The Role and Impact of Assumptions in Software Development, Maintenance and Evolution

Meir M Lehman
School of Computing, Middlesex University
The Burroughs, London NW4 4BT
mml@mdx.ac.uk
http://www.cs.mdx.ac.uk/staffpages/mml/

1 Introduction

1.1 History

This paper discusses the gradual invalidation of assumptions explicitly or implicitly embedded in software and shows that this is an inevitable, probably dominant, cause and driver of software evolution. It reasons that such evolution is a natural phenomenon having major safety, reliability and economic impact on computer usage. In the context of growing computer application, the phenomenon and its significance as this technology is ever more widely exploited are becoming increasingly apparent. Many evolution properties identified are also present in the wider context. But as demonstrated in the next section their impact are more profound in the field of software development, application and maintenance than in other fields. The problems that result are increasing. The present paper is restricted to the software areas.

Table 1 lists some of the highlights of further results obtained since the initial study. Their inclusion here is to demonstrate that being based on observations and analysis they have matured and been refined without major change over 35 years. The conclusions reached are now solidly founded, not a flash in the pan.

1.2 Evolving computing domains

Even in the computing field, evolution is not restricted to programs. Computer applications\(^1\), the wider systems to which they relate and with which they are increasingly integrated, the domains in which they are defined and executed, those where the results of computation are applied and the components of the computing system may all evolve. The last includes computer units, hardware subsystems and elements, external devices, low level elements and software in the form of programs, documentation, manuals and application and computing procedures whether written or established by practice. As long as the determining authority judges such evolution to be meaningful and beneficial they are evolved by change or replacement.

\(^1\) Unless otherwise indicated, throughout this paper the term application is to be understood as referring to all computer usage including but not limited to solution of abstract mathematical problems to be solved precisely, information systems, situations to be modelled and resolved empirically, activities supported, controlled and/or monitored etc.
Table 1. Highlights in the study of evolution

<table>
<thead>
<tr>
<th>Year</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>1968 - 69</td>
<td>The Programming Process study and report</td>
</tr>
<tr>
<td>1969 - 71</td>
<td>Identification and study of feedback as driver and controller of evolution</td>
</tr>
<tr>
<td>1969 - 02</td>
<td>Empirical studies of wide spectrum of industry-evolved systems</td>
</tr>
<tr>
<td>1974 - 86</td>
<td>Laws of Software Evolution</td>
</tr>
<tr>
<td>1979</td>
<td>SPE program classification</td>
</tr>
<tr>
<td>1988</td>
<td>First identification of the role of assumptions</td>
</tr>
<tr>
<td>1989</td>
<td>Principle of Software Uncertainty</td>
</tr>
<tr>
<td>1993</td>
<td>FEAST hypothesis</td>
</tr>
<tr>
<td>1996 - 01</td>
<td>FEAST/1 and FEAST/2 projects</td>
</tr>
<tr>
<td>2000 -</td>
<td>Proposal for development of a formalised Theory of Software Evolution</td>
</tr>
<tr>
<td>2001</td>
<td>Rules and Tools for Software Evolution Control, Planning and Management</td>
</tr>
<tr>
<td>2004</td>
<td>SPE+</td>
</tr>
</tbody>
</table>

As indicated above, evolution is, of course, not confined to computer systems. All real world systems either evolve or fall into disuse and decay. But in computing systems the phenomenon is the most intensive, has by far the most many-dimensional and critical impact and is by far the fastest recognisable change occurring in weeks or months rather than decades, centuries or even millennia. In part this is due to the fact that such systems, programs in particular, must precisely reflect the significant properties of the applications and domains being addressed and do so accurately and comprehensively whenever executed. However, programs are entirely rigid and inflexible except where a potential need for adaptation or change has been foreseen. Once created and installed program code is unambiguous and with a unique semantic. It cannot, of itself, shrink, stretch, flex or otherwise adapt to even the smallest internal or external variation or change unless deliberately programmed to do so. There is no give and take, no flexibility or self-adjustment to, for example, adapt to changing circumstances or the unanticipated. Moreover even in a single program execution, code is likely to be accessed and executed repetitively. The likelihood is, therefore, much higher than would be the case with isolated usage that any error, inaccuracy or lack of detail will be uncovered and produce unsatisfactory results. Consequent correction is a form of evolution since it widens the scope of the software, adapting or extending the domains over which it can be expected to execute satisfactorily.

Computing systems together with their operational and execution domains constitute a multi-level, multi-loop, multi-agent feedback system. They differ fundamentally from purely hard systems. For example, in the presence of and response to errors, the response times of the latter and of humans involved in their operation generally allow corrective action to be taken before serious harm befalls. Though damage may result, disaster will be rare. In software-controlled systems, on the other hand, much of the feedback involves delays much shorter than the time for human intervention and correction. It is just possible that, in the future, artificial intelligence approaches may permit a degree of automated correction in the presence of change but, as will be shown, even then satisfactory response can never be guaranteed. Development of appropriate means may permit problem reduction, decreasing error frequency and possibly their gravity. They cannot be eliminated. This is in strong contrast to the corresponding properties of the artefacts of other engineering disciplines.

In general, software is, by far, the most inflexible and error-sensitive element of the systems with which it is involved. This paper concentrates, therefore, on software and, more particularly, program evolution. Unless explicitly stated otherwise, all references to evolution in what follows are to be interpreted in this way.

### 1.3 A theory of software evolution

The systematic study of the software evolution phenomenon is still in its infancy. The groundwork has been laid. Applying phenomenological reasoning a significant number of empirical axioms, laws, principles, behavioural patterns and other characteristics have been identified and interpreted. These constitute the forerunners of potential elements and constituents of an empirical phenomenological Theory of Computing System Evolution. Parts of such a theory are candidates for formal representation. Other parts may be related to and even become part of
wider mathematical feedback and control system theory. The properties relating to the assumption phenomenon addressed here are an important part of the initial framework of this theory and of an approach to development of the means for the direction, management and control of software evolution. In terms of greater appreciation of assumptions as a contributing source of the pressures for software maintenance and to improvement of that process they constitute a topic worthy of consideration in their own right. Their wider role in software evolution is discussed in a companion paper [11].

2 The SPE program classification

2.1 Definitions

The companion paper devotes Section 2 to working definitions. These are essential for serious development of any theory. In the present paper we are more concerned with outlining the theory, stressing in particular, the role of the assumption phenomenon, conveying and spreading recognition of the latter and its implications and identifying work that should, indeed must, be undertaken to significantly improve the software maintenance process. Definitions are therefore limited to just those concepts relating to the SPE classification scheme for computer systems and programs.

2.2 The SPE scheme

E-type software is one of three classes of software defined in the SPE system classification scheme [6, 1]. The initial definition related explicitly to programs but was later extended to include computing applications, domains, systems and system elements, collectively termed systems. This paper focuses primarily on E-type systems and programs but the defining characteristics of types S and P will also be briefly discussed to help set the scene for the remainder of the discussion.

2.3 S-type systems and software

S-type software [6, 11] addresses problems that are formally defined and specified. Programs derived from such specifications can, and normally will, be required to be correct, in the full mathematical sense of the term, relative to that specification. The latter must therefore have received at least the tacit approval of stakeholders. Their satisfaction with the executable program is implied since objections to any of its properties could (in theory at least) and should have been detected and expressed during development and approval of the initial problem statement and the specification. Correct systems can be developed and delivered. Once this has been achieved the developer and supplier have met their contractual obligations.

Creation of an S-type program normally implies that individuals or groups of stakeholders have an interest in its development. If, after completion and demonstration of correctness, one or more of these people are not satisfied with the program or the results of its execution they must, by definition, be dissatisfied with the problem statement or specification as accepted. The remedy is to revise one, other or both of the former and require the creation of a new version of the software. This may be achieved by modification of the existing version but the latter must still be considered as new. Despite the demonstration of absolute correctness, hindsight or feedback can reveal that, in practical terms, with the intended purpose in mind, an S-type system does not satisfy its purpose in the required domains. Though correctness, in the strict sense, is achievable an S-type system may ultimately be shown to be unsatisfactory, absolutely or in some part of the desired operational domain. When that is the case the system must, at least conceptually, be scrapped and replaced, even though in practice its replacement may be derived from the earlier version.

2.4 P-type systems and software

The second class of software is of type P. This was originally defined as a program that solved an abstract, though not well-understood or precisely stated, problem. A formal specification could not or was not, therefore, developed and correctness proving, as required for type S, is meaningless. Subsequently, however, it became clear that every instance of type P would satisfy the definition of either type S or E. Thus until recently the type P was ignored, so it will not be further discussed here. Further work has now led to its redefinition to produce an improved version of the original scheme [1].

2.5 E-type systems and software

2.5.1 Definition

The third, E-type, designation is applied to computing applications, systems and software that operate in or interact outside the artificial world of the
model embedded in the software. They address problems or activities or execute in what is colloquially termed the *real world*. Separately and jointly, such systems display behaviour or yield concrete or abstract objects that have significance or meet specified or implied goals in the real world. The designation \( E \) was selected because of the fact that, as was shown and will be shown here, such systems must continuously be evolved in accordance with changes in the real world. If such changes are not matched by system changes the system will gradually but inevitably degenerate, becoming increasingly unsatisfactory or less useful.

By definition, the operation of \( E \)-type systems involves geographical, application and execution domains. Other domains, organisational or political for example, may also be involved. In specifying, developing and using these systems all relevant domain properties that can influence the properties of the real world phenomenon reflected in the outputs of the system, including domain boundaries, must be identified and reflected in one way or another in the application system.

For \( E \)-type systems, stakeholder and, in particular, user satisfaction with the results of system operation determines its *acceptability*. The correspondence of system behaviour and outputs to real world patterns and values during and after execution determines whether it meets the purpose for which it was created, installed and used. System acceptability is based on the satisfaction of *stakeholders* with the *results* of execution; their knowledge or conviction that these in themselves and in their impact on the real world domains elements are ‘correct’ in some defined or intuitive sense. This requirement for stakeholder satisfaction contrasts sharply with the concept of correctness as required of \( S \)-type systems.

Unless otherwise stated, all further references to programs, systems or computer applications in this paper refer to type \( E \).

### 2.5.2 \( E \)-type systems are feedback systems

Observation, analysis and reasoning about the nature of the use of computers in the real world has shown that, in general, \( E \)-type systems are and behave as complex multi-agent, multi-level, multi-loop feedback systems. This was first observed while considering the ripples on the otherwise linear growth of OS/360-70 over its first 19 releases [4] in the years following the original IBM programming process study [3]. That phenomenon was interpreted as an indicator of feedback-driven self-stabilisation of the programming process. A later paper [5] explicitly described that process as a feedback system as indicated by phenomenological reasoning, the self-stabilisation effect and the instability and ultimate break-up of the system due to excessive positive feedback effects from release 20 onwards. Measures over a number of years of changes in the physical growth, functional power, performance, complexity and other properties of industry developed and maintained systems support this system analysis based hypothesis.

More generally, behavioural studies of the process of software evolution have shown that the properties of the total system comprising operational domains of organisational entities, people, machines, computing systems, interacting processes and so on undergo continual change. Individual changes are discrete and discontinuous, but in sequences they display the hallmarks of a disciplined process.

Some of these changes are beneficial, increasing the satisfaction of users and other stakeholders and, therefore, system value (in some sense). The resultant system behaviour is similar to that of systems including positive feedback mechanisms and tends to encourage further change. Other changes degrade the system, exerting negative pressures that require action to overcome and correct degradation and to restore previously satisfactory properties. Domination of the former indicates an evolving system, though excessive positive feedback, as with OS/360-70, presents a potential for instability. The concern with the latter must be with over-compensation, since negative feedback with a strong positive component may trigger instability, with the system heading for minor or major disaster. At best it will suffer a decline in value, however defined, to its stakeholders. There are already many examples of these various situations, with some of the more costly in economic or human terms outlined in a recent analysis [12].

Despite the fact that the data provided to a computing system by means of feedback links to reflect system properties and states is continually changing and may be out of date before it can be applied, it is still used to drive, direct and, often, control system and component behaviours and interactions, much of it at very low levels of detail. To maintain the required standards of performance and safety demands checks and balances to ensure timely detection of discrepancies between the current states of the application system and the data stored in the system or embedded in program code. Effective direction and control of system maintenance and
predictable evolution requires adequate understanding of the feedback properties of the activities, systems and domains in operation. Knowledge of interactions between components and insight into the external and internal feedback properties of the systems are essential for the necessary control to be achieved.

3 The role of the computer system

3.1 The computer system as a model

In studying application systems one must distinguish between two roles that a computer, and more specifically its software, can serve. In the first of these roles it monitors, records and predicts application and domain states and behaviours by transforming, analysing and extending application system data and making it available for human interpretation and/or action. Algorithms in appropriate abstract terms that describe to the required level of precision all the relevant relationships between the subjects of the application and their separate and joint behaviours are realised by the software. The software, the software-hardware combination and the results of computation are all models of the application. They represent information on, for example, the state of the domains, progress being made in executing the application algorithms. In this role the computer is not directly linked with the application and functions as an impassionate and faithful observer. What is done with the information obtained from the computation, whether, for example, any is fed back into the application process, is of no concern.

An example of this role is found in weather observation and prediction. At present and for the foreseeable future there is no way that a computer-based weather forecasting system can directly affect future weather. Its role is to provide information that will help humans take advantage of or protect against future weather patterns or specific weather conditions. Operation is based on interpretation and extrapolation of past weather patterns in combination with the solution of complex hydrodynamic equations to provide an abstract model of weather behaviour. Computing systems restricted to such use operate as information systems where that term is used more widely than in current convention.

3.2 The computer system as a part of the application

In the second role the computer is linked in some way to the application. It receives discrete or continuous inputs directly or indirectly from human, mechanical and other elements of the application system based on which it provides computed outputs to appropriate points in the application system and its elements. It is an integral part of the application system, coupled with and having direct, possibly continual, impact on the application, its outputs and/or behaviours. Some of its output is fed back into that system to change states or processes or, for example, to offer advice or instructions to humans involved in some way with the application.

As an example, consider a computerised traffic control system receiving continuing inputs about traffic flow, road and weather conditions. On the basis of data received, appropriate calculations and output controls it provides data and signals that permits people and devices responsible for safe and efficient traffic behaviour to respond to changes and ensure continuing satisfactory traffic flow under anticipated conditions.

3.3 The computer system in practice

Increasingly computer systems combine both roles. The distinction between the roles is becoming more blurred as computerisation spreads, integration over and between applications and organisations increases, and computer activity penetrates ever more deeply into detailed levels of the application. The two roles are theoretical abstractions. A weather system, for example, even if primarily intended for forecasting, will have direct links to the physical, operational, domain. Thus, in a geographic area with frequent flooding the computer system may be directly coupled to sluice gates to facilitate their rapid closure when a threat is identified. The traffic control system, on the other hand, is likely to include an abstract computational model of traffic flow to reveal changes in flow patterns that suggest external changes to improve traffic control, flow, safety, and reliability or are injected as control signals directly into the physical domain. The distinction between roles, though not so relevant at the present level of discussion is, nevertheless, significant for improved control of the evolution phenomenon with its potential for much improved software and system maintenance and
increased functionality while maintaining satisfactory performance.

4 The size of $E$-type applications and domains

In general, any $E$-type application has an unbounded number of properties. This can be simply demonstrated. Given any two properties one may, at the very least, ask whether there is a relationship between them. Every possible answer to this question represents a new system property. This is sufficient to prove the above assertion. Admittedly, such $n$th order relationships are likely to decrease in significance as $n$ increases, so they are increasingly unlikely to have significant impact on system operation or usage. That, however, is not the point. That these relationships potentially exist is unquestionable. The issue is whether at some time in the future circumstances presently not recognised can arise in which a property now irrelevant becomes relevant.

A further source of unboundedness of numbers of properties arises from an empirical observation long experienced. As an application and its system are developed and applied one can always identify and justify another bell or whistle. The provision of new capability in a system opens up additional opportunities. In practice it is often the case that the list of candidate changes or additions not yet accepted for implementation, sometimes termed the off list, is longer than those on the in list, that is those provisionally accepted for implementation, immediately or some time in the future. Pressure for change, growth and extension is universal and often irresistible.

The entire universe with its unbounded number of sub-domains and properties is unlikely to be the active computer application domain. But, nevertheless, the Laws of Software Engineering and the discussion that follows imply that the boundaries and properties of the real world domains in which $E$-type applications are deployed cannot be absolutely or permanently defined. An unbounded number of sub-domain properties will certainly be irrelevant to use of the system, not significantly influencing system usage. But at any specific moment there will be many domain regions that are relevant and significant for the application in hand. Like the application systems themselves, these will also have an unbounded number of properties.

In addition, the real world is essentially dynamic. Only a dead world in a vacuum remains unchanging. Consideration of the resultant inevitable changes in real world properties and their possible impact on the application must take into account the fact that the application and its domains each have an unbounded number of properties. To maintain the match between the real world, the application and the computations requires the former to be updated, so that at all times computer system procedures and data accurately reflect the then current properties of the application, its procedures, the domain properties, and so on.

It is of interest to note that the very development, installation and use of a computing system intrinsically triggers evolutionary pressures. Each such activity, in itself, constitutes application, application procedures and domain changes independently of other changes in the domains. If the resultant need for adaptation elsewhere in the application system is resisted, a growing mismatch between the application, the operational domain and the computing system will cause gradual degradation of the precision, usefulness, effectiveness, the value, of the computing system, and lead inevitably to its ultimate abandonment. Change triggers change.

5 Aging

5.1 Completeness

For an $E$-type application system to be satisfactory it is necessary that its outputs produce information and behaviours that are satisfactory models of the states and behaviours of the real world application and domains as at the moment of concern. Information derived by and fed back from the computing system into the application and its domains must take into account the state of the world as at the moment of their application to that world. Errors induced by changes in value in the interval between computation and application must be below the threshold set by the required level of precision. This phenomenological truth states a completeness condition for a computer system and its software. To be satisfactory a computer system must be complete relative to the application and all its operational domains. All objects, relationships and behaviours that affect the properties and outputs of an application system at the level of detail, precision and in the timeframe of interest must be reflected in the computer system, its objects, their properties, behaviours, relationships and so on both during execution and
when the results of execution are applied. And all this is necessary in a context where the application addressed by the software system and the operational domains involved possess an unbounded number of properties and where those relevant to satisfactory operation will change with time and circumstances. Such changes must be identified, and if significant in the context of the application goals, must be reflected in system changes. As was first suggested in the 1970s, the management of software is the management of change.

5.2 A necessary (but not sufficient) condition for a program to be satisfactory

The discussion to this point and the various generalisations made include several references to stakeholder satisfaction. This demands, inter alia, the delivery of satisfactory results to users as a criterion for $E$-type system acceptability. It further requires that whenever legitimately applied, that is to provide valid or satisfactory results in the intended application and valid domains, the characteristics (relationships, values, behaviours, etc.) of all relevant application and domain properties that affect the behaviour and results of application have the same values as they will when the results of execution are applied. To satisfy the above condition for satisfactory operation requires modelling of that execution at least to the detail and precision equal to that required of the solution. This, for example, bounds the permissible computation time for an algorithm that models an event relating to changing application data to be somewhat less than that of the change, and thus implies a lower bound on the computer speeds that can be used in any application.

An $E$-type program is a compound model of the problem being addressed, the domains within which it is intended to be used and those in which the program is to be executed. Factors affecting application system behaviour at the required level of detail or precision not correctly reflected in the program in the circumstances prevailing at the time of execution may invalidate the solution. Moreover, at least in the roles described in Sections 3.2 and 3.3, the computer software implicitly includes a model of its own operation. This is, of course, not precisely possible and represents an inherent source of uncertainty, intrinsic incompleteness, imprecision and error in $E$-type systems.

6 The intrinsic software bound

The number of properties of $E$-type computer applications and their operational domains are unbounded. Having been created in finite time and being stored in finite storage media those of computer systems and their software are finite, fully characterised by a bounded number of properties. An unbounded set of properties will therefore be ignored in the system evolution process. Of these only a finite number are explicitly rejected as a result of a conscious judgment, correct or incorrect, that at the level of detail and precision required for satisfactory system operation, that is to obtain valid results, they were irrelevant. An unbounded set will have been excluded by, for example, the unconscious, and therefore unrecorded, omission of unknown, unrecognised or overlooked properties. Problems may arise if some of the latter actually are relevant within the context of the application being implemented, or if they ever become so, since completeness requires the system to reflect all of these. That such exclusion presents a real problem and threat is demonstrated by cases briefly described in Section 8.2.

Properties reflected in the software and elsewhere in the computer system are selected from the unbounded population by a process of abstraction that involves explicit or implicit references in natural language and formal statements. Unnecessary inclusion of irrelevant properties will normally be harmless but rejection must, at least at the time of execution, be valid with respect to the level of detail and precision required for satisfactory execution and results that are correct within the required degree of precision. Note that the unconsidered omission of the unbounded majority that are truly irrelevant to satisfactory operation does not violate the principles of Section 5 since in that case exclusion is, at least for the moment, of no consequence.

In any event, every property excluded knowingly or unknowingly and, therefore, not reflected in the system, implies at least one assumption; namely that its exclusion does not introduce an unacceptable error
into the computational results, i.e. that its inclusion would not contribute to satisfaction with system execution. Every exclusion involves at least this assumption of irrelevance, and it follows that an unbounded number of assumptions are reflected in every E-type software system. This occurs throughout the process, from application definition, through requirements statement, specification, all the levels of design, coding, testing, validation and reification to the executable, shipped, code and the validation of code, documentation and system. This is reflected in the following Principle, a potential theorem in the theory of software evolution, as:

Any E-type system reflects an unbounded number of assumptions any of which if invalidated by a real world change or otherwise may cause unsatisfactory computing system behaviour or computational results.

## 7 The inevitable consequence

As long as no (then) relevant properties were rejected at any point in the development or evolution processes, their omission did not inject assumptions into the system. While this is a comforting thought, it is, unfortunately, of no great consequence in relation to the issue of software maintenance and evolution. As discussed, the real world addressed in computer usage is dynamic; continual change is inevitable. As demonstrated by the examples of Section 8.2, such changes can invalidate one or more of the, originally legitimate, assumptions of irrelevance. The omission may then become a source of unsatisfactory results in execution. In addition, assumptions about the characteristics of properties consciously selected and reflected in the system may become invalid. Until the discrepancies introduced by such changes have been rectified execution may be unsatisfactory in at least some of the circumstances included in the terms of reference of the system. From the very first statement of intent and purpose it is therefore almost certain that, quite apart from those invalid or unjustified assumptions made in pursuing the process, over time invalid assumptions will appear in the software. Until these are recognised and rectified their influence on computational results is unknown. There is every reason to believe that, in a very general sense, this is a major source of the universally experienced need to continually upgrade and evolve executable software, documentation and procedures for satisfactory application of the results of execution. Hence, the conscientious control and management of assumptions must be accepted as a major task in software maintenance and evolution.

Given this understanding the Principle of Software Uncertainty, a corollary of the principle stated in Section 6, may now be introduced.

## 8 Principle of software uncertainty

### 8.1 Principle

An early formulation of this principle [8, 9] stated:

The outcome of software system operation in the real world is inherently uncertain with the precise area of uncertainty also not knowable.

More recently [10] it was restated as:

The outcome of the execution of E-type software entails a degree of uncertainty; the outcome of execution cannot be absolutely predicted.

or more fully as:

Even if the outcome of past executions of an E-type program have been satisfactory, the outcome of further executions is inherently uncertain; that is, a program may display unsatisfactory behaviour or invalid results.

Basically, these statements all make the same assertion. The facts outlined above and the inherent uncertainty implied cannot be disputed. Uncertainty arises from the nature of real world computing. The historical differences in wording are due to the context in which they were stated. What is certain is that under normal conditions, a program does not age or change spontaneously. Of itself software, as such, does not deteriorate. Uncertainty is, at least in part, due to changes, known and unknown, in the application, the operational environments, the various domains and their impact on the validity of assumptions reflected in the computer system and, most particularly, its software. Humans can be blamed for lack of foresight or insufficient care in searching for, detecting, controlling assumptions, explicit or implicit, but uncertainty and the related need for continuing evolution are intrinsic to computer use in the real world.

---

6 Since, as defined, satisfaction is a binary variable any contribution to satisfaction provided by a feature included but not required may safely be disregarded.

7 In a presentation to the Royal Society, the late Gordon Scarrott, Chief Engineer of ICL, presented the principle as an instance of Heisenberg's principle. This author has considered it as an analogue. This remains an issue to be resolved.
8.2 Empirical support for the role of assumptions

The preceding sections have outlined and provides the phenomenology that underlies the Principle of Software Uncertainty. The latter is the source of the universally experienced need for continual evolution of E-type software systems to maintain stakeholder satisfaction by maintaining the validity of their assumption sets. Empirical investigations of major system failures in four very different application areas fully support the conclusions. The cases considered were the Ariane V destruct, a CERN Accelerator failure, the Sheffield sinking and the failure of the London Ambulance System. In all cases, analysis revealed assumption(s) that led to disastrous failure. Section 8.5 will suggest a need for further studies but these four cases establish a prima facie case.

Ariane V: The European space rocket was blown up on its first test flight some seconds after launch. Its predecessor, Ariane IV, had been launched at an angle required to be within ± 0.5° of the vertical. To ensure correct launch, its software included a check on this angle that operated up to the moment of launch. The launch angle for Ariane V was changed to 4°. At some previous time it had been decided that the checking software could continue to run without harm on Ariane 5 after launch provided its output was not fed back to rocket control. On the contrary, it might be useful if a launch had to be delayed. This strategy was adopted and retained though the function was not required for Ariane 5. The check was therefore running when the latter was launched though the output was not being used. Given the eightfold increase in launch angle from the specified “vertical” ± 0.5° in Ariane 4 the computed deviation from the expected value for the angle built up very rapidly when the same software was (unnecessarily) monitoring Ariane 5. While monitoring the first Ariane 5 take-off, the accumulator in which the deviation from vertical was stored held a number between 1 and 2. The error checking software monitoring the arithmetic unit (AU) saw this apparently negative number, as determined by its leading bit, in a register that the code implicitly assumed would always be positive, assumed that the AU was faulty, switched to the back-up computer and shut down the first machine. In a short while the data in the back-up also overflowed, and it too was pronounced faulty and shut down. Since the rocket was now computerless the destruct sequence was initiated and completed. End of story, the consequence of no less than five assumptions (some related) that became invalid for one reason or another as a result of changes in the external world.

CERN Accelerator: This incident was related to the author during a lecture at CERN some years ago to provide an example of the assumption problem he had been presenting. Some months earlier the Institute had completed the construction of a 56 km diameter accelerator, twice as large as any previously built. On completion and as a first test an experiment that had previously been run on its predecessor was rerun on the new machine. Instead of the expected order of magnitude improvement in precision this trial produced a significantly different result. No explanation of this deviation was immediately available so other previously executed experiments were repeated. In each case, a different result was obtained. No explanation was forthcoming so after a wait of one or two weeks all the experiments were repeated. Much to their chagrin, the results this time were not only different to the historic results but also to those obtained in the earlier runs with the new accelerator. There was clearly a problem, but no one was able to suggest an explanation until after a few days one of their mathematicians stated that he believed that the magnitude of the error obtained was correlated to the phase of the moon. This was of course greeted with incredulity. But he persevered and came back after some days to restate, demonstrate and explain his observation. It transpired that the gravitational or tidal pull of the moon on the accelerator changed its shape from the assumed perfect circularity. Deviation from this had not been of significance in the smaller predecessor machines. For the larger machine it precisely explained the error in each of the reruns. And so it was. Once all the equations in the software had been modified to take the gravitational pull of the moon (on the shape of the accelerator but not on the circulating particles) into account, the accelerator produced the expected results. It was also noted that if an accelerator twice as large again were to be constructed the gravitational pull of the sun would also have to be taken into account. In any event, this provides a second example of an unconscious assumption that became invalid, that the moon was irrelevant to construction on earth.

Sinking of HMS Sheffield: This Destroyer was sunk during the Falkland war by an Exocet missile with the loss of 20 lives. There was an operating advance warning radar on board the ship, which should have detected and intercepted the missile on launch, destroying or diverting it. Why did the system fail? Such radar detection systems are based on detection,
analysis and identification of the electro-magnetic signature of the moving missile. For this purpose the system must have access to the signatures of all missiles against which it is to be protected. Available storage on the Sheffield installation was not sufficient to store the signatures of all known missiles. Thus the decision was taken to store only the signatures of “potentially hostile” missiles. Exocet, being French, was not classified as “potentially hostile”. Hence, though presumably detected by the radar, the absence of the signature of the missile directed at the Sheffield meant that no action was taken. That the sale of missiles to the Argentineans was not noted and acted upon has not been explained but the “incident” that bereaved twenty families and cost a ship illustrates the need not only to detect assumptions, but also to record them and keep them under constant review.

**London Ambulance System:** This system, developed at a cost of tens of millions of pounds, was installed, in service for three days and then scrapped [2]. There appear to have been two basic reasons for its failure. The first related to the fact that the appropriate union organisations in general and the ambulance crews in particular were not consulted. Had this been done then surely the second and underlying reason would have been revealed. For whatever reason the designers decided to use a screen interface between the ambulance crew and the control centre. That drivers or even anyone sitting next to them could not effectively interact with a screen or follow directions conveyed via a screen while driving at high speeds through London traffic appears not to have been recognised. Instead it appears to have been assumed that a screen was the most effective means of communication.

### 8.3 Outline analysis

Detailed analysis of the above instances is not possible in this paper. Suffice it to say that in all cases it is possible to observe with hindsight that the external circumstances and changes that caused the problem might have been foreseen. Overlooking of the lunar tidal pull in the CERN case is, probably the most excusable. That the change of launch angle from Ariane 4 to Ariane 5 did not trigger alarm bells for, at least, a review of its wider implications is perhaps understandable since while the software was still running at the moment of launch it was implicitly assumed that its output was not coupled to the remainder of the system. On the other hand, failing to consider the likelihood of the export of missiles from the producing country and their onward resale over time is very difficult to understand. The need to periodically review the list of signatures stored, particularly when going to war, should have been self-evident. The ambulance system decision with an implicit assumption is equally difficult to understand.

In summary, while not sufficient to estimate the expected percentage of failures as a result of the invalidation of assumptions, the fact that in ALL four cases described the offending assumption(s) could be identified, supported by the reasoning presented earlier, indicates that one may expect such contributions to be frequent. Some general process implications of this observation will be introduced in the next section.

### 8.4 Further work

#### 8.4.1 Universality of the assumption problem

It has not yet been possible to examine more cases to estimate the extent of the problem, the likelihood of its universality and, in particular, to investigate its relationship to application modes and areas and to development, maintenance and evolution processes, methods and support. As an extreme example consider the contrast between real time defence systems and computer games in relation to the assumption problem. The need to reveal and manage all relevant assumptions will certainly differ between them. It may also differ if one compares systems developed using different processes, waterfall, object-oriented, open source, and so on. But growing computerisation and its increasing application, integration and penetration at the detailed level of applications clearly imply that the human, social and economic impact of failures will increase the frequency and cost of this phenomenon. Thus its further analysis and, in the first place, a clear empirical demonstration of its universality deserves high priority in software engineering investigation and research.

#### 8.4.2 Empirical studies

Empirical data on the magnitude of the problem and its cost as a function of application area and process approach, for example, would prove most valuable. Studies of some tens of development projects and evolving systems, both successful and unsuccessful, spread across these and possibly other application and process variables are necessary to provide the necessary information. Such an investigation would either refute the theory presented here, restrict it to a small class of applications, or demonstrate the reality, generality and seriousness of...
the phenomenon. It this author's unqualified judgment based on more than 35 years of work and the generality of reasoning not rooted in computing technology but based on the properties of the universe we live and work in, that the third alternative, that this is a universal phenomenon that presents a major and growing problem, is by far the most likely outcome. In a world increasingly run by and dependent on computers and their software, this implies that it is a software engineering R & D area worthy of serious investment of effort and finance. The goal must be to understand, then master, the phenomenon. The development of processes, methods and tools for effective assumption control follows as a close second. The potentials for savings and hence justifiable levels of investment in each area for such development to yield more reliable operation and significant savings are indicated in Section 9.

8.4.3 Process, methods, tools

As an example of the potential for the development of processes, methods and tools for assumption management consider inspection, a widely recognised and accepted part of the software requirements, design and implementation processes. Despite its success in validation at all stages of the process this author, at least, is not aware of any organisation that searches for assumptions, the completeness of the identified set at the required level of detail and precision, verification of the current validity of the identified set and their volatility, that is their exposure, sensitivity to and likelihood of change in the future. The implication of the analysis to this point is clearly that at each moment of analysis, specification, design, implementation and validation during software and system development, maintenance and evolution, as the detailed outputs of each step are reviewed during inspection one must explicitly ask what assumptions are being made in the local and global context and what changes in circumstances, the application or its domains could invalidate any of these. Methods and tools to assist this process can surely be developed or adapted from existing mechanisms. The failure to investigate, develop and apply them is a major weakness of current software engineering R & D. At the very least, the obligation to manage and control assumptions and to identify the likelihood of change under various circumstances, in various regions of the application and its domains must receive high visibility in application definition, requirements statements and design activities. In all these cases development of methods and tools or adaptation from existing mechanisms will make software development, usage and evolution more reliable, less time consuming, less costly and more effective, yielding major safety, reliability and economic benefits. Assumption-directed research and development activity is unquestionably a challenge but also an urgent need with major potential for success and benefit.

8.4.4 Formalisation of assumption management

There is good reason to believe that the use of formal methods has real potential in appropriate aspects of assumption detection, management, control and maintenance. It could prove to be of major benefit in imposing process discipline and significantly advancing assumption management. Machine search for actual or potential relationships between domain changes, documented assumptions and system function, for example, can provide advance warning of application/domains/system mismatch and forestall failure. It can yield extra time to, for example, make system changes that correct for domain changes and so reduce impact and costs. Research and development in this area would have to start from scratch and would require the formalists to change and adapt some of their preconceived ideas. But one must be confident that the application of concepts and approaches drawn from advanced, formal, computing science is likely to provide a powerful tool in finding the solution to a problem that will, otherwise only grow and intensify in its threat and consequences.

Current discussions in the UK on challenges in computer science have identified the need to provide fast and reliable solutions to satisfy the rapidly changing information management needs of humans in society. Meeting this challenge clearly requires consideration of the stability and validity of the solutions in the face of a changing real world. Of what use, for example, is a verifying compiler that cannot take into continuing account the assumption phenomenon as discussed? Clearly reliable serious management of that phenomenon must be accepted as a fundamental element of the overall problem posed by the need for software evolution.

9 The potential benefit

A recent report of a joint working group of the Royal Academy of Engineering and the British Computer Society [12] quoted the total UK expenditure on software for the year 2003–4 as £22.6bn. It has long been known that of total life cycle
10 Final word

This author and many others have for long decreed the use of the term maintenance with respect to software. That term refers to activity which restores an artefact or other human product as closely as possible to its pristine state following deterioration as a result of use, wear and tear, weather, aging and other effects. As already mentioned, software does not, of itself, deteriorate. It is the operational environments that change to produce an effect analogous but not identical to aging. This is the observation that led to replacement of the term maintenance by evolution. As already stated in the opening remarks the latter term is a far more precise description of the process that this conference seeks to address. The term maintenance does, however, have precise meaning and deep significance in the software context. When software is evolved, the goal is to maintain the validity of the assumption set that is embedded within it and strongly influences its behaviour. Maintenance of assumptions, in turn, maintains stakeholder satisfaction in general and user satisfaction in particular through maintenance of the required level of detail and precision. That is a principle that must never be overlooked.

The presence and impact of assumptions on systems in general, computing systems in particular and software most especially has gone largely unrecognised over the past fifty or so years. It is now apparent, with hindsight self-evident, that assumptions, explicit or implicit, that have become invalid as a consequence of changes in an application or its operational domains, and the properties of those assumptions previously ignored that have now become relevant, are in large measure responsible for the need to continually upgrade and evolve software.

11 Acknowledgments

I gratefully acknowledge the many important contributions made by many colleagues over the years and, in particular, by Drs Juan Ramil and Paul Wernick to the concepts and observations presented here. Also to the anonymous referees who greatly improved the presentation by their comments.

12 References