Software Maintenance: A Tutorial

Keith H. Bennett

1. OBJECTIVES FOR THE READER

The objectives of this tutorial are:

- To explain what is meant by software maintenance
- To show how software maintenance fits into other software engineering activities
- To explain the relationship between software maintenance and the organization
- To explain best practice in software maintenance in terms of a process model
- To describe important maintenance technology such as impact analysis
- To explain what is meant by a legacy system, and describe how reverse engineering and other techniques may be used to support legacy systems

2. OVERVIEW OF THE TUTORIAL

The tutorial starts with a short introduction to the field of software engineering, thereby providing the context for the constituent field of software maintenance. The aim of the tutorial is to focus on solutions, not problems, but an appreciation of the problems in software maintenance is important. The solutions are categorized in a three-layer model: organizational issues, process issues, and technical issues.

Our presentation of organizational solutions to maintenance concentrates on software as an asset whose value needs to be sustained. We explain the process of software maintenance by describing the IEEE standard for the maintenance process. It provides a very sensible approach that is applicable to many organizations.

Technical issues are explained by concentrating on techniques of particular importance to maintenance. For example, configuration management and version control are as important for initial development as for maintenance, so these are not addressed. In contrast, coping with the ripple (domino) effect is only found during maintenance, and it is one of the crucial technical problems to be solved. We describe solutions to this.

By this stage, the tutorial will have presented the typical iterative maintenance process that is used, at various levels of sophistication, in many organizations. However, the software may become so difficult and expensive to maintain that special, often drastic, action is needed. The software is then called a "legacy system," and the particular problems of and solutions to coping with legacy code are described.

The tutorial is completed by considering some fruitful research directions for the field.

3. THE SOFTWARE ENGINEERING FIELD

Software maintenance is concerned with modifying software once it is delivered to a customer. By that definition, it forms a subarea of the wider field of software engineering, which is defined as:

> . . . the application of the systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is the application of engineering to software. [IEEE91]

It is helpful to understand trends and objectives of the wider field in order to explain the detailed problems and solutions concerned with maintenance. McDermid's definition in the *Software Engineering Reference Book* embodies the spirit of the engineering approach. He states that:

Software engineering is the science and art of specifying, designing, implementing and evolving—with economy, time limits and elegance—programs, documentation and operating procedures whereby computers can be made useful to man. [MCDER 91]

Software engineering is still a very young discipline and the term itself was only invented in 1968. Modern computing is only some 45 years old, yet within that time we have gained the ability to solve very difficult and large problems. Often, these huge projects consume thousands of person-years or more of design. The rapid increase in the size of the systems that we tackle, from 100-line programs 45 years ago to multimillion-line systems now, presents very many problems of dealing with scale, so it is not surprising that evolving such systems to meet continually changing user needs is difficult.

Much progress has been made over the past decade in improving our ability to construct high-quality software that meets users' needs. Is it feasible to extrapolate these trends? Baber [BABE 91] has identified three possible futures for software engineering:

1. Failures of software systems are common, due to limited technical competence of developers. This is largely an extrapolation of the present situation.
2. The use of computer systems is limited to those application in which there is a minimal risk to the public. There is widespread scepticism about the safety of software-based systems. There may be legislation covering the use of software in safety-critical and safety-related systems.
3. The professional competence and qualifications of software designers are developed to such a high level that even very challenging demands can be met reliably and safely. In this vision of the future, software systems would be delivered on time, fully meeting their requirements, and be applicable in safety-critical systems.

In case (1), software development is seen primarily as a craft activity. Option (2) is unrealistic; software is too important to be restricted in this way. Hence, there is considerable interest within the software engineering field in addressing the issues raised by (3). In this tutorial, we see (3), with obvious extensions to address evolving systems, as defining the goal of software maintenance.

A root problem for many software systems, which causes some of the most difficult problems for software maintenance, is complexity. Sometimes, this arises because a system is migrated from hardware to software in order to gain the additional functionality that is easy to achieve in software. Complexity should be a result of implementing an inherently complex application (for example in a tax calculation package, which is deterministic but nonlinear; or automation of the U.K. Immigration Act, which is complex and ambiguous). The main tools to control complexity are modular design and building systems as separated layers of abstraction in order to separate concerns. Nevertheless, the combination of scale and application complexity mean that it is not feasible for one person alone to understand the complete software system.

4. SOFTWARE MAINTENANCE

Once software has been initially produced, it then passes into the maintenance phase. The IEEE definition of software maintenance is as follows:

Software maintenance is the process of modifying the software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a change in environment. [IEEE91]

Some organizations use the term software maintenance to refer only to the implementation of very small changes (e.g., less than one day), and software development is used to refer to all other modifications and enhancements. However, to avoid confusion, we shall continue to use the IEEE standard definition.

Software maintenance, although part of software engineering, is by itself of major economic importance. A number of surveys over the last 15 years has shown that for most software, software maintenance represents anything between 40% and 90% of total life cycle costs (see [FOST93] for a review of such surveys). A number of surveys have also tried to compute the total software maintenance costs in the United Kingdom and in the United States. Although these figures need to be treated with a certain amount of caution, it seems clear that a huge amount of money is being spent on software maintenance.

The inability to undertake maintenance quickly, safely, and cheaply means that for many organizations, a substantial applications backlog builds up. The Management Information Services Department is unable to make changes at the rate required by marketing or business needs. End users become frustrated, and often adopt PC solutions in order to short circuit the problems. They may then find that a process of rapid prototyping and end-user computing provides them (at least in the short term) with quicker and easier solutions than those supplied by the Management Information Systems Department.

In the early decades of computing, software maintenance comprised a relatively small part of the software life cycle; the major activity was writing new programs for new applications. In the late 1960s and 1970s, management began to realize that old software does not simply die, and at that point software maintenance started to be recognized as a significant activity. An anecdote about the early days of electronic data processing in banks illustrates this point. In the 1950s, a large U.S. bank was about to take the major step of employing its very first full-time programmer. Management raised the issue of what would happen to this person once the programs had been written. The same bank now has several buildings full of data processing staff.

In the 1980s, it was becoming evident that old architectures were severely constraining new design. In another example from the U.S. banking system, existing banks had difficulty modifying their software in order to introduce automatic teller machines. In contrast, new banks writing software from scratch found this relatively straightforward. It has also been reported in the United Kingdom that at least two mergers of financial organizations were unable to go ahead due to the problems of bringing together software from two different organizations.

In the 1990s, a large part of the business needs of many organization was implemented, so that business change is now represented by evolutionary change to the software, not revolutionary change, and most so-called development is actually enhancement and evolution.

5. TYPES OF SOFTWARE MAINTENANCE

Leintz and Swanson [LEIN78, LEIN80] undertook a survey as a result of which maintenance was categorised into four different categories:

1. *Perfective maintenance.* Changes required as a result of user requests (also known as evolutive maintenance)
2. *Adaptive maintenance.* Changes needed as a consequence of operating system, hardware, DBMS, and so forth, changes
3. *Corrective maintenance.* The identification and removal of faults in the software
4. *Preventative maintenance.* Changes made to software to make it more maintainable

The above categorization is very useful to bring home to management some of the basic costs of maintenance. However, it will be seen from Section 9 that the processes for the four types are very similar, and there is little advantage in distinguishing them when designing best practice maintenance processes.

It seems clear from a number of surveys that the majority of software maintenance is concerned with evolution deriving from user-requested changes.

The important requirement of software maintenance for the client is that changes are accomplished quickly and cost effectively. The reliability of the software should at worst not be degraded by the changes. Additionally, the maintainability of the system should not degrade, otherwise future changes will be progressively be more expensive to carry out. This phenomenon was recognized by Lehman, and expressed in terms of his well-known laws of evolution [LEHE80, LEHE84]. The first law of continuing change states that "a program that is used in a real-world environment necessarily must change or become progressively less useful in that environment."

This argues that software evolution is not an undesirable attribute; essentially, it is only useful software that evolves. Lehman's second law of increasing complexity states that "as an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving the semantics and simplifying the structure." This law argues that things will become much worse unless we do something about it. The problem for most software is that nothing has been done about it so that changes are increasingly more expensive and difficult. Ultimately, maintenance may become too expensive and almost infeasible; the software then becomes known as a "legacy system" (see Section 11). Nevertheless, it may be of essential importance to the organization.

6. PROBLEMS OF SOFTWARE MAINTENANCE

There are many technical and managerial problems in striving to accomplish the objective of changing software quickly, reliably, and cheaply. For example, user changes are often described in terms of the *behavior* of the software system; these must be interpreted as changes to the source code. When a change is made to the code, there may be substantial consequential changes, not only in the code itself, but within documentation, design, test suites and so on (this is termed the *domino,* or *ripple effect*). Many systems under maintenance are very large, and solutions that work for laboratory-scale pilots will not scale up to industrial-sized software. Indeed, it may be said that any program that is sufficiently small to fit into a textbook or to be understood by one person does not have maintenance problems.

There is much in common between best practice in software engineering in general and software maintenance in particular. Software maintenance problems essentially break into three categories:

1. *The alignment with organizational objectives.* Initial software development is usually project based, with defined timescale and budget. The main emphasis is to deliver on time within budget to meet user needs. In contrast, software maintenance often has the objective of extending the life of a software system for as long as possible. In addition it may be driven by the need to meet user demand for software updates and enhancements. In both cases, return on investment is much less clear, so that the view at senior management level is often of a major activity that is consuming large resources and providing no clear quantifiable benefit for the organization.
2. *Process issues.* At the process level, there are many activities in common with software development. For example configuration management is a crucial activity in both. However software maintenance requires a number of additional activities not found in initial development. Initial requests for changes are usually made to a "help desk" (often part of a larger end-user support unit), which must assess the change (as many change requests derive from misunderstanding of documentation), and if it is viable, passed to a technical group that can assess the cost of making the change. Impact analysis on both the software and the organization, and the associated need for system comprehension, are crucial issues. Further down the life cycle, it is important to be able to perform regression tests on the software so that the new changes do not introduce errors into the parts of the software that were not altered.
3. *Technical issues.* There are a number of technical challenges to software maintenance. As noted above, the ability to construct software such that it is easy to comprehend is a major issue [ROBS91]. A number of studies have shown that the majority of time spent in maintaining software is actually consumed in this activity. Similarly, testing in a cost-effective way provides major challenges. Despite the emergence of methods based on discrete mathematics (e.g. to prove that an implementation meets its specification), most current software is tested rather than verified, and the cost of repeating a full test suite on a major piece of software can be very large in terms of money and time. It will be better to select a subset of tests that only stress those parts of the system that have been changed, together with the regression tests. The technology to do this is still not available, despite much useful progress. As an example, it is useful to consider a major billing package for an industrial organization. The change of the taxation rate in such a system should be a simple matter; after all, generations of students are taught to place such constants at the head of the program so only a one-line edit is needed. However, for a major multinational company, dealing with taxation rates in several countries with complex and different rules for tax calculations (i.e., complex business rules), the change of the taxation rate may involve a huge expense.

Other problems relate to the lower status of software maintenance compared with software development. In the manufacture of a consumer durable, the majority of the cost lies in production, and it is well understood that design faults can be hugely expensive. In contrast, the construction of software is automatic, and development represents almost all the initial cost. Hence, in conditions of financial stringency, it is tempting to cut costs by cutting back on design. This can have a very serious effect on the costs of subsequent maintenance.

One of the problems for management is that it is very difficult to assess a software product to determine how easy it is to change. This means that there is little incentive for initial development projects to construct software that is easy to evolve. Indeed, lucrative maintenance contracts may follow a software system in which shortcuts have been taken during development [WALT94].

We have stressed the problems of software maintenance in order to differentiate it from software engineering in general. However, much is known about best practice in software maintenance and there are excellent case studies such as the U.S. Space Shuttle on-board flight control software system, which demonstrates that software can be evolved carefully and with improving reliability. The remainder of this paper is focused on solutions rather than problems. The great majority of software in use today is neither geriatric nor state of the art, and this tutorial addresses this type of software. It describes a top-down approach to successful maintenance, addressing:

1. Software maintenance and the organization
2. Process models
3. Technical Issues

In particular, we shall focus on the IEEE standard for software maintenance process, which illustrates the improving maturity of the field.

7. ORGANIZATIONAL ASPECTS OF MAINTENANCE

In 1987, Colter [COLT87] stated that the major problem of software maintenance was not technical, but managerial: software maintenance organizations were failing to relate their work to the needs of the business, and, therefore, it should not be a surprise that the field suffered from low investment and poor status in comparison to initial development, which was seen as a revenue and profit generator.

Initial software development is product oriented; the aim is to deliver an artifact within budget and on time. In contrast, software maintenance is much closer to a service. In many Japanese organizations, for example [BENN94], software maintenance is seen at senior management level primarily as a means of ensuring continued satisfaction with the software; it is closely related to quality. The customer expects the software to continue to evolve to meet his or her changing needs, and the vendor must respond quickly and effectively or lose business. In Japan, it is also possible in certain circumstances to include software as an asset on the balance sheet. These combine to ensure that software maintenance has a high profile with senior management in Japan.

Like any other activity, software maintenance requires financial investment. We have already seen that maintenance may be regarded simply as a drain on resources, distant to core activities, by senior management in a company, and it becomes a prime candidate for funding reduction and even close down. Software maintenance thus needs to be expressed in terms of return on investment. In many organizations undertaking maintenance for other internal divisions, the service is rarely charged out as a revenue-generating activity from a profit center. In the U.K. defence sector, there has been a major change in practice in charging for maintenance. Until recently, work would be charged to Government based on the time taken to do the work plus a profit margin. Currently, competitive tendering (procurement) is used for specific work packages.

Recently there has been a trend for software maintenance to be *outsourced;* in other words, a company will contract out its software maintenance to another that specializes in this field. Companies in India, China, and other coutries are becoming increasingly competitive in this market. This is sometimes done for peripheral software, as the company is unwilling to release the software used in its core business. An outsourcing company will typically spend a number of months assessing the software before it will accept a contract. Increasingly, *service-level agreements* between the maintenance organization (whether internal or external) and the customer are being used as a contractual mechanism for defining the maintenance service that will be provided. The U.K. Central Computer and Telecommunications Agency has produced a series of guidelines on good practice in this area, in the form of the Information Technology Infrastructure Library [ITIL93].

When new software is passed over to the customer, payment for subsequent maintenance must be determined. At this stage, primary concerns are typically:

- Repair of errors on delivery
- Changes to reflect an ambiguous specification

Increasingly, the former is being met by some form of warranty, to bring software in line with other goods (although much commodity software is still ringed with disclaimers). Hence, the vendor pays. The latter is much more difficult to resolve, and addresses much more than the functional specification. For example, if the software is not delivered in a highly maintainable form, there will be major cost implications for the purchaser.

Recently, Foster [FOST93] has recently proposed an interesting investment cost model that regards software as a corporate asset that can justify financial support in order to sustain its value. Foster uses his model to determine the optimum release strategy for a major software system. This is, hence, a business model, allowing an organization the ability to calculate return on investment in software by methods comparable with investment in other kinds of assets. Foster remarks that many papers on software maintenance recognize that it is a little understood area but it consumes vast amounts of money. With such large expenditure, even small technical advances must be worth many times that cost. The software maintenance manager, however, has to justify investment in an area that does not directly generate income. Foster's approach allows a manager to derive a model for assessing the financial implications of the proposed change of activity, thereby providing the means to calculate both cost and benefit. By expressing the result in terms of return on investment, the change can be ranked against competing demands for funding.

Some work has been undertaken in applying predictive cost modeling to software maintenance, based on the COCOMO techniques. The results of such work remain to be seen.

The AMES project [HATH94, BOLD94, BOLD95] is addressing the development of methods and tools to aid application management, where application management is defined as "the contracted responsibility for the management and execution of all activities related to the maintenance of existing applications." Its focus is on the formalization of many of the issues raised in this section, and, in particular, customer-supplier relations. It is developing a maturity model to support the assessment of this relationship in a quantitative and systematic way.

8. PROCESS MODELS

Process management is defined as "the direction, control and co-ordination of work performed to develop a product or perform a service." [IEEE91] This definition, therefore, encompasses software maintenance and includes quality, line management, technical, and executive processes. A mature engineering discipline is characterised by mature, well-understood processes so it is understandable that modeling software maintenance, and integrating it with software development, is an area of active concern [MCDER91]. A software process model may be defined as "a purely descriptive representation of the software process, representing the attributes of a range of particular software processes and being sufficiently specific to allow reasoning about them." [DOWS85]

The foundation of good practice is a mature process, and the Software Engineering Institute at Carnegie-Mellon University has pioneered the development of a scale by which process maturity may be measured. A questionnaire approach is used to assess the maturity of an organization and thus also provides a metric for process improvement. More recently, the BOOTSTRAP project has provided an alternative maturity model from a European perspective.

In order to promote the establishment of better understood processes, the IEEE has published a standard for software maintenance [IEEE] and the next section describes this in detail. This reflects the difference between maintenance and initial development processes. It represents well many of the elements of good practice in software maintenance. The model is based on an iterative approach of accepting a stream of change requests (and error reports), implementing the changes, and, after testing, forming new software releases. This model is widely used in industry, in small-to-medium-sized projects, and for in-house support. It comprises four keys stages:

1. **Help desk.** The problem is received, a preliminary analysis undertaken, and, if the problem is sensible, it is accepted.
2. **Analysis.** A managerial and technical analysis of the problem is undertaken, to investigate and cost alternative solutions.
3. **Implementation.** The chosen solution is implemented and tested.
4. **Release.** The change (along with others) is released to the customer.

Most best practice models (e.g., that of Hinley [HINL92]) incorporate this approach, though it is often refined into much more detailed stages (as in the IEEE model described in the next section). Wider aspects of the software maintenance process, in the form of applications management, are addressed in [HATH94].

9. IEEE STANDARD FOR SOFTWARE MAINTENANCE [IEEE98]

9.1. Overview of the Standard

This new proposed standard describes the process for managing and executing software maintenance activities. Almost all of the standard is relevant for software maintenance. The focus of the standard is in a seven-stage activity model of software maintenance, which incorporates the following stages:

1. Problem identification
2. Analysis
3. Design
4. Implementation
5. System test
6. Acceptance test
7. Delivery

Each of the seven activities has five associated attributes; these are:

1. Input life cycle products
2. Output life cycle products
3. Activity definition
4. Control
5. Metrics

A number of these, particularly in the early stages of the maintenance process, are already addressed by existing IEEE standards.

As an example, we consider the second activity in the process model, the analysis phase. This phase accepts as its input a validated problem report, together with any initial resource estimates and other repository information, plus project and system documentation if available. The process is seen as having two substantial components. First of all, feasibility analysis is undertaken, in which the impact of the modification is assessed, alternative solutions investigated, short- and long-term costs assessed, and the value of the benefit of making the change computed. Once a particular approach has been selected then the second stage of detailed analysis is undertaken. This determines firm requirements of the modification, identifies the software involved, and requires a test strategy and an implementation plan to be produced.

In practice, this is one of the most difficult stages of software maintenance. The change may effect many aspects of the software, including not only documentation, test suites, and so on, but also the environment and even the hardware. The standard insists that all affected components shall be identified and brought into the scope of the change.

The standard also requires that at this stage a test strategy be derived comprising at least three levels of test including unit testing, integration testing, and user-orientated functional acceptance tests. It is also required to supply regression test requirements associated with each of these levels of test.

9.2. Structure of the Standard

The standard also establishes quality control for each of the seven phases. For example, for the analysis phase, the following controls are required as a minimum:

1. Retrieval of the current version of project and systems documentation from the configuration control function of the organization
2. A review of the proposed changes and an engineering analysis to assess the technical and economic feasibility and to assess correctness
3. Consideration of the integration of the proposed change within the existing software
4. Verification that all appropriate analysis and project documentation is updated and properly controlled
5. Verification that the testing organization is providing a strategy for testing the changes and that the change schedule can support the proposed test strategy
6. Review of the resource estimates and schedules and verification of their accuracy
7. The undertaking of a technical review to select the problem reports and proposed enhancements to be implemented and released. The list of changes shall be documented.

Finally, at the end of the analysis phase a risk analysis is required to be performed. Any initial resource estimate will be revised, and a decision that includes the customer is made on whether to proceed on to the next phase.

The phase deliverables are also specified, again as a minimum, as follows:

1. Feasibility report for problem reports
2. Detailed analysis report
3. Updated requirements
4. Preliminary modification list
5. Development, integration, and acceptance test strategy
6. Implementation plan

The contents of the analysis report is further specified in greater detail by the proposed standard.

The standard suggests that the following metrics be taken during the analysis phase:

1. Requirement changes
2. Documentation area rates
3. Effort per function area
4. Elapsed time
5. Error rates generated, by priority and type

The standard also includes appendices that provide guidelines on maintenance practice. These are not part of the standard itself but are included as useful information. For example, in terms of our analysis stage, the appendix provides a short commentary on the provision of change on impact analysis. A further appendix addresses supporting maintenance technology, particularly reengineering and reverse engineering. A brief description of these processes is also given.

9.3. Assessment of the Proposed Standard

The standard represents a welcome step forward in establishing a process standard for software maintenance. A strength of the approach is that it is based on existing IEEE standards from other areas in software engineering. It accommodates practical necessities, such as the need to undertake emergency repairs.

On the other hand, it is clearly oriented toward classic concepts of software development and maintenance. It does not cover issues such as rapid application development and end-user computing. Nor does it address executive-level issues in the process model nor establish boundaries for the scope of the model.

The process model corresponds approximately to level two in the SEI five-level model. The SEI model is forming the basis of the SPICE process assessment standards initiative.

Organizations may well be interested in increasing the maturity of their software engineering processes. Neither the IEEE standard nor the SEI model give direct help in process improvement. Further details of this may be found in [HINL92]. Additionally, there is still little evidence in practice that improving software process maturity actually benefits organizations, and the whole edifice is based on the assumption that the success of the product is determined by the process. That this is not necessarily true is demonstrated by the success of certain commodity software.

It is useful to note that the International Standards Organisation [ISO90] has published a draft standard for a process model to assess the quality (including maintainability) of software. Many technical problems in measurement remain unsolved, however.

10. TECHNICAL ASPECTS OF SOFTWARE MAINTENANCE

10.1. Technical Issues

Much of the technology required for software maintenance is similar to that needed for initial development, with minor changes. For example, configuration management and version control are indispensable for both. Information relating to development and maintenance will be kept in a repository. For maintenance, the repository will be used to hold frequently occurring queries handled by the help desk. Metrics data for product and process will be similar. CASE tools, supporting graphical representation of software, are widely used in development and maintenance. These topics are described in other chapters of this book; here we concentrate on issues of specific importance to maintenance.

In our description of the IEEE standard process model, the need for impact analysis was identified. This is a characteristic of software maintenance that is not needed in initial software development. We shall present further details of this technique as an example of the technology needed to support software maintenance.

In the above process model, it was necessary to determine the cost of making a change, to meet a software change request. In this section we therefore examine how impact analysis can help this activity. To amplify the analysis needed, the user-expressed problem must first of all be translated into software terms to allow the maintenance team to decide if the problem is viable for further work or if it should be rejected. It then must be localized; this step determines the origin of the anomaly by identifying the primary components to the system that must be altered to meet the new requirement.

Next, the above step may suggest several solutions, all of which are viable. Each of these must be investigated, primarily by using impact analysis. The aim is to determine all changes that are consequent to the primary change. It must be applied to all software components, not just code. At the end of impact analysis, we are in the position to make a decision on the best implementation route or to make no change. Weiss [WEIS 89] has shown, for three NASA projects, the primary source of maintenance changes deriving from user problem reports:

Requirements phase	19%
Design phase	52%
Coding phase	7%

He noted that 34% of changes affected only one component and 26% affected two components.

10.2. The Problem

One of the major difficulties of software maintenance that encourages maintainers to be very cautious by nature is that a change made at one place in the system may have a ripple effect elsewhere, so consequent changes must be made. In order to carry out a consistent change, all such ripple effects must be investigated, the impact of the change assessed, and changes possibly made in all affected contexts. Yau defines this as:

> Ripple effect propagation is a phenomenon by which changes made to a software component along the software life cycle (specification, design, code or test phase) have a tendency to be felt in other components. [YAU87]

As a very simple example, a maintainer may wish to remove a redundant variable *X*. It is obviously necessary to remove all applied occurrences of *X* too, but for most high-level languages the compiler can detect and report undeclared variables. This is, hence, a very simple example of an impact that can be determined by *static analysis*. In many cases, ripple effects cannot be determined statically, and dynamic analysis must be used. For example, an assignment to an element of an array, followed by the use of a subscripted variable, may or may not represent a ripple effect depending on the particular elements accessed. In really large programs containing pointers, aliases, and so on, the problem is much harder. We shall define the problem of impact analysis as the task for assessing the effects for making the set of changes to a software system [WILD93].

The starting point for impact analysis is an explicit set of primary software objects that the maintainer intends to modify. He or she has determined the set by relating the change request to objects such as variables, assignments, and goals. The purpose of impact analysis is, hence, to ensure that the change has been correctly and consistently bounded. The impact analysis stage identifies a set of further objects impacted by changes in the primary sector. This process is repeated until no further candid objects can be identified.

10.3. Traceability

In general, we require traceability of information between various software artefacts in order to help us assess impact in software components. Traceability is defined as:

> Traceability is a degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor–successor or master–subordinate relationship to one another. [IEEE91]

Informally, traceability provides us with semantic links that we can then use to perform impact analysis. The links may relate similar components such as design documents or they may link between different types, for example, for a specification to code.

Some types of traceability links are very hard to determine. For example, altering the source code in even a minor way may have performance implications that cause a real-time system to fail to meet a specification. It is not surprising that the majority of work in impact analysis has been undertaken at the code level as this is the most tractable. Wilde [WILD89] provides a good review of code level impact analysis techniques.

Many modern programming languages are based on using static analysis to detect or stop the ripple effect. The use of modules with opaque types, for example, can prevent at compile time several unpleasant types of ripple effect. Many existing software systems are unfortunately written in older languages, using programming styles (such as global aliased variables) that make the potential for ripple effects much greater and their detection much harder.

More recently, Turver and Munro [TURV94] have described an approach that has placed impact analysis within the overall software maintenance process. The major advance is that documentation is included within the objects analysed; documentation is modeled using a ripple propagation graph and it is this representation that is used for analysis. The approach has the advantage that it may be set in the early stages of analysis to assess costs without reference to the source code.

Work has also been undertaken recently to establish traceability links between HOOD design documents [FILL94] in order to support impact analysis of the design level.

In a major research project at Durham, formal semantic-preserving transformations are being used to derive executable code from formal specifications and in reverse engineering to derive specifications from existing code. The ultimate objective is to undertake maintenance at the specification level rather than the code level, and generate executable code automatically or semiautomatically. The transformation technique supports the derivation of the formal traceability link between the two representations, and research is underway to explore this as a means of enhancing the ripple effect across wider sections of the life cycle (see, e.g., WARD93, WARD94, WARD94a, YOUN94, BENN95, and BENN95b for more details).

11. LEGACY SYSTEMS

11.1. Legacy Problems

There is no standard definition of a legacy system, but many in industry will recognize the problem. A legacy system is typically very old and has been heavily modified over the years to meet continually evolving needs. It is very large, so that a team is needed to support it; none of the team were around when the software was first developed. It will be based on old technology and be written in out-of-date languages such as Assembler. Documentation will not be available. Testing new releases is a major difficulty. Often, the system is supporting very large quantities of live data.

Such systems are surely a candidate for immediate replacement. The problem is that the software is often at the core of the business; replacing it would be a huge expense, and while less than ideal, the software works and continues to do useful things.

An example of a legacy system is the billing software for a telecommunications company. The software was developed 30 years ago, when the company was owned by the Government, and the basic service sold was restricted to a telephone connection to each premises. The system is the main mechanism for generating revenue; it supports a huge on-line database of paying customers.

Over the years, the software has been maintained to reflect the changing telecommunications business: from government to private ownership; from simple call charging to wide-ranging and complex (and competitive) services; from single country to international organization, with highly complex VAT (value added tax) systems. The system now comprises several million lines of source code.

Although the process of maintenance to meet continually evolving customer needs is becoming better understood, and more closely linked with software engineering in general, dealing with legacy software is still very hard. It has been estimated that there are 70 billion lines of COBOL in existence, and still doing useful work. Much of the useful software being written today will end up as legacy software in 20 years time. Software that is 40 years old is being used in mission-critical applications.

It is easy to argue that the industry should never have ended up in the position of relying on such software. It is not clear that steps are being taken to avoid the problem for modern software. There seems to be a hope that technology such as object-oriented design will solve the problems for future generations, though there is as yet little positive evidence for this.

In this section, we shall analyze why it might be useful not just to discard the legacy system and start again. In the subsequent section, we shall present solutions to dealing with legacy systems.

11.2. Analysis of Legacy Systems

In some cases, discarding the software and starting again may be the courageous, if expensive, solution, following analysis of the business need and direction, and the state of the software. Often, the starting point has to be taking an inventory of the software, as this may not be known. As a result of analysis, the following solutions for the legacy system may be considered:

- Carry on as now, possibly subcontracting the maintenance.
- Replace software with a package.
- Reimplement from scratch.
- Discard software and discontinue.
- Freeze maintenance and phase in new system.
- Encapsulate the old system and use as a server for the new system.
- Reverse engineer the legacy system and develop a new software suite.

In the literature, case studies addressing these types of approaches are becoming available. The interest of this tutorial is focused on reverse engineering, as it appears to be the most fruitful approach. Increasing interest is being shown in encapsulation as a way of drawing a boundary round the legacy system. The new system is then evolved so that it progressively takes over functionality from the old, until the latter becomes redundant. Currently, few successful studies have been published, but these support the move to distributed open systems based on client–server architectures.

11.3. Reverse Engineering

Chikofsky and Cross have defined several terms in this field that are now generally accepted. Reverse engineering is:

> . . . the process of analysing a subject system to identify the system's components and their inter-relationships, and to create representations of the system in another form or at higher levels of abstraction. [CHIK90]

It can be seen that reverse engineering is passive; it does not change the system or result in a new one, though it may add new representations to it. For example, a simple reverse engineering tool may produce call graphs and control flow graphs from source code. These are both higher-level abstractions, though in neither case is the original source code changed. Two important types of reverse engineering are redocumentation, which is the creation or revision of a semantically equivalent representation within the same relative abstraction layer, and design recovery, which involves identifying meaningful higher-level abstractions beyond those obtained directly by examining the system itself.

The main motivation is to provide help in program comprehension; most maintainers have little choice but to work with source codes, in the absence of any documentation. Concepts such as procedure structures and control flow are important mechanisms by which the maintainer understands the system, so tools have been constructed to provide representations to help the process.

If good documentation existed (including architectural, design, test suite documentation, etc.), reverse engineering would be unnecessary. However, the types of documentation needed for maintenance are probably different from those produced during typical initial development. As an example, most large systems are too big for one person to maintain, yet the maintainer rarely needs to see a functional decomposition or object structure; he or she is trying to correlate external behavior with internal descriptions. In these circumstances, *slicing* offers help. Slicing is a static analysis technique, in which only those source code statements that can affect a nominated variable are displayed.

Pragmatically, many maintainers cover line-printer listings with notes and stick-on pieces of paper. In an attempt to simulate this, Foster and Munro [FOST87] and Younger [YOUN93] have built tools to implement a hypertext form of documentation that is managed incrementally by the maintainer, who is able to attach "notes" to the source code. An advantage of this approach is that it does not attempt to redocument the whole system; documentation is provided, by the maintainer, in the form preferred, only for the "hot spots." Those parts of the code that are stable, and are never studied by the maintainer (often large parts), do not have to be redocumented, thereby saving money.

For a description of a reverse engineering method, see [EDWA95].

11.4. Program Comprehension

Program comprehension is a topic in its own right, and has stimulated an annual IEEE workshop. Documentation is also an active area; see, for example, Knuth's WEB [KNUT84] and also Gilmore [GILM90] for details of issues concerned with psychology. In [YOUN93], there is a useful list of criteria for software maintenance documentation:

- Integrated source code, via traceability links
- Integrated call graphs, control graphs, and so on
- Integration of existing documentation (if any)
- Incremental documentation
- Informal update by maintainer
- Quality assurance on the documentation
- Configuration management and version control of all representations
- Information hiding to allow abstraction
- Team use

It may be decided that active change of the legacy system is needed. Restructuring is the transformation from one representation to another at the same relative level of abstraction, while preserving the system's external behavior.

Lehman's second law argues that such remedial action is essential in a system that is undergoing maintenance, otherwise the maintainability will degrade and the cost of maintenance correspondingly increase. Examples include:

- Control flow restructuring to remove "spaghetti" code
- Converting monolithic code to use parameterized procedures
- Identifying modules and abstract data types
- Removing dead code and redundant variables
- Simplifying aliased/common and global variables

Finally, reengineering is the examination and alteration of the subject system to reconstitute it in a new form, and the subsequent implementation of the new form.

Reengineering is the most radical (and expensive) form. It is not likely to be motivated simply by wanting more maintainable software. For example, owners of on-line systems produced in the 1960s and 1970s would like to replace the existing character-based input/output with a modern graphical user interface. This is usually very difficult to achieve easily, so it may be necessary to undertake substantial redesign.

11.5. Reverse Engineering and Reengineering

In [BENN93], a list of 26 decision criteria for considering reverse engineering is presented. In abbreviated form, these are:

Management criteria

- Enforcing product and process standards (such as the IEEE draft standard introduced above)
- Permit better maintenance management
- Legal contesting of reverse engineering legislation
- Better audit trails

Quality criteria

- Simplification of complex software
- Facilitating detection of errors
- Removing side effects
- Improve code quality
- Undertaking major design repair correction
- Production of up-to-date documentation
- Preparing full test suites
- Improving performance
- Aligning with practices elsewhere in the company
- Financial auditing
- Facilitate quality audits (e.g., ISO9000)

Technical criteria

- To allow major changes to be made
- To discover the underlying business model
- To discover the design and requirements specification
- To port the system
- To establish a reuse library
- To introduce technical innovation such as fault tolerance, graphic interfaces, and so on
- To reflect evolving maintenance processes
- To record many different types of high-level representations
- To update tool support
- Disaster recovery

It is useful to amplify two of the above points. First, many legacy systems represent years of accumulated experience, and this experience may now no longer be represented anywhere else. Systems analysis cannot start with humans and hope to introduce automation; the initial point is the software that contains the business rules.

Second, it is not obvious that a legacy system, which has been modified over many years, does actually have a high-level, coherent representation. Is it simply the original system plus the aggregation of many accumulated changes? The evidence is not so pessimistic. The current system reflects a model of current reality, and it is that model we are trying to discover.

11.6. Techniques

Work on the simplification of control-flow and data-flow graphs has been undertaken for many years. A very early result showed that any control graph (using, e.g., unstructured go-to's) can be restructured into a semantically equivalent form using sequences, if-then-else-fi conditionals, and loops, although this may cause flag variables to be introduced. A good review of this type of approach can be found in the Redo Compendium [ZUYL93]. This work is generally mature, and commercial tools exist for extracting, displaying, and manipulating graphical representations of source code. In [WARD93], an approach using formal transformations is described that is intended to support the human maintainer, rather than act as a fully automated tool. This work shows that much better simplification is achievable, such as the conversion of monolithic code with aliased variables to well-structured code using parameterized procedures.

Much research in reverse engineering, especially in the United States, has been based on the program plan or cliché approach, pioneered by Rich and Waters [RICH90]. This is based on the recognition that many programs use a relatively small number of generic design ideas, which tend to be used over and over again. Reverse engineering should then attempt to find such plans in existing source code, by matching from a set of patterns in a library. This would appear to have had some modest success, but there are many open issues. For example, how should patterns be represented? How generic are they? And how good is the matching process? This approach shares many of the problems of libraries of reusable components.

Most researchers aim to make their approach source language independent, so that different languages may be handled by adding front ends. Thus design of intermediate languages is an important issue. In [ZUYL93], an approach called UNIFORM is described.

Ward [WARD93] uses a formally defined, wide-spectrum language, WSL, as the heart of his system. A wide-spectrum language is used as the representational format because only one language is then needed, for both low and high levels of abstractions and intermediary points. The approach has been shown to work for large (80K line) assembler programs, and also for very challenging benchmark cases such as the Schorr–Waite graph-marking algorithm. Further details are given in [BULL92] and [BULL94] ([BULL94] also contains a useful review of other transformation systems).

Cimitile and his colleagues have undertaken much research on producing tools and methods for discovering abstract data types in existing code [CANF94]. Sneed [SNEE91, also NYAR95, SNEE93] has presented his experience in reverse engineering of large commercial COBOL systems using partial tool support.

It is encouraging to observe that most new, promising approaches to reverse engineering address two basic properties of legacy systems:

- They are very large, and "toy" solutions are not applicable.
- They must be taken as they are, not how the engineer would like them to be. Often this means "one-off" solutions.

12. RESEARCH QUESTIONS

Although software maintenance tends to be regarded in academic circles as of minor importance, it is of major commercial and industrial significance. It is useful to end this tutorial with a brief review of promising trends.

There are many interesting research problems to be solved that can lead to important commercial benefits. There are also some grand challenges that lie at the heart of software engineering.

How do we change software quickly, reliably, and safely? In safety-critical systems, for example, enormous effort is expended in producing and validating software. If we wish to make a minor change to the software, do we have to completely repeat the validation or can we make the cost of the change proportional in some way to its size? There are several well-publicised cases in which very minor changes to important software have caused major crashes and failures in service. A connected problem lies in the measurement of how easily new software can be changed. Without this, it is difficult to purchase software in the knowledge that a reduced purchase price is not to be balanced by enormous maintenance costs later on. Almost certainly, the solution to this problem will involve addressing process issues as well as attributes of the product itself. This is a major problem for computer science. A new approach is described in [SMIT95].

In practice, much existing software has been evolved in ad-hoc ways, and has suffered the fate predicted by Lehman's laws. Despite their often central role in many organizations, such legacy systems provide a major headache. Management and technical solutions are needed to address the problems of legacy systems, otherwise, we shall be unable to move forward and introducing new technology because of our commitments and dependence on the old.

It is often thought that the move to end-user computing, open systems, and client service systems has removed this problem. In practice, it may well make it considerably worse. A system that is comprised of many components, from many different sources, by horizontal and vertical integration, and possibly across a widely distributed network, poses major problems when any of those components change. For further details of this issue, see [BENN94b].

13. PROFESSIONAL SUPPORT

Over the last 10 years, professional activity in software maintenance has increased considerably. The annual International Conference on Software Maintenance, sponsored by the IEEE, represents the major venue that brings academics and practitioners together to discuss and present the latest results and experiences. Also relevant is the IEEE workshop on program comprehension. The proceedings of both conferences are published by the IEEE.

In Europe, the main annual event is the annual European Workshop on Software Maintenance, organized in Durham. This is mainly aimed at practitioners, and, again, the proceedings are published.

There is a journal—the *Journal of Software Maintenance—Practice and Experience*—which appears bimonthly and acts as a journal of record for significant research and practice advances in the field.

Finally, aspects of software maintenance are increasingly being taught in university courses, and Ph.D. graduates are starting to appear who have undertaken research in the field.

14. CONCLUSIONS

We have described a three-level approach to considering software maintenance in terms of the impact on the organization, on the process, and on technology supporting that process. This has provided a framework with which to consider maintenance. Much progress has been made in all three areas and we have briefly described recent work on the establishment of a standard maintenance process model. The adoption of such models, along with formal process assessment and improvement, will do much to improve the best practice and average practice in the field of software maintenance.

We have also described a major problem that distinguishes software maintenance: coping with legacy systems. We have presented several practical techniques for addressing such systems.

Thus, we have presented software maintenance not as a problem but as a solution. However, there are still major research issues of strategic industrial importance to be solved. We have defined these as, first, to learn how evolve software quickly and cheaply, and, second, how to deal with large legacy systems. Whereas modern technologies such as object-oriented systems claim to improve the situation, this is largely a hope, and there is yet little evidence that they do indeed do so. Such technology may introduce new maintenance problems; see, for example, [SMIT92, TURN93, TURN95] for new testing methods associated with object-oriented programs. As usual, there are no magic bullets, and the Japanese principle of *Kaizen,* the progressive and incremental improvement of practices, is likely to be more successful.

ACKNOWLEDGEMENTS

Much of the work at Durham in software maintenance has been supported by SERC (now EPSRC) and DTI funding, together with major grants from IBM and British Telecom. I am grateful to colleagues at Durham for discussions that led to ideas presented in this paper, in particular to Martin Ward and Malcolm Munro. A number of key ideas have arisen from discussions with Pierrick Fillon. Thanks are due to Cornelia Boldyreff for reading drafts of this paper.

REFERENCES

[BABE91] Baber, R. L., Epilogue: Future Developments, in *Software Engineer's Reference Book,* Ed. McDermid, Butterworth-Heinemann, 1991.

[BENN93] Bennett, K. H., An Overview of Maintenance and Reverse Engineering, in *The REDO Compendium,* Ed. van Zuylen, Wiley, 1993.

[BENN94] Bennett, K. H., Software Maintenance in Japan. Report published under the auspices of the U.K. Department of Trade and Industry, September, 1994. Available from the Computer Science Department, University of Durham, South Road, Durham, DH1 3LE, UK.

[BENN94b] Bennett, K. H., Theory and Practice of Middle-out Programming to Support Program Understanding. in *Proceedings of IEEE Conference on Program Comprehension,* Washington, 1994, pp. 168–175

[BENN95] Bennett, K. H., and Ward, M. P., Formal Methods for Legacy Systems *Journal of Software Maintenance: Research and Practice, 7,* 3, 203–219, May–June 1995.

[BENN95b] Bennett, K. H., and Yang, H., Acquisition of ERA Models from Data Intensive Code in *Proceedings of IEEE International Conference on Software Maintenance,* Nice, France, October 1995, pp. 116–123.

[BOLD94] Boldyreff, C., Burd, E., and Hather, R., An Evaluation of the State of the Art for Application Management, in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, 1994, pp 161–169.

[BOLD95] Boldyreff, C., Burd, E., Hather, R. M., Mortimer, R. E., Munro, M., and Younger, E. J., The AMES Approach to Application Understanding: A Case Study, in *Proceedings of the International Conference on Software Maintenance,* IEEE Computer Society Press, 1995. pp. 182–191.

[BULL94] Bull, T., Software Maintenance by Program Transformation in a Wide Spectrum Language. Ph.D. thesis, Department of Computer Science, University of Durham, 1994.

[BULL92] Bull, T. M., Bennett, K. H., and Yang, H., A Transformation System for Maintenance—Turning Theory into Practice, in *Proceedings of IEEE Conference on Software Maintenance,* Orlando, Florida, USA, 1992.

[CANF94] Canfora, G., Cimitile, A., and Munro, M., RE2: Reverse Engineering and Reuse Re-engineering, *Journal of Software Maintenance: Research and Practice, 6,* 2, 53–72, March–April 1994.

[COLT87] Colter, M., The Business of Software Maintenance, in *Proceedings of First Workshop on Software maintenance,* University of Durham, Durham, 1987. Available from the Computer Science Department, University of Durham (see [BENN94]).

[DOWS85] Dowson, M., and Wilden, J. C., A Brief Report on the International Workshop on the Software Process and Software Environment, *ACM Software Engineering Notes, 10,* 19–23, 1985.

[EDWA95] Edwards, H. M., Munro, M., and West, R., *The RECAST Method for Reverse Engineering,* Information Systems Engineering Library, CCTA, HMSO, ISBN:1 85 554705 8, 1995.

[FILL94] Fillon, P., *An Approach to Impact Analysis in Software Maintenance,* M.Sc. Thesis , University of Durham, 1994.

[FOST87] Foster, J., and Munro, M., A Documentation Method Based on Cross-referencing, in *Proceedings of IEEE Conference on Software Maintenance,* Austin, Texas, 1990.

[FOST93] Foster, J., *Cost Factors in Software Maintenance,* Ph.D. thesis, Computer Science Department, University of Durham, 1993.

[GILM90] Gilmore, D., Expert Programming Knowledge: A Strategic Approach, in *Psychology of Programming,* Ed. Hoc, J. M., Green, T. R. G., Samurcay, R., and Gilmore, D. J., Academic Press, 1990.

[HATH94] Hather, R, Burd, L., and Boldyreff, C., A Method for Application Management Maturity Assessment, in *Proceedings of Centre for Software Reliability Conference,* Dublin, 1994.

[HINL92] Hinley, D. S., and Bennett, K. H., Developing a Model to Manage the Software Maintenance Process, in *Proceedings of Conference on Software Maintenance,* Orlando, Florida, IEEE Computer Society Press, 1992.

[IEEE91] IEEE Std. 610.12-1990, *IEEE Standard Glossary of Software Engineering Terminology,* IEEE, 1991.

[IEEE98] *IEEE Std. 1219-1998, Standard for Software Maintenance.* IEEE, 1998.

[ISO90] *International Standards Organisation Information Technology—Software product evaluation—Quality characteristics and Guidelines for Their Use.* ISO/IEC JTC1 Draft International Standard 9126.

[ITIL93] *The IT Infrastructure Library,* Central Computer and Telecommunications Agency, Gildengate House, Upper Green Lane, Norwich, NR3 1DW.

[KNUT84] Knuth, D. E., Literate programming, *Computer Journal, 27,* 2, 97–111, 1984.

[LEHE80] Lehman, M. M., Programs, Lifecycles, and the Laws of Software Evolution, in *Proceedings of IEEE, 19,* 1060–1076, 1980.

[LEHE84] Lehman, M. M., Program Evolution, *Information Processing Management, 20,* 19–36, 1984.

[LEIN78] Lientz, B., Swanson, E. B., and Tompkins, G. E., Characteristics of Applications Software Maintenance, *Communications of the ACM, 21,* 466–471, 1978.

[LEIN80] Leintz, B., and Swanson, E. B., *Software Maintenance Management,* Addison-Wesley, 1980.

[McDER91] SERB McDermid, J. (Ed.), *Software Engineering Reference Book,* Butterworth-Heinemann, 1991.

[LEVE93] Leveson, N. G., and Turner, C. S., An Investigation of the Therac-25 Accidents, *IEEE Computer, 26,* 7, 18–41, July 1993.

[NYAR95] Nyary, E., and Sneed, H., Software Maintenance Offloading at the Union Bank of Switzerland, in *Proceedings of IEEE International Conference on Software Maintenance,* Nice, France, pp. 98–108 October 1995.

[RICH90] Rich, C., and Waters, R. C., *The Programmer's Apprentice,* Addison-Wesley, 1990.

[ROBS91] Robson, D. J., Bennett, K. H., Cornelius, B. J., and Munro, M., Approaches to program Comprehension, *Journal of Systems Software, 14,* 1, 1991.

[SMIT92] Smith, M. D., and Robson, D. J., A Framework for Testing Object-Oriented Programs, *Journal of Object-Oriented Programming, 5,* 3, 45–53, June 1992.

[SMIT95] Smith, S. R., Bennett, K. H., and Boldyreff, C., Is Maintenance Ready for Evolution? in *Proceedings of IEEE International Conference on Software Maintenance,* Nice, pp. 367–372, October 1995 IEEE Computer Society Press, (1995).

[SNEE91] Sneed, H., Economics of Software Re-engineering, *Journal of Software Maintenance: Research and Practice, 3,* 3, 163–182, Sept. 1991.

[SNEE93] Sneed, H., and Nyary, E., Downsizing Large Application Programs, in *Proceedings of IEEE International Conference on Software Maintenance,* Montreal, 1993, pp. 110–119, IEEE Computer Society Press, 1995.

[TURN93] Turner, C. D., and Robson, D. J., The State-based Testing of Object-Oriented Programs, in *Proceedings of IEEE Conference on Software Maintenance, Montreal, September 1993, 302–310.*

[TURN95] Turner, C. D., and Robson, D. J., A State-Based Approach to the Testing of Class-Based Programs, Software—Concepts and Tools, 16, 3, 106–112, 1995.

[TURV94] Turver, R. J., and Munro, M., An Early Impact Analysis Technique for Software Maintenance, *Journal of Software Maintenance: Research and Practice, 6,* 1, 35–52, Jan. 1994.

[WALT94] Walton, D. S., Maintainability Metrics, in *Proceedings of Centre for Software Reliability Conference,* Dublin, 1994. Available from Centre for Software Reliability, City University, London, U.K.

[WARD93] Ward, M. P., Abstracting a Specification from Code, *Journal of Software Maintenance: Practice and Experience, 5,* 2, 101–122, June 1993.

[WARD94] Ward, M. P., Reverse Engineering through Formal Transformation, *Computer Journal, 37,* 9, 1994.

[WARD94a] Ward, M. P., Language Oriented Programming, *Software—Concepts and Tools,* 15, 147–161, 1994.

[WEIS89] Weiss, D. M., *Evaluating Software Development by Analysis of Change,* Ph.D. dissertation, University of Maryland, USA.

[WILD93] Wilde, N., Software Impact Analysis: Processes and Issues, Durham University Technical Report 7/93, 1993.

[YAU87]Yau, S. S., and Liu, S., Some Approaches to Logical Ripple Effect Analysis, Technical Report, SERC, USA, 1987.

[YOUN93] Younger, E., Documentation, in *The REDO Compendium,* Ed. van Zuylen, Wiley, 1993.

[YOUN94] Younger, E., and Ward, M. P., Inverse Engineering a Simple Real Time Program, *Journal of Software Maintenance: Research and Practice, 6,* 197–234, 1994.

[ZUYL93] van Zuylen, H. (Ed.), *The REDO Compendium,* Wiley, 1993.