Information Processing & Management Vol. 20, No. 1 2, pp. 19–36, 1984
Printed in Great Britain.

0306-4573/84 $3.00 + .00
Pergamon Press Ltd.

PROGRAM EVOLUTION

M. M. LEHMAN
Department of Computing, Imperial College of Science and Technology, London SW7 2BZ, England

Abstract—Following a brief review of the development of the concept of *program evolution*, the paper identifies some of the program attributes that are changed over system life-time in response to evolutionary pressures. It recognises that their evolution may, in part, be due to inadequate specification, design and implementation technologies; the processes by which a system is conceived and implemented, and the primitives in terms of which this is done. Basically, however, evolution is recognised as intrinsic to the very nature of computer application, of computing systems, and, most significantly, of programs.

All systems evolve. The peculiar feature of computing systems is their rate of evolution. Analysis of this phenomenon leads to the identification of levels of evolutionary development; thence to the concept of a *continuous* programming process supported by an *integrated support environment*. One model of such a process is outlined and then decomposed into a sequence of orthogonal elements or steps that can all be described by a common paradigm. The former provides a conceptual framework for the design of integrated support environments, the latter its core concept.

1. PROGRAM EVOLUTION

1.1 *Historical summary*

Program evolution is now widely accepted as a fact of life. The phenomenon was first recognised in the late 1960s as continuing program *growth* [1]. The growth then discussed related to improvement in functional capability. For the sequence of releases of a given system at least, this was assumed to be related to program size as determined by counts of program modules, lines of code or storage requirements.

Collection of relevant data and their interpretation subsequently suggested the concept of *Program Growth Dynamics* [2, 3]. Its refinement led to the realisation that observed phenomena should be interpreted as program *evolution*. This represented more than a change of name. It produced, for example, the hypotheses that were later formulated as *Laws of Program Evolution* [4, 5]. Continuing investigation has given rise to the beginnings of a discipline, *Program Evolution Dynamics*; yielding insight[6–8], practical tools and management guidelines[8–10] and most recently a new view of the programming process itself[11, 12].

It must now be accepted that evolution is, ultimately, not due to shortcomings in current programming processes. It is intrinsic to the very nature of computer usage; computing applications and the systems that implement them. This perception has led to the SPE program classification[8] and thence to the concept of *continuous programming processes* supported by *vertically integrated support environments.*

1.2 *The SPE classification*

In the SPE classification an S-type program or system is *defined* as one for which the *only* criterion of success in its creation is equivalence, in some sense, to a *specification*. The P-type, which falls between the other two, is not considered here. An E-type is one *embedded* in its operational environment, implementing an application *in* that environment, as suggested by Fig. 1.

E-type systems have no intrinsic boundaries. The programs that implement them cannot have permanent and demonstrably *satisfactory* specifications since the *variety* of features that can be built into such systems is unlimited, meaningful permutations unbounded. Selection of features must take into account the perceived need, inter-related properties such as performance and cost and the nature of the operational environment once the system is installed. Moreover the very act of system *installation* changes that environment. And as

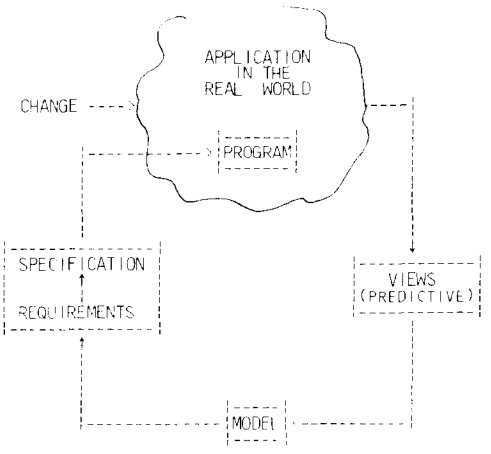

Fig. 1. E-Type programs.

operational experience is gained, perception of the problem and of possible solutions continuously advance, whilst exogenous pressures operate at all times to modify the environment, the problem and solution technology still further.

Now the acceptability of an E-type program relates to satisfaction in *usage* and it follows that the level of satisfaction will change with time. The real test of satisfaction occurs *after installation*, on the basis of system usability, performance and adaptability. But satisfaction of initial specifications is, at most, relevant for a limited period whose duration will depend on the foresight of the design team and the rate of change in the operational environment. Clearly, a specification that defines the *ultimate* E-type system cannot be conceived.

1.3 *Evolutionary traits*

As E-program correctness is determined by user-satisfaction rather than by equivalence to a specification, such equivalence, if it can be demonstrated, is merely a means to an end. It indicates a high likelihood of immediate satisfaction. Continued satisfaction demands continuing change. The system will have to be adapted to a changing environment, changing needs, developing concepts and advancing technologies. The application and the system should evolve. The human effort to achieve this may, however, be withheld. Thus either the system evolves or its effectiveness and that of the application it supports will, inevitably, decline (First Law[4]).

As a system evolves its complexity increases unless specific complexity-control effort is applied[4]. Complexity growth occurs because managerial guidelines seldom include its control as an objective. Instead they tend to focus on deadlines and on the cost-effectiveness of the *local* process or of the resultant system. No real *cost* is attached to structural deterioration for which the penalty lies in the future. Complexity growth is therefore in part a consequence of weaknesses in current process-management practice. Advances in the process and in process-management can overcome this.

When, however, system change reflects extension of an application, it generally implies an increase in the complexity of *what is being done* [13]. It leads, inevitably, to increased system complexity; though such increase may be hidden by the use of higher level (*internally* more complex) primitives; VLSI components for example.

The preceding paragraphs have described evolution as a phenomenon of functional and complexity growth. Current programming practice actually *exploits* evolution to achieve satisfactory levels of system attributes. Correctness, reliability, performance, capacity are all achieved *iteratively*. To the extent that their improvement reflects developing human perception and ambition triggered by use of the system or by change in the operational environment, evolutionary development is inescapable. The lack of an adequate development technology, a supporting engineering science and evaluation calculi are, however, a strong contributing factor; in fact the ultimate cause. The knowledge, understanding and techniques to permit *ab initio* design and construction of a system with, at least, an initially

satisfactory level of all attributes, simply do not exist. Their development, the emergence of a software engineering discipline, could significantly reduce reliance on iterative evolution of such function and quality factors. Note that once the essential evolutionary nature of software is appreciated one need not, indeed should not, distinguish between *initial development* of a system and its subsequent enhancement or extension in a *maintenance* process. Software does not, of itself, deteriorate and so need not be "maintained" in the traditional engineering sense. At most, a system is seen as no longer achieving its full, perhaps newly recognised, potential. Whether one begins with an application concept or with an existing system, all work undertaken to produce a system with more, less or different attributes or characteristics, constitutes evolutionary development. The term *maintenance* is, therefore, inappropriate in the context of software. Use of the term should be abandoned; replaced by *evolution.*

1.4 *Process dependent and intrinsic evolution*

The above summary of some aspects of program evolution suggests that it is, in part, a consequence of the *process* of programming. Thus its rate may perhaps be reduced through the development and application of more advanced design and implementation technologies. It has, however, also been suggested that significant evolutionary pressure arises from the very nature of computer application and therefore of computing systems and programs[8]. *Intrinsic* evolution must be accepted as a fact of life.

With current practice, an average of about 70% of the life-time expenditure on a program is incurred after initial intallation. The main cost of a program is not incurred in its creation but in its subsequent evolution. To achieve an appropriate balance between initial product quality, continuing satisfaction and some desired life-time expenditure distribution, requires improved understanding of both intrinsic and process dependent modes of evolution; that is, a clearer insight into the basic nature and dynamics of the act of creation of a computing application through the development *or modification* of system constituents. The remainder of this paper analyses the evolution processes to lay the foundations for the systematic development of a systematic application and program development process; the *software technology process.*

It should be stressed that the analysis to follow is as relevant to VLSI "hardware" as it is to software. As element numbers per chip increase, the latter will display all the characteristics of complexity, invisibility, evolution and uncertainty that have plagued software for over two decades. The resultant problems add, of course, to those arising from the technology itself. Moreover, as the manufacturing process becomes automated, chip functionality will be defined by its formal inputs as transformed by that process. It may even be a design option whether a chip functional-specification is used to control a manufacturing process or as source code for a program subsequently to be stored in and executed by a simpler chip. VLSI and software technologies can, therefore, be expected to have much in common. Thus in the remainder of this paper "program' is to be interpreted as including both soft and VLSI implementations.

2. SYSTEMS EVOLUTION

It is a truism to assert that all natural and artificial[14] systems evolve. It may therefore be asked why the *dynamics* of program evolution (as distinct from its mechanics) should prove of interest when similar concern has not developed to any significant degree in the study of other systems. The answer to this question is both simple and revealing.

The time scale over which *natural* systems evolve is such that significant change is observable only over many human generations. The natural scientist cannot, therefore, observe change and development *as it occurs.* He operates as an archaeologist and historian, deducing the occurrence, mechanics and nature of evolution from static remains, relics and records of past events.

Artificial systems fall into several classes that vary widely in their characteristics. Consider first *socio-economic* systems such as cities. These too evolve. Noticeable change may occur in a matter of months, but developments that change the structure and character of a city extend over a human generation or more. The sociologist, by and large,

does not monitor continuing change and evolution. He deduces it by comparing his observations with historic records. Because the rate of global change is relatively slow, its *dynamics* are, at most, *deduced* not experienced.

Engineering artifacts evolve more rapidly. The motor car and the aeroplane, for example, have each seen eight to ten generations in as many decades. New models, incorporating minor improvements are released periodically. Modifications may be introduced at any time and even retro-fitted to instances already operational. But the cost of total redesign and retooling is such that an essentially new system only appears once in ten years or so. Thus during his career the average aeronautical engineer, for example, will be involved with no more than three or four generations of aircraft. And if he experiences that many, he does so as an apprentice, as a mature engineer and as a senior manager respectively, say. His viewpoint, involvement and responsibility is different for each generation. He experiences and views successive generations as a sequence of static instances, albeit of ever more advanced characteristics. Finally, consider *programs*. These constitute "the fruit fly" of artificial systems, undergoing continuing change and rapid evolution. The reasons are manifold[8]. The frequency and speed with which programs are executed, draws almost immediate attention to any shortcomings or mismatch and to developing or emerging opportunities. This leads to a constant stream of proposals for enhancements. That is, change proposals emerge rapidly because of the intimate coupling between the computing system in execution, operational personnel and the application environment. It takes much bitter experience to demonstrate the cost of the effort subsequently required to implement the associated changes. Once made, proposals for change are too easily accepted since their implementation involves little physical effort. The intellectual effort required is, in general, under-estimated and under-rated.

As a consequence, it has been universal experience that software systems evolve at rates reflected by the release of new versions at intervals ranging from less than one month to some two years. Mini-releases containing corrections and minor modifications may even be interspersed between these. The user, the salesman, executives and developers each experience the sequence of releases as a stream. At every stage of their career they are *actively* exposed to a sequence of system releases. They *experience* system evolution as a *dynamic* process influenced by and in turn influencing the environments in which it exists. The *Program Evolution Dynamics* studies of the last ten years have shown that the dynamics of that process may be modelled; *the models reflecting the discipline that underlies and regulates human society and the effort that implements change.*

3. THE CURRENT PROGRAMMING PROCESS AND THE IDEAL

The software engineering and programming processes as currently practiced have themselves evolved over some three decades. As the state of the art in electronic computing advanced, methods, techniques and tools were conceived, developed, implemented and used to solve *specific* problems as these arose in *specific* environments. Together these form an ever-growing set of process primitives, from which total processes have been created by *ad hoc* association. These serve the needs of each individual production environment; additional elements being created as necessary to achieve local efficiency, effectiveness and cost-effectiveness.

Current programming processes developed during the formative years of a new technology. They had therefore to be assembled by *ad-hoc* association of such methods, procedures and tools as were available at any given time. Such bottom-up development inevitably leads, for example, to process discontinuities and to local rather than global optimisation. Once an adequate primitive set of methods, techniques, procedures and tools exists, one may, however, *design* a process top-down by decomposition and successive refinement guided by whatever criteria one chooses to adopt. One may, for example, try to synthesize a process using only available primitives. Alternatively one may seek to produce an *ideal* process which is then approached as closely as possible. Primitives may have to be defined and developed to fit needs not otherwise satisfiable, or to achieve significant gains in product quality or in process effectiveness or responsiveness. The latter approach has been adopted here. The paper seeks to identify an ideal process to serve as a base from which practical processes may subsequently be constructed.

4. LEVELS OF EVOLUTION

4.1 *The feedback controlled evolutionary system*

It has been argued that changes in the operational environment constitute a significant source of evolutionary pressure. In part such changes are due to evolution of the environment itself in response to forces unrelated to the application addressed by the system. In part they are due to experience with system operation which, of itself, suggests corrections and enhancements to, or enlargement of the scope of, the application. That is, installation and operation of a system modifies the operational environment. E-type programs, as they have been termed, therefore include an implicit model of their own operation. They must be designed from a viewpoint of the application universe as it *will be* when the system is installed. The process of specifying and designing them is *essentially* predictive. It must be based on foreseeing the new perceptions of application need and system potential that will develop as a consequence of system usage. However good that *foresight*, it must be imprecise[15] and pressure for corrective adaptation will inevitably develop.

Computing systems, however, are not *self*-adaptive. Selection and management of change is the responsibility of managers, who may well resist pressures and opportunities. But if a system is not adapted to its evolving conceptual and physical environment it becomes ever less satisfactory in the users' eyes. Ultimately it must be abandoned and replaced. Thus whether change is implemented continuously, by successive modification, change upon change upon change, or whether it is revolutionary, an outdated system being entirely replaced, or whether these modes alternate, the system will evolve on the basis of feedback provided, for example, by accumulated learning experience.

The system comprising the application and computing systems in their operational and system-implementation environments constitutes a multi-loop feedback system with both change reinforcing (positive) and change opposing (negative) feedback paths. At worst, the feedback leads to instability; always to continuing pressure for change. The rate of evolution, even though subject to management decision, will depend on the characteristics of the feedback paths. With current practice, four major paths and hence four levels of evolution may be identified. It will be shown that, of these, the two higher levels (slower rates) are largely intrinsic and unavoidable, though effective prognosis and prediction can reduce the rate of change. The two lower levels are largely process dependent. The development of improved, systematic, software engineering practices based on full understanding of why and how computing applications and software evolve, can minimise the evolutionary element.

4.2 *Evolution over generations*

The highest of the four feedback levels, that currently drive program evolution, arises from pressures that reflect changes in the operational and technological environment. The mechanism is similar to that which drives the evolution of socio-economic systems and engineering artifacts. As suggested in section 2 these evolve at a rate expressible in terms of decades or human generations. The increasing use of computers may tend to accelerate the process, but will be counterbalanced by an increasing need for stability and increasing system malleability. Relevant time scales are therefore unlikely to change dramatically. System life-time will continue to span ten to twenty years.

As indicated in Section 2, evolution of a system over successive generations also covers successive generations of the personnel associated with that system. The *individual* thus experiences evolution as a *static* phenomenon recognisable from separate instances of the system. Its dynamics are only observable by the historian. The impact of this mode of evolution on process technology is therefore minimal.

4.3 *Evolution through successive releases*

During the life-time of each generation, the program *release*, at present, provides a mechanism for the controlled implementation of changes and their transmission to many users. That mechanism has been evolved and refined to a form peculiar to the software industry. Its special nature is due to the fact that software requires intellectual, rather than physical, effort to change. Software rleases are therefore created by modification and

change to the implementation itself, that is the code and the documentation, rather than by the creation of new instances as is the case for other artificial systems. A release may consist of a single change or of a number of unrelated corrections, enhancements and additions. Whatever the mix, observation suggests that the average work content of a sequence of releases, stabilises to a constant level. This consequence of the feedback nature of system evolution is linked to the effort required for each involved individual to regain and retain familiarity with the system[7].

When setting release content and interval objectives, managers can apply alternative strategies[8]. But increasing societal dependence on computers implies a need for fast response to error reports and design deficiencies. In the early operational life of a system, release intervals of order one month are common. As the system ages, complexity and complexity control effort increase and with average release content constant[7], the release interval eventually stretches to two or three years. The determinants include the work to be achieved, user resistance to installation of a new system and delays in their mastering and appraising its new characteristics. In any event, the loop-delay in release-based evolution is conveniently expressed in months.

This second level of the current modes of evolution has been extensively studied, measured and modelled[8]. It is not further considered in the present paper.

4.4 *Decimal or sub-releases*

Sub-releases are sometimes interposed between main releases to achieve fast response for the fixing of minor faults or blemishes or to provide urgently desired enhancements. This practice has some impact on the evolution dynamics of the system to which it is applied and to the parameters of the resultant process. The latter is, however, not qualitatively different to one in which releases of this category are not used. Nor is a sub-release sequence over long enough to define a coherent sequence of evolutionary levels. Sub-release may therefore, in general, be treated as part of the release process.

4.5 *Developmental evolution*

4.5.1 *The ideal process.* Consider now the *process* whereby an individual release, sub-release or isolated change is developed from conception to eventual operation. Present industrial practice is the outcome of *ad hoc* evolution driven by expanding demand for computer applications, and for the programs to implement them. Its limitations provide the motivation and justification for seeking to unify and advance software technology so as to achieve an economical process that can be *planned* and *controlled.*

At initiation of an E-type development project, the picture of *what* is to be achieved and *how*, is, at best, fuzzy. Specification and design evolve iteratively as a consequence of feedback via various paths. The latter are often *ad hoc* and pooly defined, making analysis difficult if not impossible. To be amenable to analysis, the process should have well defined structure. This may be obtained by first identifying an *ideal* process; a contiguous, non-iterative, sequence of orthogonal sub-processes that jointly represent a set of sub-transformations necessary and sufficient for the transformation of a computer application concept into an operational system. An analysis in terms of current process concepts, exemplified by Fig. 2(a) that yields such a process, has been given elsewhere[11, 12]. The time required for each of the activities indicated will typically be in the order of weeks and the process is third in the schema proposed in Section 2.1.

The programming process, illustrated in Fig. 2(a), is expressed in terms of activities that are widely recognised and pursued in the current industrial process. More generally, one should view the process as a sequence of linguistic transformations[16] with each model a representation of both the problem to be solved (or more precisely its formal specification), and of the system to be constructed. This is illustrated by Fig. 2(b).

From either viewpoint the process may be described as the *transformation* of a computer application *concept* into an *operational system* and its continuing *adaptation* to evolution of the operational environment. This transformation is complex. To achieve a practical process it must be decomposed into a series of sub-transformations. These define a structure of sub-processes, execution of which defines a system implementing the

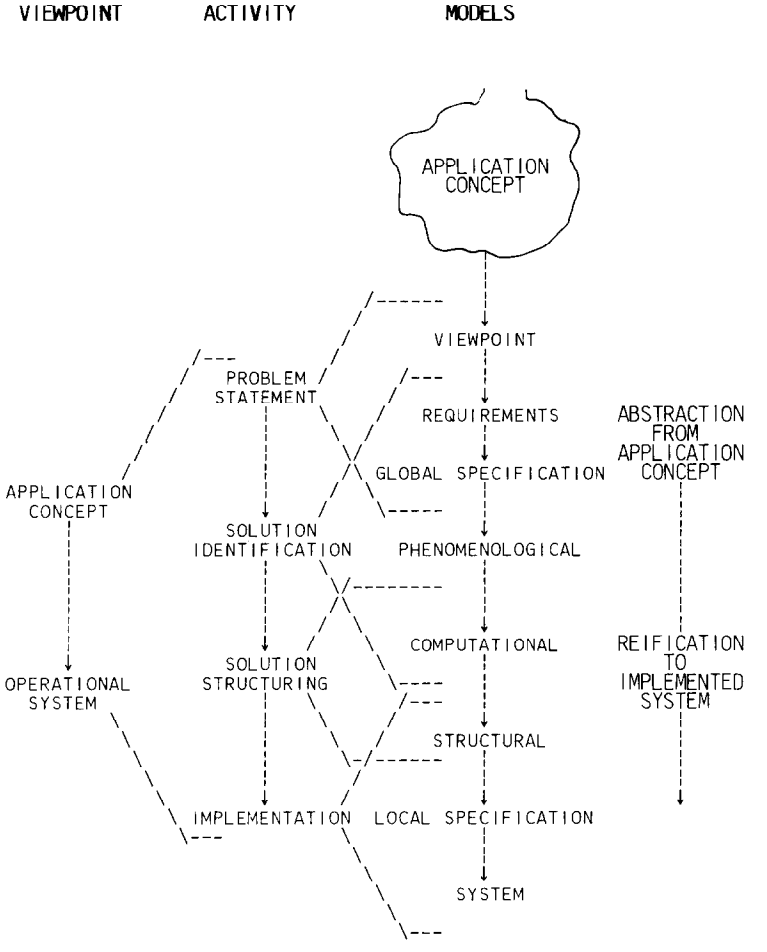

Fig. 2(a). An ideal process in terms of current process concepts.

application concept. The system is represented as "*a model of a model of..... a model of a computer application concept in its operational environment*". Each of the constituent models represents an abstraction of both the application concept *from* which it derives *and* of the system *to* which it is advancing by a process of reification. The models are *double* abstractions, a fact that has important bearing on the concept of a continuous, that is coherent, total software process supported by an integrated support environment.

The above reflects the dynamic view of the process. Its static counterpart regards each step of the development process as producing a theory for which the neighbouring steps provide models. In particular, the real world at one extreme and the operational system at the other are each models of the theories provided by the intermediate steps[17].

Note that the term "ideal" must be understood in the sense of the thermo-dynamicist's "ideal cycle", in that it is believed not to be attainable in practice. Exogenous change in the operational environment and the consequent pressure for software adaptation is essentially unpredictable and cannot be accommodated without iteration. The following sections will suggest that iteration is also inescapable in specific software development. For example, total understanding of a problem and creative development of the best, in some sense, design for a computing-based solution can, in general, only be achieved iteratively[15]. A linear process with only orthogonal activity cannot be achieved

4.5.2 *Iteration.* Even the linear ideal process suggested by Fig. 2 is not unique. Any orthogonal set of steps that together cover the necessary and sufficient activity required to produce the target system serves the same purpose. But the sequential process is an idealisation that cannot actually be achieved. Thus the detailed role of each step is not

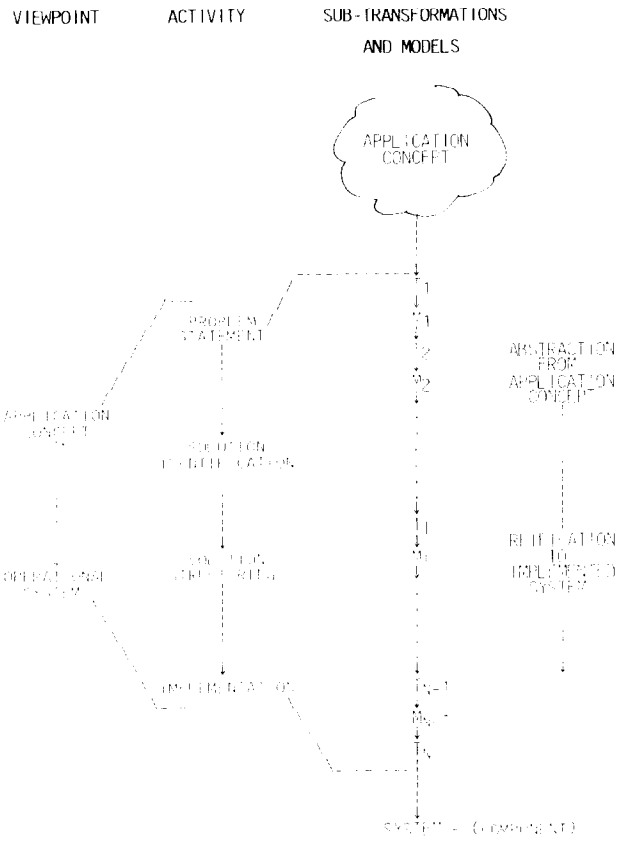

Fig. 2(b). An abstract ideal process.

significant. It is the structure that provides the starting point for the development of a practical process.

The desire for a programming process that is linear is not new. Such an objective has been implicit in the search for improved programming methods since programmed computers were first invented. The development of high-level languages, for example, could be interpreted in those terms. The trend towards formal specification and the search for techniques that automate the transformation required to convert a program specification into an operational program[18, 19] contains similar implicit objectives.

The specification of any system embedded in an application environment is, however, inherently incomplete. A linear process without evaluation based human decision is, thus at best, only possible when the *structural model* has been defined; that is, when the evolving E-type design has been decomposed into a structure of S-type elements. It has been hypothesised that the latter decomposition is always possible[8]; in fact a programmer should never be required to commence a *programming* task until elements to be created have been fully specified. Nevertheless, the fact that a selection must be made between meaningful alternatives indicates that, to some extent at least, iteration must be used. In fact, with presently available software process technology, it will occur at three distinct levels in any practical implementation of processes based on the ideal illustrated by Fig. 2. These levels correspond to the three lowest of the levels of evolution identified above. From the outlines that follow it will be seen that they reflect the organisational feedback paths involved in the various activities.

Section 4.3 discussed system evolution via the release mechanism. The pressures that drive this process, and the justification of any changes undertaken, arise from continuing information exchange amongst technical personnel, from similar exchanges with users and from exogenous change. This may be reflected in a model derived from that of Fig. 2(a), by means of a feedback connection over the entire process as shown in Fig. 3 by the outer

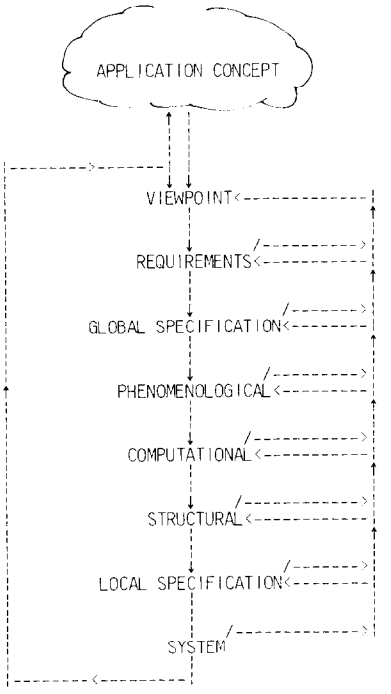

Fig. 3. Iteration on the ideal process.

loop. The information fed back reflects the experience and insight accumulated during development, implementation, installation and *usage* of the system.

No calculus is readily available to support linear progression from model to model over the steps that derive the succession of models that collectively capture and embody the output of the software process in the form of evolving design detail. Nor is there a calculus that permits forward evaluation of a design over that process. *Validation* of design decisions at each step, based on assumptions about the remaining process and about the primitives available to implementors, is ad hoc[20]. In the absence of adequate methods, errors, weaknesses and omissions are uncovered only as the process proceeds. Thus every now and again earlier decisions must be reviewed; the models that embody them revised. Such review is currently, generally, casual, little attempt being made to update any but the most recent documentation. In the future, increasing emphasis will have to be placed on explicit review of all models affected by a change, by explicit iteration over the feedback oaths encompassing one or more steps, as indicated in Fig. 3.

The third level of iteration *occurs* in that portion of each sub-transformation concerned with design of the new features to be added to the model at each stage. It arises because no analytical design method is available. An iterative approach based on intuitive trial and evaluation must therefore be used. This is discussed further in Section 4.8.

The above discussion has presented iterative design and implementation, the evolutionary programming process, as the consequence of inadequate design theory. An adequate theory is essential if a technology to cover the total software process is ever to be achieved. There exists, however, a fundamental dilemma that complicates the development of such a theory and may well frustrate it. It stems from the fact that system design is two dimensional; creative design must be explored in, at least two directions.

On the basis of Figs. 2 and 3, one may hypothesise processes in which each model in the sequence is completed before proceeding to the next. A specification, for example, can be considered as being developed by a question and answer process that produces a tree-like structure. When the tree has been developed down to leaves about which all questions have been directly and uniquely answered, specifically or by "don't care", the specification is complete and provides the input to the next transformation. Were such a process possible at each stage of the overall process, if the human questioners and decision

takers were all-wise, then this procedure could define a sequential macro-process, but with iterative loops for tree development.

Alternatively one may visualise a procedure that limits the descent at each step to one level of refinement; it being recognised that the optimum decomposition may well be a function of decisions still to be taken in future steps. One proceeds, therefore, along several steps of the transformation sequence, before returning to advance the design further by additional decomposition.

4.5.3 *A practical process.* At the present time there does not exist a method of programming based exclusively on one or other of these alternative approaches. Whether either can yield such a method remains an open question[15]. Strong grounds exist, however, for believing that they do not; that a linear, non-iterative, process cannot be achieved. Any practical process must represent a compromise between the two extreme methods, a development from the abstract ideal including, at the very least, several levels of iteration.

Which direction to pursue, and how far to proceed in the refinement process at any given stage of the development, is probably one of the most difficult and critical decisions of the software design process. The software engineer must decide when to iterate, to which model in the sequence of process models to return and which of many alternative design paths to explore. The decisions taken determine the future course of the process. *Process design* is thus itself a critical *process activity*, an activity that cannot be concentrated in an initial, or any, process step. It must be accepted as an ongoing consideration distributed over the entire life of the software.

In summary, the total programming process is inherently an iteration of iterative steps. Their execution implements evolutionary development, progress towards the final goal by refinement and by resolution of imprecision or incompleteness in original concepts and in individual design and implementation decisions. As progress is made through the process, the full set of models which collectively constitute the system model must be maintained consistent each in itself and with one another[13].

4.6 *The role of the ideal-process concept in a theory of program evolution*

Despite the fact that the "ideal process" is almost certainly not attainable in practice, the search for an integrated, maximally mechanised (tool supported), software development process can still benefit from the concept; identification of its structure, components and properties. The concept constitutes a useful abstraction to aid formulation of a *theory of program evolution.* Such a theory is regarded as an essential precursor to the establishment of a *coherent process* for software development. The coherent process is, in its turn, vital if the much used term of "integrated programming support environment" is to assume real meaning; if such systems are to be constructed. It should be noted that the term "development" is here used in its fullest sense to include continuing evolution of the software to adapt it to the needs and opportunites of a changing operational environment. *Programs must not only be good in the first place, they must be adapted to remain good despite exogenous changes.*

Is development of a theory of program evolution a meaningful objective? The mere fact that the programming process consists of a series of steps that "yield development via a series of changes", satisfies both Oxford[21] and Webster[22] definitions of "evolution". But is this all? Can similarities with other evolutionary phenomena be identified and prove helpful in achieving understanding of that process or, conversely, of the evolution of other forms of complex systems?.

Consider the steps of an E-type application development. These involve selection between alternatives, *natural selection* with *survival of the best.* The process relies heavily on human perception for injection of the consequences of exogenous change analogous to *mutation. Adaptation* to environmental changes plays a key role. These facts suggest that software evolution may have much in common with that occurring in other artificial and in biological systems. The significant difference may lie, primarily, in its reliance on iteration rather than on parallel development and on the rate of evolution.

In any event one may conclude that the overall theoretical structure, regarded as key to significant advances in the emergence of a discipline of software engineering, may be developed from the postulate of an "ideal process" and its instantiation as in Fig. 2.

Its practical approximations will be based on iteration. However, with advances in technology, the dependence on iterative development for the lower levels of the total process will decrease as analytic design and validation techniques are developed for guiding and controlling the selection process. Examination of the process step, lowest in the levels, of the evolutionary hierarchy will further clarify this issue and provide additional insight into program evolution.

4.7 *The step paradigm*

4.7.1 *Its core.* A recent publication[12] presented a paradigm describing the activity required in each of the steps that together realise the release development process. After a brief discussion of the paradigm, the present paper will isolate its evolutionary component to determine the degree to which such evolution is intrinsic or technology dependent.

The core activity of an elementary step is illustrated by Fig. 4. At the highest level of abstraction it represents a *transformation* of an *input* (model) into an *output* (model). The transformation may implement changes in representation but is primarily directed at achieving some refinement to advance transition to the object system. The process of notational change, restructuring and refining of the input model, in ways to be discussed in Section 4.7.3, is termed *design.*

Input and output models also provide a means of communication between designers and between them and their clients. The models must therefore be accessible in a structure and notation that makes them comprehendable to humans, who have to base decisions on their understanding and appreciation of them. A suitable representation for human comprehension may, however, not be the most appropriate for optimum decomposition and refinement. At a first level of decomposition the core of the paradigm, must therefore be shown as a sequence of three sub-steps. The first transforms a communication oriented representation to an internal, manipulation oriented, form. The third, if needed, produces a form appropriate for the output interface. In between there is the *design* step.

Design is achieved by the application of human judgement and decision, on the basis of defined immediate objectives and long-range goals. It must consider all potential inputs to and the desired output from the current step and the total process, the constructs or primitive elements available for the current step, the nature and power of the remaining process and the primitives available to it.

Structuring at each step facilitates intellectual mastery of the total complex. If interfaces and interconnections between the identified parts can be completely specified, it also permits division of further design activity, amongst participants or groups, for that step or for the remaining process. The potential activity split is indicated in Fig. 4 by the dotted lines out of the design box.

The preceding discussion has indicated the *role* of the design step. Its own design, to produce a practical process in a specific context for example, requires systematic decomposition, structuring and refinement of the basic concept; that expressed above for example. A preliminary analysis aimed at determining a lower level paradigm, is outlined in Section 4.7.3. Discussion of the step paradigm must, however, first be completed.

4.7.2 *The complete paradigm.* In the absence of precise design calculi, the activity outlined by Fig. 4 must be supported by activities that address questions such as, "are we building the system right?", and "are we building the right system?"[23]. Ideally, each step of the design process must, in the most general sense, be validated. The complete first level description of the step paradigm, illustrated by Fig. 5, indicates how this might be achieved.

Each step-transformation produces a model that, if satisfactory, represents the input to the next step of the development process. What is "satisfactory" in this context? The input and output transformations are purely representational in nature. They involve no

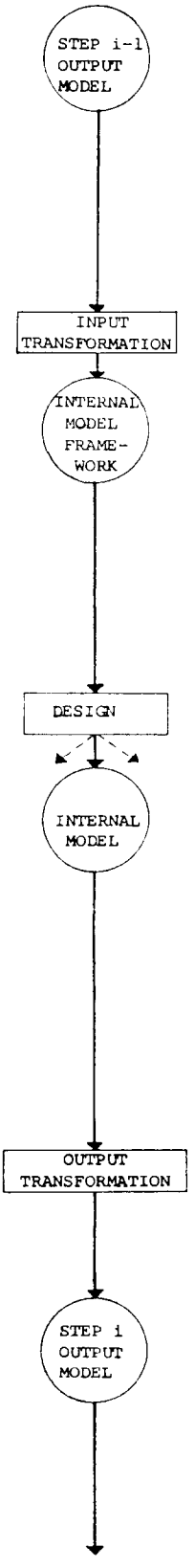

Fig. 4. The step paradigm core.

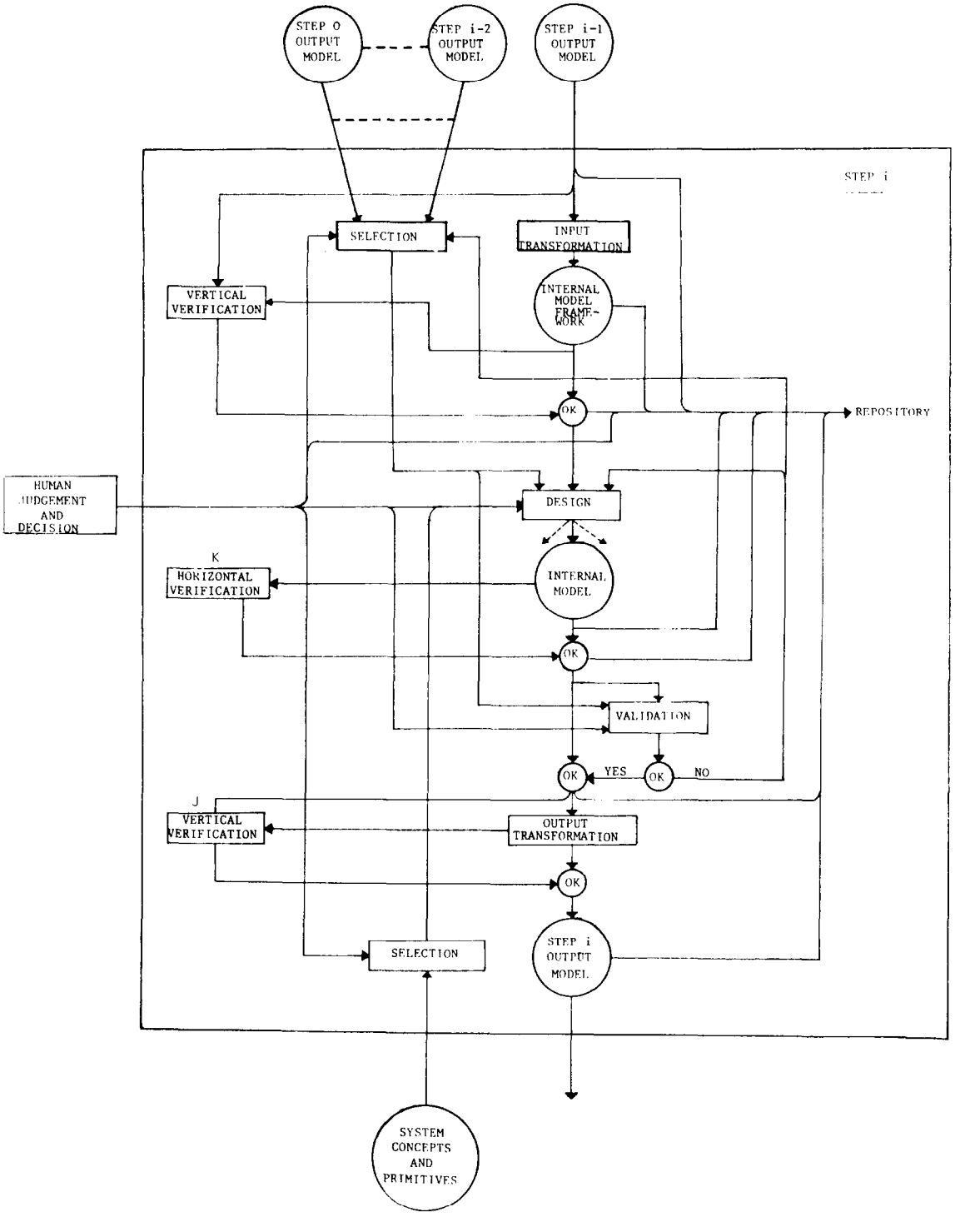

Fig. 5. The full step paradigm.

changes arising from design decisions. Hence the requirement is equivalence, in some sense, between each pair of inputs and outputs. Demonstration of such equivalence is, here, termed *vertical verification*. One could achieve it by *proving* equivalence, by a demonstration that each constituent sub-step of the transformation is correct (*constructive correctness* [24]) or by demonstrating once and for all that the *transformer is correct*, all in the context of the current step and its primitives.

On completing a design step it must be shown that the resultant model is, itself, consistent and that it is complete in relation to the features that were to have been added or the problems that were to have been resolved in the current step. Such a demonstration is, here, termed *horizontal verification*.

When an acceptable model, in this sense has been achieved, it should be determined whether that model, *as an intermediate step in the total design process*, is *likely* to lead to an acceptable, or even optimum, operational system. In the absence of appropriate calculii, this judgement is imprecise, but should be based on an assessment of the current model, on where additional detail is required and how it might be developed, on implications of precursor models as to further generalisations or features that must be achieved, on the capability of the remaining process and on the primitives available for implementation of the remaining steps and the final system. The process of developing this assessment is termed *validation*. In the absence of a methodology that leads to a satisfactory design in a single step, the design step becomes iterative. Each iteration requires validation to determine completion of the sub-process. Horizontal verification is, of course, also desirable, obligatory in fact, in each iteration.

The above discussion has outlined the direct, first-level, elements of a paradigm covering the many steps of a practical software process. Though the issue cannot be explored here, it will be self-evident that the process as described, demands simple access to extensive records that contain the state and histories of the various models and of the processes that produced them. Perceptions and decisions that underlie design decisions, and the reasoning that produced them, must also be preserved. Finally information relevant to the planning, management and evaluation of the entire process and its relationships to the environments in which it is executed, must also be recorded. The whole of this requirement is indicated in Fig. 5 by the lines converging on "repository".

4.7.3 *A preliminary design sub-step paradigm.* The present paper is intended to address the issue of evolution in the programming process and to develop, at least, the outlines of a theory of program evolution. It is therefore appropriate to pursue further refinement of the step paradigm only to the extent that identification of further detail assists its development. The specific objective must be to pin down and to clarify more precisely the origin of true evolution, as distinct from technology dependent evolution, in the design process.

The discussion of earlier sections presented the process of program design or development as a sequence of transformations. This view is particularly appropriate when considering the mechanistic aspects of the process. However, in so far as the transformations include a creative element that requires human involvement, it is more appropriate to describe the process as one of refinement, as in section 4.7.1. The development of a lower level design paradigm may therefore be based on a decomposition and refinement of the *process of refinement*.

In his original paper [25], Wirth defined refinement as the *addition of detail*. In his introduction he states that "... the program is refined in a sequence of refinement steps. In each step, one or several instructions of the given program are decomposed into more detailed instructions". In his conclusion this is expressed as "In each step a given task is broken up into a number of subtasks". As Wirth clearly recognised, this process requires human decision. The way in which an element is decomposed will affect, at least some, attributes of the final product of the refinement process.

Wirth also discussed the need to define and *structure* associated data. More generally, structuring is, in fact, an integral part of the refinement process. As detail is added, the internal elemental structure will expand in a way that is dependent both on the original or higher level structure and on the process of refinement. At some stage, it may be advantageous to restructure the emerging element, by associating its primitives in a

different pattern without changing its internal semantics. Such restructuring may, for example, improve representational clarity and pave the way for further refinement, though it does not add detail in the Wirthian sense. Whatever the reason for undertaking it, refinement by restructuring is a further element of the design process.

As the process is followed, it will occasionally be convenient to focus on one aspect of the emerging design, possibly even to the exclusion of others. Aspects that are then temporarily ignored will subsequently have to be re-introduced. This too is an element of the refinement process.

The above forms of refinement involve evolution because no calculus exists to implement them without iteration. Their evolutionary content is strictly process dependent. However, as refinement proceeds, new insights will inevitably develop into the nature of the application, the properties of possible solutions and potential extensions of both. This developing insight may lead to actual change of one or more of the models, or even to change of the original application concept. It is a mode of refinement that represents evolution in the fullest sense of the term.

4.8 *Evolution in the process step*

The previous section outlined a standard paradigm that describes the stepped activities that, together, achieve the transformation of an application concept into an operational system. The key elements of the constituent steps are the transformations that, jointly, achieve abstraction of the application concept and its reification into an operational system. They reflect the *design* activity that successively refines individual models to achieve the *incremental development* that the step is to provide.

After, at most, a representational transformation of the output of the previous stage, each input model is modified and extended by refinement to add detail, modify structure or impose change. This yields a new model sequence, with consistent and compatible elements, that represents progress towards the target system. The process is guided by the objectives of the current step and by knowledge of the remaining development process and the primitives in terms of which it is to be accomplished. It involves creative thinking; judgements and decisions based on knowledge, understanding, experience and intuition.

It is the *design* sub-step which drives system evolution. Refinement decisions impose verification obligations. They must also include consideration of the remaining process and its primitives, that is evaluated by *validation* procedures. Ideally, these should provide the best assessment possible, *at the current stage*, of the degree of satisfaction that can be expected from the system *likely* to emerge. Because of the imprecise nature of current technology, progress is iterative. That is, there exists, in general, no precise method for selecting the form or content of refinement to be applied. There is no systematic linear technique for selection between alternative structures, algorithms and primitives. In the absence of appropriate methods, or if insight develops in a way that suggests benefit can be derived by a change propagated across the sequence of models, iterative refinement must be applied possibly across several steps.

In summary, where formal or analytic techniques exist, a single application of the design sub-step can produce an acceptable output model. In the absence of such techniques, the design will evolve over several passes through a design and validation loop. Where validation methods are adequate, iteration and lowest level evolution may be confined to the internal step concerned. Where they are insufficiently refined or when *changes* affecting higher level models are introduced so that true evolution occurs, iteration must span several steps or extend evolution over two or more releases.

In any event, evolution clearly features at the step level of the programming process. At this lowest level the real times involved are of the order of days or weeks. It represents a lower level of evolution than that of the release-development process. At both these levels, however, evolution arises from processes tending to one or other of the alternatives identified at the end of section 4.5.2. There is some hope that in the future, development of adequate design and validation techniques will permit refinement of the design process to *reduce* reliance on evolution at the lowest levels. Whether they can ever be made sufficiently precise to confine evolution to the Release Sequence and Generation levels remains to be seen.

5. STRUCTURE OF THE EVOLUTION PROCESS

The above discussion has briefly introduced a concept of hierarchical evolution and identified natural levels of a process implementing it. Table 1 summarises facts relevant to an associated theory of software evolution.

Table 1. Levels of evolution

Level	Involvement	Feedback via	Time Unit	Mainly
Step-local Design	Individual	Designer's Perception	Days to Weeks	Process Dependent
Release	Project Group	Subsequent Process Step	Weeks to Months	Process Dependent
Release Sequence	Organisation	Users and Developers	Months to Years	Intrinsic
Generations	Society at Large	Real World Environment	Years to Decades	Intrinsic

Urgency may, occasionally, force an *ad hoc* system change to be implemented outside the established process. In general, however, system evolution will be constrained, so that change is achieved within processes set up at each level. Table 1 suggests orderly progression, a property that is highly desirable if a related theory is to be palatable. The pattern has a simple interpretation. It reflects the fact that evolution is achieved by human activity in a societal framework. Intervals that represent natural time constants in that framework, in the life and activity of individuals, groups, organisations and society at large, must have an impact on the programming process. The paper demonstrates that they appear naturally from an analysis of that process, both current and abstract. Their appearance is a hopeful sign that that analysis may bear further fruit. What are the immediate implications, particularly on the process and its support?

6. SOFTWARE PROCESS SUPPORT

One of the important conclusions of this study follows from the demonstration that the software process is a phenomenon that can be studied systematically in the context of the environments within which it is pursued. This view has always been implicit in Belady and Lehman's Evolution Dynamics studies. It reflects the tight linkage with the society within and for which computers operate. Many of its properties are a consequence of that relationship. We ignore the relationship at our peril[7, 8, 10].

The notion of software evolution as a partially natural phenomenon leads to the concept of an *ideal continuous process* extending over the system life-cycle. During the last twenty years, *ad hoc* processes have evolved, assembled from equally *ad hoc* methods and tools. Each process has been adapted to the development environment in which it is to operate. What is now known and available as a result of this process evolution provides a rich set of primitives. These, in conjunction with the understanding achieved, yield the target implementation primitives for a top-down analysis that can determine a structured process approaching the ideal; and a basis for its practical implementation.

It is now widely realised that an effective programming process must be supported by an adequate set of tools. Such a tool kit must be coherent and integrated. The coherent view of the total programming process based on a theory of program evolution, provides a conceptual framework for the development and implementation of a methodology, a set of compatable methods and an integrated tool kit for their support. Space constraints prevent further exploration here. Preliminary discussion may be found with rapidly increasing frequency in the literature[7, 8, 11, 12, 26–30]. There is clearly much to be done. The present paper together with the referenced literature provides the concepts and a systematic and unified base for such an effort. The practical implementation of these concepts could rapidly follow.

Acknowledgements—The author has developed the concepts presented, over many years in collaboration with a number of colleagues. The continuing association with L. A. Belady is well known. More recently, Professor W. M. Turski has acted as a constructive critic, contributing insight, concepts and refinements. The author is also deeply endebted to Drs. G. Benyon Tinker, P. G. Harrison, C. Potts and V. Stenning. Acknowledgement is also due to ERO and its Director of Information Sciences, G. M. Sokol for continuing encouragement and support including that under contract number DAJA-37-80-C0011.

REFERENCES

[1] M. M. LEHMAN, *The programming process*, IBM Res. Rep. RC 2722, Dec. 1969, 47p.

[2] L. A. BELADY and M. M. LEHMAN, *Programming systems dynamics or the meta-dynamics of systems in maintenance and growth*, p. 30. IBM Res. Rep. RC 3546, IBM Res. Div., Yorktown Heights, NY 10598, Sept. (1971).

[3] L. A. BELADY and M. M. LEHMAN, An introduction to growth dynamics. Proc. of the Conf. on Stat. Comp. Perf. Eval., Brown Univ., 1971, *Statistical Computer Performance Evaluation*, pp. 503–512. Academic Press, New York (1972).

[4] M. M. LEHMAN, Programs, cities and students—limits to growth?, Inaugural Lect. 14 May 1974, *ICST Inaug. Lect. Ser.*, Vol. 9, 1970–1974; *Programming Methodology*, (Edited by D. GRIES), pp. 42–69. Springer Verlag, New York (1979).

[5] L. A. BELADY and M. M. LEHMAN, A model of large program development. *IBM Systems J.* 1976, **15**(3), 225–252.

[6] L. A. BELADY and M. M. LEHMAN, Characteristics of large systems. *Research Directions in Software Technology*, (Edited by P. Wegner), Part 1, Chap. 3. pp. 106–142. Sponsored by the Tri-Services Commission of the DoD, MIT Press, Cambridge, Mass. 1979.

[7] M. M. LEHMAN, On understanding laws, evolution and conservation in the large-program life cycle. *J. Systems and Software* 1980, **1**(3), 213–222.

[8] M. M. LEHMAN, Programs, life cycles and laws of program evolution. *Proc. IEEE* p. 1060–1076, **68**(9), 1980, Special Issue on Software Engineering, and with more detail as, Programs, programming and the software lifecycle. *System Design*, pp. 262–291. Infotech State of the Art Report, ser. 6, no. 9, Pergamon Infotech Ltd., Maidenhead, (1981).

[9] M. M. LEHMAN and F. A. PARR, Program evolution and its impact on software engineering. *Proc. 2nd Int. Conf. on Software Engineering*, pp. 350–355. San Francisco, IEEE Cat. No. 76CH-1125-4C (1976).

[10] M. M. LEHMAN, Laws of software evolution—rules and tools for programming management. *Why Software Projects Fail, Proc. Infotech State of the Art Conf.*, pp. 11/1–11/25. 9–11 April 1978.

[11] M. M. LEHMAN, The environment of program development and maintenance—programs, programming and programming support. *Systems Architecture, Proc. of the Sixth ACM European Regional Conf., ICS'* 81, pp. 273–284. ICS Science and Technology Press, Guildford, March (1981).

[12] M. M. LEHMAN, Programming productivity—a life cycle concept. *Proc. Comp Con*81, *productivity an Urgent Priority*, pp. 232–241. IEEE Cat. No. 81 CH-1702-0, Sept. (1981).

[13] G. BENYON TINKER, P. G. HARRISON and M. M. LEHMAN, Program Complexity, Imp. Col. Res. Rep. 83/?

[14] H. A. SIMON, *The Sciences of the Artificial*. M.I.T. Press, Cambridge, Mass, (1969), 123 p.

[15] M. M. LEHMAN, Human thought and action as an ingredient of system behaviour. *Encyclopaedia of Ignorance* (Edited by R. DUNCAN and M. WESTON-SMITH), pp. 347–354. Pergamon Press, London (1977).

[16] M. M. LEHMAN, V. STENNING and W. M. TURSKI, *Another Look at Software Design Methodology*, Imperial College, Dept. of Comp. Res. Rep. 83/13, 25 p. June (1983).

[17] W. M. TURSKI, Specification as a theory with models in the computer world and in the real world. *Infotech State of the Art Report 'System Design*, Ser. 9, No. 6, pp. 363–377. Pergamon Infotech, Maidenhead (1981).

[18] J. DARLINGTON, Program transformation: an introduction and survey. *Comp. Bull.* 1979, **2**(22), pp. 22–24.

[19] R. A. KOWALSKI, *Logic for Problem Solving*. North Holland Elsevier, New York (1979).

[20] M. M. LEHMAN, The role of executable metric models in the programming process, an unsolicited research proposal. Mar./June 1981. Presented at *IEEE Workshop on Rapid Prototyping, Baltimore*, Mass, 19–21 Apr. (1982). *ACM Software Engineering Notes* 1982, **7**(5).

[21] *The Oxford English Dictionary*, Vol. III, p. 354. defns. nos. 1, 5, 7. Clarendon Press, Oxford (1933).

[22] *Webster New Collegiate Dictionary*, 1959 *Edn*, p. 286, also Unabridged New International edn, p. 789, Merriam, Springfield, Mass. (1979).

[23] B. W. BOEHM, *Software Engineering Economics*, p. 37. Prentice-Hall, Englewood Cliffs, New Jersey (1981).

[24] E. W. DIJKSTRA, A. constructive approach to the problem of program correctness. *Bit* 1968, **8**, 174–186.

[25] N. WIRTH, Programming development by stepwise refinement. *Commun. ACM* 1971, **14**, (4) 221–227.

[26] D. A. DOLOTTA and J. R. MASHEY, An introduction to the programmer's workbench. *Proc. 2nd Int. Conf. on Software Engineering*, pp. 164–168. San Francisco, CA, IEEE Cat. No. 76CH-1125-4C, Oct. (1976).

[27] A. E. HUTCHINGS, R. W. MCGUFFIN, A. E. ELLISTON, B. R. TRAUTER ANand P. N. WESTMACOTT, CADES—software engineering in practice. *Proc. 4th Int. Conf. on Software Engineering*, pp. 136–152. Munich, Germany, IEEE Cat. No. 79CH-1479-5C, Sept. (1979).

[28] J. N. BUXTON, *Requirements for ada programming support environment—Stoneman.* US Dept. of Defence, Washington, D.C., Feb. (1980).

[29] W. E. RIDDLE and R. E. FAIRLY, Software development tools. *Proc. Pingree Park Workshop, May* 1979. Springer-Verlag, New York (1980).

[30] C. K. S. CHONG HOK YUEN, *A phenomenology of program maintenance and evolution*, Ph.D. Thesis, Dept. of Computing, Imperial College of Science and Technology, London, Nov. (1980).