis**: As complex feedback systems *E*-type software processes evolve strong system dynamics and the global stability tendency of other feedback systems.
> **Supporting observation**: Processes adopted, applied and improved in industry, will naturally evolve positive feedback to drive organisational growth and negative feedback controls — checks and balances.
> This hypothesis actually includes three sub-hypotheses or assertions:
> - The *software evolution process* for *E*-type systems constitutes a complex feedback system.
> - Where present, feedback is likely to constrain the global benefits derived from forward path changes to the process, however effective they may appear locally.
> - Major improvement requires process innovation to change system dynamics by modification of feedback mechanisms.
>
> A lemma also follows: slow progress in process improvement may be due, at least in part, to lack of **attention** to feedback phenomena?

5 FEAST/1

The above statement and a fuller phenomenological analysis of the nature and likely consequences of feedback in the global software process as briefly outlined in section 4.2 and 4.3 provide the starting point for a systematic investigation to confirm the validity and relevance of the hypothesis, to develop means for its exploitation and to strengthen the theoretical base for process improvement. Over the past two years three workshops at Imperial College with wide, international, participation [6] and with the primary objective of arousing interest in the approach have laid the foundations for a collaborative, international, multidisciplinary investigation. A two year project named FEAST/1 is now funded by EPSRC under grants numbered GR/K86008 and GR/L07437. It will commence in the Autumn of 1996 with Professors B Rustem, V Stenning and the present author as Principle Investigators.

For the past two years a core group consisting of Professors V Stenning and W M Turski and Dr D E Perry has been meeting at intervals both to clarify some of the basic concepts that underlie the hypothesis and, therefore, the proposed investigation. Some of their early discussion concentrated on seeking to understand the meaning of *positive* and *negative* in the context of feedback and on a search for examples. It proved possible to generate hypothetical instances of feedback in their understanding of a typical industrial process. Examples based on real experience in actual processes proved more illusive. Hence the importance of the project plan to include studies of a series of industrial projects. Access to such projects will be provided by the FEAST/1 committed industrial collaboration, BAe, ICL, Logica and MOD. The core group also recognised a need for and organised a series of three FEAST workshops at Imperial [6] whose main idea was to expose the hypothesis and under lying concepts to a wide international forum.

The specific conceptual difficulty encountered by the core group in its discussions related to the fact that in analysing feedback in the software process one must take cognizance of the distinction between feedback *control* and *information* feedback which the recipient absorbs (perhaps), interprets (correctly or incorrectly) and acts upon or ignores. Control feedback we can hope to analyse and model in a systematic and rigorous fashion. After all this has been done for many years in, for example, control theory applied to economic modelling [2] and, more recently, in studying and seeking to improve business processes [19]. Information feedback on the other hand, poses more difficult problems. Person to person information flow is clearly not amenable to rigorous analysis. Psychologists might care to attach different probabilities to various reactions. The software engineer can hardly expect to provide a meaningful model. When it comes to the consequences of cumulative information flow in the larger process the situation may be a little more hopeful. Despite the fact that each individual acts on his or her own in decision taking they do take information received into account. With numerous individuals taking numerous decisions one may ask whether the net result in their impact on the process is not, in some sense, *normally* distributed. This was, in fact, precisely the approach taken in seeking to explain the third law [10, 11] and was shown to be the case in, at least, one instance [5].

A brief synopsis of the core group discussions and conclusions may be found in [17].

6 Preliminary Results

Systematic work on the project according to the current plan [leh96b] must await its formal start in the Autumn of this year. But some small progress has already been made. In particular, Logica plc, has given us growth data on one of their system. This banking transaction system with some one hundred user sites has now seen over twenty documented releases and sub releases. That is the records extend to a total of twenty three release sequence numbers (RSN). Figure 2 plots system size modules as a function of the RSN.

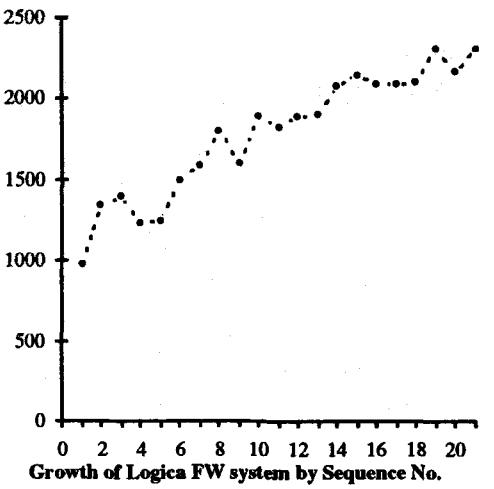

Fig. 2. The growth of System FW

This curve has strong similarities with the OS/360 growth curve reproduced in figure 1. It provides, therefore additional circumstantial real world evidence that the software process is self stabilising. This data would seem to negate the 1970s suggestion that, for the reasons summarised in section 3 inferences from OS/360 or 1970s data were not more widely applicable. At first glance at least the data as plotted appears to support the first and/or sixth and third laws directly. A final judgment must of course await detailed exploration of this and of further data which will become available once the project is under way. For example, whether the observed growth is due to the first or sixth law phenomenon, or to both will require detailed examination of the changes applied to each release.

A brief analysis by Turski of the minimal data of figure one [tur96] has also proved most valuable produced a truly remarkable result. In summary he has shown that, despite the ripples, the data fits very closely to what he has termed an inverse square growth law. That is if S_i is the size in modules of the release with RSN_i:

$$S_{i+1} = S_i + \underline{E}/(S_i)^2, (1 \leq i \leq n-1)$$

where E_i is a constant that represent the work done (in unidentified units) to take the system from RSN_i to RSN_{i+1} and $\underline{E}$ is the average E_i computed

across the full set of observed values. The closeness of the fit is illustrated in figure 3.

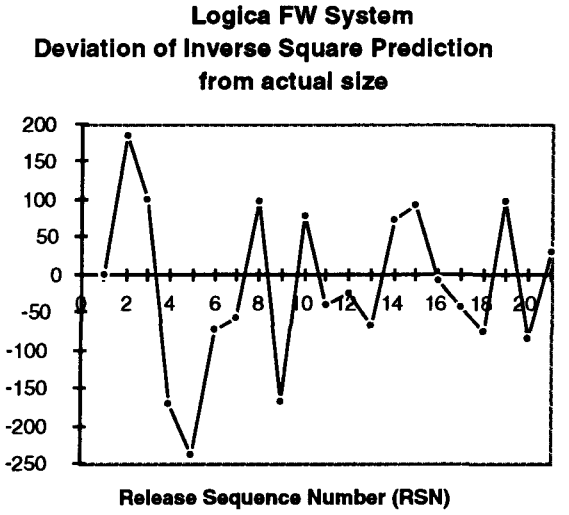

Fig. 3. Difference between Actual size and Inverse System Size Prediction for FW

This relationship is, of course, entirely compatible with the view that growing complexity (second law) is a constraining growth factor. Perhaps just as significant is the fact that such a square law growth is very typical of a system dominated by its own system dynamics. The ability to closely fit a constant effort parameter $\underline{E}$ appears equally to support the third law. So the data plotted in figures 2 and 3 provides further circumstantial support for the laws and the hypothesis. Perhaps the most astonishing result obtained by Turski was in relation to the constant E. He computed $\underline{E}$ in the first instance from all twenty pairs of datapoints i = 1 to i = 23.

An even more impressive initial result obtained by Turski is illustrated by figure 4. This shows the average error in size prediction and its standard deviation if E is estimated from the first j data point pairs ($p \leq j \leq 20$).

This suggests that, at least for FW, the system dynamics is so strong that its parameters are fixed quite early in the life cycle of the evolving system. This is a remarkable result which, if verified for other systems, has profound implications on system evolution, its planning and management.

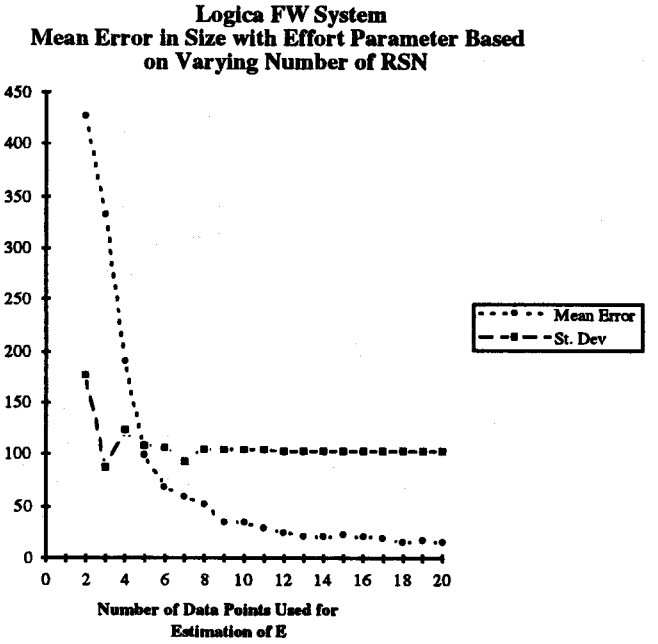

Fig. 4. Difference between Actual size and Inverse System Size Prediction for FW

7 Summary

It is not possible to do more than to provide these initial results in this paper. The size data as presented above, and possible fits, needs further examination. Additional data on FW is required. Similar data from other systems, different organisations and different developments must be obtained and analysed. Further the above black box approach must be complemented by a white box, or systems dynamics, approach, to identify the nature of global software process structure, to isolate feedback loops, to determine their characteristics and how they constrain process improvement. Above all, if it is demonstrated that the feedback control phenomenon is widespread, the software process is constrained by the process dynamics, means must be developed for the mastery and exploitation of that knowledge and understanding.

Acknowledgments

My grateful thanks are, above all, due to Dewayne Perry, Vic Stenning and Wlad Turski who have put up with my obstinate refusal to abandon my beliefs and have co-operated over a two year period in formulating and developing the

material presented here, and much more. Also to Wlad for permitting me to outline the results of his first analysis before his much fuller paper on the topic has been published. Logica plc have been most kind in giving us access to their FW data and in permitting its publication in the present form. Finally to EPSRC for their two grants to support phase two of the longer FEAST investigation and the continued association of Perry and Turski with that endeavour. Also to the Department of Trade and Industry for the earlier grant under the ESF extension that permitted us to establish the recent foundations of FEAST.

References

1. Baumol W J: Macro-Economics of Unbalanced Growth; The Anatomy Of Urban Cities. Am. Econ. Rev. June 1967, pp. 415-426.
2. Becker R S, Hall B, and Rustem B: Robust Optimal Control of Stochastic Nonlinear Economic Systems. J. of Economic Dynamics and Control, n. 18, 1994,pp. 125-148.
3. Belady L A and Lehman M M: An Introduction to Program Growth Dynamics. In Statistical Computer Performance Evaluation. W Freiburger (ed), Academic Press, New York, 1972, pp. 503-511.
4. Brooks F: The Mythical Man Month. Addison-Wesley, Reading, MA., 1975, 2nd ed. 1993, p. 206.
5. Chong Hok Yuen C K S: Phenomenology of Program Maintenance and Evolution. PhD Thesis, Dept. of Comp., Imp. Col., 1981.
6. M M Lehman (ed.): Preprints of the three FEAST Workshops, Dept. of Comp., ICSTM, 1994/5.
7. Lawrence M J: An Examination of Evolution Dynamics. Proc. Proc. 6th Int. Conf. on Softw. Eng., Tokyo, Japan, 13-16 Sep. 1982. IEEE Comp. Soc. ord. n. 422, IEEE cat. n.81CH1795-4, pp. 188-196.
8. *[4]Lehman M M: The Programming Process. IBM Res. Rep. RC 2722, IBM Res. Centre, Yorktown Heights, NY 10594, Sept. 1969.
9. *Lehman M M: Programs, Cities, Students, Limits to Growth?, Inaugural Lecture, May 1974. Publ. in Imp. Col ofSc. Tech. Inaug.l Lect. Ser., vol 9, 1970, 1974, pp. 211-229. Also in Programming Methodology, (D Gries ed.), Springer, Verlag, 1978, pp. 42-62.
10. *Lehman M M: Laws of Program Evolution - Rules and Tools for Programming Management. Proc. Infotech State of the Art Conf., Why Software Projects Fail, Apr. 1978, pp. 11/1 11/25.
11. Lehman M M: On Understanding Laws, Evolution and Conservation in the Large Program Life Cycle. J. of Sys. and Software, 1:3, 1980, pp. 213-221.
12. *Lehman M M: Programs, Life Cycles and Laws of Software Evolution. Proc. IEEE Special Issue on Software Engineering, 68:9, Sept. 1980, pp. 1060-1076.
13. Lehman M M and Belady L A: Program Evolution — Processes of Software Change. Academic Press, London, 1985, pp. 538.
14. Lehman M M: Uncertainty in Computer Application and its Control Through the Engineering of Software. J. of Software Maintenance: Research and Practice, 1:1, Sept. 1989, pp. 3-27.

[4] Papers identified by an * in the references listing are reprinted in [leh85]

15. Lehman M M: Software Engineering, the Software Process and Their Support. IEE Softw. Eng. J. Spec. Iss. on Software Environments and Factories, Sept. 1991, 6:5, pp. 243-258.
16. Lehman M M: Process Improvement - the Way Forward. Proc. CAiSE 95, Jyvaskyla, June 1995, Lect. Notes in Comp. Sci., Springer Verlag, pp. 1-11.
17. Lehman M M, Perry D E and Turski W M: Why is it so hard to find Feedback Control in Software Processes? Invited Talk, Proc. of the 19th Australasian Comp. Sc. Conf., Melbourne, Australia, 31 Jan - Feb 2 1996. pp. 107-115..
18. Lehman M M and Stenning V: FEAST/1: Case for Support, ICSTM, March 1996.
19. Ould M A: Business Processes - Modelling and Analysis for Re-engineering and Improvement. Wiley, Chichester, 1995.
20. Scarrott G A: Private communication, June 1993.
21. W M Turski: Reference Model for Smooth Growth of Software Systems. U. of Warsaw, June 1996,Submitted for publication, available for perusal only from Prof. W M Turski, Dept. of informatics, University of Warsaw, Banacha 2, 00-903 Warsaw 59, Poland.
22. Wilkes M V, Wheeler D J and Gill S: The Preparation of Programs for an Electronic Digital Computer. Addison Wesley Press Inc., 1951, 167 pp.