Evolution in Software and Related Areas

M M Lehman
Department of Computing
Imperial College
180 Queen's Gate, London SW7 2BZ
tel. +44-20-7594 8214, fax +44-20-7594 8215
mml@doc.ic.ac.uk

J F Ramil
Computing Dept., Faculty of Maths and Computing
The Open University
Milton Keynes MK7 6AA, U.K.
tel. +44 - 1908 - 65 4088, fax +44 - 1908 - 65 2140
j.f.ramil@open.ac.uk

ABSTRACT

After briefly discussing the meaning of the term *evolution* in the context of software, its technology, the software process and related domains, the paper describes some of the facets and implications of the evolution phenomenon as identified during many years of active interest in the topic, most recently during the FEAST (Feedback, Evolution And Software Technology) projects.

Keywords

Empirical Studies, Software Process, Feedback, Process Modelling, Process Improvement, Software Engineering, *SPE* Program Classification, Theory.

1. INTRODUCTION

The term *evolution* describes a phenomenon encountered in many different domains. Classes of entities such as natural species, societies, cities, artefacts, concepts, theories, ideas, for example, are said to evolve in time, each in its own context. In all these, and many other, instances the term refers to continued progressive change in the properties or characteristics of some material or abstract entity or of a set of entities. This *process* of change in one or more of the class attributes leads to the emergence of new properties or to improvement, in some sense. In general, the change will be such as to adapt the elements of the class 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. Alternatively or at the same time, evolution of the class may remove properties no longer appropriate. Changes are generally incremental and small relative to the entity as a whole but exceptions may occur.

In the same way that the properties of individual or classes of entities displaying evolutionary behaviour vary widely so do those of evolution *processes*. To distinguish between instances of the latter one may start by classifying them according to one or other of their characteristics. A study of common factors and differences between classes teaches one more about each. It also provides clues to the relationship between evolutionary and other properties of each class and between classes and may suggest how individual evolutionary patterns and trends may be predicted, directed and controlled.

The present paper is limited to a discussion of software evolution *per se*. One may be able to increase understanding of the phenomenon by studying evolution in other domains, biological systems (e.g., [wei70; SEEC02]) or scientific theories for example. Comparing and contrasting the behaviours, and other evidence that supports interpretation of that behaviour, may tempt one to suggest common underlying mechanisms and drivers across various domains. The present paper argues, however, that one must proceed with caution. Fundamental mechanisms such as feedback appear to drive evolution in many domains, but there are also differences to be recognised, even across the limited number of software related domains, termed *areas*, discussed in this paper.

2. RESEARCH VIEWS OF THE EVOLUTION CONCEPT

The term *evolution* may, generally, be interpreted and studied from several distinct points of view. In the software context, for example, the more widespread approach sees the important evolution issues, those most worthy of study, as those that focus on the *how* of software evolution. It explores the methods and means whereby a software system may be implemented from *ab initio* conception to operational realisation and evolved to adapt and extend it to be more satisfactory in a changing operational environment. The view focuses on mechanisms and tools whereby progressive change and growth may be achieved in a systematic and controlled manner. Within this view, a more restricted approach limits use of the term *evolution* to *software change*. Defect fixing, functional extension and restructuring for example, are explicitly excluded (e.g., IWPSE 2001, [FFSE 01]).

An alternative approach, less frequently invoked but equally important, view is concerned with the *what* and the *why* of evolution. It addresses issues of the *nature* of the evolution phenomenon, its drivers and its impact. The importance of this view, the main focus of this paper, becomes apparent with recognition of the fact that, as discussed below, the software evolution process is, and must be treated as, a multi-agent, multi-level, multi-loop feedback system [leh94]. Consideration of the *what* and the *why* view has so far been limited (e.g., Lehman *et al*, 1969 – 2001, [cho81; kem99; raj00; ant01]). It is based on a conviction that more insight into, and better understanding of, the evolution phenomenon must lead to improved methods for planning, managing and implementing it. It can, for example, help identify areas in which research effort is most likely to yield significant benefit.

The co-existence of these views, and possibly others, indicates that there is no single established view of *evolution* in the context of software [ben00]. The present paper is intended to expose some of the many wide-ranging implications to be considered in addressing evolution related issues. In so doing, the paper may contribute to the development of a unified view based on common understanding. This is surely required if progress in this research area is to be accelerated and is to achieve results that have real impact on practice [mit01].

3. PROGRAM EVOLUTION AND ITS DYNAMICS

It is, by now, generally accepted, e.g., [pfl98a,b; ben00] that, as formally stated in Lehman's first law of software evolution [leh74] software must be continually adapted, enhanced and extended if it is to remain satisfactory in use. Concern about the cost and impact of this universal experience was first publicly voiced and discussed at the Garmisch Conference in 1968 [nau68]. At that time it was expressed in terms of the continuing need for software *maintenance*. At about the same time one of the present authors (mml) studied the programming process within IBM [leh69] though his report did not become generally available till much later [leh85]. *Inter alia,* the report examined and modelled the continuing change process being applied to IBM's OS/360[1] operating system. It derived preliminary and very simple models of that system's evolution from measures of software release characteristics and proposed use of these models, or more refined versions, as tools to support planning and management of sequences of software releases [bel72; leh80]. Following identification of the software process as a feedback system, a phenomenon termed *Program Growth Dynamics*, later renamed *Program Evolution Dynamics*, was identified and its study initiated [leh74]. The study produced valuable insight into the nature and properties of software and the software *process*, based on models of the release process for OS/360 and other software systems [leh80,85] and tools for release management [leh80].

Lehman and Belady's work on program growth dynamics and software evolution had considered program evolution dynamics as an intrinsic characteristic of *large* programs. A widespread view was, and remains, that a program is *large* if it contained more than some arbitrary number, perhaps, for example, 100K lines of deliverable source code (KLOC). Lehman was critical of this view on the grounds of its arbitrariness. Instead he proposed a definition that classified a program as *large* if "... it was designed, or had been developed or maintained in a management structure involving at least two groups..." [leh79], and therefore involves at least two levels of management. That fact alone could, it was asserted, account, directly or indirectly, for many of the observed behavioural phenomena.

Even with this definition, however, concerns remained about the suitability of *large* (or any other descriptor of "physical" attributes of a software system) as the primary characteristic of the evolutionary character. This led to a classification scheme that did not involve the concept of size *per se*. Instead, it identified programs of types *S*, *P* and *E* respectively [leh80,82]. The defining properties of these types are discussed below. It is the *E*-type, and its intrinsic need to undergo evolution that is of particular relevance.

3.1 *SPE* Program Classification

The *SPE* program classification scheme has been discussed many times, most recently in [leh02]. The paragraphs below follow this recent description briefly addressing each type in turn.

S-type programs are defined here as those addressing a problem with a computational solution in an abstract and closed, for example mathematical, domain. It follows that the *sole* criteria for accepting satisfactory completion of a *contract* for the development of such a program is that the completed product *satisfies* a specification reflecting the behavioural properties expected from program execution [leh80,82,85]. This presupposes existence of an appropriate *formal* specification, accepted by the prospective users as *completely* defining the problem to be solved or the *need* to be fulfilled. Such a specification becomes a contract between a supplier and the representative of such users. Whether the results of execution are useful in some sense, whether they provide a solution to the specified problem, will be of concern to both users and producers. However, once the specification has been contractually accepted and the product has been shown to satisfy it, by definition the contract has been fulfilled. Thus, if results do not live up to expectations or need to be adjusted, that is, if their properties need to be redefined, rectification of the situation requires a new, revised, specification to be drawn up and a new program to be developed. Depending on the details of changes required, such a new version may be developed from scratch or obtained by modification of that rejected. However achieved, it is a *new* program.

These views of *S*-type programs imply that the specification expresses *all* the properties that the program is *required* to possess to be deemed satisfactory or acceptable and that it is *correct* in the full mathematical sense relative to the specification. With the exception of situations, such as when decidability issues arise, e.g., [apt86], demonstration of *correctness*, by means of a *proof* for example [hoa69,71], is not a matter of principle but of mathematical skill and the availability of appropriate tools and resources.

The designation *S* was chosen to indicate the decisive role that the *specification* plays in determining *required* product properties. Pfleeger [pfl98a,b] assumed an alternative interpretation, suggesting that *S* stands for *static*, a property which, as mentioned above and further clarified below, distinguishes it from the intrinsically evolutionary type *E*.

The *E*-type were originally defined as "...programs that mechanise a human or societal activity..." [leh80]. This description was subsequently extended to include all programs that "...operate or address a problem or activity in the real world...". They are intrinsically evolutionary, must be evolved as long as they remain in active use if they are to remain satisfactory to stakeholders, hence the designation *E*. The need and demand for continued change, correction, adaptation, enhancement and extension cannot be avoided if such software is to remain operationally satisfactory. Section 5 and 6 below elaborate on this and related assertions.

[1] In general, references in this paper to OS/360 refer to both that system and to its successor OS/370, having been renamed as part of the introduction of the 370 hardware.

To make the classification as inclusive as possible a type *P* was also included in the original schema [leh80]. This type covers programs intended to provide a solution to a problem that can be formally stated even though approximation and consideration of the real world issues are essential for its solution. Subsequently, it was suggested that the type could, in general, be treated as one or other of the other two. But there are classes of problems, chess players, for example, that do not satisfactorily fit into the other classes. Thus, for completeness the type *P* must be retained. When *P*-type programs are used in the real world, however, they acquire such *E*- type properties as, for example, the intrinsic need for evolution as discussed in this paper. They also acquire inherent uncertainty in the acceptability of the results of their execution [leh89,90,02]. Hence they are not considered separately in the present paper.

3.2 *E*-type Evolution as a Feedback Driven Phenomenon

Data on the evolution of a variety of systems, of differing sizes, from different application areas, developed in significantly different organisations and with distinct user populations has been collected over the years (Lehman *et al* 1969-2001). The accumulated observations and data can, in general be interpreted as being consistent with the hypothesis that software evolution is a recognisable phenomenon [leh80] in which *feedback* plays an important role. The wider implications of these observations were, however, only recently recognised [leh94]. Figure 1 is the earliest example of supporting evidence. It plots the growth over releases of OS/360. The plot shows an essentially linear trend to release rsn 19 but with a *superimposed ripple*. Similar ripples have been observed in many of the other systems whose studies followed in the seventies, eighties and more recently, in the FEAST projects [www01]. While remarkably similar in the general trends they display, the growth patterns of the various systems studied differ in some of their detail. Of immediate interest is the fact that, unlike OS/360 where the *long-term* growth rate over releases was positive and linear until instability set in after rsn 20, for five of the six FEAST systems this rate, although also positive, was predominantly declining. For four of the latter, however, growth recovery points were clearly visible. This is further discussed in section 6.

The cause of the OS/360 instability is believed to have been due to excessive positive feedback as evidenced by the growth of rsn 19 to rsn 20 by an increment significantly greater than that of any of its predecessors. The data plotted in figure 1 provides, therefore, two pieces of evidence, the ripple effect and an unprecedented growth increment followed by instability, supporting the 1971 hypothesis [bel72], that the software evolution process is a feedback system. That paper observed that "...from a long-range point of view the rate of system growth is self-regulatory, despite the fact that many different causes *control the selection of work implemented*[2] in each release, with varying budgets, increasing number of users desiring new functions or reporting faults, varying management attitudes towards system enhancement, changing release intervals and improving methodology". As interest in the software process and process improvement spread it became apparent to one of the present authors that the feedback observation had direct practical implications. This conclusion was encapsulated in the FEAST (*F*eedback, *E*volution *A*nd *S*oftware *T*echnology) hypothesis [leh94] which, in one of its formulations states that, apart from those based on less *mature* processes[3], software evolution processes are multi-agent, multi-level, multi-loop feedback systems. They must be seen and treated as such if sustained process improvement is to be achieved. Some of the implications of this hypothesis are discussed in publications generated by the FEAST projects. As its role and impact became more evident (for example as an abstraction of the mechanisms underlying many of the observed characteristics of the *E*-type evolution process such as the first seven laws of software evolution), the feedback observation was enshrined as an eighth law [www01].

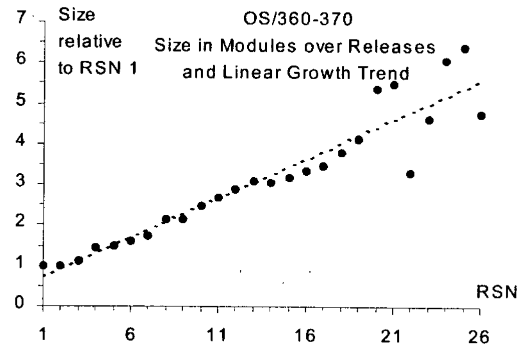

Figure 1. The growth of OS/360 over releases as a function of release sequence number (rsn)

4. AREAS OF SOFTWARE *RELATED* EVOLUTION - Summary

The discussion in section 3 has concentrated primarily on the phenomenon of program evolution. However, evolution in the wider software related domain it is not confined to sequences of programs and artefacts of the programming process such as specifications, designs, and documentation. Other entities involved in software development and maintenance, such as programming paradigms, languages, object definitions, usage domains and practices, applications and the very processes of software evolution themselves also evolve. These various evolution processes affect, interact and impact one another. To truly master software evolution one must understand and master the evolutionary properties of all these entities individually and collectively. Ideally and to the extent that this can be done, their evolution must be jointly planned, and controlled. In the first instance, however, one must focus on individual aspects. Consideration of the consequences of interactions between them can then follow, so are mentioned here only in passing.

Classification of areas of evolution in software and software related domains, including software technology, can be based on

[2] Italicised here to expose the *control* inputs that are all at least partly dependent on feedback mechanisms.

[3] For a discussion of the process *maturity* concept and its practical assessment see, for example [pau93; zah97].

various alternative approaches. The list that follows represents just one such schema:

I. The lowest level at which evolution is a factor is in the *implementation* of programs and software systems from initial statement of an application concept to the final, released, installed and operational code and supporting documentation. The implementation of a set of changes and/or enhancements to an existing system is a sub-area. The entire area and its processes, often referred to as *development* relies heavily on information and instruction feedback based on both formal and informal mechanisms. At the start of an *E*-type system development, for example, knowledge and understanding of the details of the application to be supported or the problem to be solved as well as approaches and methods of their solution are often undefined, even arbitrary [tur81]. The relative benefits of alternatives can often not be established except through trial and error and even the results of these are unlikely to be comprehensive or conclusive. The development process is a learning process in several dimensions. These include both the matter that is being addressed and the manner in which this is done. Such experiences based feedback involving evaluation of past results leads to an evolutionary process. Development and change activities typify the constituents of area I in the present schema.

II. At the next level up, consider a *sequence* of *versions*, *releases* or *upgrades* of a program or software system. These incorporate changes that rectify or remove defects, make provision for alternative operational environments and/or implement desired improvements or extensions to system functionality, performance, quality and so on. They are made available to users by means of what is generally termed a *release process* (e.g., [bas96]).

Basically all these changes are regarded by stakeholders as improvements to the program in one sense or another. Otherwise they would not be undertaken. As already observed, this process is widely referred to as program *maintenance*. Many people over the years have, however, recognised that that term is inappropriate, even misleading, in the software context. After all, as used in other areas, *maintenance* describes an activity seeking, in general, to rectify wear and tear or other deterioration that has developed in, say, an artefact. The purpose is to return it as near as possible to its original initial, pristine state. But software as such is not subject to wear and tear. It does not deteriorate *per se*. The deterioration that users sense arises from, for example, changes in the *purpose* for which the software was acquired, the changing properties of the operational domain, advancing technology or the emergence of competitive products. Such deterioration will often be a consequence of *assumptions*, that have become invalid as a result of external changes but are, nevertheless still implicitly or explicitly reflected in the software. In this context, therefore, the term *maintenance* refers more appropriately to maintenance of the *validity of the assumption set* reflected in the software and its documentation, as well as its satisfactory behaviour as judged by the stakeholders, users for example. In short, software is *evolved* to maintain the validity of its embedded assumption set, its behaviour under execution and its compatibility with the world *as it now is*.

The preceding discussion has distinguished between areas I and II. The first covers the *development* of an entire system *ab initio* (or of a change to an operational version of a system) whereas the second addresses adaptation and enhancement of a developed system *over a sequence* of releases. Changes and additions constituting the release also require specification, design and implementation. Thus area II involves area I activity and evolution.

III. *E*-type software supports *E*-type *applications* and the latter must also evolve. Applications span the range from pure computation to co-operative computer supported humans-organisations-machines activity. Introduction into use of successive software versions by the user community inevitably changes the activity supported. It also changes the operational domain. Furthermore, it often opens up new opportunities for performance and functional enhancement, efficiency and cost reduction. New standards of satisfactory operation are recognised and changes to the system are initiated to achieve them. Installation and operation of an *E*-type system, drives an unending process of system and application evolution.

As an application evolves so, inevitably, do the processes and domains within which it operates or is pursued. Installation and operation of a new or improved system will also impact other processes and domains with which the application, its processes and its operational domain interacts or is associated. Hence one must to consider, or may be forced into adopting, evolutionary software changes. Evolution of an element of a system ripples out and spreads throughout its wider context and encompassing domains.

Evolution of the processes and domains referred to in the previous paragraph may be regarded as sub-areas of area III or as areas in their own right. The former approach is adopted in this initial analysis.

IV. The *process* of software evolution refers, in general, to the aggregate of all activities that implement one or other of the above levels of evolution. It is variously estimated that between 60 and 80 percent of the resource applied to evolve a software system over its lifetime is incurred after first release [pgs96], in area II evolution and may reach 95 percent or more in defence applications. Hence, there are sound economic and logistic reasons to improve the process to achieve lower costs, improved quality, faster response to user needs for change and so on. Societal dependence on computers and on the software that gives them their functional and computational power is increasing at ever increasing rates. Process improvement to reduce human exposure to the consequences of high software costs, computer malfunction and delays in adaptation to changing circumstances demands improvement of the means whereby evolution is achieved. And the improvement achieved must produce improvements in quality, cost, and implementation-time according to the needs of each application area and application domain. The process evolves, driven by experience and a changing real world, including evolving external processes, technological advances and so on.

V. Achieving full understanding and mastery of the software evolution process, a complex multi-loop feedback system, remains a distant goal. *Process modelling*, using a variety of approaches, is an essential tool for study, control and improvement of the process, e.g., [pot84]. The *models* facilitate reasoning about it, the exploration of alternatives and change impact assessment, for example. The process evolves. So must models of it.

5. AREA I: SOFTWARE *AB INITIO* IMPLEMENTATION[4]

Ab initio implementation of a program or changes to an existing program requires execution of a series of discrete, iterated, and often overlapping steps by interacting individuals and teams, using a variety of, in general computer based, tools. Their joint action over a period of weeks, months or years produces the desired program or a new *version* or *release* of an existing program. The many steps or stages in such development differ widely. In probably the first published model of the software process, the Waterfall model [roy70] and its subsequent refinements, e.g., [boe76,88], the various steps are identified by terms, in current terminology, such as *requirements elicitation and analysis*, *specification*, *high level design*, *detailed design*, *coding*, *unit test*, *integration*, *system test*, *documentation* and so on. Execution of these steps is not purely sequential. Execution of any step, for example, may reveal an error in one or more earlier steps or may suggest an improvement to the detailed design or underlying assumptions associated with them. The process may also be described as one of *successive transformation* [leh84]. It is driven by human analytic and creative power as influenced and modified by developing insight and understanding. Feedback from later steps leads to iteration over earlier steps. Changes in the external world that must be reflected in the system, also serve as drivers.

Successive steps are likely to operate at many different conceptual and linguistic levels of abstraction and require different transformational techniques. The aggregated effect is, effectively, a process of successive refinement [wir71] that transforms the original application concept into its operational implementation, the software system. It is in this sense that the use of the term *development* is legitimate. A high level view of what is going on suggests, however, that the process must also be recognised as *evolutionary* since the transformational steps are elements in a successive-transformation paradigm [leh84] that satisfies the definition of evolution. They add or modify functional and computational detail in a reification process as described in the LST (*L*ehman, *S*tenning, *T*urski) paradigm (figure 2). That view may be appear too high a level of abstraction, too remote from the complexity of the industrial software process to be relevant in the present context. It is, however, briefly discussed here because it provides insight into the nature of the software development process, helps illustrate fundamental issues that emerge during software *ab initio* implementation and also to exemplify one practical significance of the *SPE* classification (section 3.1).

[4] Area I also encompasses he implementation of a set of changes or enhancements to an existing system.

5.1 LST Paradigm

The LST paradigm identifies each step in the implementation process as the transformation of a specification into a model of that specification (figure 2). Alternatively, it may be described as the transformation of a design into an implementation. The transformation step is not finalised until its output has been *verified* as being correct in the strict mathematical sense where *correctness* is a precise *relationship* between input (i.e., the specification) and output (i.e., the implementation). For *S*-type applications, this transformation process will, if faithfully followed, eventually lead to a satisfactory program, since to be deemed satisfactory and acceptable, by definition and subject to the provision below the specification must express all the properties that the program is required to possess. For *E*-type applications, on the other hand, there can, inherently, be no *guarantee* that the transformation process will yield a satisfactory program. That is so, even if a formal specification together with means for demonstrating the correctness relationship is available. At best, means such as *testing* under appropriate conditions, for example, can be used to *increase confidence* that the process output will prove satisfactory when executed in the real world. Such alternative means of validation can, however, not be absolute. To paraphrase Dijkstra [dij72], testing can only demonstrate the presence of defects, never their absence.

It has long been recognised that any statement about the *absolute* correctness of an *E*-type program is meaningless in the context of *E*-type applications. Why? *E*-type specification and program are necessarily finite, but mirror applications and domains having an unbounded number of attributes [leh89,90,02]. Moreover, some of the real world attributes upon which successful execution may depend are subject to unpredictable change. But neither the specification nor the program can reflect such changes without (considerable) delay and then only with a risk that the program change introduced is incomplete or in error. For each instance of execution, however, the program will be judged as satisfactory or otherwise in terms of the domain as it is during execution.

Intrinsically, *E*-type programs address applications for which the specification *cannot* at each moment in time, reflect fully and unambiguously *all* the properties that the program must display to remain satisfactory. Hence, even though parts, or even the whole, of an *E*-type program may be shown to satisfy a formal specification, use of the term correctness in the *E*-type application domain is meaningless. Problem are due to changes in the application or the operational domain and to unavoidable delays in adapting the program and/or procedures and documentation to match those changes, not to faults in the program.

Nevertheless, the use of formalisms for, for example, partial specification of *E*-type applications may provide benefits [lam00]. In association with other techniques they may, for example, provide means for isolating, characterising and minimising inherent uncertainties or inconsistencies, and facilitate means to address the inescapable need to evolve the system and its parts in a systematic and controlled fashion as, when and in the time frame required.

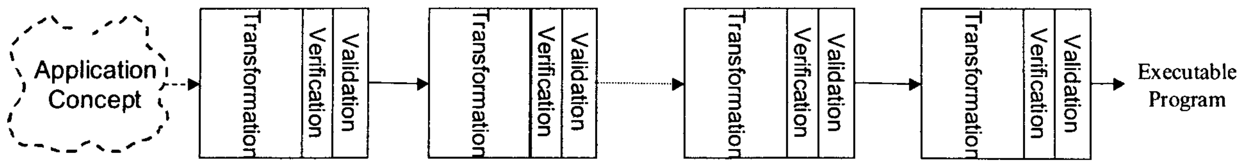

Figure 2. LST Program Implementation Paradigm

Lamsweerde also highlights the need to accompany a formal specification with a precise, informal definition of its interpretation in the domain of interest. The systematic evolution of these informal objects is a worthwhile activity in the context of *E*-type evolution. We refer to this again briefly later in this section when discussing the role of assumptions.

As briefly stated above, the LST paradigm, requires demonstration that the model output of each step satisfies its specification. Each model also requires *validation* – termed *beauty contest* in the LST paper. This serves the purpose of confirming (or otherwise) that, to the stage of refinement reached, the current model is *likely* to lead to a product that will satisfy the purpose for which it is being developed. Failure to validate implies a weakness, indicating that the final product may be unsatisfactory in the context of the intended purpose and domain. The problem may have arisen from changes in the purpose or the domain of the intended application, from oversights in generating the specification or from introduction by the transformation process of properties that are considered unacceptable in terms of the intended purpose though they are neither excluded by the specification nor incompatible with it. Whatever the source of failure, the source of the unacceptable properties must be identified and rectified by changing one or more of the transformation procedures, their inputs or the specification. If the latter is at fault it must be modified or, for *S*-type programs, replaced by a new one. If, however, both verification and validation are successful, or perhaps more appropriately, not unsuccessful, the new model becomes the specification of the next transformational refinement step.

5.2 *S*-type Software

By definition, *S*-type software is, contractually, acceptable if it has been shown to satisfy its specification. Its properties will reflect the specification in its entirety and verification is sufficient to determine its contractual acceptability. This implies that the completed specification is believed to be complete as far as the intended purpose of the program is concerned; that the problem to be addressed is fully understood and unchanging. Thereafter, it is primarily the knowledge, understanding and experience of the implementers that drives the implementation process. Learning during the course of that process is largely restricted to determination of methods of solution, or of identification and selection of the *best* method, in the context of constraints applying in the solution domain. Feedback in *S*-type program implementation is restricted. It may well be present at a low levels of development such as requirements design, coding or documentation. It plays a secondary role and is unlikely to dominate the implementation process.

Though verification is sufficient for *S*-type programs from a contractual point of view, business considerations require its *validation* since it determines the likely acceptability of the final product. Technically it shifts the responsibility for product acceptance to the client since, contractually, its definition was terminated by satisfactory completion of the *verification*, that is by acceptance of the specification by the client. If the latter proves deficient, refinement is required, the development process is abandoned and a *new* one based on a *new* specification is initiated. In practice, that new process may well take advantage of the earlier one. Nevertheless, conceptually the development from an initial concept and the derived specification, consists of a series of discontinuous open loop processes rather than a feedback-driven continuous evolutionary process. For *S*-type programs too evolution occurs but over a sequence or *generations* of instances not over the lifecycle of a single, isolated program as for type *E*.

5.3 *E*-type Software

In the case of *E*-type systems the problem to be solved relates to the real world. Thus an application (or change to an application) to be developed and the domain (or change of domain) within which it is to be solved are not, in general, clearly and uniquely defined. There will always be fuzzy aspects. If they are recognised they may be firmed up as knowledge and deeper understanding of the problem or application, of the operational domain and of acceptable solutions is acquired. This normally occurs during development as managers, developers and, possibly, clients take arbitrary decisions. In this situation, feedback plays a *crucial* role.

Parts of the multi-dimensional domain boundary of an *E*-type system will be well defined by, for example, prior practice or related experience. Its operational range will, therefore, be determined. Other parts of the boundary are adopted on the basis of compromise or recognised constraints. Still others will be uncertain, undecidable, possibly introducing inconsistencies. This situation may be explicitly acknowledged or remain unrecognised until exposed by chance or during system operation. The application domains have unclear, fuzzy, possibly fluctuating, *boundaries* that must be continually reviewed.

In relation to evolution, any initial fuzziness in development involves at least two separate and distinct aspects. The first relates to the intrinsic unbounded nature of any application and its operational domain. Initially the latter is neither precisely defined nor bounded in extent and in detail. Such uncertainty is removed by a *bounding process* that determines the domain over which the application is to be valid, used and supported, or to which it is to be adapted to provide a satisfactory solution in some defined time frame at acceptable cost. However, once the system is in operation, a need or desire to extend the area of validity to regions of the domain or to detail previously excluded will inevitably arise. If left alone, the latter, in particular, will become irritants and performance inhibitors. Equally, feeding back the need for modification or domain extension to the implementers and requiring them to satisfy newly emerging needs, changing constraints or changed environmental circumstances will exert pressure for system evolution.

The second concern relates to the boundaries of the system to be implemented. As anyone with experience in systems analysis, specification and design knows, the list of properties and function that could be included in a system is potentially unbounded. It is always in excess of what can be accommodated within the resources and time allocated for system implementation. Thus, from the point of view of potential coverage, the boundaries of the final system are arbitrary. But, unlike those of the domain, once developed and installed they become *solid*, determined, at any moment in time, by the installed hardware and software. A user requiring a facility not included within this boundary will, in the first instance, use stand-alone software to provide the required facility. It may be possible to couple such software tightly to the system for greater convenience in co-operative execution. But, however the additional function is invoked and the results of execution passed to the main system, additional execution overhead, time delays, performance and reliability penalties and sources of error are incurred. The omissions become onerous, a source of performance inhibitors and user dissatisfaction. The inevitable result is a request for system extension. The history of automatic computation is rich with examples of function first developed and exploited as stand alone application software, migrating inwards to become, at least conceptually, part of an operating or run time system and ultimately integrated into some larger application system. In some instances the migration continues until the function is implemented in hardware (chips) as exemplified by language and graphics support. The evolving computing system may be seen as an *expanding universe* with an inward drift of function from the domain to the core of the system. The process is driven by feedback about the strengths, weaknesses, effectiveness and potential of the system as recognised during use of the system or its outputs.

E-type programs and systems are the entities of ultimate concern of software technology. So is the process of system evolution over versions, releases and upgrades that maintain system applicability and viability, its value in a changing world. Their development and adaptation cannot be covered by an exhaustive and complete theory and partly to human involvement in the applications. It is equally due to the partial arbitrariness of procedures in business, manufacturing, government, the service sector and so on. Finally, it also relates to actual or potential imprecision of the operational domain boundaries [tur81]. Properties such as these make implementation and use of the systems a learning experience. The system is intrinsically evolutionary.

Any program is a bounded, discrete and static reflection of an unbounded, effectively continuous and dynamic application domain. The boundaries and other attributes of the latter are intrinsically fuzzy but must be fixed before, during and after development, primarily by operational, economic, time, and technology considerations and constraints and the striving for growth of human individuals and organisations. Some boundaries are determined explicitly in processes such as requirements analysis and specification, others involve explicit or implicit assumptions adopted and embedded in the system during the evolution process. Fixing the detailed properties, such as those of human/system interfaces or interactions between people and the operational system will include trial and error. Fine design detail cannot be based on one-off observation, requirements elicitation, intuition, conjecture or statistics alone. It arises from continuing human experience, judgement and decision by development staff and users. Development changes perception and understanding of the application itself, of facilities that may be offered, of how incompatibilities may be resolved, what requirements should be satisfied by the solution, possible solutions and so on. In combination such considerations drive the process onwards, by experience and learning based feedback, to its final goal, an operational system that is satisfactory by criteria considered appropriate during development and on its completion and acceptance. Subsequent evolution over versions or releases is, however, inevitable for maintenance of that satisfaction.

5.4 Component-based Architectures

The LST paradigm described above and the distinction between the nature of the *evolution* process in the context of *S*- and *E*-types systems may appear abstract and without practical implications. That is certainly not so. Moreover, an increasing trend to the use of component-based architectures, reuse and COTS will make their significance more widely appreciated. Why? Because these approaches are conceptually based on elements, that must, implicitly and in isolation, at least, have been assumed to be *fully* specified, that is to be of type *S*. In practice, however, to maintain stakeholder satisfaction and adaptability to evolving needs and a changing environment such components require' the malleability and evolutionary characteristics of *E*-type systems. They too must be subject to evolution [leh00c].

6. AREA II: SOFTWARE SYSTEM EVOLUTION

The relationship of *E*-type software to the real world may be described as a *model-like reflection.* In the accepted mathematical meaning of the term *model*, both the *application* in its real world operational domain and the *program implementation* are models of a common specification obtained by an abstraction process from a vision and understanding of the application. The program is the end result of a reification process. Program and application may, and will normally, possess other properties compatible with the specification. The term *model-like reflection* is used to convey the fact that in addition to reflecting all properties identified by its specification, a program must remain compatible with the real world application in its operational domain. Thus, software maintenance may be defined as *maintenance of reflective validity between a program and the various domains within which it is* *developed and used.* Continuing maintenance reconciles the system with its application, operational domains and stakeholders views all continually changing. The area II evolution process over a sequence of version seeks to *achieve* and *maintain reflective equilibrium* [lac96] between them. It culminates in a release process making the evolving system available to users.

As already indicated, evolution is intrinsic to *E*-type software, systems and applications. Given the appropriate economic, social and technical conditions, they co-evolve over a succession of *versions, upgrades* or *releases* that is an adaptation, improvement (in some sense) and/or extension of them. It represents one step in an ongoing, complex, *evolution* process. The sequence of releases transforms the system away from one satisfying the original concept to one that successively supports changing circumstances, needs and opportunities in a changing world. If conditions to support such evolution do not exist, then the program will gradually lapse into uselessness as a widening gap develops between the real world as mirrored by the program and the real world as it now is (First Law of Software Evolution, [leh74,78,80,97]).

As mentioned above and described in a number of papers since then, the study of software system evolution emerged from a wider software process study [leh69,85]. *Inter alia*, the original study examined empirical data on the growth of the IBM OS/360 operating system. It concluded that system evolution, as measured for example by growth in size over successive releases, *displayed a remarkable degree of regularity. This was unlikely to have been the consequences of planning and decisions by process participants or actively sought by them* but was more likely to be the consequence of the dynamic forces [for61] to be associated with the feedback nature of the software process. The empirical data that first suggested this conclusion is exemplified by figure 1 above. This plots system size measured in number of modules - a surrogate for the functional power of the system - against *release sequence number* (rsn). This relatively stable growth trend was ended by a period of instability that preceded break up of the system into separate systems, VS/1 and VS/2.

When plotted over releases, the *long term* growth trend of OS/360 up to rsn 20, as displayed in figure 1, was close to linear. The superimposed ripple suggests self-stabilisation around that trend. It is described as *self*-stabilisation because no indication could be found that management sought linear growth, that, in fact, growth considerations played any conscious part in defining individual release content. The latter was a consequence of the aggregation of individual management considerations and decisions; based on many inputs from many sources and co-ordinated, to a greater or lesser extent, by release management. The stabilisation suggested by the ripple must reflect the consequence of the organisational integration of all these inputs in and via a complex, multi-loop feedback structure mediated by managers, policies, established and *ad hoc* practices and other mechanisms.

The stabilisation phenomenon triggered the first realisation that feedback might be playing a role in determining the pattern of growth in functional power of an evolving system, and other attributes, of system evolution. As discussed above, this conclusion was reinforced by the observation of OS/360 post rsn 20 instability [bel72].

6.1 Recent Empirical Studies

Follow-on studies in the 70s and 80s (e.g., [leh85]) identified further evidence of a degree of commonalty in evolutionary behaviour across systems that led eventually to eight *Laws of Software Evolution* encapsulating these behavioural invariants [leh74,78,80,97; www01]. Following formulation of the FEAST hypothesis [leh94], the FEAST/1 and /2 studies, further explored the evolution phenomenon. Figure 3 provides one example of the similarity between the observation of OS/360 (figure 1) growth and observations some 30 years later of the growth of one of the systems studied in FEAST. That study, in fact, analysed growth data from six systems - a financial transaction system, two real time systems, an information system, an operating system kernel, and a defence system. The study identified qualitative and quantitative similarities and the interested reader is referred to the relevant literature for details [www01]. It was concluded, directly or indirectly, that the FEAST data was generally consistent with refined versions of seven of the eight laws of software evolution. They did suggest some, relatively minor, modification of earlier overall results but, overall, strengthen confidence in the universality of the phenomenon of *E*-type software evolution, at least under the paradigms and within the evolution environments represented by the systems studied.

An example of such similarities is provided by the fact that five of the six FEAST systems display a positive, but predominantly declining, *long term* growth rate over releases. For these systems the growth trends were closely replicated by several growth models [tur96; leh97,01c], one of them termed the *inverse square model*. This takes the form $S_{i+1} = S_i + E/S_i^2$ where S_i is the predicted size of the release with sequence number "i" measured in appropriate units and E is a model parameter as determined from data on the growth history of the system [tur96]. Moreover, in four of these five systems a mid-life break-point (rsn 9-10 in figure 3), with different degrees of prominence in each of the four cases, can be observed in the long term growth trends. This suggested that changes, in, for example, the evolving system, the evolution process and/or the environment led to growth rate recovery. The presence of segments provides behavioural support for the view that evolutionary stages must be understood [ben00; raj00], but raises interesting questions such as the extent to which the occurrence of such break-points can be interpreted in a manner consistent with the other observations (e.g., the laws). These matters will be clarified if the proposal to develop a software evolution theory is realised [leh00b,01b]. The sixth system - (a defence system) was excluded from the area II analysis because it related to an *ab initio* development over a sequence of company internal releases. That is, the project was primarily concerned with area I activity though some features of area II activity also present [wer98].

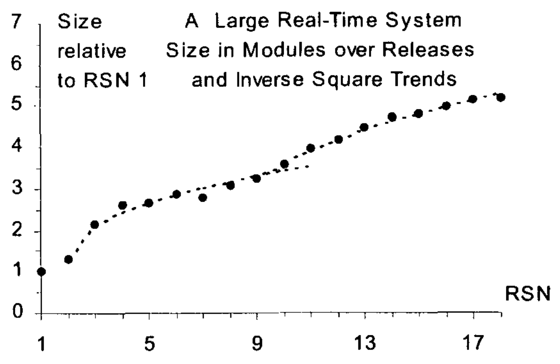

Figure 3. Growth trend of one of the systems studied in the FEAST projects (dots) with Inverse Square Models fitted to two individual segments (dashes).

The observed positive but declining growth rate trends in the FEAST systems is consistent with a hypothesis that declining growth rate may be attributed, at least in part, to growing complexity of the evolving system and application as change is applied upon change. Such complexity growth can, of course, be compensated for by complexity control, also termed *anti-regressive* activity [leh74] based, for example, on re-engineering, system restructuring, *refactoring* [fow99], and so on. Of course, such compensation involves a cost in resources and a trade-off between immediate short-term gains and long-term evolution sustainability. System dynamics models [for61, ven99] of the process reproducing this phenomenon suggest that maintenance of a close to linear growth is achievable at the price of allocating an adequate level of resource to anti-regressive activity [kah01].

6.2 The Role of Feedback in Area II

The hypothesis that feedback plays an important role in the release based area II evolution processes are, in part at least, driven and controlled [leh85,91,94] is consistent with both interpretation of the observed data sets and anecdotal observations. Feedback mechanisms are, still, the most likely underlying unifying factor for the observed commonalties. Drivers of that feedback may also play a direct role. Such drivers include, but are not limited to, *defect reports* from the field, domain changes due to installation, operation and use of the system, changing user needs, recognition of new opportunities, advances in technology, even the economic climate. However, detailed quantitative testing of this hypothesis (and of its negation) by, for example, *control theoretic* or *system identification* methods, e.g., [lju87], presents many challenges.

The above feedback mechanisms reflect *experience* that changes user *perception*, their *understanding, desires* and *ambitions* and hence underlying application and system concepts, assumptions and abstractions. And these combine to produce pressure for software change, pressure that in most cases is exerted in and on the process by one or other of a variety of feedback channels and controls that demand action on the part of the supplier. Nor is the source of feedback confined to developers and the user community who exploit the accumulated insight. Many contribute to the change process. But in all cases information feedback is a major driver, with the nature of the path and mechanism, the degree of authority of decision makers for example, determining its impact.

The information propagates along paths involving human interpretation and significant delays. The people involved have a direct impact on the information, that is on feedback characteristics. Process internal feedback paths are relatively short and involve experts in the application, the development process and the target system. Their feedback is based on expert interpretation. In control theoretic terms it can be interpreted as a low level of amplification and delay but with possibly significant levels of noise and distortion. It is the outer external user, long delay and influential (i.e., high amplification) feedback pressures, the *business-based* loops that are believed to be primarily responsible for the characteristics of *release* dynamics.

6.3 Further Work in Area II

Some of the issues mentioned clearly suggest issues for further research. When clarified, they will help advance understanding of area II evolution. Other themes of further research relate to the role and impact of evolution environments comprising applications areas, their operational domains, the industrial implementation and marketing domains, economic conditions and so on. The FEAST projects, being confined to just five distinct combinations of these various, and other, factors, did not permit the identification of specific dependencies or relationships. We mention just one example. None of the FEAST systems were instances of open source development. Others, however, have investigated evolution of instances of such domain. They have reported [gdf00; suc01] that the evolutionary trends of the open source systems studied (e.g., Linux) differ in detail from those of the commercial systems studied in FEAST. They also display positive growth but with instances of increasing growth rate. Explanations of this fundamental difference have been advanced. The proliferation of functional duplicates (clones) in open source, together with the participation of an unrestricted pool of developers, for example, may explain the anomaly. The difference, however, may also be pointing towards possible differences in evolutionary behaviour across domains (open source, commercial), suggested in [pir88; suc01]. When studying the impact of specific development environment, for example, one might reasonably expect some impact from *business-based* loops (see further discussion of this issue below). These, of course, must be expected to differ significantly over domain. Further study is clearly required. Studies of the role of the impact of different technologies on long term evolution over releases offers similar opportunities for further investigation. A study of similarities and differences between across domains may provide additional insight and understanding of possible opportunities for improvement in software product quality and reliability (however defined), increased evolution productivity and rates of evolution, for easing the cost and ease of long-term evolution and the periodic introduction into operation of new versions of an evolving systems.

Last, but not least, one of the challenges in making the assessment of similarities across systems arises from the informality of the current statement of the laws of software evolution. This makes it difficult to make explicit and unambiguous links between observed behavioural patterns and the laws. Investigation of systematic ways of formalising the links between observations and summary statements is an interesting research topic. It is hoped that investigations into a formal theory of area II software evolution will address these and related issues. Plans are described in a companion paper in this volume [leh01b].

6.4 Evolutionary Development

Even though areas I and II reflect two distinct levels of software evolution, attention should also be drawn to specific development approaches that cut across those areas. As one instance that highlights these, consider the Evolutionary Development approach [gil81,88]. This develops and fields an *ab initio* or new evolutionary development in a sequence of releases each involving one or more new components or chunks of functionality. The parts are developed, integrated, installed and introduced into use in a predetermined order. Users become progressively exposed to a system of increasing functionality and power. The system evolves by leaps and bounds. With this divide and conquer approach the complexity of the task undertaken in any release interval is greatly reduced. Hence the degree and complexity of validation and of rework is greatly reduced. On the other hand, it does increase the amount of design partitioning and planning activity that allocates change elements between successive releases. The amount of regression testing and revalidation, for example, will also increase significantly.

Application of the approach depends on being able to architect and decompose the system so that constituent parts may be interconnected, part by part, to yield viable *sequences of evolution processes with systems* of increasing functionality and power. As a result, constituent parts of the total desired evolutionary change are exposed to system internal interactions and real usage much sooner, systematically and more progressively than would be the case if real world operation were to await the system's total completion. Learning and user feedback is taken into account long before development is completed. Since introduction of the system into real world usage invariably reveals faults not previously detected, early user exposure in smaller chunks is likely to greatly simplify fault removal.

Evolutionary development has, we believe, been industrially applied in practice but we are not aware of any empirical assessment of its effectiveness in relation to more conventional development approaches. The method is based on recognition of the fact that a major problem in real-world system development is that of uncertainty and risk associated with fixing the required system properties. A related issue is the lack of a theoretical framework to guide selection of system properties during requirements analysis, specification and design. Up to the arrival of component based architectures, the common and widespread industrial paradigm for release based evolution was one or other derivative of the waterfall model[5]. This involves many arbitrary decisions as progress is made in system development. These are not fully validated or rejected until the system has been fielded and is in regular use. If systematically and thoughtfully applied, Evolutionary Development may help overcome the resultant problems. If the process is adequately planned and faithfully implemented the approach should yield clear net benefits.

[5] We do not here discuss the relationship of these to Boehm's spiral model [boe88].

7. AREA III: EVOLUTION OF THE APPLICATION IN ITS DOMAIN

Continued evolution is not confined to the software or even to a wider application *system* within which the software may be embedded. It is inherent in the very nature of computer *application*. The activity that software supports and the problems it solves, also evolve. Such evolution is, in part, driven by human aspiration for improvement and growth. But more subtle forces are also at play. Installation and use of the system changes both the activity being supported and the domain within which it is pursued. When installed and operational, the output of the process that developed the software *ab initio*, or evolved an existing system, changes the attributes of the application and the domain that defined the process in the first place. In association with the application and the operational domain as defined and bounded, the development process, as outlined in the discussion of area I, clearly constitutes a feedback loop as illustrated in figure 4. Depending on the manner in and degree to which changes impact use of the system and on loop characteristics such as the amplification/attenuation and delays, the overall feedback at this level can be negative or positive resulting in stable growth or instability.

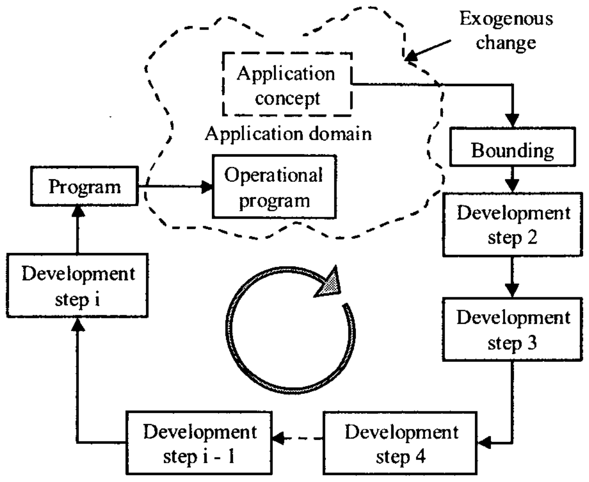

Figure 4. Evolution of the Application in its Domain as an Iterative Feedback System. Internal Process Loops not shown

In many instances, however, the phenomenon of application evolution is more complex than indicated in the preceding paragraph. In particular, it may not be self-contained but a part of the phenomenon and process of *co-evolution*. As government, business and other organisations make ever greater use of computers for administration, internal and external communication, marketing, security, technical activity and so on, the various applications become inextricably interdependent, exchanging and sharing data, invoking services from one another, making demands on common budgets. The inescapable trend is towards the integration of all internal services, with the goal, for example, of minimising the need for human involvement in information handling and communication to avoid delays and errors and to increase security. And such integration is seen as needing to gradually extend to clients systems, their customers, suppliers and other external organisations.

With this scenario, the rate at which an organisation can grow and be adapted to changing conditions, new opportunities, competitive challenges and advancing technology increasingly depends on the rate at which it can evolve the software systems that support its activities. More generally, in the world of today and, even more so, tomorrow, organisations, whatever their activity or sphere of operation, the domains within which they operate, the activities they pursue, the technologies they employ and the computer software which links, co-ordinates and ties all together will be inter-dependant. All must co-evolve, each one advancing only at a rate that can be accommodated by the others. And those rates depend not only on the various entities involved but also on the processes pursued and the extent to which these can be improved. Software is at the very heart of co-evolution, the means whereby it is achieved. Change to any constituent element of the global system will almost inevitably imply software change.

8. AREA IV: SOFTWARE PROCESS EVOLUTION

Over the past decade, computers and the software that gives them their functional capability, have penetrated ever more deeply into the very fabric of society, individually and collectively. The world at large has become more and more dependent on the *timely* availability of *satisfactorily operating* software at a *cost* that is commensurate with the *value* that the software yields when executed. But, as discussed above, as the world changes, even *S*-type software must be adapted and extended to yield satisfactory results at each moment in time. Errors or delays in this continuing process are likely to yield significant cost penalties due to incorrect or unacceptable behaviour. They can constrain, even throttle, organisations limited by out of date capabilities, *legacy* software. This has resulted in major investment in developing and applying software *process improvement* [e.g. zah97] paradigms such as CMM [pau93], SPICE [ele98], Bootstrap [kuv93] and ISO 9000 with its derivatives. These have and are being explored and applied the world over.

It is certain is that the software *process* as variously practised today is far from perfect, expensive, the source of delay in many computer dependent projects and of the major defects and deficiencies so often displayed by allegedly completed software products. Moreover, as new software technologies, Object Orientation, Component Based Architecture, UML or Java, for example, emerge they call for and suggest new approaches to software implementation and evolution. The net result is that software implementation and evolution processes also evolve. The fact that the new technologies are different in principle to earlier practice and that no comprehensive scientific base or framework exists for software technology means that process evolution efforts must rely on intuition, experience, emerging insight, inventiveness and feedback.

8.1 Process Improvement: Two Approaches

Two approaches to process improvement can be identified, the theory based and the empirical. The former, essentially addressing the *how* approach mentioned in section 2, is exemplified by the work of WG 2.3 [gri73]. This group has been meeting, initially informally, and since 1971 formally as an IFIP working group, to discuss its members' work and views on various aspects of programming methodology. The approach is bottom up, looking at how individuals might approach program development for problem solution. The groups' many positive contributions, owe much to earlier work by members of the group. Such work included Dijkstra's much quoted observation that "GOTOs are considered dangerous" [dij68], the concepts and procedures of structured programming [dij72] and the concepts of program correctness proving [dij68; hoa69,71]. The programming approaches developed and discussed by the group have provided basic concepts of modern programming methods as applying, in particular, to *S*-type programs. As such, they are crucial to the development of improved programming processes. They provide the basis for individual programmer practice [hum97] and, for example, for the development of defect free code. In general, however, the group recognises that the methods and techniques they advocate cannot readily be *directly* applied to the *large E*-type systems that are the principle concern of evolution studies. By definition, *S*-type programs do not evolve. As already observed, when they no longer satisfy their intended purpose a *new* program must be developed to replace them.

Thus, in general, the above approach has not been able to conquer the issue of largeness, however defined, though it has made fundamental and important contributions. It has also identified another basic problem in software evolution, the consequences and mastery of concurrent evolution at different semantic levels. Turski affirms that "...the problem of adapting existing software to evolving specifications remains largely unsolved, perhaps is algorithmically insoluble in full generality..." [tur00]. The feedback-system characteristics of the process support this conclusion.

The empirical approach, exemplified by the work of the FEAST group [leh94; www01] and of Kemerer [kem99], follows the *scientific method* by starting its consideration, based on the *what* and *why* approach introduced in section 2. This begins with observation, measurement, interpretation and hypothesis formation. In FEAST this was based on data recording the evolution of industrially developed and evolved software systems. The development of black box [e.g., leh97,01c] and white box (system dynamics) [e.g., wer98, kah01] observations-based models based on these hypotheses and observations permitted reasoning about the findings and has provided foundations for the gradual development of an empirically based theory. It is not possible to discuss the findings of these studies further here and the interested reader is referred to the referenced literature. It is, however, worthy of note that the eight laws of Software Evolution are a direct outcome, over a period of some 30 years, of such empirical observation and interpretation. The accumulated observations, insight and derived understanding and have already led to practical rules for software release planning, management and control [leh01a]. The time is now ripe for development of a formal Theory of Software Evolution [leh00a,01b].

8.2 *In vitro* and *in vivo* Process Evolution

Any instance of the process is *transient*, *ephemeral*. Once executed it is gone forever. It will normally have been pre-planned in outline with details adopted as progress is made. However unanticipated circumstances and unexpected conditions, specification changes, performance problems, budget changes, for example, are the norm and lead to process adjustments, adaptations and changes *on the fly*. Such unplanned changes, are error prone and therefore, in principle, undesirable though frequently triggered by observation of the results or consequences of past activity or by perception of what lies ahead. These may lead to a change to the planned upcoming process activity or a need to backtrack or iterate. In any event there is a complex mixture of feedback and *feed forward* based on individual and collective interpretation, intellectual judgement and decision by humans that will determine how to proceed. Whenever people are involved some degree of freedom exists; otherwise their activity could be mechanised. That freedom relates to what is done, what is not done and how the former is done. Hence the process can sensibly only be pre-planned and defined to a limited extend and to some arbitrary level of detail. It can be enforced only at a comparatively coarse level of granularity. Enforcement of a process may be specified at a high level of detail in specific circumstances (e.g., life critical, such as medical or aerospace software), but this can, itself lead, for example, to defect injection, inadequate treatment of unforeseen circumstances, high cost or serious time delays while authorisation to deviate is obtained. In commercial environments, subject to strong resource, time-to-market and other constrains expectation that the process will be carried out as planned may be naive in the extreme. The process will inevitably evolve, not only through pre-planning, *in vitro*, but also dynamically, *in vivo*.

9. AREA V: PROCESS MODEL EVOLUTION

Software processes are the aggregate of all constituent activities and the relationships between them. This comprises all the constituents, that are required to transform a computer application concept into a satisfactory operational system. Improvement of the process is achieved by improvement of its parts and the effectiveness, reliability and speed of interactions. The constituents themselves are comprised of both operational and managerial activity. Though at a sufficient high level the process steps can be seen as instances of a successive transformation paradigm (section 5.1), detailed enactment of a software process requires a wide variety of interacting entities and their activities. Many of these are outside the core transformational steps but are nevertheless needed to address the fuzziness of the application concept, to enable the orderly interaction of many stakeholders, and to ensure that the process outcome is achieved within the relevant economic, schedule and quality constraints. Real world processes are very complex. Understanding how they act and interact requires models that permit assessments that can subsequently be validated by observation and measurement of real world events.

9.1 Process Models and Process Improvement

Process models in general, and process programs [ost87,97] in particular, have been a major focus of software technology research for some time. Interest in the former went public with the first International Process Workshop [pot84]. It was, however, Osterweil's keynote address at ICSE 9 [ost87] that triggered widespread interest in *process programming* as a modelling technique, though serious questions about the approach were also raised [leh87] (section 10). At about the same time, system dynamics and other types of *behavioural* process models were used as a tool to achieve improved software project management [abd91] in the context of *ab initio* software development (area I evolution). Precursor models exploring the process as a dynamic system may also be found in several earlier papers [leh85].

Process models, of whatever kind, facilitate understanding of processes and communication about them. They are indeed essential as vehicles for communication and reasoning, a role greatly enhanced if they are formal. More specifically, they provide means for systematic and disciplined examination, evaluation, comparison and improvement of processes and, using process enactment and simulation, preliminary measurement, exploration, and evaluation of proposed changes [tul89]. But to remain of value the models must be adapted and evolved as the concepts, methods and processes they reflect advance and as the applications and software at which they are directed become ever more complex, ever larger and ever more integrated. Inevitably, all process models evolve. If any model is to serve a useful purpose it must reflect the process as the latter evolves. The inevitability of process evolution has already been discussed (section 8).

As mentioned in section 1, such process evolution will in most cases proceed slowly in incremental steps. Incremental changes that are local to the immediate process, introduction of a new step for example, may not, however, suffice to yield visible benefit. The ultimate measure of improvement is the impact as observed from outside the process, at the *global* level [leh94]. Typically, improvement goals include reduction of *overall* cost and of the elapsed time required to transform initial concepts into an operational system, time to analyse and remove a defect and release the correction to users and reduction in the total number of legitimate defect reports or the average rate of their submission once the system is in use. In this regard, behavioural process models that address these concerns can be useful (e.g. [wer98, kah01]. Process improvement can be developed and evaluated on all process models. The realisation and fine tuning can only be achieved, and ultimately utilised as a decision making tools, on *calibrated global* process models which, because of the feedback nature of the process, must reflect its dynamics.

9.2 Feedback Role in Process Model Evolution

Evolution of a process and its model must be linked. What is the nature of that linkage? Where impetus for change comes from a need to adapt a process to specific conditions or circumstances, model evolution is a consequence of process evolution or of creative or intuitive insights into possible improvements. Initial evaluation may often be obtained by implementing, exploring, and comparing alternative changes in the model, by enactment or otherwise, before incorporating the selected change in the process. In any event changes made to the process, whether premeditated or on the fly (something that should rarely, if ever, be done) must be reflected in a change to the model if the latter is to retain its validity and value. Where the pressure for evolution comes from recognition of a need for improvement, the process model can play a seminal role being used to design and evaluate the change before implementation. However exploited, the information that drives improvement is garnered from observation and previous experience.

Model evolution is also feedback driven. The flow will be from within the organisation, from other software developers and from process experts and practitioners [leh91]. Disciplined, and directed effort in process improvement is typified by the work of the Software Engineering Institute at Carnegie Mellon University [hum89]. A possible shortcoming is that their work does not explicitly focus on process models, feedback direction and control or the process dynamics. But those, in essence, are among the issues addressed and exploited.

10. AN ILLUSTRATION OF CONTRASTS BETWEEN EVOLUTION AREAS

While recognising the interactions between them, the above discussion of various levels at which software evolution phenomena occur has not suggested that there are similarities in the phenomena themselves. The reverse is, in fact, the case. Software *process* evolution, for example, clearly differs significantly from that of software itself. This is shown by the fact that the relationships between a software process and its models differ fundamentally from those between *E*-type software and the problem or application processes of which the software is a model-like reflection.

Two issues termed here the *computation process/software* contrast and the *software process/process model* contrast respectively are of particular interest, if only for historical reasons. Wherein lies the difference between these two? When software development processes and the models that describe them are considered, the focus of concern is the process even though the source of evolutionary change may have originated in a study of a model, a *process program* [ost87, 97] for example. Moreover, a proposed change and its consequences may be explored by use of a model and be evaluated by its enactment. Nevertheless, and even where a model-driven support environment is used to directly guide the process [tay89], the focus of concern remains with the process *in execution*. The latter is the consequences of humans interpreting specifications, processing directives, choosing directions, taking decisions, following methods, and applying tools. The proof of the pudding lies in the eating. The process model is a broad-brush tool to permit reasoning about the process but the consequences of executing the process depends on specific actions of individuals. The process models are incomplete; at best a high level guide to the process. As for other human activities [har02], plans and process models do not provide a precise and complete representation of the process to be, or actually, followed. If they are intended to be so, and this is enforced, the model becomes a straightjacket and bottleneck, a source of constraint in a domain where the unexpected and unanticipated is a daily occurrence. This must be contrasted with executable software. Once accepted this is accepted as providing a *precise*, *detailed* and *complete* representation of the actuality required or desired. Software completely and absolutely *defines* a *process of computation* subject, at most, to intervention by an operator or the use in premeditated fashion of input parameters. Such intervention will often not have been carefully or fully evaluated on the basis of a total overview of the situation. Thus it represents not only a slow down of the computational process but is also a possible source of error. Intervention of this sort will become ever more rare as computer systems become more ubiquitous and integrated, more the master than the slave.

In so far as the software is concerned, the languages in the various steps of development are all formal. Each definition will be absolute in the context of that language. Process models, on the other hand, are a *partial reflection* of a *process of software development*. It is the *product* of that process that is of concern. Changes to the model are incidental. They describe changes, proposed or implemented; concepts to be translated into reality by people. They are evaluated in terms of their impact on the process in execution. A process change may be conceived and incorporated in a model. The acid test comes with *execution of an instance of the process*. Determination of success or failure, improvement or deterioration of the process is judged on the basis of process dependant attributes of the product, its cost or quality for example.

The *ultimate* concern with *E*-type software is with the application process, and the consequences of program execution in the real world [leh91]. One is concerned with the development or evolution of software systems to be used by a changing population of (largely anonymous) people with different degrees of understanding, skill and experience. The concern will, in general, be with user community behaviour. Only in exceptional instances can code make provision for individual misuse, and that only if such misuse was be anticipated. An essential ingredient of successful software design is insulation of the system from user behaviour. Relative to the process/process model relationship, for software, therefore, the direction of interaction is reversed. Computer applications evolve, *inter alia*, in response to changes to the software and exogenous changes in the domain. This even though such changes may have been inspired by observation of real world processes, as influenced or controlled by execution of that software. Software changes tend to drive application changes or at least co-evolve with them.

There are also other significant differences. For example, *process* quality, productivity and cost concerns relate to the process, not its model. For software the reverse is the case. Quality, productivity, and cost concerns as visualised by the software engineer do not to relate primarily to the application but to the software as a model-like reflection of the application in its domain. Concern about the above factors does arise in connection with the application, but these must, in the first place, be addressed by application experts. Deficiencies will, in general, be overcome by changes to system requirements and specification, to be subsequently reflected future versions of the software.

Consider, finally, the time relationship between model changes and process changes and the nature of the feedback loops that convey the interactions. For the process the key word is *immediacy* whereas for software there is, in general, significant relative delay in feedback. One could go on listing the differences. The analysis as given suffices to indicate that the thesis that "software processes are software too" cannot not be taken too literally.

More generally, the analysis suggests that fundamental differences occur amongst the various co-existing co-evolving software-related domains, contributing to the difficulty of attempting to manage and control evolution phenomena. Understanding of similarities, but also differences, appears to be crucial. The degree to which commonalties exist between the evolution of such entities as those listed in the opening paragraph of this paper is even more obscure, with the only immediately recognisable commonality the fact that, in one way or another, feedback plays a role in most, if not all of them. These are matters which are beginning to be discussed (e.g., [SEEC02]).

11. FINAL REMARKS

The brief discussion of Gilb's evolutionary development in section 6.4 has indicated that the scheme proposed is not as precise as one might have hoped for. There are other examples In bringing one more in these concluding remarks a more general point can be made.

Rapid Prototyping [luq89] also combines views from areas I and II of evolution. This suggests that there might be advantages in addressing them and indeed, the other areas described in this paper, simultaneously. In particular it must be recognised that the low level evolution of area I is, in fact, applied to implement the evolution of the individual entities in the other areas. Thus while compartmentalisation has very clear benefits as an aid to understanding it remains arbitrary. It is certain that in industrial situations, for example, parallel evolution will occur simultaneously in several of the areas described. All must be individually and collectively considered and managed to ensure maximum benefit.

The objective of this discussion has been to expose the wider and crucial role of evolution and feedback in a number of domains related to software. Only recently has serious thought been given to this topic and firm conclusions must await future intensive directed study. The analysis presented here must be accepted as preliminary and exploratory. Its principal conclusions stem from the conjecture that evolution in the whole of software technology is, at least in part, feedback-driven. The characteristics of individual phenomena are partly functions of the properties of the feedback loops, with process evolution displaying fundamentally differences to other instances of evolution in software related domains. There is still much to be learned in this area. The nature, impact and control of evolution in software must become a major focus of future research and development.

12. ACKNOWLEDGMENTS

Many thanks are due to the FEAST collaborators and to our colleagues in academia for many discussions. All have helped to generate, prune and sharpen over the years the ideas exposed here. We are grateful to the UK EPSRC for funding that has supported work on this topic for more than five years.

13. REFERENCES[6]

[abd91] T. K. Abdel-Hamid and S. E. Madnick, *Software Project Dynamics - An Integrated Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1991, 264 pps.

[ant01] A. Antón and C. Potts, *Functional Paleontology: System Evolution as the User Sees It*, ICSE 23, Toronto, 12-19 May, 2001, pp. 421 – 430

[apt86] K. R. Apt and D. Kozen, *Limits for Automatic Program Verification of Finite-State Concurrent Systems*, Inform. Proc. Letters, v. 22, n. 6, 1986

[bas96] V. R. Basili *et al*, *Understanding and Predicting the Process of Software Maintenance Releases*, ICSE 18, Berlin, March 25-29, 1996

[bel72] L. A. Belady and M. M. Lehman, *An Introduction to Program Growth Dynamics*, in W. Freiburger, editor, Statistical Computer Performance Evaluation, Academic Press, New York, 1972, pp. 503-511

[ben00] K. H. Bennett and V. T. Rajlich (2000), *Software Maintenance and Evolution: a Roadmap*, in A. Finkelstein (ed.), The Future of Software Engineering, ICSE 2000, June 4-11, 2000, Limerick, Ireland, ACM Order Nr. 592000-1, pp 75 - 87

[boe76] B. W. Boehm, *Software Engineering*, IEEE Trans. on Comp. vol. C-25, n. 12, pp. 1226 - 1241

[boe88] *id.*, *A Spiral Model of Software Development and Enhancement*, Computer, v. 21, May 1988, pp. 61-72

[cho81] C. K. S. Chong Hok Yuen (1981), *Phenomenology of Program Maintenance and Evolution*, PhD thesis, Department of Computing, Imperial College

[SEEC02] Symposium on Software Evolution and Evolutionary Computation, U. of Hertfordshire, U.K, 7-8 Feb 2002, forthcoming

[dij68a] E. W. Dijkstra, *A Constructive Approach to the Problem of Program Correctness*, BIT 8, 3, 1968, pp. 174 - 186

[dij68b] *id.*, *GOTO Statement Considered Harmful*, Letter to the Editor, CACM, v. 11, n. 11, 1968, pp. 147 - 148

[dij72a] *id.*, *Notes on Structured Programming*, in Dahl, Dijkstra and Hoare, *Structured Programming*, Acad. Pr. 1972, pp. 1 - 82

[dij72b] *id.*, *The Humble Programmer*, ACM Turing Award Lecture, CACM, v. 15, n.10, Oct. 1972, pp. 859 – 866

[ele98] K. El Eman *et al*, *SPICE: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE CS Press, 1998

[gri78] D. Gries, *Programming Methodology—A Collection of Articles by Members of IFIP WG2.3*, Springer V., NY, 1978, p. 437

[FFSE01] Intl. Session on Formal Foundations of Software Evolution. 13 March 2001, Lisbon, Portugal http://prog.vub.ac.be/poolresearch/FFSE/FFSE-Workshop.html

[for61] J. W. Forrester, *Industrial Dynamics*, MIT Press, Cambridge, Mass., 1961

[fow99] M. Fowler, *Refactoring: Improving the Design of Code*, Addison-Wesley, New York

[gil81] T. Gilb, *Evolutionary Development*, ACM Softw. Eng. Notes, April, 1981

[gil88] *id.*, *Principles of Software Engineering Management*, Addison-Wesley, Wokingham England, 1988

[gdf00] M. W. Godfrey and Q. Tu, *Evolution in Open Source Software: A Case Study*, Proc. Intl. Conf. on Software Maintenance, ICSM 2000, 11-14 Oct., San Jose, CA, pp. 131-142

[har02] M. Hartswood *et al*, *"Cunning Plans": Some Notes on Plans, Procedures and CSCW*, RQ Newsletter, issue 25, Jan. 2002, http://www.resg.org.uk <as of Jan 2002>

[hoa69] C. A. R. Hoare, *An Axiomatic Basis for Computer Programming*, CACM, v. 12, n.10, Oct., 1969, pp. 576 - 583

[hoa71] *id.*, *Proof of a Program FIND*, CACM, v. 14, n. 1, Jan., 1971

[hum89] W. S. Humphrey, *Managing the Software Process*, Addison-Wesley, Reading, Mass., 1989

[6] An '*' indicates that the paper has been reprinted as a chapter in [leh85].

[hum97] *id., Introduction to the Personal Software Process(SM)*, Addison-Wesley, Reading, Mass., 1997

[lac96] A. R. Lacey, *A Dictionary of Philosophy*, 3rd Ed., Routledge, London, 1996, 386 pps

[kah01] G. Kahen *et al*, *System Dynamics Modelling of Software Evolution Processes for Policy Investigation: Approach and Example*, J. of Sys. and Softw., v. 59, 2001, pp. 271-281

[kem99] C. F. Kemerer and S. Slaughter, *An Empirical Approach to Studying Software Evolution*, IEEE Trans. on Softw. Eng., vol. 25, n. 4, July/August 1999, pp. 493 – 509

[kuv94] P. Kuvaja *et al*, *Software Process Assessment and Improvement-The Bootstrap Approach*, Blackwell, 1994

[leh69] *M. M. Lehman, *The Programming Process*, IBM Research Report RC2722M, IBM Research Center, Yorktown Heights, New York, Sept.

[leh74] **id., Programs, Cities, Students—Limits to Growth,* Imp. Col. 1974, Inaug. Lect. Series, Vol.9, 1970-1974, pp. 211 - 229; also in Gries, 1978

[leh78] **id., Laws of Program Evolution-Rules and Tools for Programming Management*, Proc. Infotech State of the Art Conference, Why Software Projects Fail, April 9-11, 1978, pp. 1V1- IV25

[leh79] *id., The Environment of Design Methodology*, Keynote Address, In Proc. Symp. on Formal Design Methodology, Cox TA (ed.). Cambridge, UK, 9-12 Apr. 1979, pp. 17-38, pub. by STL Ltd, Harlow, Essex, UK, 1980.

[leh80] **id., Program Life Cycles and Laws of Software Evolution,* Proc. IEEE Spec. Iss. on Softw. Eng., Sept. 1980, pp. 1060-1076

[leh84a] *id., Program Evolution*, Symposium on Empirical Foundations of Computer and Information Sciences, Georgia Institute of Technology, in J. of Information Proc. and Management, v. 19, n. 1, 19, 38, 1984

[leh84b] *id., A Further Model of Coherent Programming Process*, Proc. Softw. Process Workshop, Egham, Surrey, 6 – 8 Feb. 1984, IEEE Cat. no . 84 CH 2044-6, pp. 27-35

[leh84c] M. M. Lehman, Stenning V and Turski W. M., *Another Look at Software Design Methodology*, ACM SigSoft Softw. Eng. Notes, v. 9, n. 2, pp. 38 - 53, April 1984

[leh85] M. M. Lehman and L. A. Belady, *Program Evolution—Processes of Software Change,* Academic Press, London, 1985

[leh87] M. M. Lehman, *Process Models, Process Programs, Programming Support*, Invited Response to a Keynote Address by L. Osterweil, Proc. 9th ICSE, Monterey, CA., March 30-April 2, 1987, pp. 14-16

[leh91] *id., Software Engineering, the Software Process and Their Support*, IEE Software Engineering J., Spec. Iss. on Softw. Environments and Factories, 6(5), Sept. 1991, pp. 243 – 258,

[leh94] *id., Feedback in the Software Evolution Process* , Keynote Address, CSR Eleventh Annual Workshop on Software Evolution: Models and Metrics, Dublin, 7-9 Sept. 1994, Workshop Proc., Information and Software Technology, sp. is. on Software Maintenance, v. 38, n. 11, 1996, Elsevier, 1996, pp. 681-686

[leh96] *id., Laws of Software Evolution Revisited,* pos. pap., EWSPT96, Oct. 1996, LNCS 1149, Springer Verlag, 1997, pp. 108-124

[leh97] M. M. Lehman *et al*, *Metrics and Laws of Software Evolution - The Nineties View*, Proc. Fourth Int. Symp. on Software Metrics, Metrics 97, Albuquerque, New Mexico, 5-7 Nov. 1997, pp 20-32. Also in K El Eman and N H Madhavji (eds.), Elements of Software Process Assessment and Improvement, IEEE CS Press, 1999, pp. 343-368

[leh00a] M. M. Lehman, *An Approach to a Theory of Software Evolution*, EPSRC Proposal, Case for Support Part 2, Dept. of Comp. ICSTM, Dec. 2000, rev. version Sept.2001

[leh00b] M. M. Lehman and J. F. Ramil, *Towards a Theory of Software Evolution - And its Practical Impact*, inv. talk, Proc. ISPSE 2000, Kanazawa, Japan, Nov 1-2, in Katayama T *et al* (eds.), IEEE Comp. Soc. Press, Los Alamitos, CA, pp. 2 – 11

[leh00c] *id., Software Evolution in the Age of Component Based Software Engineering*, IEE Proc. Softw., sp. Iss. on Component Based Software Engineering, v. 147, n. 6, Dec. 2000, pp. 249 - 255, earlier version as Tech. Rep. 98/8, Imp. Col., London, Jun. 1998

[leh01a] *id., Rules and Tools of Software Evolution Planning, Management and Control*, Annals of Softw. Eng., Spec. Iss. on Softw. Managmt., v. 11., 2001, pp. 15-44

[leh01b] *id., An Approach to a Theory of Software Evolution*, IWPSE 2000, this volume

[leh01c] M. M. Lehman, J. F. Ramil and U. Sandler, *An Approach to Modelling long-term Growth Trends in Software Systems,* ICSM 2001, 6-10 Nov., Florence, Italy, pp. 219 – 228

[leh02] M. M. Lehman and J. F. Ramil, *Software Uncertainty*, Soft-Ware 2002, 1st Intl. Conf. on Computing in an Imperfect World, Belfast, North Ireland, 8-10 April

[lju87] L. Ljung, *System Identification - Theory for the User*, Prentice Hall, Englewood Cliffs, NJ 1987

[luq89] Luqi, *Software Evolution through Rapid Prototyping*, IEEE Computer, v. 22, n. 5, May 1989, pp. 13 – 25

[mar02] J. Marciniak (ed.), *Encyclopaedia of Software Engineering*, 2nd. Edition, Wiley, 2002

[mit01] R. T. Mittermeir, IWPSE 2001, this volume

[nau69] P. Naur and B. Randell (eds.), *Software Engineering*, Report on a Conf. Sponsored by the NATO Sc. Comm., Garmisch, Germany, 7-11 Oct. 1968, Brussels, Sc. Aff. Div., NATO, 1969, 231 pps, http://www.cs.ncl.ac.uk/people/brian.randell/ home.formal/NATO/

[ost87] L. Osterweil, *Software Processes Are Software Too*, Proc. of the 9th Int. Conf. on Softw. Eng., Monterey, CA, March 30-April 2, 1987, IEEE Comp. Soc. Pub. 767, pp. 2-13

[ost97] *id., Software Processes Are Software Too - Revisited: An Invited Talk on the Most Influential Paper of ICSE 9*, Proc. ICSE 19, Boston, May 17-23, 1997, pp. 540-548

[pau93] M. C. Paulk, *et al*, *Capability Maturity Model*, ver. 1.1, IEEE Software, v. 10, n. 4, 1993, pp. 18 - 27

[pgs96] T. M. Pigoski, *Practical Software Maintenance*, Wiley, 1996, pp. 384

[pir88] S. S. Pirzada, *A Statistical Examination of the Evolution of the UNIX System*, PhD Thesis, Imperial College, London, 1988

[pfl98a] S. L. Pfleeger, *Software Engineering – Theory and Practice*, Prentice Hall, 1998

[pfl98b] S. L. Pfleeger, *The Nature of System Change*, IEEE Software, 15, 3, May-June, pp. 87 – 90, 1998

[pot84] C. Potts, *ed., Proceedings of the Software Process Workshop,* Egham, Surrey, Feb., 1984

[raj00] V. T. Rajlich and K. H. Bennett, *A Staged Model for the Software Life Cycle*, Computer, Jul., pp. 66 - 71

[roy70] W. W. Royce, *Managing the Development of Large Software Systems*, IEEE Wescon, Aug. 1970, pp. 1–9

[suc01] G. Succi, J. Paulson and A. Eberlein, *Preliminary Results from an Empirical Study on the Growth of Open Source and Commercial Software Products*, EDSER-3 Wkshop, co-located with ICSE 2001, May 14-15, Toronto

[tay89] R. N. Taylor *et al*, *Foundations for the Arcadia Environment Architecture*, SIGPLAN Notices v. 24, n. 2; Softw. Eng. Symp. on Practical Software Development Environments, spec. iss. Proc. ACM SIGSOF7/SIGPLAN

[tul89] C. Tully, *Representing and Enacting the Software Process*, Proc. 4th Int. Worksh. on the Softw. Proc., Jan. 1989, ACM SIGSOFT Softw. Eng. Notes, June 1989

[tur81] W. M. Turski, *Specification as a Theory with Models in the Computer World and in the Real World*, Infotech State of the Art Report v. 9, n. 6, 1981, pp 363 - 377

[tur87] W. M. Turski and T. S. E. Maibaum, *The Specification of Computer Programs*, Addison-Wesley, Wokingham

[tur96] *id., A Reference Model for the Smooth Growth of Software Systems*, IEEE Trans. SE, v. 22, n. 8, pp. 599 - 600

[tur00] *id., An Essay on Software Engineering at the Turn of the Century*, in T. Maibaum (ed.): Fundamental Approaches to Software Engineering, Proceedings of the Third International Conference FASE 2000. March/April 2000. LNCS 1783, Springer-Verlag, Berlin, pp. 1 – 20

[lam00] A. van Lamsweerde, *Formal Specification: a Roadmap*, in A. Finkelstein (ed.), The Future of Software Engineering, 22nd ICSE, Limerick, Ireland, 2000, ACM Order N. 592000-1, pp. 149-159

[ven99] Vensim 1999, *VENSIM 4 (c) Reference Manual*, Ventana System Inc., Harvard, MA

[wei70] *G. M. Weinberg (1970), *Natural Selection as Applied to Computers and Programs,* General Systems, v. 15

[wer98] P. Wernick and M. M. Lehman (1998), *Software Process White Box Modelling for FEAST/1,* ProSim '98 Workshop, Silver Falls, OR, 23 Jun. 1998. As a rev. version in J. of Sys. and Softw., vol. 46, nos. 2/3, 15 Apr. 1999

[wir71] N. Wirth, *Program Development by Stepwise Refinement,* CACM v. 14, n. 4, April, pp. 221-222

[www01] FEAST projects web site, Dept. of Computing, Imp. College, see http://www.doc.ic.ac.uk/~mml/feast

[zah97] S. Zahran, *Software Process Improvement - Practical Guidelines for Business Success*, SEI Series in Software Engineering, Addison-Wesley, Harlow, England, 1997

[zur67] F. W. Zurcher and B. Randell, *Iterative Multi-Level Modelling - A Methodology for Computer System Design*, IBM Res. Div. Rep. RC-1938, Nov. 1967. Also in Proc. IFIP Congress 1968, Edinburgh, Aug 5 - 10, 1968, pp D-138 - 142

This paper represents a broadening and revision of an article to appear in the Encyclopedia of Software Engineering, 2nd edition, edited by J Marciniak.