Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Information Processing Letters 88 (2003) 33–44

Information Processing Letters

www.elsevier.com/locate/ipl

Software evolution—Background, theory, practice

Meir M. Lehman [a,*], Juan F. Ramil [b]

[a] *School of Computing, Middlesex University, Bounds Green Road, London N11 2NQ, UK*
[b] *Computing Department, Faculty of Maths and Computing, The Open University, Walton Hall, Milton Keynes MK7 6AA, UK*

Abstract

This paper opens with a brief summary of some 30 years of study of the software evolution phenomenon. The results of those studies include the *SPE* program classification, a principle of software uncertainty and laws of *E*-type software evolution. The laws were termed so because they encapsulate phenomena largely independent of the people, the organisations and the domains involved in the evolution of the *E*-type systems studied. Recent studies have refined earlier conclusions, yielded practical guidelines for software evolution management and provide a basis for the formation of a theory of software evolution. Given the volume of published material and the extent of recent discussions on the topic (see, e.g., [Proc. ICSM, Montreal, 2002, p. 66]), this paper is restricted to an overview that exposes the significance of the evolution phenomenon and its study to the wider community, providing a basis for the future and, in particular, development of a theory of software evolution.

Keywords: Software engineering; Software design and implementation; Safety/security in digital systems; Software evolution; Theory of software evolution; Laws of software evolution; Software uncertainty principle; Software process; Best practice; Assumptions; Maintenance

1. Introduction—the early days

The term *evolution* describes a phenomenon encountered in many different domains and which can involve concrete and abstract entities or sets of entities. Classes of entities such as natural species, societies, cities, artefacts, concepts, theories, ideas, for example, all evolve in time, each in its own context. The common factor is that all undergo continued progressive change in some of their attributes. This *process* of change leads to improvement in some sense, often to the emergence of new properties. In general, the change will be such as to adapt the individual entities or the class as a whole so that they maintain or improve their fitness to a changing environment. The change may make them more useful or meaningful or otherwise increase their value in some sense. The evolution process may also remove properties no longer appropriate. Evolutionary changes are generally incremental and small relative to the entity as a whole but exceptions may occur. This general view of evolution defines precisely the use by the present authors of the term in the software context.

The *software evolution phenomenon* was first identified in the late 60s [15] though not termed as such till 1974 [16]. Its study was pursued intermittently during

* Corresponding author.
E-mail addresses: mml@mdx.ac.uk (M.M. Lehman), j.f.ramil@open.ac.uk (J.F. Ramil).
URLs: http://www.cs.mdx.ac.uk/staffpages/mml/, http://mcs.open.ac.uk/jfr46.

0020-0190/$ – see front matter
doi:10.1016/S0020-0190(03)00382-X

the 70s and produced results as exemplified in [21]. The work that led to the discovery and exploration of the phenomenon had been seeded by a nine month 1968/9 study of the IBM programming process [15, 21]. The outcome of that study, recently judged to be as relevant today as it was then, led to the Lehman–Belady collaboration. Their joint work initiated systematic empirical study of the growth of software systems and changes in other evolutionary attributes over real or *pseudo* time as measured, for example, in release sequence numbers—rsn [8].

The data that triggered the studies was first drawn from IBM's OS/360-70 and subsequently from other systems [17,19–21]. The insight and overall picture obtained from these studies indicated that software evolution could be systematically studied and exploited. They also suggested that to some extent, software evolution was a *disciplined phenomenon* as illustrated, for example, by the regularity of functional growth patterns. Models of such patterns permitted the forecasting of future overall system growth and growth rates. At a high level of abstraction, some evolutionary characteristics were remarkably similar across the systems studied, despite differences in the detailed patterns of short-term behaviour. This was, of course, counter intuitive since the software process is conceived, directed, planned, managed, implemented and controlled by humans, whose decisions at each stage of the process are assumed to direct and drive the process. Thus, one would expect evolutionary behaviours to vary significantly from application to application, organisation to organisation, system to system, time to time, release to release. Such similarities in overall trends were identified and further supported as the study progressed, the spectrum of applications addressed widened and the number of differing development, marketing and user organisations increased. More importantly, the observed patterns of behaviour appearing yielded common phenomenological interpretations. Differences in behaviour were, of course, also apparent. However, apart from one early study [38], investigations to relate observed differences in evolutionary behaviour to the characteristics of the different circumstances involved in the systems studied were not undertaken and provide opportunities for further investigation.

For practical reasons, the early studies concentrated on the behaviour of what were then termed *large*[1] software systems and the organisations that developed, maintained, evolved and used them [3,4]. They suggested that the system formed by the evolving software and the organisations involved in or relating to its evolution behave like and constitute a *feedback system*, subsequently described as a multi-level, multi-loop multi-agent, self-stabilising, feedback system [24]. The models developed were relatively simplistic being limited by data availability. The investigations outlined above were, however, not confined to modelling and analysis of growth data and the dynamics of that growth. They also included, for example, search for conceptual and theoretical models that reflect understanding of the phenomenon and the forces driving it, e.g., [4]. This led to important advances in understanding of the software evolution phenomenology as encapsulated in a set of laws of software evolution [3,16,17,19–21]. The investigation also triggered a change in terminology from Software Growth Dynamics to Software Evolution [16].

2. The second wave

The work outlined above went largely unnoticed by the mainline Computer Science and Software Engineering communities. Gradually, however, the concept of software evolution as a phenomenon to be investigated began to attract other investigators, e.g., [12,14,9,2].

A major conceptual advance came with formulation of the *software uncertainty principle* [22,23, 34], the FEAST (*F*eedback, *E*volution *A*nd *S*oftware *T*echnology) hypothesis and the FEAST projects [26, 28]. The wider significance of these was not, so much, in themselves as in their implications to real world computer usage. The insights that emerged are directly relevant to the widespread and growing striving for software process improvement [24,25]. The overall results of these project studies and many of the conclu-

[1] The term *large* is, generally, used to describe software whose size in number of lines of code is greater than some arbitrary value. For reasons indicated elsewhere, it is more appropriate to define a large program as one developed by processes involving groups with *two or more* management levels [18].

sions to which they led have been widely reported [45, 31,32,34–36].

3. Evolution: Phenomenon and activity

The studies reflected in the overview above have demonstrated that *software evolution* is a *phenomenon* that may be systematically studied. This represents a *nounal* view of the term *evolution* [29] focusing on the nature of evolution, its causes, properties and characteristics, its consequences, impact, management, control and exploitation. An alternative approach takes a *verbal* view [29]; is concerned with improvement of means, languages, methods, tools, for example, whereby evolution is implemented over the many process activities.

These orthogonal views are mutually supportive. Their joint further development is critically important to a society increasingly dependent on computers, and hence software. In response to the computer usage itself and to other influences, the domains and applications in and to which computers are applied change. System functionality and behaviour must keep pace. Defects must be fixed, functionality refined and extended, performance improved. The system must be adapted in response to application and domain changes, operational extension and need or desire for new capability. As business operations, and in general, organisational behaviour, becomes ever more dependent on software, the consequences of a delay in implementation range from frustration to disaster. Thus improvement in means to support evolution will also, in general, have an impact on risk mitigation, usability, quality, timeliness, economic benefit and so on. Increased understanding of the nature, characteristics and impact of evolution will also yield benefit to the software process in general, evolution planning, process management and process improvement in particular [32].

As understanding of the causes, properties and implications of software evolution grows, planning, control, execution of process improvement will become more systematic and effective. It will deepen insight into the types of activities, methods and tools required, identify and determine those likely to be beneficial, when and how they should be used and how they relate to one another. Studies of these views of software evolution must be joined and move forward together.

The approach to the study by Lehman and his collaborators assumed the nounal interpretation. Results obtained have yielded striking evidence that at a high level of abstraction the software process and the products it produces display a degree of regularity, patterns, trends, long term limits, similar evolutionary behaviour across a variety of commercially developed and marketed software systems. Despite low level behavioural variations, this similarity permits growth models of the same form, though differing parameter values, to be fitted to empirical data. Significant improvement of the predictive power of later models exploiting growth stages [5] and having similar qualitative patterns [40,41] has further increased confidence in the models and the reality of the software evolution phenomenon.

Collecting data from different systems, coping with questions of compatibility, identifying differences, fitting models to various attributes, assembling and combining evidence from different studies, understanding characteristics of individual stages, interpretation of observations [40,41] all raise non-trivial issues requiring investigation. Appreciation of these, and challenges raised, has, however, not undermined confidence in the results. Qualitative commonalities in the evolution of so many *E*-type widely used systems provide a solid basis for confidence.

4. Laws of *E*-type software evolution

The laws reflecting the observed evolutionary behaviour of large *E*-type[2] software systems and processes implementing their evolution are currently stated in natural language. They encapsulate aspects of the common behaviour of a variety of systems over a spectrum of application areas, development organisations, system characteristics and technologies. It may be noted that pressures and constraints originating outside the technological aspects of the software engi-

[2] The *SPE* application and software classification scheme is now well recognised. Many of the references cited above, e.g., [21,34], include brief discussion and there is no necessity to expand further here.

neering process appear to have a far greater influence on the behaviour than the specifics of the technologies.

Over the years the laws have been refined and extended, driven by interpretation of models of further metric data that has become available. It is not possible to discuss details here. Interested readers are referred to the literature [16,17,19,20,25,27,39–41] and it worthy of note that recent public discussion [10] produced general agreement as to their continuing relevance. An earlier paper [9] had suggested that in the context of Open Source software development some rewording of the laws might be required. A paper to be presented later this year in that context [2] supports the conclusion of continued relevance.

The laws of software evolution as developed and over a ten-year period were formulated and presented individually. Possible relationships between them were not discussed until formulation of the FEAST hypothesis and, its restatement as the eighth, Feedback System law, suggested one springing from the feedback nature of the process. It is expected that formalisation of the observations and interpretations in a theory of software evolution will, *inter alia*, lead to better understanding of any inter-relationships.

The laws and their empirical support have been criticised on several grounds [14,3,8,9]. As a result of the FEAST studies some of these issues are now better understood [39], can be addressed and refuted by techniques such as qualitative abstraction and simulation [40,41,43].

Amongst these concerns was the fact that the laws addressed activity conducted by and dependent on human intellectual processes, decision taking and implementation. In particular, critics suggested that it was inappropriate to term the observations *laws*. However, when the use of the term was first presented [16], it was pointed out that the statements reflect phenomena outside the immediate control of those implementing system evolution. They emerged from observation of behavioural phenomena in the real world of software development and evolution. These phenomena were a reflection of the attitudes and behaviours of many groups and individuals engaged directly in software creation and evolution. They also relate to managers and, via feed forward and feedback, to other stakeholders in the end product. As such, the influence of technical activity at the direct development (process) level, however significant locally, appeared to be of little consequence at the global level.

That is, the activities of those directly involved in evolution process are, in general, restricted to local areas of the evolving system. The causes that underlie observed behaviours as described, for example, by the laws stem from group, organisational and societal behaviours and the multi-level, multi-loop, multi-agent feedback system that links and aggregates these. The individuals involved in the process each have their area of experience and expertise. No matter the degree of experience and understanding of individuals and groups, no matter the methods and tools used, the scope of action is constrained by complex relationships between them, the feedback network structure of the evolution process. This results, for example, in a form of organisational inertia loosely termed *inherited evolution dynamics*.

Software engineers do not, in general, have the time, experience, viewpoint or appreciation to explore potential benefits that could be exploited by taking the complex feedback system properties of the process into account. Moreover the causes of the behaviours and phenomena addressed by the statements will, in general lie outside their range of expertise. In general, they can do little or nothing to modify the implied behaviours. The behavioural assertions can be explained and accepted in terms of group, organisational and societal behaviour, market and economic forces and so on. Thus they must be accepted as *laws*.

It may well be that as knowledge and understanding of the phenomenon increases it will become reflected in organisational and process improvements and, in particular, in new forms of technology which help to overcome the limitations of individual system evolvers, constraints expressed by the laws. As paradigms evolve, new approaches such as the increasing desire to use COTS-based designs [30] are adopted behaviours may change, the laws as currently stated may need to be modified, added to or even dropped. However, it has been reasoned, e.g., [30], that, in the long run, such adjustments are likely to be minor. One can question specifics of the laws as presently stated, but the fact that it is meaningful to formulate and acknowledge such laws and take them into practical consideration is now widely accepted, e.g., [10,2].

One limitation of the studies to date must be recognised. Apart from some recent work on Open Source

evolution [9,2] whose initial findings suggest minor changes to the laws as presently formulated, evolution studies have been largely limited to what may be termed the *classical* industrial, waterfall-like, software development process. The FEAST investigations, in particular, were exclusively based on data obtained from the latter. The evolution of systems developed by Object Oriented, Incremental and Evolutionary Development, Extreme, Agile and Component Intensive development paradigms appear not to have been investigated. Phenomenological and theoretical analysis suggests, however, that basic concepts and conclusions will not change significantly unless very radical changes occur in software process practice, technology and their wider domains. It is not unlikely that the laws as stated are relevant to the evolution of other artificial [42] systems. Observations and reasoning along these lines suggests that a general framework and theory of software evolution can be developed [31,33].

5. An approach to theory formation

The approach taken in FEAST and its antecedents to the study of software evolution was inspired by the traditional view of the scientific method involving empirical observation, measurement, hypothesis testing (in the form of growth model fitting, for example) and phenomenological interpretation. This paralleled the view of the roots of that method expressed by Kelvin in a much quoted statement:

> "... first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it. I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be" [11].

Given the numbers that result from such measurement, one looks for patterns, regularities, and trends and develops preliminary hypotheses, phenomenological and mathematical models. At any point in time, available mathematical approaches may prove inadequate. Thus scientific advances are, often, accompanied by development of new mathematical concepts and tools. Given appropriate models, one searches for interpretations in the domain of interest and refines any preliminary hypotheses and extends them individually or as a set. Further observation and, when possible, real or *gestalt* experiments must be undertaken to support or reject these. Validation (or rejection) follows. As the number of validated hypotheses builds up, one looks for relationships between them and develops seeds of a theory from the collection of observations, measurements, models, hypotheses, relationships, etc. Inferences from such theory lead to an iterative search for new data, new hypotheses and so on. This describes the approach taken in the FEAST studies and their predecessors over many years. The study of industrial software evolution is, however, severely limited by the difficulty of scaling-up software evolution experiments to industrial settings. Despite this, the FEAST results provide a set of empirical generalisations relating to software evolution. These include, but are not limited to, the laws, a principle of software uncertainty [22,23,34] and the FEAST hypothesis [24].

6. A theory of software evolution

For many years now it has been observed that, apart from that provided by *programming methodology* as established by the work of WG2.3, software engineering has no comprehensive theoretical base, e.g., [37,21,5]. The programming methodology base is vital to guiding the structure, underlying quality and implementation of the evolving software products and the evolvability of the resulting system. It lies at the heart of improving the *means* whereby evolution can be effectively achieved. But, though critical, programming methodology plays a relatively local part in the process. As indicated by FEAST results and also by other observers, long-term evolvability and evolution is likely to be heavily dependent on more global mechanisms subject to behaviours implied by the laws. These include forward and feedback loops and mechanisms that involve players such as business executives, other stakeholders, organisational and individ-

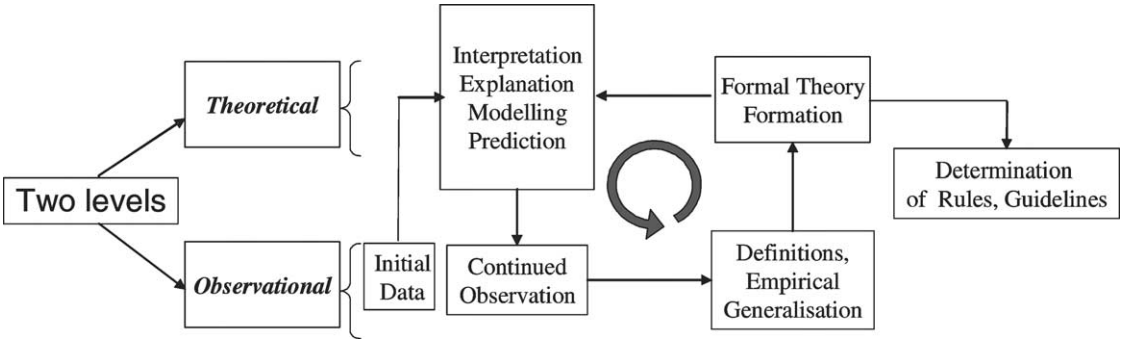

Fig. 1. An approach to the formation of a theory of software evolution

ual users, governments and economies, in the total evolution process. All influence the rate and direction of software system evolution, the disciplined and sustained improvement of the global software processes and system evolvability. A sound conceptual base with predictive and explanatory power, would contribute significantly to integration of these many influences, strengthen software engineering in general and guide improvement of software evolution processes. Continuing, systematic pursuit of that goal would greatly benefit from such a base, a theory of software evolution [31].

The history of science reflects many different ways of achieving an empirically grounded theory applicable to a natural, artificial or hybrid phenomenon. One particular approach which is relevant to the search for a software evolution theory is that emanating from Carnap's studies [6] of theory formation. The theory envisaged will be based on observation, hypotheses and assumptions that follow. Given the confidence widely expressed in recent years in the laws of software evolution,[3] one may start by considering these as *empirical generalisations* as defined by Carnap [6]. The wider conclusions of the nounal aspects of software evolution studies may be encapsulated in a fuller, initial set to include other generalisations of repeated real world observations of *software evolution processes* and evolution of their *products*. These, in turn, must be supplemented by generalisations about the domains in which software is developed, used and evolved. Together with insight and *understanding* of the phenomenology reflected and *relationships* and *dependencies* identified, they provide a basis for the development of an encompassing theory. Confirmation obtained from the developing theory by further observation of the real world then provides partial validation, increasing confidence in the validity of the developing theory to the level of detail reached and over the domain in which the observations were made. Given such a partial theory one may then seek to formalise and extend it. This may involve refinement or rejection of previously formulated empirical generalisations, identification of additional ones, derivation of implications and formalisations to produce definitions, axioms and theorems. Iteration and continuing experimentation then gradually builds up a fuller, but never complete, theory accepted as valid until shown to be inadequate or incorrect. The main activities involved in the proposed theory formation approach are depicted in Fig. 1.

To undertake the above requires formal definition of terms used, or their acceptance, at least temporarily, as undefinable, but this remains a challenge. All are candidates for rejection or acceptance as formalisable generalisations. Once begun, one can initiate selection and statement of axioms followed by the identification and proofs of theorems. An outline proof of the software uncertainty principle has been generated [31] to provide an example of the approach. Its completion awaits wider discussion, formal definition of the terms used in its statement and progress in the formation process.

Given an emerging theory, one may then formally develop its interpretations in the real world of industrial software development and derive practical implications for the planning, management, control and evaluation of system evolution, vital activities in a so-

[3] For example, as expressed recently [10].

ciety ever more dependent on computers and their software, and that, at present is done informally, often intuitively. At that point, the research approach outlined above will have the potential to merge the search for achieving full *understanding* of the software evolution phenomenon and the development of effective, reliable, predictable and on time means for its achievement.

The empirical generalisations referred to above provide initial inputs for development of an empirical theory and its formation as a formal theory. Rooted in earlier findings, supplemented by the results of the FEAST projects, the efforts of other groups in Europe, North America and Japan, and the related insight and understanding achieved, development of a such a theory appears to be within reach [31,33]. Its development could represent a first step in the development of a more general theory to support the discipline of Software Engineering. Moreover, software has long been regarded as the fruit fly (*Drosophila*) of artificial systems [42]. Thus the theory of software evolution could, in turn, provide a basis for development of a general theory of artificial systems evolution. But that is many years, possibly decades, away from realisation.

7. Example

Any approach to theory formation, such as that outlined, begins with formulation of a set of definitions, to be revised and extended as development proceeds. Their seeds are derived from empirical *observation* of the phenomenon to be reflected in the theory. Their interpretation leads to *assumptions* judged reasonable in relation to the domain being addressed. Once accepted, they become the source for the derivation of *inferences*. The initial set of definitions provides a basis for an emerging theory that, eventually, one will seek to formalise, in part or completely. Together with the observations they constitute axioms from which new and established implications may be formally derived as theorems. In the absence of such derivation the observations remain, at best, hypotheses. Practical application such as, for example, methods and guide lines for program development and management then emerge as corollaries.

The empirical aspects of this approach to theory formation is illustrated by the example provided by the lists that follow. These provide definitions, observations and implications that, when formalised, are believed to suffice for a proof of the principle of software uncertainty [23]. Such formalisation and proof has not yet been undertaken.

List 1 (*Provisional definitions*).

(1) The *real world* encompasses the entire universe and all happenings in it.
(2) *E*-type *operational domains* and their attributes are, respectively, sub-domains and sub-sets of the real world and its attributes.
(3) An *E*-type *application* addresses a problem or supports an activity in a specified *E*-type operational domain.
(4) An *E*-type *specification* abstract an *E*-type application, representing in a set of statements the recognised attributes required to define a solution in accord with the application needs or terms of reference.
(5) An *E*-type *program* is a set of computer executable instructions defining a solution to an *E*-type application.[4]
(6) A program is *satisfactory* as long as it is compatible with the solution required and the operational domains within which it is executed.

List 2 (*Observations*).

(1) The real world has an unbounded number of attributes.
(2) It is dynamic with its attributes continually changing.
(3) It may be partitioned in an unbounded number of ways into domains that, in general, each possess an unbounded number of attributes.
(4) The designed and implemented attribute set of an *E*-type program, as distinct from the totality of attributes of such programs in execution, is necessarily bounded.
(5) Attributes of an *E*-type application must be appropriately addressed in the implementing program to make the latter satisfactory.

[4] A program also is a model of an *E*-type specification.

List 3 (*Inferences*).

(1) Every E-type operational domain, though abstracted by a finite specification, has an unbounded number of attributes.[5]
(2) By design and implementation E-type specifications and programs have a bounded number of attributes which reflect an unbounded number of assumptions (at least one per each unaddressed attribute).[6]
(3) Every E-type program is essentially incomplete and there will be attributes of the operational domain not addressed by it.
(4) Assumptions about the operational domain reflected in E-type specifications and programs may become invalid as a consequence of changes in the real world so invalidating either one or both.
(5) Though both are models of the same specification, an E-type program and its operational domain may be or may become incompatible.
(6) E-type program execution entails a degree of uncertainty, sustained satisfaction cannot be guaranteed.[7]
(7) Program evolution activity consists primarily of maintenance of the validity of the assumption sets reflected in it.
(8) Progressive change, that is evolution, of E-type programs is inevitable if satisfaction is to be maintained.

8. One practical implication

As already indicated, the proposed theory is not purely of theoretical import. It, and theorems developed in it, should yield a rich source of *process improvements* and *best practice*. This is exemplified by first pointing the reader to technical and management guidelines derived from the laws [32] and the principle of software uncertainty and a brief discussion of their practical implications. That analysis introduces, *inter alia*, the impact of assumptions on computer software, computer systems and their users. The latter is worthy of much wider attention than it has, with some exceptions [44], received. It provides just one, specific, example of practical conclusions and guidelines resulting from the study of software evolution and from the formation of a conceptual framework that will eventually be provided by a theory of software evolution. Its mention here also draws attention to a neglected phenomenon with major practical consequences.

A central ingredient of an informal, demonstration [31] of the validity of the principle of software uncertainty, and of the more recent formulation outlined in Lists 1–3, is the observation that all E-type software has embedded within it reflections of an uncountable number of assumptions by omission or commission, conscious or unconscious, known or unknown. This follows from the fact that any real world computer application and its operational domain each have a potentially uncountable number of properties. Having been developed by humans, with finite resources in finite time, the static software (as distinct from the software in execution) has a finite number of attributes fixed by design and implementation. As a finite model-like reflection of unbounded domains E-type software is essentially incomplete [34].

Assumptions may relate to the application being addressed, to software functionality, application systems within which it executes, computer systems on which it runs, geographical, economic and societal domains on and in which it operates and which it supports, the processes by which it is produced, adapted and evolved and so on. Any conscious underlying assumptions may have been subjected to appropriate review. Others will be the consequences of, for example, decisions taken sometime during system conception, specification, design, implementation, installation and operation. Others will be the consequence of overlooking or being ignorant of facts or situations that can affect the workings of the software, the results of execution or its impact on the operations or domains served.[8–10] Still others were entirely inconsequential or irrelevant, at least at the point in time

[5] Some of which will change with usage and the passage of time.

[6] The assumptions issue is exemplified towards the end of the paper by three identified examples. The examples are the failure of the London Ambulance Service software, the Ariane 5 rocket disaster and the initial failure, during commissioning, of a new CERN accelerator.

[7] That is, sustained compatibility between the program and application it addresses, the domains within which it is executed.

[8] Consider, for example, the failure of the London Ambulance Service Computer Aided Despatch System in 1992 where ambu-

at which, implicitly or explicitly they were adopted and embedded. Whatever the source of each assumption, and even supposing that all those where validity has meaning, were valid when relevant decisions were taken they may become invalid as a consequence of changes in the domains within which the software executes, with which it interacts and which it supports. The results of *E*-type system execution will be influenced to some degree by invalid assumptions. Given an uncountable number of assumptions, one cannot, *a priory*, know absolutely which are invalid and what the impact of such invalidity will be. The degree of satisfaction to be derived from execution of any *E*-type system cannot be guaranteed.

9. Practical application

Except for inferences (7) and (8) the above reasoning provides a basis for a proof of the principle of software uncertainty. This is not just of theoretical interest, having important practical implications. Before addressing that issue, one point must be noted. In all of software systems recently examined, it has been possible to show that the *underlying* cause of unsatisfactory operation or of failure was one or more, probably implicit, assumptions that from the outset were unjustified, or that became invalid as a result of changes external to the software system (see footnotes to preceding section). There are good reasons to believe that this observation may, in fact, be generalised and applied to a high proportion of software, computer and computerisation project failures. They may be explained by assumptions about the properties of the operational domains, specification, design, implementation processes, interfaces, usage procedures and so on.[11] One may confidently assert that assumptions play a critical role in the birth, life and death of software systems. Hence, as many as possible must be identified, captured, questioned, confirmed and reviewed when adopted, as appropriate, thereafter. Moreover such justification must not concentrate on circumstances as they are then. The nature and direction of possible future changes, and their likelihood, must also be taken into account. Anything that influences viewpoints adopted or decisions taken must be captured and stored in a structured fashion for subsequent systematic review that scans the database when appropriate. The intervals at which examinations should be scheduled, review frequency, the extent of each review, triggers for unscheduled reviews and so on will depend on the criticality of the application, volatility of the domains, the error sensitivity of the domain, the likelihood of change, the occurrence of perceived system or external changes and so on.

The real world is dynamic and undergoes continual change. Even assumptions, conscious and justified at the moment of adoption, can eventually become invalid. As for the unknown unbounded who knows. Hence, total management and control of assumptions, though the goal, is, in practice, not possible. Resource and time considerations limit the frequency and detail assumption database review to check for continuing validity, a need for correction. As in all engineering, compromises must be made, decisions taken and implemented. Given those cognitive and managerial constraints it must be accepted that an *E*-type system *cannot* be made or maintained absolutely valid and up-to-date. But in current practice, industrial or otherwise conscious, explicit and continual attention to assumptions is the exception rather than the rule. Apart from [44], the authors do not know of, for example, any specified *inspection* procedures at any process stage that calls for systematic questioning and recording of assumptions, their presence and continued validity. We must do better than that. The management of assumptions over system lifetime must become an essential part of every software process to maximise the likelihood of sustained satisfactory operation of the software as it evolves.

Recognition of the role, largely inadvertent, played by assumptions in the exploitation of computers and in the lifecycle of *E*-type software explains a fact that has been recognised since computers came into common usage, the continuing, ubiquitous, need for, so-called, software maintenance and upgrading. Computer systems must provide sound solutions to the problems ad-

lance drivers reactions and their inability to operate a complex interface whilst driving was overlooked in the requirements phase. There was an implicit assumption that they could [13].

[9] Another example is provided by the Ariane 5 rocket disaster which took place during its maiden flight on 4 June 1996 [1].

[10] A third example is the initial failure, during commissioning of the LEP accelerator in 1989 [7].

[11] An investigation of this hypothesis is being planned for the Software Forensic Center at Middlesex University.

dressed or the processes supported and/or controlled. It is stakeholder and, in particular, user satisfaction and assumption set validity that need to be *maintained*.

A software system does not, in general, adapt itself to changing situations or domains, though auto-update mechanisms, triggered internally to supply and apply software changes supplied from outside the system come close to this. In certain applications, in safety critical or defence systems, for example, and to the extent permitted by economic considerations, developers provide mechanisms to accommodate future external changes by applying necessary software changes. But it is the domains within which a system operates that change, often in unforeseen ways. Situations where a need is anticipated and an auto-change mechanised are the exception rather than the rule. Human intervention is required to maintain stakeholder satisfaction, by system *adaptation* to ensure continuing consistency that meets changing needs and desires of stakeholders and to continue to support the operational domains satisfactorily. That is achieved by *changing* software, documentation and/or usage procedures, *evolving* the total system, not by restoring the software to its pristine beauty, as is the case when physical artefacts are maintained. In a strict sense software does not decay, need not be maintained. It must be evolved to remain satisfactory to its users and to the community it serves.

The uncertainty principle highlights the risks associated with reliance on software for critical real time control decisions—as in weapon systems for example. There might be, of course, reasons for doing so. But it must be understood and accepted that ignoring the inherent uncertainty that is involved and the implications of the evolutionary pressures and embedded assumptions that are intrinsic to computer systems poses challenging problems. 'Human activity-business processes-software' systems must be designed and operated as safely as possible, with software remaining the slave, not the master in the decision and implementation chains. Society ignores this fact of life at its peril.

10. Further work

To this point, the discussion has mainly referred to the evolution of software as reflected in a *series of releases or upgrades*. Evolution that closely affects the continuing value, quality, cost and/or timeliness of software can and does, however, occur at many product and process levels [36].

A fuller discussion of the levels, the role and the wider impact of evolution may be found in [36]. Changes at any of these levels may have an impact on E-type software product properties or behaviour. Potential impact must be identified and considered during conception, specification, design, planning, management and execution of the processes that develop the products and maintain the software operationally satisfactory in a changing world. Hence, empirical study of the evolution phenomenon at all its levels is of considerable interest.

The extension of results of software-related investigations to the evolution of other artificial [42] systems or even, more broadly to other areas as exemplified in the introduction to this paper is likely to lead to further conclusions. But that remains to be demonstrated.

11. Final remark

Software evolvability, the ability, *inter alia*, for responsiveness and timely implementation of needed changes, will play an ever increasing more critical role in ensuring the survival of a society ever more dependent on computers. This requires means to ensure timely adaptation of the ever more integrated computer systems to maintain compatibility between the sub-systems and the forever changing circumstances with, within and under which they operate. The goal here has been to convince the wider Computer Science and Software Engineering community that study of the software evolution phenomenon is of theoretical and practical importance and must be widely pursued.

Acknowledgements

Sincere thanks are due to many colleagues and collaborators for sharing their insights, many useful discussions, and constructive criticism. In particular it is a great pleasure to acknowledge the contributions of Wlad Turski and Dewayne Perry to the FEAST projects funded by the UK EPSRC and for their major contributions during four-year appointments as SVFs. One of us (mml) must also express his

profound gratitude to Turski for over twenty years of a relationship as teacher, critic, collaborator and, above all, friend, also to his wife, Chanka, for supporting the resultant lengthy absences from home and hearth.

References

[1] http://www.ima.umn.edu/~arnold/disasters/ariane5rep (as of July 2003).
[2] A. Bauer, M. Pizka, The contribution of free software to software evolution, in: IWPSE 03, Amsterdam, 2 September 2003.
[3] L.A. Belady, M.M. Lehman, Programming system dynamics or the metadynamics of systems in maintenance and growth, IBM Res. Rept., T.J. Watson Res. Centre, Yorktown Heights, NY, RC 3546; Also as Chapter 5 in [21].
[4] L.A. Belady, M.M. Lehman, An introduction to program growth dynamics, in: W. Freiburger (Ed.), Statistical Computer Performance Evaluation, Academic Press, New York, 503–511; Also as Chapter 6 in [21].
[5] K.H. Bennett, V.T. Rajlich, Software maintenance and evolution: A roadmap, in: A. Finkelstein (Ed.), The Future of Software Engineering, in Conjunction with ICSE 22, Limerick, Ireland, 4–11 June 2000.
[6] R. Carnap, Philosophical Foundations of Physics, Basic Books, 1966.
[7] CERN Bulletin 09/98; 23 February 1998, http://bulletin.cern.ch/9809/art1/Text_E.html (as of July 2003).
[8] D.R. Cox, P.A.W. Lewis, The Statistical Analysis of Series of Events, Methuen, London, 1966.
[9] M.W. Godfrey, Q. Tu, Evolution in open source software: A case study, in: Proc. ICSM, San Jose, CA, 11–14 October 2000, pp. 131–142.
[10] N.H. Madhavji, Introduction to the panel session Lehman's laws of software evolution, in context, in: Proc. ICSM, Montreal, Canada, 2002, p. 66.
[11] W.T. Kelvin, Popular Lectures and Addresses, 1891–1894.
[12] B.A. Kitchenham, System evolution dynamics of VME/B, ICL Tech. J. (1982) 42–57.
[13] http://www.cs.ucl.ac.uk/staff/A.Finkelstein/las.html (as of July 2003).
[14] M.J. Lawrence, An examination of evolution dynamics, in: Proc. ICSE 6, Tokyo, 13–16 September 1982, pp. 188–196.
[15] M.M. Lehman, The programming process, IBM Res. Rept. RC 2722, December 1969, 46 pp.; Also as Chapter 3 in [21].
[16] M.M. Lehman, Programs, cities, students, limits to growth?, in: Inaugural Lecture, in: Imperial College of Science and Technology Inaugural Lecture Series, Vol. 9, 1974, pp. 211–229; Also in: D. Gries (Ed.), Programming Methodology, Springer, Berlin, 1978, pp. 42–62; Also in [21].
[17] M.M. Lehman, Laws of program evolution—rules and tools for programming management, in: Proc. of the Infotech State of the Art Conference, Why Software Projects Fail, 1978; Reprinted as Chapter 12 in [21].
[18] M.M. Lehman, The environment of design methodology, in: T.A. Cox (Ed.), Proc. Symp. on Formal Design Methodology, Cambridge, UK, 9–12 April 1979, STL Ltd, Harlow, Essex, UK, 1980, pp. 17–18.
[19] M.M. Lehman, On understanding laws, evolution and conservation in the large program life-cycle, J. Syst. Software 1 (3) (1980).
[20] M.M. Lehman, Programs, life cycles and laws of software evolution, in: Proc. IEEE (Special Issue on Software Engineering), 1980, pp. 1060–1076; With more detail as "Programs, programming and the software life-cycle", in: System Design, Infotech State of the Art, Rept. Se 6, No 9, Pergamon Infotech Ltd, Maidenhead, 1981, pp. 263–291; Reprinted as Chapter 19 in [21].
[21] M.M. Lehman, L.A. Belady, Program Evolution—Process of Software Change, Academic Press, London, 1985.
[22] M.M. Lehman, Uncertainty in computer application and its control through the engineering of software, J. Software Maint. Res. Practice 1 (1989) 3–27.
[23] M.M. Lehman, Uncertainty in computer application, Technical Letter, Comm. ACM 33 (5) (1990) 584.
[24] M.M. Lehman, Feedback in the software evolution process, keynote address, in: CSR 11th Annual Workshop on Software Evolution: Models and Metrics, Dublin, 7–9 September 1994; also in: Information & Software Tech. (Special Issue on Software Maintenance) 38 (11) (1996) 681–686.
[25] M.M. Lehman, Laws of software evolution revisited, in: Proc. EWSPT '96, Nancy, October 1996, in: Lecture Notes in Comput. Sci., Vol. 1149, Springer, Berlin, 1997, pp. 108–124.
[26] M.M. Lehman, V. Stenning, FEAST/1: Case for support, ICSTM, DoC, EPSRC Proposal, Nov. 1995/March 1996, 11 pp.
[27] M.M. Lehman, D.E. Perry, J.F. Ramil, W.M. Turski, P. Wernick, Metrics and laws of software evolution—the nineties view, in: Proc. Metrics '97, Albuquerque, NM, 5–7 November 1997, pp. 20–32;
Also as Chapter 17 in: K. El Eman, N.H. Madhavji (Eds.), Elements of Software Process Assessment and Improvement, IEEE CS Press, Los Alamitos, CA, 1999, pp. 343–368.
[28] M.M. Lehman, FEAST/2: Case for support, DoC, Imp. Col., London, EPSRC Proposal, July 1998, 11 pp.
[29] M.M. Lehman, J.F. Ramil, G. Kahen, Evolution as a noun and evolution as a verb, in: SOCE 2000 Workshop on Software and Organisation Co-evolution, Imperial College, London, 12–13 July 2000.
[30] M.M. Lehman, J.F. Ramil, Software evolution phenomenology and component based software engineering, IEE Proc. Software (Special Issue on Component Based Software Engineering) 147 (6) (2000) 249–255.
[31] M.M. Lehman, J.F. Ramil, An approach to a theory of software evolution, in: IWPSE 2001, Vienna, 10–11 September 2001; Also in: Proc. IWPSE 2001, IEEE CS Press, Los Alamitos, CA, 2002.
[32] M.M. Lehman, J.F. Ramil, Rules and tools for software evolution planning and management, Ann. Software Engrg. (Special Issue on Software Management) 11 (2001) 15–44.
[33] M.M. Lehman, SETH—Approach to a theory of software evolution, case for support, Part 2, DoC, Imperial College, Lon-

don, September 2001, http://www.cs.mdx.ac.uk/staffpages/mml/seth_p2.pdf (as of July 2003).

[34] M.M. Lehman, J.F. Ramil, Software uncertainty, software 2002, in: D. Bustard, W. Liu, R. Sterritt (Eds.), Soft-Ware 2002, 1st Internat. Conf. on Computing in an Imperfect World, Belfast, North Ireland, 8–10 April 2002, in: Lecture Notes in Comput. Sci., Vol. 2311, Springer, Berlin, 2002, pp. 174–190.

[35] M.M. Lehman, J.F. Ramil, An overview of some lessons learnt in FEAST, in: Proc. WESS '02, Montreal, 2002.

[36] M.M. Lehman, J.F. Ramil, Software evolution and software evolution processes, Invited Contribution to Special Issue on Process-Based Software Engineering, Ann. Software Engrg. 14 (2002) 275–309.

[37] P. Naur, B. Randell (Eds.), Software Engineering, Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 October 1968, January 1969, p. 231.

[38] S.S. Pirzada, An statistical examination of the evolution of the Unix system, Ph.D. Thesis, Dept. of Computing, Imperial College, London, 1988.

[39] J.F. Ramil, Laws of software evolution and their empirical support, invited panel statement, in: Proc. ICSM '02, Montreal, 3–6 October 2002, p. 71.

[40] J.F. Ramil, Continual resource estimation for evolving software, PhD Thesis, Dept. of Computing, Imperial College, London, January 2003.

[41] J.F. Ramil, N. Smith, Qualitative simulation of models of software evolution, Special Issue on Software Process Simulation Modelling, J. Software Process, Improvement and Practice (2003), in preparation.

[42] H.A. Simon, The Sciences of the Artificial, 3rd Edition, MIT Press, Cambridge, MA, 1996, 231 pp.; first published 1969.

[43] N. Smith, J.F. Ramil, Qualitative simulation of software evolution processes, in: WESS '02, Montreal, 2 October 2002, pp. 41–47.

[44] S. Uchitel, D. Yankelevich, Enhancing architectural mismatch detection with assumptions, in: Proc. 7th IEEE Int. Conf. on the Engineering of Computer Based Systems (ECBS 2000), Scotland, UK, April 2000.

[45] http://www.cs.mdx.ac.uk/staffpages/mml/ (as of July 2003).