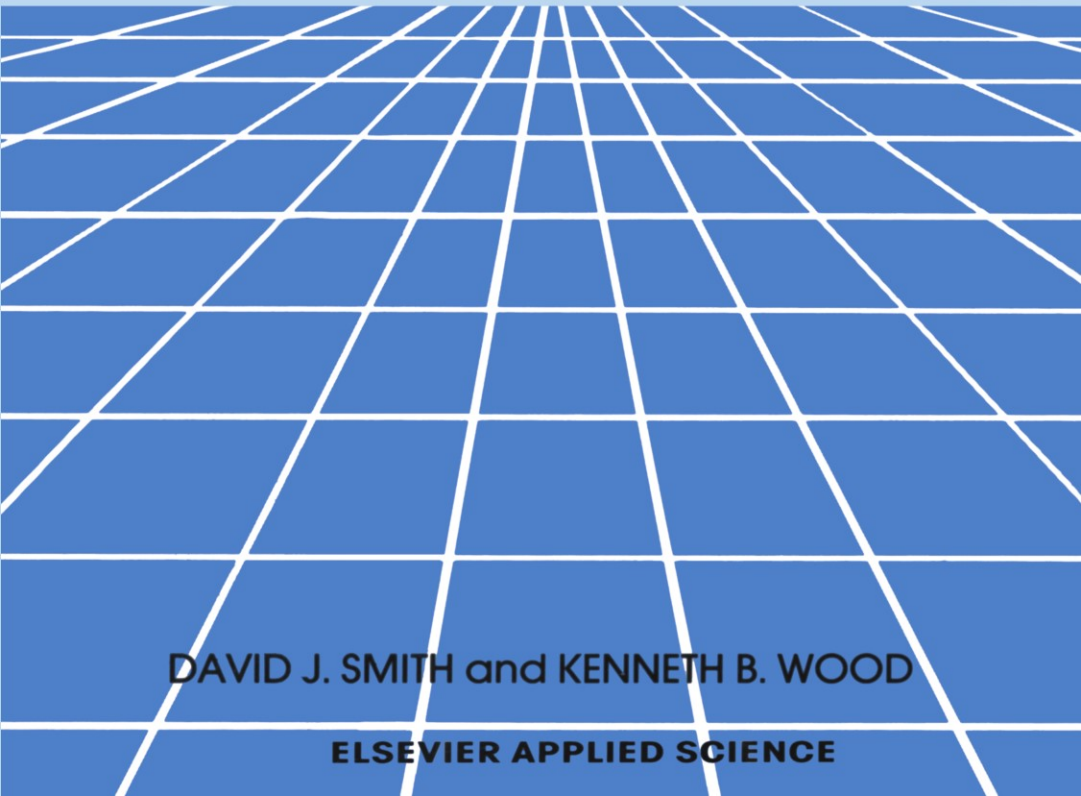
ENGINEERING QUALITY SOFTWARE

A Review of Current Practices
Standards and Guidelines including
New Methods and Development Tools

SECOND EDITION



DAVID J. SMITH and KENNETH B. WOOD

ELSEVIER APPLIED SCIENCE

ENGINEERING QUALITY SOFTWARE

Second Edition

ENGINEERING QUALITY SOFTWARE

A Review of Current Practices, Standards and Guidelines including New Methods and Development Tools

Second Edition

DAVID J. SMITH
B.Sc., C.Eng., F.I.E.E., F.I.Q.A., F.Sa.R.S.
Tonbridge, Kent, UK

and

KENNETH B. WOOD
Fleet, Hampshire, UK

ELSEVIER APPLIED SCIENCE
LONDON and NEW YORK

ELSEVIER SCIENCE PUBLISHERS LTD
Crown House, Linton Road, Barking, Essex IG11 8JU, England

Sole Distributor in the USA and Canada
ELSEVIER SCIENCE PUBLISHING CO., INC.
655 Avenue of the Americas, New York, NY 10010, USA

WITH 3 TABLES AND 36 ILLUSTRATIONS

First Edition 1989
Reprinted 1990

Softcover reprint of the hardcover 1st edition 1986

British Library Cataloguing in Publication Data

Smith, David J. (David), 1946–
Engineering quality software.
1. Computer systems. Software. Quality control.
I. Title II. Wood, Kenneth B.
05'.14

ISBN-13:978-94-010-6996-0 e-ISBN-13:978-94-009-1121-5
DOI: 10.1007/978-94-009-1121-5

Library of Congress Cataloging-in-Publication Data

Smith, David John, 1943–
Engineering quality software: a review of current practices, standards, and guidelines, including new methods and development tools / David J. Smith and Kenneth B. Wood.—2nd ed.
p. cm.
Bibliography: p.
Includes index.
ISBN-13:978-94-010-6996-0
1. Software engineering. 2. Computer software—Quality control.
I. Wood, Kenneth B. II. Title.
QA76.758.S55 1989
005.1—dc20

Preface to the Second Edition

During the 18 months since the publication of the 1st edition the practice of software quality and the availability of tools and guidance for its implementation has increased dramatically.

The emphasis on the need for formal methods has increased and calls for certification of safety critical software are now common. In particular this 2nd edition:

—Expands the treatment of static analysis and includes a comprehensive but simple example in order to illustrate clearly the functions of each analyser in Chapter 8.
—Describes formal requirements languages more fully in Chapter 6.
—Updates the compendium of available guidelines and standards in Chapter 5.
—Expands the description of the many high level languages in Chapter 9.
—Improves and expands the exercise into a 49 page case study consisting of a documentation hierarchy for a safety system in Chapter 14. It is seeded with deliberate errors and ambiguities and now includes guidance in finding them.

D. J. Smith
K. B. Wood

Preface to the First Edition

Towards the end of the 1960s there was an awareness of the problem of software-related failures. It became clear that, with tolerably reliable hardware, those failures arising purely from aspects of the software could dominate the total number. Furthermore, compared with hardware failures they were extremely difficult to diagnose and impossible to predict due to the limited visibility which exists in computer software. There was, even at that time, a general feeling that they related to the structure of the design process which generated the software.

The result of this was that, by the late 1960s, the term 'structured programming' had emerged. This heralded the development of block-structured high level languages (Chapter 9) as a means of disciplining the production of code. Various programming standards and methods were created around this idea.

Later in the 1970s the test phase became more formalised and automated test tools were introduced. It was realised then that test specifications should be generated from a formal analysis of the requirements and of the design as it progressed since, with software, test relies on exercising each function, and combinations of functions, rather than simply accumulating test time.

By now the software quality activity was an established feature of the better organisations and various systems, standards and guidelines began to emerge (Chapters 4 and 5).

In the early 1980s it was thought that the design review activities similar to code inspection and structured walkthrough should be automated. The result was the development of static analysers which are a major contribution to the production of error-free code (Chapter 8).

Ironically, the most important feature of software design—the

requirements specification—is only now receiving adequate attention. Hitherto, and for that matter now, requirements have been expressed in natural language wherein lies the potential for the ambiguities and omissions which ultimately lead to software failures. Chapter 6 deals with the formal techniques which are emerging in this area.

The following dates help to put this into perspective.

1965	The problem revealed.
1970–1975	Structured programming.
1975–1980	Testing tools.
1980–1985	Code verification and validation.
1985 onwards	Formal specification expression.

Software development must evolve by means of more formally structured and automated techniques. As a result, the role of the software engineer will evolve from programmer to that of software and systems engineer.

This book reviews the current state of the art in Parts 1 and 2 and looks forward in Parts 3 and 4.

D. J. Smith
K. B. Wood

Contents

PART 3. SOFTWARE QUALITY ENGINEERING—AN IDEAL APPROACH

Acknowledgements

Particular thanks are due to two good friends and colleagues:

Mike Forrester, whose informed and thorough criticism led to significant improvements in the text; and

Len Nohre, whose painstaking study of the manuscript revealed many points for discussion and subsequent improvement.

Thanks are due to Stuart Pegler and Barry Price of British Gas Midlands Research Station for helpful comments on Chapter 10.

Thanks also go to Ron Bell of the Health and Safety Executive for reviewing Chapter 5 and for assistance with the section relating to safety guidelines.

Bob Malcolm of CAP kindly allowed us to use his 'Get in boats' illustration of structured programming in Chapter 4.

The second edition owes much to RTP Software Ltd, of Farnham, Surrey for permission to make use of the MALPAS static analysis example in Chapter 8.

Many thanks are also due to Dr Paul W. Banks and Mr John Dixon for their major contributions to the case study which is Chapter 14.

Last, but by no means least, we must thank our wives, who have had much to contend with.

PART 1

The Background to Software Engineering and Quality

The first three chapters define the terms which are necessary for an understanding of the subject. Causes of failure are introduced as well as the idea of the software design cycle. The basic software quality problem is explained. Safety critical software is addressed.

Chapter 1

The Meaning of Quality in Software

1.1 QUALITY—WHAT IS IT?

The popular view of quality is still a subjective concept which perpetuates the idea that the more elaborate and complex products somehow offer a higher level of quality than their humbler counterparts. Whilst this misconception is well understood amongst 'quality' professionals the temptation remains to equate sophistication, instead of simplicity of function, with quality.

Traditionally, quality is defined as the adherence to some agreed specified performance. It follows that failures are observed as items where the specification is not met.

A 'good' product requires not only conformance to the specification but a 'good' specification in the first place. It is not unknown for a good quality product to emerge despite an inadequate specification but it must be said that this arises purely as a result of enlightened design and is by no means a desirable state of affairs. On the other hand the 'good' specification is no guarantee of an adequate product since the task of achieving conformance still remains.

This rigid concept therefore requires that the total performance criteria are foreseen since failures cannot be said to occur in respect of functions which have not been defined.

A problem arises in respect of our ability to foresee and define these numerous and complex requirements, to say nothing of the complications arising from the combination of operating modes and environments. As systems become larger and more complex this difficulty becomes more evident and often the result is prolonged negotiation resulting in modifications to the requirements specification lasting throughout the design and even into the test and commissioning phases.

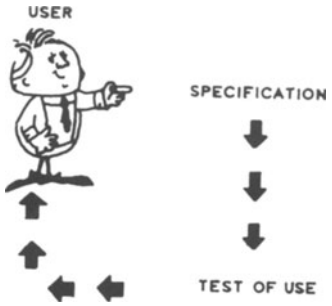


Fig. 1.1.

Figure 1.1 illustrates the simple concept of comparison with a specification as a means of a demonstrating adherence to quality. Figure 1.2, however, introduces the additional complication arising from

(a) Our limited ability to foresee the requirement;
(b) Our actual perception of the performance.

The problem, which for that matter is not confined to software engineering, is that the specification is a statement of what a potential user perceives as the field requirements. Since the specification is limited to one's perception of these requirements it is unlikely that it will cover all needs and eventualities. This is compounded by the further difficulty in interpreting the actual field performance since incidents, or 'failures', are seldom observed when they occur but are seen as 'second-hand' events.

A significant proportion of 'failures' are either:

Unwanted incidents which are not catered for in the specification and are therefore formally not failures,

or

The result of various data values or program states which are no longer available and therefore cannot be diagnosed.

The quality tasks facing the Software Engineer are therefore:

(1) Verify the requirement;
(2) Validate the design;
(3) Perform adequate tests.

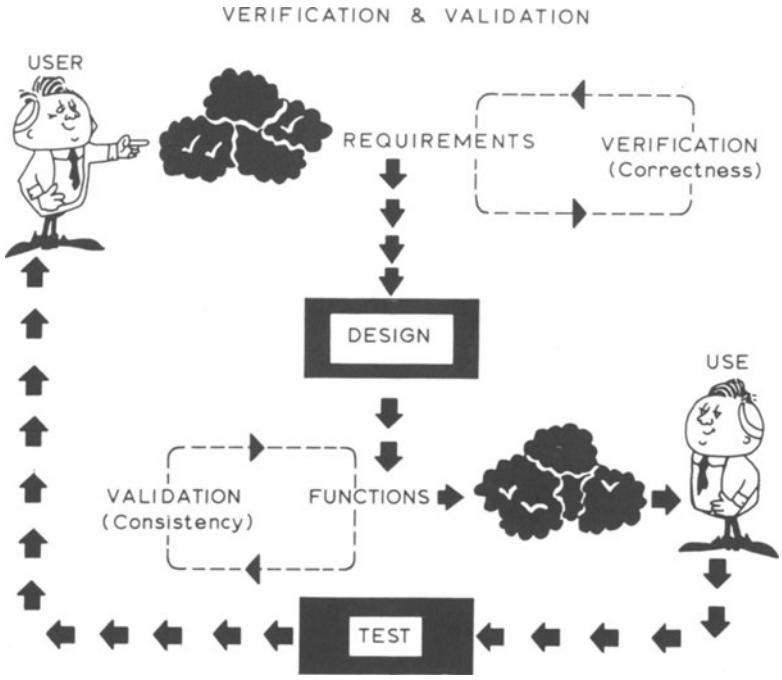


Fig. 1.2. The quality problem.

1.2 QUALITY—THE ELUSIVE ELEMENT

Quality is an elusive feature of a product. The main reason, as we have seen, stems from the inability to specify a requirement in its entirety. The perception of quality in software is seen principally in terms of the time that a software system operates 'correctly'.

At first sight software reliability might be easier to achieve than for hardware since Fig. 1.3 reminds us that, in hardware, failures arise from three basic causes which include the design itself:

(a) Early failures related to manufacturing imperfections, in other words populations of inherent failures due to microscopic flaws;
(b) So-called random failures, assumed to be due to fluctuations in stress;
(c) Wearout failures due to mechanisms of physical change.

Due to the abstract nature of software it can show neither of the last

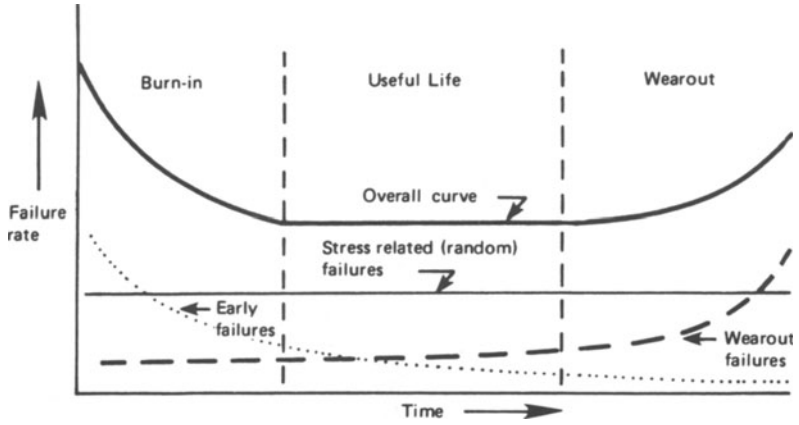


Fig. 1.3.

two characteristics since it has no physical entity. There remains, then, only that inherent population of defects which arise from being unable to foresee the total logic of a software package.

It is far easier to address the reliability of hardware designs when the object is purely to eliminate undesirable interactions between component parts. It does not require a genius to spot the stupidity of a common pneumatic supply feeding two 'independent' braking systems. Independence is not, however, so obvious in complex software programs; thus the equivalent type of interacting software fault is frequently found.

Furthermore, modifications are a fact of life in engineering and many features are added to programs well after their original conception. The inclusion of code to bridge boundaries between previously isolated program modules is thus a real possibility.

1.3 THE SOFTWARE PROCESS—CRAFT OR SCIENCE?

The piecemeal 'craft' approach to software design, otherwise known as programming, has much in common with its historic counterpart. Craftsmen, and more recently the electronic designers of the 1960s, acquired their skills by a combination of imitation and practice. They were unwilling to document their activities and preferred to work alone. Although the skills and broad patterns of design were carried from task to task, two designs were seldom if ever identical. In

software this tendency leads to poor design and error-prone code since the benefits of evolving standard reliable packages are not readily proliferated from design to design or from programmer to programmer.

The last 25 years have seen a radical change in approach to electronic hardware design. The vast size and complexity of these electronic packages no longer permit the use of such undisciplined methods. Recognised circuit functions are performed by proven packages and components which have become readily accepted standards used by electronic engineers.

It would be good to report that computer programming has also undergone this transformation and that software is generated by means of proven structures and standard routines. Alas, this is the exception rather than the rule, although there are signs that we are moving in the right direction and the remainder of this book will describe the current practice as well as indicating future trends.

Much lip-service is paid to the need for formal methods and these are currently receiving much attention in both the academic and industrial worlds. They extend to the requirements specification level already discussed and both logical and mathematical rules are being applied to this activity. An engineering approach to the specification and to programming is thus beginning to appear and will ultimately allow products to be defined and developed with the same confidence that applies to physical designs. However, the full realisation of this ideal is still very much part of a vision of things to come.

This need for formalising structures is reflected in much current teaching, and already some programming curricula and textbooks advocate and explain these techniques. An embryo generation of software engineers is thus at hand.

1.4 BLENDING ENGINEERING DISCIPLINE AND SOFTWARE DESIGN

The tailoring of traditional quality disciplines to the software design process has resulted in a 'checklist' approach to software quality. Although this has led to significant benefits it by no means guarantees error-free code and, furthermore, it has helped to perpetuate the misconception that high software reliability can be achieved solely by the application of these qualitative quality control techniques. In any

case there is no accurate means of quantifying the potential improvement in error rate which will result from such an approach.

Two fundamental changes to the approach are needed.

In the first case formal methods need to be seen as an integral feature of an engineering design process to be applied by the system designers themselves. Additional levels of surveillance will be of value but the primary exponents of quality-related techniques must be the designers. This requires a fundamental change in the training and motivation of software staff.

Secondly, more formally based automatic tools are needed in validating and verifying the correctness of specifications and of code if realistic confidence levels are to be established during the production of systems.

Requirements specification methodologies, static and dynamic test tools and higher level structured languages will all contribute to this improvement.

We may expect to see a radical change in software design management. Key areas are:

Formal specification. The use of mathematical (probably automated) methods.

Design. Interfaces between modules will be better defined. Libraries of re-usable software will exist. Proof of correctness of specifications and verification of code will be largely automated or achieved during the design process.

Test. Test specifications will be generated by and from the formal languages. Test beds will be automated. Tests will be static and dynamic (Chapter 8).

Modifications. Since requirements change with time then so does the code. Formal methods supported by tools will make it easier to determine which parts of the program need to be changed and it becomes less likely that bugs will propagate.

1.5 THE CONFLICT BETWEEN QUALITY AND TIME

The cost of quality and reliability is no new concept. It recognises the trade-off between the costs incurred by these techniques and the penalty costs of failures. Experience suggests that the curve in Fig. 1.4 is biased towards the right and that failure costs far outweigh those of prevention.

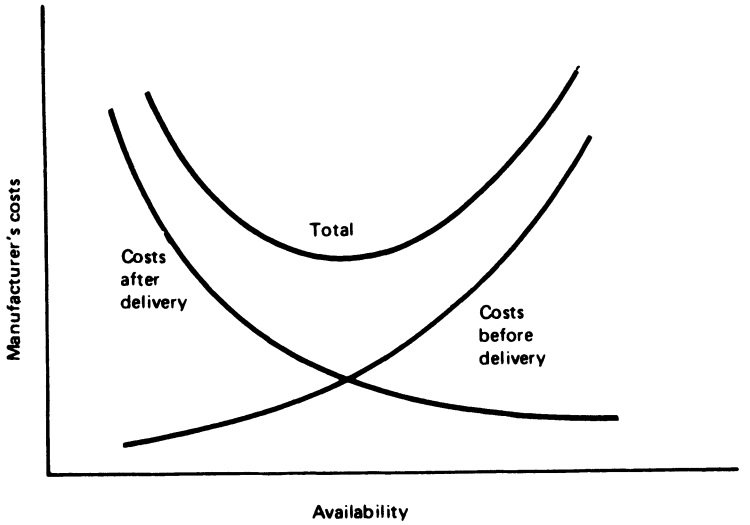


Fig. 1.4.

Figure 1.5 reinforces this view by reminding us that the cost of failure prevention is lower, and for that matter less uncertain, the earlier that it is incurred. Pennies spent during the design phase therefore reap better dividends than pounds wasted on diagnosis and modification later on.

Activities, however, have to be planned and the temptation is to save money by saving time during specification and design. The temptation to begin coding too early is often a result of overwhelming

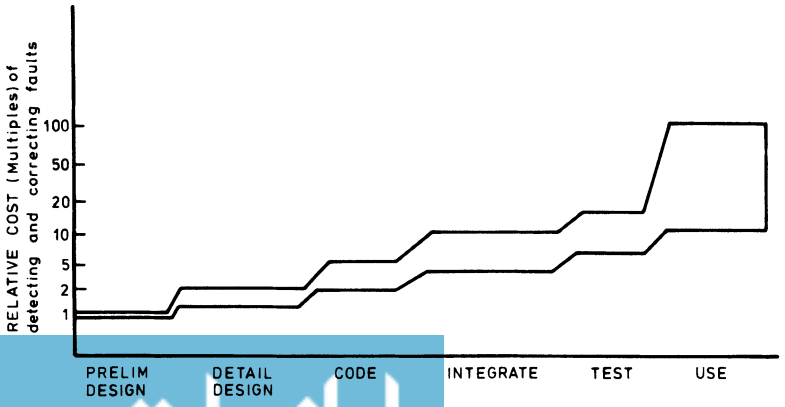


Fig. 1.5.

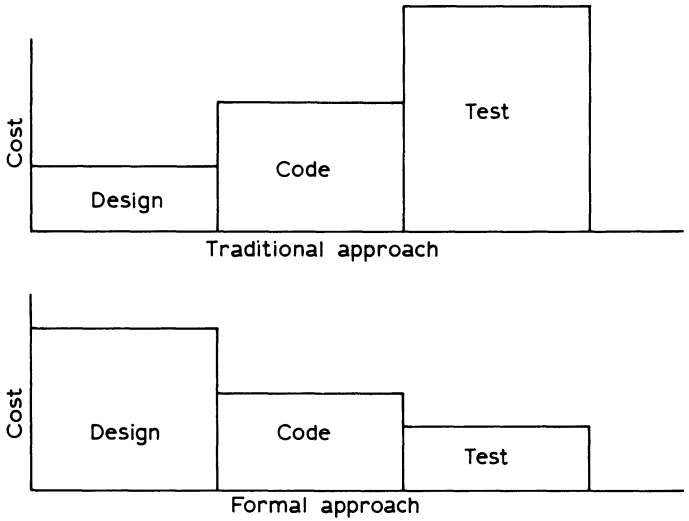


Fig. 1.6.

schedule and commercial pressures. It is then too late to recover the situation by adding additional manpower once the problems begin. In fact, the addition of programmers, as a remedial measure, usually results in delaying rather than improving the schedule.

The message is clear. Formal methods must be employed and the appropriate resources committed at the beginning of the project. The evidence is also clear.

This approach will result in more reliable software at a lower schedule cost than by less disciplined approaches. Would that every project manager agreed!!

Figure 1.6 compares the distribution of costs between the traditional approach and the more formal methods. More expenditure on design simplifies the coding process and substantially reduces the number of errors to be revealed during test. The total cost and schedule is thus reduced. The following chapters will outline the traditional methods and explain the newer formal techniques and tools.

1.6 THE DECLINE OF HARDWARE AND THE RISE OF SOFTWARE

The last two decades have witnessed a rapid transition from wholly hard-wired electronic systems in which functions have been executed

as circuit features towards a single-circuit architecture which can execute, via software, a vast range of hitherto specific circuit functions. It is this 'multi-application' feature of the computer architecture which has given rise to

(a) The increase of functions per unit of hardware.
(b) The favouring of software solutions as the preferred method of system design.

The original goal which motivated and catalysed this trend was the promise of flexible and lower-cost systems whereby hardware was minimised and easily modified functions were achieved through low-cost programs. The reality, however, is that these benefits have been largely thrown away due to the poor economy arising from unstructured and poorly managed software development.

The current scenario is therefore one of largely software-implemented functions whereby costs remain too high due to inefficient methods.

Only by the introduction and use of formal methods and automated programming tools will these potential economies be realised through the reduction of the currently excessive test and debug activities. Unless these changes take place, there will be a tendency to revert to hardwired solutions even though they may be less efficient. The greater predictability of such solutions will be a positive attraction, particularly in safety critical systems. The 'unpredictability' of software is already leading to the introduction of 'hardware feedback', that is to say diversity, in order to prevent software failures from leading to hazardous situations. This is in no small measure a result of the poor record of software development in recent years.

Chapter 2

Software Failures—Causes and Hazards

2.1 ADVANTAGES AND DISADVANTAGES OF PROGRAMMABLE SYSTEMS

A programmable system is any equipment or device which, having a computer architecture (i.e. arithmetic and logic capability plus a memory), relies on a set of sequential programmed instructions in order to function. This is known as a Von Neumann architecture. The set of logical commands is referred to as software and usually consists of binary numbers stored within the system. The term software also embraces the design documents which are needed in order to produce this code or program.

Some of the types of programmable system are typified by computing and real time control applications—both of these are software systems—and can be seen in three broad categories:

(a) *Mainframe computing*. This can best be visualised in terms of systems which provide a very large number of terminals (several hundred) and support a variety of concurrent tasks. Typical functions provided are interactive desktop terminals or bank terminals. Such systems are also characterised by the available disc and tape storage which often runs into hundreds of megabytes.

(b) *Minicomputing*. Here we are dealing with a system whose CPU may well deal with the same word length (32 bits) as the mainframe. The principal difference lies in the architecture of the main components and, also, in the way in which it communicates with the peripherals. A minicomputer can often be viewed as a system with a well-defined hardware interface to the outside world enabling it to be used for process monitoring and control.

(c) *Microprocessing*. The advent of the microcomputer is relatively recent but it is now possible to have a 32-bit architecture machine as a desktop computer. These systems are beginning to encroach on the minicomputer area but are typically being used as 'personal computers' or as sophisticated work stations for programming, calculating, providing access to mainframes, and so on.

The boundaries between the above categories have blurred considerably in recent years to the extent that minicomputers now provide the mainframe performance of a few years ago. Similarly microcomputers provide the facilities recently expected from minis.

The complexity of the architecture has increased in all three cases and this has, to some extent, maintained a difference between them. One example is the use of RISC (Reduced Instruction Set Computers). These are architectures in 32-bit microcomputers which have provided a considerable performance enhancement. They involve fewer instructions in the set but more efficient processing (see 10.6). At the other end of the spectrum 'pipelining architectures' and specialised 'bolt-on' systems, to enhance number crunching, use parallel architectures and are becoming the norm. Pipelining involves more sophisticated CPUs for which the FETCH instruction brings a number of instructions into a buffer for speedier execution.

Real time applications have expanded more than any other technology in recent years. These include:

Communications
Telephone signalling, mobile radio, telemetry, satellite.

Domestic
Appliance control and timing, security.

Transport
Auto landing, railway signalling, fuel management.

Energy
Process shut-down and control, fire detection, power stabilisation, telemetry.

Health
Diagnostics, body monitoring, blood analysis.

Industry
Robotics, process data gathering, process control, automated clerical and stock control methods.

Commerce
Word processing, calculations, elevators, graphics.

Finance

Banking systems, dealing systems, cash dispensers.

From both reliability and operating standpoints there are advantages and disadvantages arising from programmable devices.

Reliability advantages

Less hardware (fewer devices) per circuit due to high levels of integration.
Fewer device types.
Consistent architecture (configuration) leading to a common approach to hardware design.
Easier to support several models in the field with spares.
Simpler to modify or reconfigure.
Provides a running log for investigation.
Allows self-test.

Safety advantages

Removes human operators from hazardous areas.
Provides sophisticated process interlocks.
Interprets rates of change of process parameters and gives timely warning of potentially hazardous conditions.
Provides early warning diagnostics.
Provides centralised displays and graphics on VDUs.

Reliability and safety disadvantages

Difficult to 'inspect' software for inherent faults.
Difficult to impose standard approaches to software design.
Difficult to control software changes.
Testing and validation of high-scale integration devices and their associated software is difficult owing to the high package density and consequent lack of visibility and interface to the functions.
Impossible to predict software failure modes and rates.
Exhaustive testing is impossible because of time constraints since the number of permutations in software is extremely high.
More susceptible to common-mode failures.
Easier to corrupt data and programs.

Various features of software have a direct bearing on its quality. These will be addressed throughout this book and include:

Usability and ease of diagnosis.
Traceability of requirements through the code.
Visibility of functions in the code.

Documentation clarity and consistency.
Spare capacity in timing and memory resources.
Fault tolerant design techniques.
Ability to operate in a degraded mode under fault conditions.

2.2 SOFTWARE-RELATED FAILURES—FAULT, ERROR, FAILURE

The question arises as to how a software failure is defined. Unlike hardware failures there is no physical change which causes a unit to

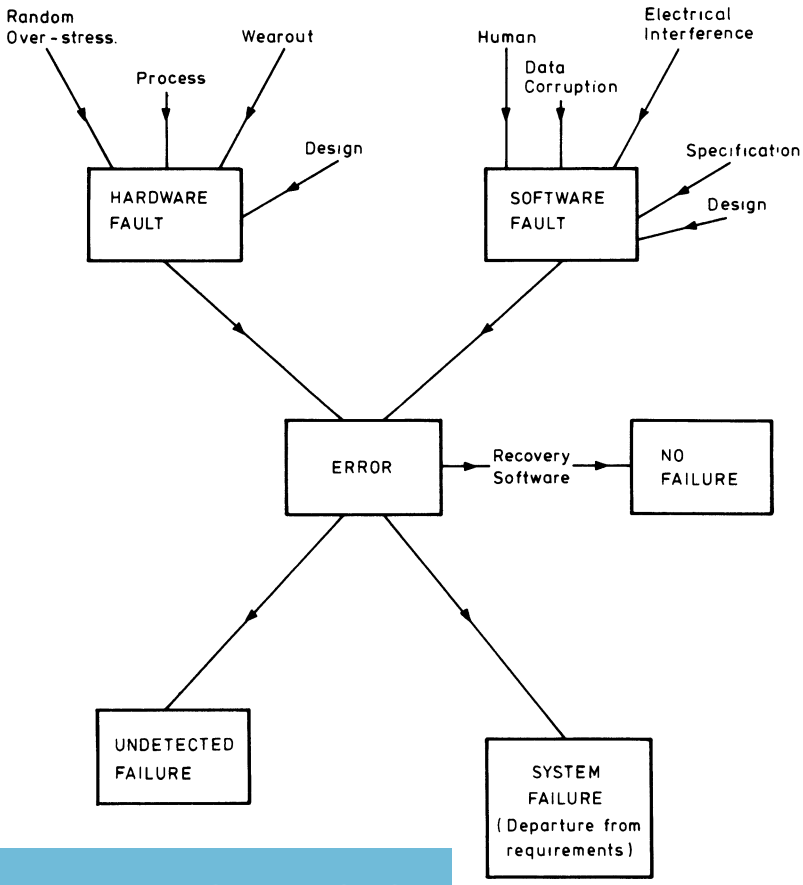


Fig. 2.1.

cease functioning. Software failures are in fact errors which, owing to the complexity of programs, do not always become evident immediately. Unlike the hardware 'bathtub curve', there is no wearout feature since the population of bugs can only (save for modifications) decrease. Figure 2.1 illustrates the concept of fault/error/failure.

Faults may occur in both hardware and software. Software faults—often known as *bugs*—will arise as a result of particular parts of the code being used for the first time or because of corruption due to some outside influence.

The presence of a fault in a program does not necessarily result in either an error or failure. A long time may elapse before that portion of the code is used under the circumstances which lead to failure.

A fault (bug) may lead to an *error*. This is the condition whereby the system is in an incorrect state. A data value or an instruction is thus incorrect and only when that particular part of the code is executed is the error revealed.

An error may propagate to become a *failure* if the system does not contain some error recovery logic capable of dealing with and minimising the effect of the error.

A failure, be it hardware- or software-related, is the termination of the ability of an item to perform its specified function.

2.3 CAUSES OF FAULTS

Faults are caused at all stages in the specification, design and coding process. There is evidence that the majority of errors (over 60%) are committed during the requirements and design phases. The remaining 40% occur during coding. That is not to say that coding is not a part of the design but it is only the final activity in a much larger process. The more complex the system the more faults will be likely to stem from ambiguities and omissions in the specification stages. The major sources of fault are:

(a) *From the requirement specification*
  Incorrect requirements due to:
    Model not a good fit to the physical situation.
    Incorrect document cross-references.
  Inconsistent or incompatible requirements:
    Two references give conflicting information.
    Conventions not consistent.

Requirement unclear or illogical.
Requirement omitted (e.g. handling of invalid inputs).

(b) *From the design*

Unstructured approach to the design breakdown (i.e. detail is considered first).
Lack of proper reviews.
Lack of change control.
Specification was misunderstood.

(c) *From coding*

Semantic errors involving incorrect use of statements.

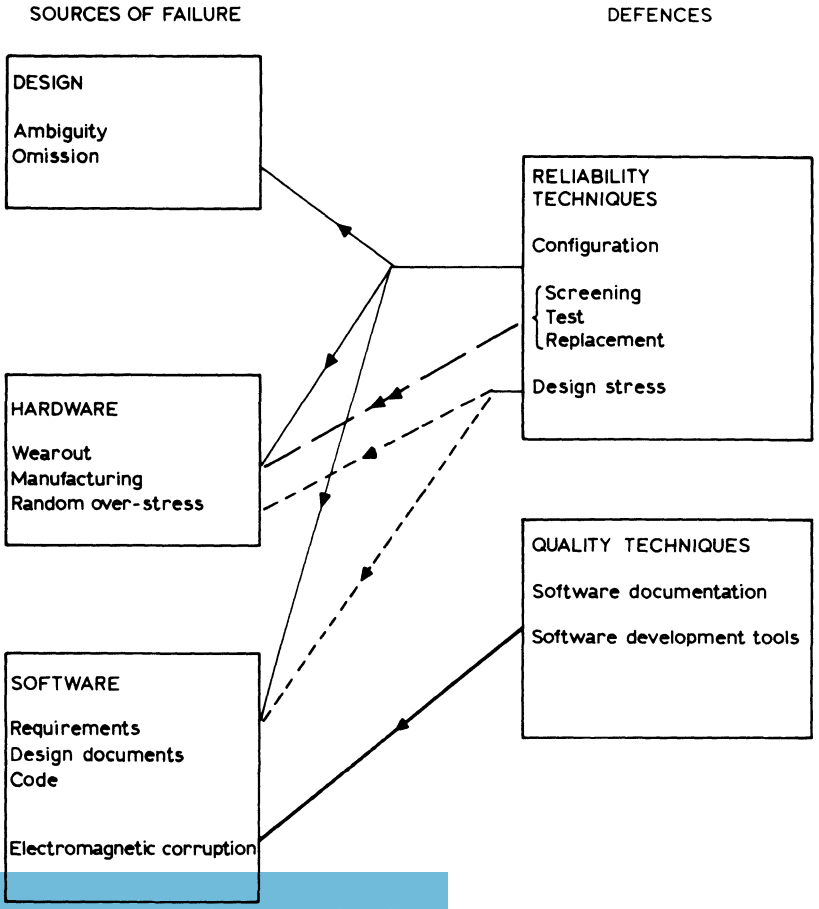


Fig. 2.2.

Logical errors in translating the design into code.
Detailed syntax errors which may have escaped detection by the compiler.
Poor data validation (e.g. no default condition after a data input).
Variables not initialised, or used incorrectly.
Insufficient arithmetical accuracy.
Insufficient range checks (e.g. divide by zero).
Type mismatch (e.g. string used as a variable).
Residual errors in compilers.

Figure 2.2 gives an overall picture summarising the sources of hardware and software failures and the defences against them.

2.4 SAFETY CRITICAL SOFTWARE

Applications of programmable equipment now include equipment carrying out safety functions. Examples are fire and gas detection apparatus, shut-down and control systems in process plant, medical electronics, aircraft controls, machine tool control, nuclear plant and weapon systems. The consequences of failure under these circumstances are often severe and thus attract particular attention.

Using software in potentially hazardous situations leads to two main difficulties:

(a) Due to the complexity of software failure modes the possibility of hazardous failures is greater.
(b) Since the use of software makes failures difficult to predict it is difficult to perceive if the integrity of a system is adequate.

There are four basic design philosophies which should be considered when incorporating software elements into a safety system.

(1) The software directly controls the safety-related function. Figure 2.3(a) shows a contact providing an input to a programmable electronic system (PES) which, in turn, provides an output. The PES output operates a solenoid whose contact causes some safety action. It is possible to imagine the PES causing an unwanted operation or even failing to operate when required. Due to the uncertainty associated with software failures this solution is seldom favoured.

(2) The system retains hardwired control. Figure 2.3(b) also shows this arrangement whereby the PES carries out functions whilst the

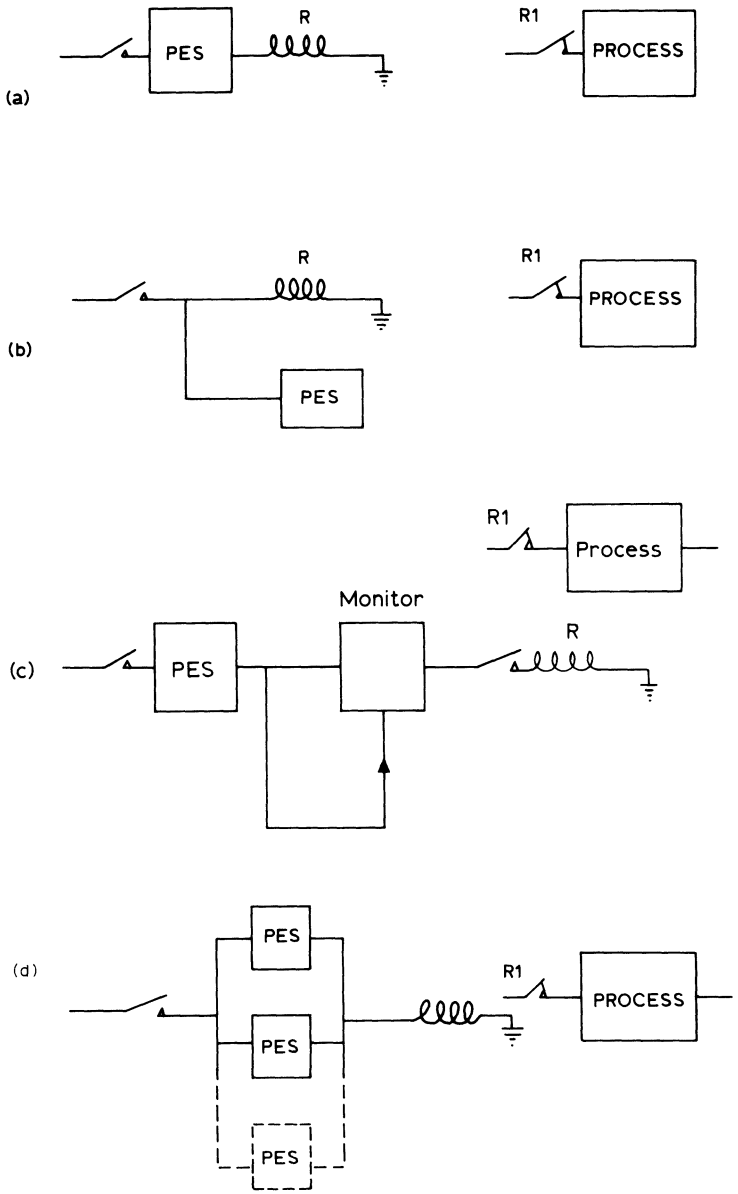


Fig. 2.3.

solenoid is still operated from the input. Currently this solution is usually favoured.

(3) Figure 2.3(c) shows an alternative application of the override principle. In this case a hardwired monitor examines the outputs and disconnects the PES from the solenoid in the event of predetermined undesirable combinations of output.

(4) Diverse software is employed whereby duplicated or triplicated systems are provided such that each system is separately designed and programmed. Some measure of protection against software failure is thus obtained and the outputs from the diverse systems can be compared and voted. This is shown in Fig. 2.3(d).

These alternatives and their relative merits are discussed in Sections 5.4.1 and 10.1.

Various guidelines—particularly the HSE and CEC documents described in Chapter 5—call for assessments to be carried out on safety-related programmable systems. The assessments are required to address:

(a) The overall design configuration (Section 10.1).
(b) Hardware reliability for the hazardous modes (see *Reliability and Maintainability in Perspective,* 3rd edn, David J. Smith, Macmillan, London, 1987).
(c) Qualitative features of the system and its software (see checklists).

In any assessment it is important to identify and then define the boundary of what constitutes the 'safety-related system'. This enables the failure modes of interest to be defined. A clear statement of the safety requirements is also necessary so that definition of hazardous failure can be unambiguous. Some guidelines and standards are then needed in order to give criteria for judgement. The documents mentioned above provide such guidance.

A new initiative which is still under way at the time of writing, is the production of a new Def-Stan by the Ministry of Defence. The purpose of this new standard is to lay down requirements for the production of software for safety-critical systems produced for military use. The main theme of Def-Stan 00-55 is that formal methods are to be used, in particular, the use of mathematical methods to formally verify programs against their specifications. It is expected that this new standard will come into use in late 1989.

2.5 QUANTIFYING SOFTWARE RELIABILITY

In Section 1.2 it was stated that software failures arise from a population of design faults. This population will decline as failures arise and each cause is removed by means of a modification. The rate of occurrence of these failures is dependent on the use of different paths in the program, in order to reveal the faults, rather than the passage of time.

Numerous attempts have been made to model the distribution of software failures. The statistical prediction of failures based on the extrapolation of test data is difficult since time-related models do not necessarily hold good for future periods of code execution. Furthermore, the data contained in the observed distribution may not contain the information relevant to the tail of the distribution which is where predictions are required. The assumptions made by the various models tend to be difficult to achieve in practice. For example, it is often assumed that manpower levels are constant, whereas in practice they vary throughout the project. This will be addressed in Section 12.4.

Another approach to modelling software reliability involves an attempt to identify measurable complexity and size parameters (known as *metrics*) which can be correlated with failure rates. The problem in this case is that the number of complex interacting variables which govern software quality is probably greater than can be realistically modelled. Unlike hardware there is no simple repeatable model involving the failure rate of specific elements. These techniques are discussed in Section 12.3.

No amount of statistical failure analysis will directly contribute to quality in the specification and design of software and it is the purpose of this book to address the qualitative techniques which will influence software quality. The techniques described in the following chapters, therefore, concentrate on the prevention, detection and removal of errors.

Chapter 3

The Effect of the Software Life-cycle on Quality

3.1 THE MEANING OF 'LIFE-CYCLE'

The idea of a life-cycle is a convenient model which serves two purposes. Firstly, it allows one to represent the process of conception and production in a graphical and logical form and, secondly, it provides a framework around which quality assurance activities can be built in a purposeful and disciplined manner.

The conception, design and use of software is an evolutionary process. That is to say, it is produced through successive stages of specification, design and modification. Each assessment of a piece of software, be it by a review of the requirements, the design, the code or, later, by tests and field use, results in changes. This process should involve successive tiers of specification and design where each step is verified against the requirements of the preceding stage. Thus a form of reliability growth applies and a viable software product is evolved.

This top-down succession of activities is commonly called a software life-cycle, and is described in diagrammatic form in Fig. 3.1. This commonly used model of software development is often called the 'waterfall model' because of its similarity to a set of cascading waterfalls. It depicts the software life-cycle as a set of linked but discrete processes with inputs downwards to successive stages and feedbacks upwards to provide verification against previous stages and a final validation of the requirements.

The first stage in this model is the definition of requirements. Always the first stage in any problem-oriented process, it is in fact the most difficult to achieve. The problem lies in the communication between customer and developer. The former often does not know what he wants and, as a result, the latter will have difficulty in

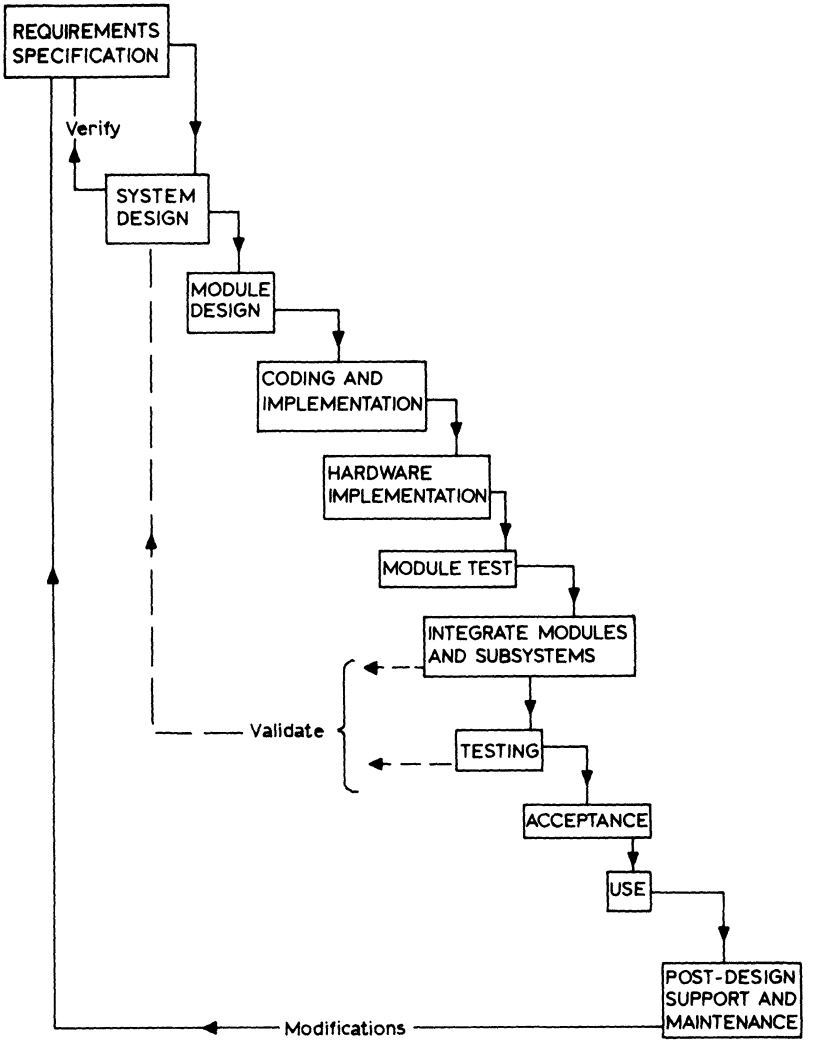


Fig. 3.1. One software life-cycle diagram.

formalising a specification which accurately translates the requirements into design. Even in cases where the customer has a precise idea of what he wants the development of the system tends to result in a modification of the customer's original perception of the final need. This leads to conflict between customer and developer. This requirements stage can be compared with the preparation of a legal

document. The more precise it becomes the more difficult it is to understand. If precision is omitted, then the scope for ambiguity and misunderstanding increases.

Assuming for the moment a satisfactory set of requirements, the next stage, which takes us into design, is usually for the developer to respond with some form of functional specification. This defines, at the interface between user and system, the externally apparent functions of the system as the developer perceives them. This equally important stage defines what the developer is to implement and thus provides the first level of visibility to the customer of the eventual product. It is vitally important that the correspondence between functions and requirements be ascertained to ensure that it is possible to trace requirements throughout the remainder of the life-cycle.

From here onwards one becomes involved in design iteration of some form. At the highest level a system design is established which will allow the separation of software components from non-software components and the definition of the interface between them. Very often an architectural design will be generated in order to establish a framework. Software design is then performed by use of an established 'top-down' methodology. It is here that the greatest effect on quality can be seen since the quality of the design will greatly determine the final quality of the product. The lowest level of the software design will provide the basis for coding and will largely define the structure of the program. In this way the design should solve the overall problem leaving only minor difficulties for the programmer. In the event that some design features are not implementable, then it may be necessary to resort to redesign and iteration until the problem is solved.

The next stage involves testing the coded design at various levels. At the lowest level the programmer must debug his code in isolation and then present it for integration to the system of which it is part. Prior to integration it will go through some form of verification. As integration progresses and external functions begin to appear the customer is often provided with the opportunity to view the potential product. This can prove invaluable in demonstrating both that the system is being developed as originally specified and also to detect points of misunderstanding in interpreting the original requirement.

Finally an acceptance of the system will take place, at which stage the customer may 'sign-off' the system, possibly with some defects still acknowledged. In-service use should also be viewed as part of the system life-cycle right up to the point of obsolescence.

Acceptance testing usually concentrates on demonstrating the functional aspects of the product. In reality it is delivery and operation which usually reveal the degree to which the product meets the customer requirements.

Finally there is an ongoing maintenance activity whereby faulty or missing requirements are revealed and addressed. Unfortunately, at this stage the cost of rectification is very large.

This life-cycle model is not the only one available. It has, however, the advantage of being the most common and is well understood. The opportunities for feedback within it are many and thus, provided that the system developer wishes, the propagation of faults can be largely prevented by the detection and verification activities at each stage.

It is important, in the software life-cycle, to distinguish between the two words

specification
and
design

The *specification* (the top part of Fig. 3.1) is a description of requirements and is usually written by the user. It is vital that such a requirements specification is an accurate statement of what is needed (see Chapters 4, 5 and 6). An incorrect or ambiguous statement, at this stage, will simply be reflected into the design and may not become evident until it is revealed, during test, that the product is not what was required.

Designing is the process of breaking down that specification into a logical hierarchy of successive descriptions resulting eventually in program code. The middle part of Fig. 3.1 deals with this activity. The *design* itself is the set of documents and code listings which are produced.

3.2 ACHIEVING QUALITY SOFTWARE

There are a number of activities and methodologies which contribute to software quality. It is important to understand that these fall into two categories:

(a) Quality activities which attempt to identify and remove errors. These disciplines also minimise the probability of errors being committed.

(b) Specification and programming methods and their associated automatic tools which make it highly unlikely that certain types of software error will be committed by removing many of the traditional steps in software design.

The former group, commonly referred to as software quality, consists of activities which will be described in detail in Chapter 4, including the following areas. Current standards and guidelines to their application are described in Chapter 5.

Configuration management. This is the essential quality activity of keeping control of the issue and status of all hardware, documents, firmware and software. Control of changes is additionally important since there is no visibility to the software on discs and PROMs other than via labelling and documentation.

Documentation standards. Since the only visibility to software is via documentation, a formal structure of specifications, design documents (e.g. diagrams) and code listings is essential. Naturally these should be appropriate to the size of the software package. A few pages of requirements and flowcharts may well suffice for a small calculation package, whereas several volumes would be necessary to describe a large real time control project for some process control or communications system.

Programming and design standards. The objective is to write clear structured software using well-defined modules whose functions are easily understood. There is no prize for complexity and there are various methods for developing structure, including flow, hierarchical and structured diagrams as well as pseudo code (a form of high level code using English language statements).

Design reviews. These are assessments of the design against the requirements. Since the early design reviews are carried out before the product is ready for demonstration or test they take the form of an assessment of the functions provided by the design specifications and flow charts. Code inspection and walkthrough (discussed in Chapter 7) are a part of later reviews and involve detailed examination of the code in order to assess its capability of carrying out the requirements and its conformance to standards.

Test and integration. Testing involves a number of methods. These are described in detail in Chapter 8. It is important to plan a structured set of tests which build on each other so that confidence in modules leads to testing of subsystems (groups of modules) until the

entire system is integrated and ready for functional and environmental test. Types of test include:

Formal proof of correctness.
Validation of code by inspection or walkthrough.
Static tests.
Dynamic tests involving test data.

Failure feedback. Repetition of a proven piece of code with the same inputs will only continue to generate the previous result. Hence, software faults will only be revealed by exercising the program in different ways. This arises in field use, as a result of which reliability growth will be observed following the modifications which arise from field failures. The more structured and formal the system of failure feedback, the faster will be the reliability growth. There are three distinct areas of failure data collection:

(1) Informal reporting prior to an item becoming subject to configuration control.
(2) Failure reporting during test.
(3) Field failure reporting from the user.

The above traditional quality activities involve setting standards and auditing progress against the stated requirements. Although this provides an environment for reducing the likelihood of errors, it does not impact on the fundamental reasons for their existence.

The second group of activities (formal and automated methods) are far more effective since they seek to replace error-prone activities with formal and automated tools. The primary objective of this book is to establish the benefits to be derived from these tools and to describe reasons for their development. Parts 2 and 3 will outline the current methods and practices and will explain the evolving methodologies, tools and attitudes which need to dominate software design if its reliability is to equal or exceed the associated hardware.

3.3 CURRENT PRACTICE

A large number of designers ignore both of the above approaches. It is tempting to proceed with coding when only a part of the total design is conceived. Indeed, commercial pressures make this almost inevitable.

Many designers pay lip-service to the techniques of quality and go so far as to install quality manuals and procedures. Frequent audits against these standards and procedures give the impression of achieved quality. However, these methods alone do not prevent the individual from producing the uncoordinated code which typifies the problem of software design.

A few design teams succeed in applying the traditional quality techniques, mentioned above, in a realistic and conscientious way. The result is much higher-quality software than would otherwise have been the case and, more important, the bugs are discovered and rectified at the earliest opportunity. As shown in Fig. 1.5 of Chapter 1 this minimises the time and cost associated with reliability growth.

A very small number of organisations have yet invested in the design and programming tools which essentially replace and supersede the quality techniques by preventing rather than removing software faults. These tools vary from simple aids which concentrate on one aspect of software development right through to full 'environments' of tools which provide facilities for design capture and expression, code production, context- and syntax-sensitive editors, test aids and configuration management. These so-called Integrated Program Support Environments (IPSEs) are just beginning to see the light and are likely to have a strong impact on software development in large systems.

The next chapter gives a comprehensive outline of current software quality techniques. Although they do not guarantee the absence of errors, if properly applied they will greatly improve the quality of the software product. Chapter 6 onwards is devoted to the techniques which will further improve the situation.

3.4 QUALITY CONTROL AND QUALITY ASSURANCE

Much is written concerning the meaning of quality, quality control and quality assurance. It is not the purpose of this book to address the differences in any detail but, nevertheless, a few words are called for.

The term *quality* refers to the whole concept of specifying, designing and implementing software and hardware which meets the requirements of the user. It involves all stages in the 'life-cycle' and thus addresses the various methods and tools which can be used to achieve it.

Quality control is the activity of measuring achievements and performance against some preceding specification or standard.

Quality assurance embraces the activities which check that the quality control process is taking place and that the standards, methods and tools are adequate.

The organisational split between these activities varies between companies and there is no 'correct' way of apportioning tasks and responsibilities.

What matters is that the tasks are carried out.

PART 2

Current Quality Systems and Software Standards

The next two chapters describe the current quality methods and provide a review of the published Standards and Codes of Practice.

Chapter 4

The Traditional Approach to Software Quality

4.1 QUALITY SYSTEMS

There is nothing particularly new in the idea of software quality. It has evolved from inspection methods and procedures into systems of control and is characterised by the requirement for conformance to formal procedures. It is, however, rooted in the manufacturing activities which have dominated hardware production for several decades. Hardware quality is achieved by bringing together conforming parts and materials by means of proven processes.

The design of software, on the other hand, has no equivalent model. Simple conformance of individual program modules to some specification is certainly no guarantee of system quality and, in any case, it is rarely possible to test for total conformance until all the system modules are brought together as a whole. In other words, although it is possible to validate each module of code without reference to the system, the inter-relationship of coded modules is far more subtle than is the case with hardware pieceparts.

Currently quality systems concentrate on establishing the existence of standards and controls and seek, by means of audit, to verify that they are applied. They operate by specifying various areas for control and requiring that suitable audits take place to verify that they are being applied.

The role of quality management, in these systems, is to monitor conformance with procedures and standards by means of:

Establishing quality plans.
Quality management reporting.
Review meetings.
Quality audits.

In some cases the software quality function is set up as a separate activity from the existing hardware quality. Due to the slightly different skills involved, or perhaps to the craft approach which still pervades the production of software, the activity often evolves as a separate responsibility. This can lead to a blinkered approach to the resolution of problems due to the polarisation of failures into 'hardware problems' and 'software problems' when, more often than not, a solution can be found in a compromise involving both. The principles of quality are common to both and there should be a focal responsibility for this function within any organisation.

The remainder of this chapter describes the areas and extent of current software quality methods.

4.2 QUALITY ORGANISATION, MANAGEMENT AND REVIEW

There has to be an identifiable organisational responsibility for software quality. The important thing is that the actual function can be identified—actual titles are not very important. In a small organisation individuals often carry out a number of different tasks and the quality activities may well be adequately carried out by someone who combines them with other duties.

There should be a quality manual for the organisation and a quality plan together with specific documents for each project. Whereas the quality manual describes the tools and procedures available, the quality plan consists of specific methods drawn from the manual, for that project. These should be produced by the design team but controlled independently of the design activity. They may not have these fancy titles but what is important is the intent. Quality plans must be used effectively and not just left on the shelf.

Specialist quality staff are essential to ensure that customer requirements are adequately planned at the pre-contract stage. During the project the quality engineer will review progress with the project manager and others.

The quality plan must embrace:

How, when and by whom reviews are to be carried out.
What standards, procedures and codes of practice will be applied.
The extent of quality control of subcontractors.
Methods of failure reporting.

Test strategy and the control of testing.
Extent of customer involvement in project quality.

Another feature of quality management is the ongoing review of the quality system. Systems should evolve to meet the requirements of an organisation and the types of product which it develops. It is therefore essential that systematic reviews be held to establish:

Adherence to current procedures.
Effectiveness of current procedures.
Need for additions/changes to the procedures.

4.3 DESIGN DOCUMENTATION

This will vary, according to the size and complexity of the product, from a complex hierarchy of specifications, as illustrated in Fig. 4.1, to a few pages consisting of:

A functional description.
Design text and diagrams.
A program listing (lines of source code).

In more complex systems the structured hierarchy (Fig. 4.1) is essential. It represents a top-down approach whereby the requirement is *decomposed* from the user requirements specification through the various levels of design specification to the source code listings.

It is now widely accepted that the so-called 'top-down' approach provides the best result. Briefly the approach takes the whole system and decomposes it functionally into major subsystems. Each subsystem can then be decomposed in a similar manner until there are codable modules. The major advantage of the 'top-down' approach is that it treats the system as a whole and not as piecemeal components as in 'bottom-up' design (see Section 4.7).

It is the authors' experience that lack of a visible documentation structure, reflecting the system in the manner described above, nearly always coincides with design problems and delays and vastly higher failure rates during test and commissioning.

The documents should include:

User requirements specification. This describes the functions to be performed by the system. It should be complete and unambiguous and

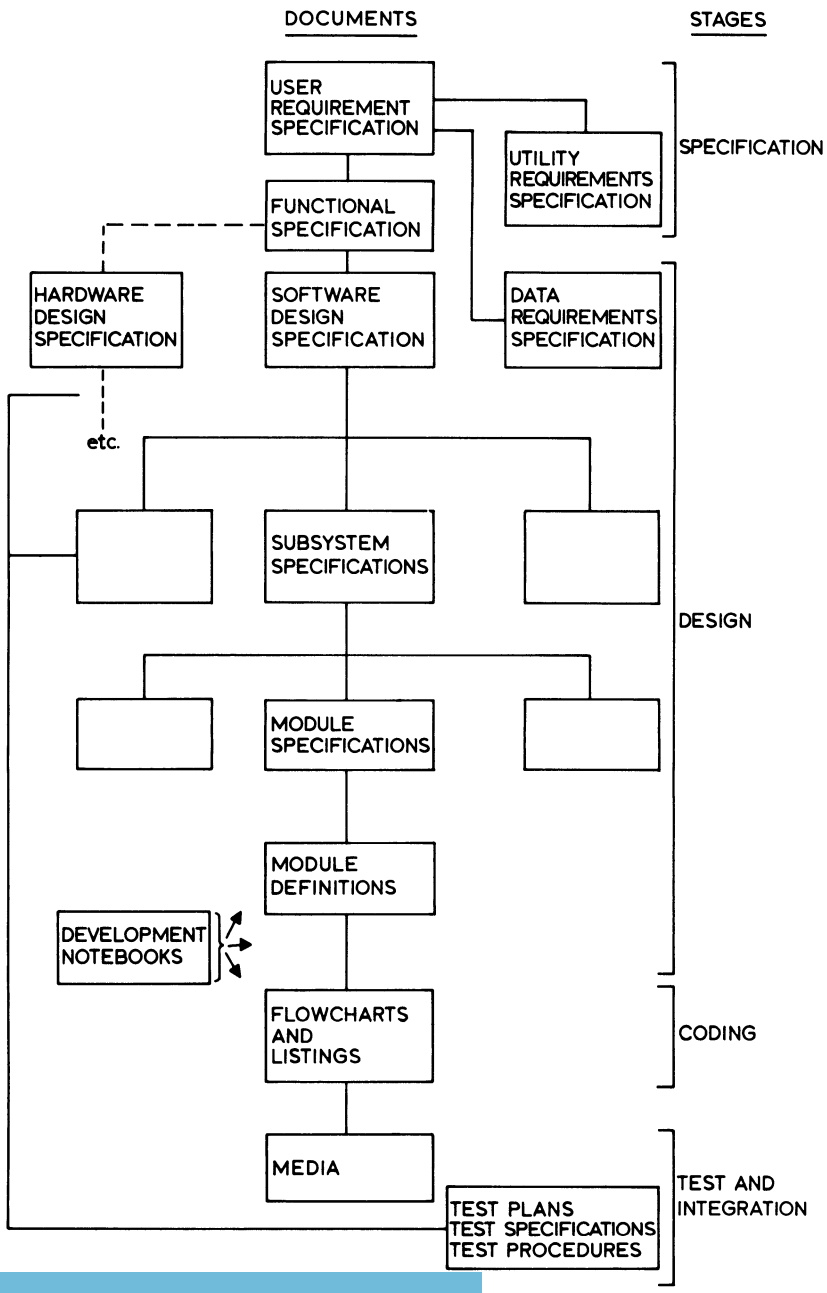


Fig. 4.1.

should be quantitative wherever possible in order to facilitate the assessment of whether the requirements have been met.

Since the user requirements specification is a formal contractual document it is important that its preparation is a thorough and painstaking process involving all interested parties. A good structure is described in the IEE document (see Section 5.4.2) and includes:

(a) An overview providing a perspective of the system within the plant or total environment in which it is to be employed. It should indicate the overall objectives of the system.
(b) System objectives setting out in full detail the objectives as they relate to the operating requirements.
(c) System interfaces outlining how it is required to communicate with the world. This includes both human and electrical/data interfaces.
(d) System environment which covers all the features which would affect the hardware and software design.
(e) System attributes describing and listing the parameters which constrain the design.
(f) Test considerations including diagnostic requirements which will affect operation and maintenance.
(g) A commercial section addressing licensing, etc.

Ideally the user requirements specification should be a pre-contractual document since it specifies *what* the contract will provide. In practice, however, it may well be subject to negotiation and change after the contract has been established, in which case both parties must carefully evaluate the performance, cost and schedule implications.

Functional specification. It takes the requirements of the user specification and describes the actual processing functions which are to be carried out in order to achieve those requirements. It will address language, memory requirements, data bases, the partitioning of the system into subsystems, inputs, outputs, interface communications, data flow, etc.

The important difference between the functional specification and the user requirements specification is that whereas the latter states *what* is required, the former describes *how* it will be achieved in terms of functions. It is usually prepared by the supplier in response to the user requirements specification and, once agreed, becomes a part of the contractual documentation against which acceptance will ultimately be reviewed.

During the detailed process of generating the functional specification, deficiencies and ambiguities in the user requirements specification will arise. These must be negotiated and documented as mentioned above.

Software design specification. Once the software and hardware functionality has been separated, one can proceed to consider the software aspects separately. This proceeds in a top-down manner until subsystem or module specifications are derived, according to size and complexity. Whereas the functional specification addresses the externally apparent functions of the system, the software design specification addresses the way in which those functions will be implemented in software.

Subsystem specifications. If the system is small enough it may be possible to do without subsystem specifications and to move directly to module specifications. In large systems, however, subsystems will ultimately consist of modules and the subsystem specification will describe the functions of the subsystem, the data flow between modules and the interfaces to the other subsystems.

A typical subsystem specification would commence with a brief (one-paragraph) description of the subsystem function. For example:

'The Graphics subsystem receives inputs of the status of fire and gas detection devices. It processes this data to construct mimics of zones and their fire and gas alarm status. This information is passed to the VDU subsystem.'

The specification will carry on to describe how the subsystem is decomposed into modules and what procedures they carry out. Interfaces to other subsystems may be described by means of data flow diagrams.

There will also be a breakdown of the subsystem into modules (basic coded units) together with a brief description of each module. A statement concerning the operating system, memory requirements and hardware environment would also be included.

Module specifications. This is the 'foundation' of the hierarchy since it contains the basis of the system code. A module of code should not be greater than can be easily perceived by a single person understanding its total function—typically 100 lines. G. J. Myers wrote 'Write a sentence describing the purpose of the module. If the sentence is a compound sentence, containing a comma or more than one verb, then the module is probably more than one function.' The module

specification will describe the inputs and outputs of the module and the logic to be performed by it. The flowchart will be part of the specification.

Module definitions. This is a more detailed level of design than the module specification. It involves the actual coding of the module specification requirements and addresses the testing and performance of that module. It includes:

The module specification.
Diagrams (see Section 4.5).
Source code listing.
Test specification.
Test procedure.
Test results.

This package then constitutes a total description of the module and its design and test history.

Utility requirements specification. This should contain a description of the hardware requirements including the operator interface, hardware memory requirements, processor hardware, data communications hardware, software support packages, etc.

Development notebooks. It is an excellent thing for each designer to have a development notebook—a looseleaf file containing his written notes, listings, changes, correspondence, etc. This can be an invaluable aid to diagnosis during testing when the reasons for certain lines of code or some changes, have become blurred in one's memory. Furthermore, with the turnover of design staff it is possible that the person in question may not be available when problems arise.

Requirements matrices. These provide a graphical system of cross-referencing between specifications as well as a method of checking off each requirement against the test specifications.

4.4 CONFIGURATION MANAGEMENT AND CHANGE CONTROL

This requires procedures covering:

Modifying the software.
Modifying the hardware.
Reporting discrepancies.
Disposal of documents and media.

Keeping secure masters.
Modification approval.
Labelling documents and media.
Segregating non-conforming items.

It is vital to ensure that changes are formally documented and controlled, particularly since there is no visible change to the software media (tapes, discs, PROMs). Any hardware change can render the

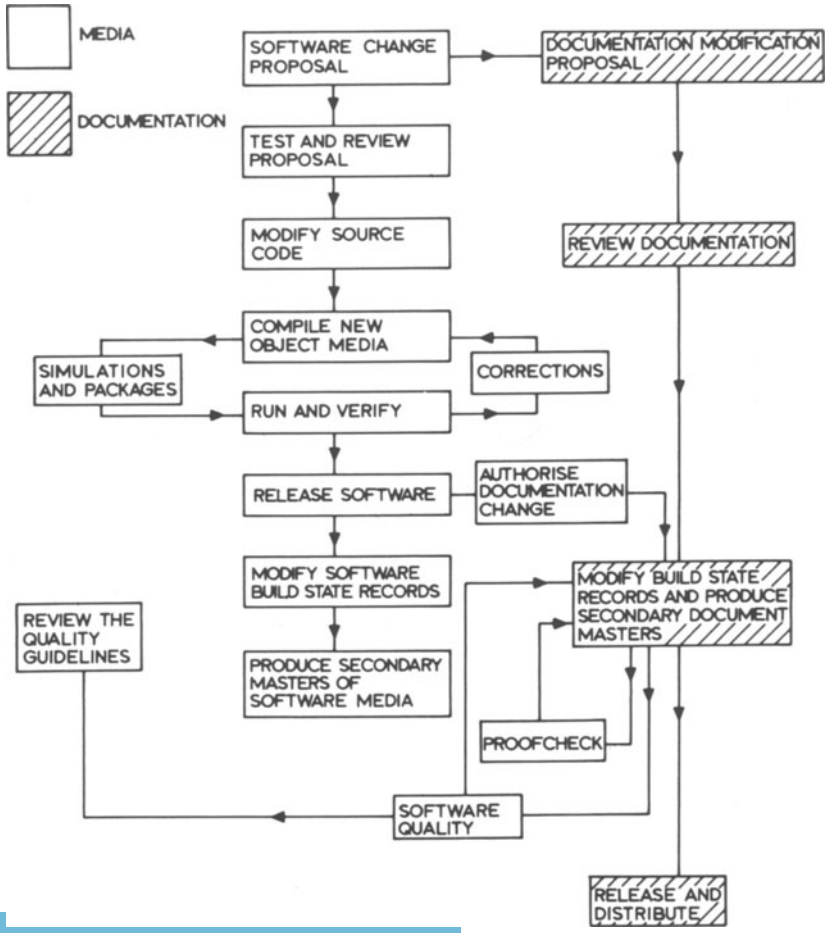


Fig. 4.2.

software incompatible, therefore the build state of the two must be carefully controlled. A PROM containing the incorrect issue of software, even though only a single statement may have been changed, can cause a system to malfunction. This may well be a hazard in a process control or safety system.

Figure 4.2 shows a typical outline of a change system which co-ordinates both documentation and media.

The library/bureau function may well provide a focal point for this configuration and change control. Other library functions are:

Storage and identification of media and documents.
Build state records.
Spare copies and security.
Audit reports.
Codes of Practice.
Test documents.

A good rule is that a document first becomes subject to configuration control once it is used by another person. The library will then take control by checking completeness and format and will provide a unique identity and issue status.

Media should have visible labelling containing serial and issue status and this information should also be present in the software itself—that is to say, written on to the disc, encoded on the tape leader, programmed into specific locations of the PROM.

Arrangements for secure storage are important and 'insurance copies should be made weekly (or even daily) for storage in a different location. Banks provide this function and many organisations make use of the facility.

4.5 PROGRAMMING STANDARDS

4.5.1 General Rules

The human brain is not well adapted to retaining random information; hence standardised rules and concepts substantially reduce the probability of error.

A standard approach to creating files, polling output devices, handling interrupt routines, etc., constrains the programmer to use proven methods.

A further step in that direction is the use of standard subroutines to perform common functions within the system. Re-inventing the wheel is both a waste of time and an unnecessary source of error. Examples are:

Extended memory addressing (EMA) buffer management.
EMA table access.
System error routines.
Commonly used data structures.

The undisciplined use of GOTO statements in high level language is dangerous and leads to difficulties in perceivability of the functions when reading source code. This is known as spaghetti code. Modern block structured languages often contain no GOTO statement.

Modules should have only one entry point and it is desirable that they have only one clearly defined exit.

To avoid data corruption the use of globals should be minimised so that modules can only access data local to their subsystem. Where global data (e.g. EMA buffers, system data) is required then standard subroutines must be used for access.

A good guide to module size was given in Section 4.3 (G. J. Myers' quotation). In practice this might be 30–60 lines of code plus 20 lines of comment, but the ultimate criterion is its total perceivability in order to grasp the function.

4.5.2 Structured Programming

This involves the decomposing of the design (as discussed above) in a strictly logical manner. If the design has been well structured, then the coding activity should be almost routine after reading the module specification. The following light-hearted example, in a sort of pseudo code, illustrates the difference between a structured and unstructured approach.

Unstructured

```
⟨Embark⟩
Begin
        Get in boat
        Reach for bags
        If bags in reach then GOTO 'out'
        Again: Shout for help
```

Structured

```
⟨Embark⟩
Begin
        Get in boat
        Reach for bags
        IF in reach THEN
        put bags in
```

```
    IF help comes THEN                ELSE
    Begin                             Begin
        Take bags                         While no one around do
                                                shout for help
        GOTO 'out'                        Take bags
    END                                   put bags in boat
    GOTO 'Again'                      END
    'Out': put bags in boat           Sailaway
    Sailaway                      END
END
```

Looking at the two examples it can readily be seen that the same problem has been tackled in two ways. In the unstructured case it is necessary mentally to add lines and arrows to perceive the loops. In the structured case the flow is always forwards. Note that, in the structured case, no GOTO statement is present.

4.5.3 Describing the Modules

Earlier in this chapter it has already been emphasised that the requirement is to arrive at well-defined modules via a process of logical definition. There is no prize for complexity.

There are several methods of developing the module on paper:

Flow diagrams are a method of graphical representation as illustrated in Fig. 4.3. They are less popular now as a result of more formal block structured languages. There is a potential fault inherent in Fig. 4.3. Consider what happens if $N < 0$. Is there an endless loop?

Hierarchical diagrams provide a breakdown by task and then detail as illustrated in Fig. 4.4.

Warnier diagrams use both horizontal and vertical dimensions, and Fig. 4.5 shows the same problem as Fig. 4.4 in Warnier form.

Structured box diagrams, sometimes known as Nassi–Shneiderman diagrams, are another format and the same task is shown again in Fig. 4.6.

Pseudo code uses English language statements as illustrated in the 'Embark' example above.

These aspects of programming are developed further in Section 7.3, which addresses the more up-to-date methods.

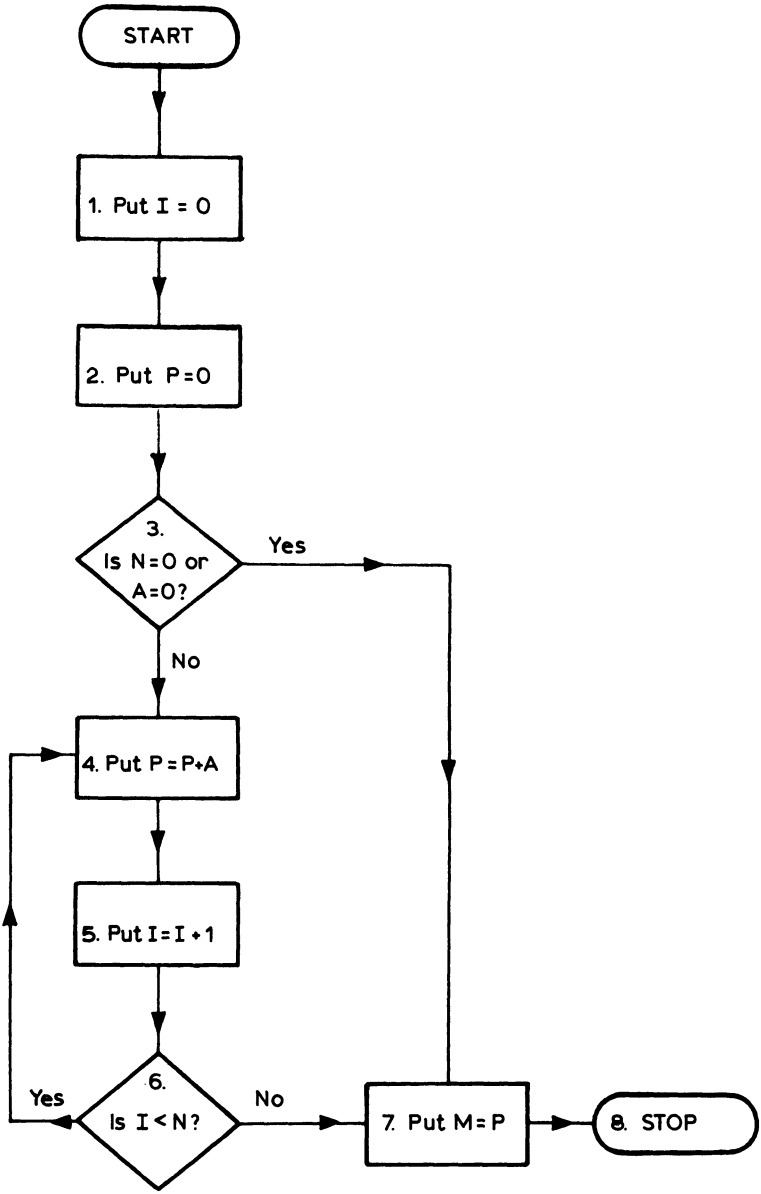


Fig. 4.3.

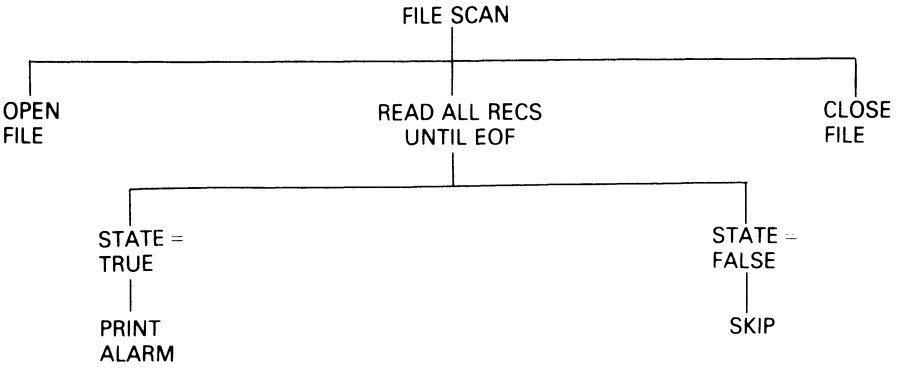


Fig. 4.4. Hierarchical diagram.

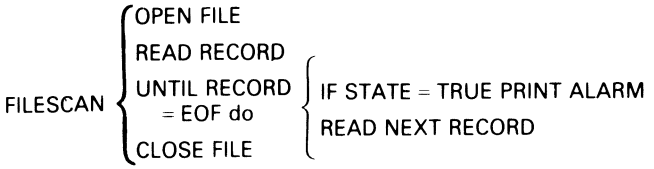


Fig. 4.5. Warnier diagram.

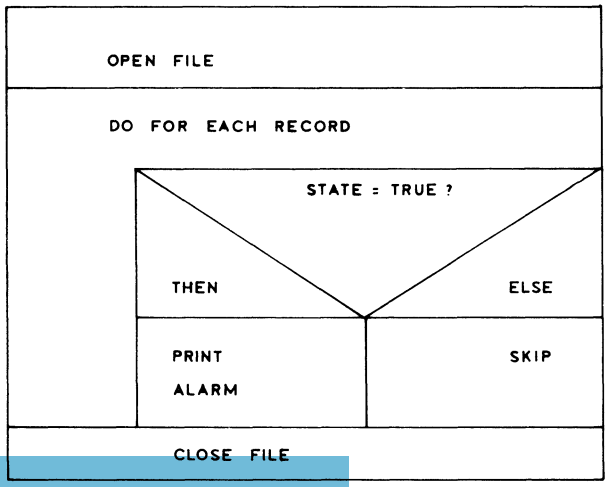


Fig. 4.6. Structured box diagram.

4.6 DESIGN REVIEWS

Two common misconceptions about design review are:

That they are schedule progress meetings.
That they enable one to appraise the designer.

These are both dangerous misunderstandings of the purpose of design review and will result in its not being effective. Its purpose is to verify the *design,* at specific milestones, against the requirements—not to establish reasons for delay. Chapter 7 deals with the function and conduct of design reviews in some detail.

The value of design review is usually underestimated but it is most definitely the major activity in establishing software quality. Being a formal review highlights the need for a baseline against which to review specifications. Section 7.4 addresses this further.

Design reviews can involve code inspection and walkthrough, both of which involve the systematic review of code against its specification. This is also addressed in Section 7.5 and Chapter 8 along with the automated tools available.

4.7 TEST AND INTEGRATION

Currently much of the focus, in software quality, is on test. Although this is an important area it cannot be stressed enough that the major part of the quality effort should have been expended on the earlier design activities. Nevertheless, there needs to be a hierarchy of test documents and a structured approach to testing from coded modules upwards to system test.

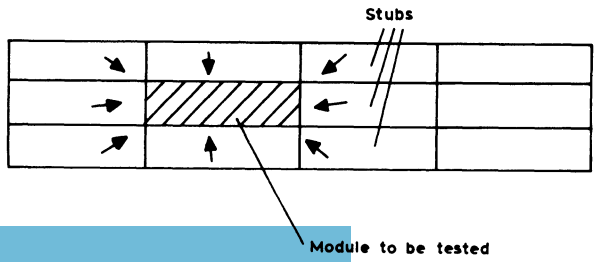


Fig. 4.7.

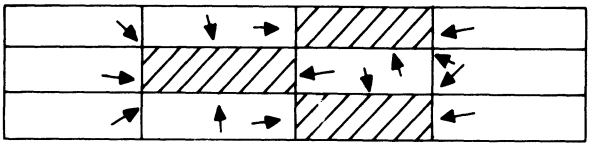


Fig. 4.8.

The alternative to top-down integration involves the testing of individual modules as they are produced. It is therefore necessary to provide a minimum simulated interface to replace each of the as-yet uncoded modules. These simulated modules are known as *stubs*. Figure 4.7 illustrates the concept. As additional modules become available they replace the stubs as shown in Fig. 4.8.

This is also discussed in Section 8.4.

4.8 SUBCONTRACTED AND BOUGHT-IN SOFTWARE

4.8.1 Shelf versus Custom Software

In terms of evaluation, and of confidence in the final product, the quality problem is much the same. The circumstances of a proprietary (off-the-shelf) product are that the visibility to the purchaser is greatly reduced. Custom software, however, is usually produced on a project basis such that the various activities of specification, design, test, review and commissioning are visible to the customer.

In the case of 'off-the-shelf' software the evaluation is usually restricted to:

Evidence of field experience (if any).
Data from other users.
A vendor appraisal.

4.8.2 Vendor Appraisal

The requirement for a vendor appraisal is stated in most of the quality systems (Section 5.3) and this is necessary for both of the above types of purchase. Clearly, in the 'off-the-shelf' case, it is even more critical due to its being the main point of assessment.

The features which should be investigated are, broadly, those which have been outlined in this chapter. In Chapter 5 the systems and procedures which should be looked for are outlined.

A small vendor may well exercise adequate control by the use of

simple, perhaps informal, documents. It should therefore always be the aim to establish that genuine control is being applied rather than efforts being directed to the production of impressive manuals and standards which are not actually being used.

4.8.3 Field Experience and History

This will apply to proprietary software packages rather than to custom packages. The following questions should be addressed:

(1) Who has purchased and used the package?
(2) Is there evidence of documented defects and subsequent corrective action?
(3) How long has the package been in active use by consumers?

If it proves possible to contact users they might be asked:

(1) Do you keep a detailed log of the results obtained in using the package?
(2) What is your experience with the vendor in following up and resolving problems?
(3) Is the documentation adequate to permit easy modification of the package?

4.9 AUDIT

This involves an assessment of the controls used in the design and management process and an evaluation of their effectiveness. The systems and guidelines (Chapter 5) provide a basis for the audit and, in most cases, offer checklists as an aide memoire against which to examine an organisation.

There are advantages and disadvantages in the use of checklists. On one hand they provide a means of ensuring that each question is remembered and thus enable the auditor to select the most appropriate questions. Furthermore, the checklists can be revised and updated as additional lessons are learned and, thus, one is always presented with the total of previous experience.

On the negative side, however, there is a temptation to expect yes/no answers and this can lead to a false view of the situation. In most cases it is the reason for the answer rather than the answer itself which provides the real information. For example, 'Is a high or low

level language being used?' has no right or wrong answer. The reasons given for the answer are, however, of great importance.

In Section 11.3 the planning and conduct of an audit is described.

CHECKLIST 4.1 MANAGEMENT

(1) Is there a senior person with responsibility for software quality and does he have adequate competence and authority to resolve all software matters?
(2) Is there evidence of regular reviews of software standards?
(3) Is there a written company requirement for the planning of a software development?
(4) Is there evidence of software training?
(5) Is there good housekeeping of listings, specifications and computer hardware?
(6) Is there a quality manual or equivalent documents?
(7) Is there a formal release procedure for deliverable software?
(8) Does every programmer have a dedicated VDU terminal or reasonable access to one?
(9) Is there a Quality Plan for each development including:
    Organisation of the team;
    Milestones;
    Codes of Practice;
    QC procedures including release;
    Purchased software;
    Documentation management;
    Support utilities;
    Installation;
    Test strategy?
(10) Is there evidence of documented design reviews? The timing is important. So-called reviews which are at the completion of test are hardly design reviews.
(11) Is there evidence of defect reporting and corrective action?
(12) Is there a fireproof media and file store?
(13) Are media duplicated and stored in separate locations?
(14) Are the vendor's quality activities carried out by people not involved in the design of the product that they are auditing?
(15) Is account taken of media shelf life?
(16) Are there audits of documentation discrepancy?
(17) Are the quality activities actually planned through the project?

CHECKLIST 4.2: DOCUMENTATION HIERARCHY AND CONTROL

(1) Is there an adequate structure of documentation?
(2) Are all the documents available?
(3) Do specifications define what must not happen as well as what must?
(4) Is the format of the documents consistent?
(5) Is change control in operation?
(6) Are development notebooks in use? (If so, audit a sample for completeness.)
(7) Are the requirements of the higher level specifications accurately reflected down through the other documents to module level?
(8) Are there a significant number of parameters left 'To Be Determined'?
(9) Is 'automatic coding' (use of coding lines on flowcharts) in use?
(10) Do actual documents and firmware (PROMs) correspond to the build state records? (Do sample checks.)
(11) Are maintenance manuals:
  (a) Adequately detailed and illustrated?
  (b) Prepared during the design?
  (c) Objectively tested?
(12) Are operating instructions adequately detailed and illustrated?
(13) Is there a standard or guide for flowcharts, diagrams or pseudo code in the design of modules?
(14) Are there written conventions for file naming and module labelling?

CHECKLIST 4.3: PRODUCT DOCUMENTATION

Duplicate sufficient of this Checklist for the number of subsystem module and changenote audits.

Date ..... Module .....
System ..... Auditor .....
Subsystem .....

(1) Are coding and format standards observed (block structures, headers)?

(2) Is issue control correctly applied?
(3) Is the documentation complete?
(4) Are there marked up documents present?
(5) Is the module a self-contained, perceivable unit?
(6) Is there one entry and one exit?
(7) Does the module implement all the parameters of the specification?
(8) Is the program adequately commented?
(9) Is it easy to cross-reference to other specificatons?
(10) Are there any off-page connectors?
(11) Are GOTO statements used?

Comments

CHECKLIST 4.4: CHANGE CONTROL

(1) Is there a named documentation controller?
(2) Is there a documentation plan (list of all documents)?
(3) Is there a distribution list for each document?
(4) Are there written rules for the holding of originals?
(5) Is there a written procedure for the release of both the media and the documents? (Audit by looking at examples.) N.B. There may well be levels of release (e.g. internal and external).
(6) Is there a record of all amendments?
(7) Is there a written change control procedure?
(8) Is there a named change controller?
(9) Is there evidence of software quality control:
Audit records;
Named software quality engineer?
(10) Is there evidence of hardware/software change coordination?
(11) Are all issues of program media accurately recorded?
(12) Are all software patches identified, recorded and dealt with?
(13) Is there a system for the removal and destruction of obsolete documents from all work areas?

Note:

(a) Sample the effectiveness of the change system by taking samples from the work areas.

(b) Documentation control needs to be realistic and flexible. A good rule is that a programmer needs to exercise formal documentary control over a piece of code only once a change will affect another person.

CHECKLIST 4.5: PROGRAMMING STANDARDS

(1) Is there a document defining program standards?

(2) Is each of the following covered:
- Block lengths;
- Size of codable units (module size);
- Use of globals;
- Use of GOTO statements;
- File security;
- Operator error security;
- Unauthorised use security;
- Recovery conventions;
- Data organisation and structures;
- Memory organisation and backup;
- Error correction software;
- Automatic fault diagnosis;
- Range checking of arrays;
- Use of PROM, EPROM, RAM, disc, etc;
- Structured techniques;
- Treatment of variables (i.e. access);
- Coding formats;
- Code layout;
- Comments (REM statements);
- Rules for module identification?

(3) Is there reference to structured programming?

(4) Is there a library of common program modules?

(5) Is the 'top-down' approach to software design in evidence?

(6) Is high level or low level language used? Has there been a conscious justification?

Chapter 5

Current Standards and Guidelines

5.1 THE NEED FOR STANDARDS

The recognition of a need for standards dates back many centuries. Probably the earliest example is that of weight standards to ensure uniformity and understanding when purchasing quantities of goods. A highly industrial society naturally develops standards to cope with the compatibility problems of rapid technological change and the need to trade globally.

The software industry has discovered over the years that the use of standards can lead both to economies in production and to ensuring that particular requirements are universally understood. In just the same way that two manufacturers, one of nuts and the other of bolts, must set standards to enable their products to be of use to a consumer, there is also a need for the software industry to ensure the compatibility of its tools and products. A proliferation of different languages, operating systems, etc., may lead to diversity and choice but it also minimises portability and interoperability, which does neither the supplier nor the producer much good.

5.2 HOW STANDARDS EVOLVE

There are a number of ways in which standards are produced and evolve. The first is by the identification, by a group or organisation, that efficiency can be achieved by adhering to common methods and tools on some topic. Alternatively an independent body might propose a new standard in order to initiate discussion and consultation which will lead to that standard being established in some form. A very

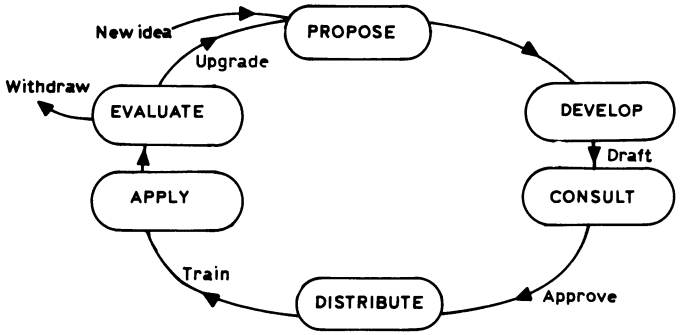


Fig. 5.1.

common route is that whereby manufacturers join together and define some common standard which they all agree to work to in order that the consumer's requirements of interoperability can be met. There are also benefits arising from a reduction in competition by agreeing to a standard approach.

Standards tend to evolve from changes in market conditions or from changes in the user's requirements. Standards must clearly be reviewed on a regular basis in order to ascertain their suitability for the conditions at that time.

Figure 5.1 is an attempt to represent the evolution of standards.

5.3 A SUMMARY OF CURRENT QUALITY SYSTEMS

There are a number of quality systems which have been developed over the last decade. Naturally, there is a degree of overlap in the requirements since many are developments based on previous documents. Table 1 gives an outline.

5.3.1 UK Defence Standard 05-21

This was evolved, in the early 1970s, from the NATO AQAP 1 standards. Essentially it lays down the areas for procedures in design, procurement, manufacture and installation for a total system of quality. Each paragraph is expanded in Standard 05-22 which describes the requirements in detail. The basic requirement is that one operates adequate controls in order to run the particular business in question so that products conform to specification. This provides the flexibility

TABLE 1

		Quality systems model for quality assurance in design/development, production, installation, and servicing	*Note*	*See Section*
Civil	ISO	ISO 9001: 1987	European	5.3.6
	CEN	EN 29001		
	Belgium	NBN X 50-003		
	Canada	CSA Z299.1-85		
	France	NF X 50-131		
	Netherlands	NEN 2646		
	Norway	NS 5801		
	Switzerland	SN 029 100A		
	United Kingdom	BS 5750: Part 1 (ISO 9001—1987)	QAS 3302 addresses software	5.3.2
	United States	ANSI/ASQC Q91—1987		
	West Germany	DIN ISO 9001		
Military	NATO	AQAP-1	replaces 05/21	5.3.3 (5.3.1)
	United States (defence)	MIL-Q-9858A		
	UK MOD	Def-Stan 00-55	Not yet published Replaces 00-16	5.3.5 (5.3.4)

which permits both simple and complex sets of quality procedures to satisfy the standard, provided that they are adequate for the type of business and product. The requirements are sufficiently general that they cover the design and production of both hardware and software. Although specific software controls are not mentioned they are implied by calling for adequate product documentation and controls during all phases of design and manufacture.

The 05-2*x* series embraces:

05-21 Systems for design and manufacture.
05-24 Systems for manufacture.
05-29 Systems for inspection.

The requirements of the lower standards (higher numbers) are contained within the higher ones. In other words the requirements stated in 05-21 are those of 05-24 with additional controls covering the design and engineering activities.

The associated standards are:

05-22 Describes 05-21 in detail.
05-26 Covers calibration.

The 05 series was replaced from 5 September, 1985 by the AQAP series (see Section 5.3.3).

5.3.2 British Standard 5750 (1987)

This followed in the late 1970s as a civilian standard for quality systems and is very similar to the 05-2*x* series. In fact any quality system which conforms to BS 5750 will almost certainly conform to the appropriate Defence Standard 05-2*x*. In a similar way the Part numbers of BS 5750 embrace design and manufacturing as did 05-21 and 05-24. BS 5750 was reissued in 1987.

Part 1 Specification for design, manufacture and installation.
Part 2 Specification for manufacture and installation.
Part 3 Specification for final inspection and test.

Parts 4, 5 and 6 are detailed guides to Parts 1, 2 and 3 respectively.

Technical schedule QAS 3302 is one additional feature of BS 5750 and provides some additional specific requirements for the quality of software. These include control of records, Codes of Practice, aspects of software design, subcontracted software and software design tools.

BS 5750 has been offered as an International Standard, ISO 9001 (See Section 5.3.6).

5.3.3 NATO Standards—AQAP Series

The AQAP series is similar to the 05 series described above.

AQAP 1 Similar to 05-21 and BS 5750 Pt 1.
AQAP 2 Similar to 05-22.

AQAP 13 is specific to software and addresses Codes of Practice. It is written as a supplement to AQAP 1 which is the broader quality standard. Nevertheless AQAP 13 may be used as a stand-alone document. It covers organisation, system review, planning, documentation, corrective action, design review, configuration management, subcontract software, support, purchasing, testing and handling. Furthermore, it specifically calls for a quality plan, which is not the case with the other standards.

AQAP 14 is guidance on 'Evaluation of a contractor's QA system for compliance with AQAP 13'.

5.3.4 UK Defence Standard 00-16

This is called 'Guide to the Achievement of Quality in Software' and was published, as Issue 1, in 1980. It is a comprehensive outline of software quality activities as they are currently applied and covers all types of software and firmware.

There are three major areas of guidance and ten appendices which feature checklists for each major software activity. The three areas are:

Pre-contractual activities. This addresses the establishing of requirements and the overall planning of the software life-cycle.

Codes of Practice. This section identifies the areas where standards are needed, such as documentation, configuration management, design review, test and subcontract.

Software quality procedures. Here the activities are listed and described.

The appendices are:

(A) Pre-contractual activities.
(B) Planning for software quality.
(C) Design and programming techniques and methods.
(D) Documentation.
(E) Configuration management.
(F) Design reviews.

(G) Tests.
(H) Trials procedures.
(J) Transfer to customer.
(K) Subcontracting of software.

5.3.5 UK Defence Standard 00-55

Although not yet published in draft, some information has been made available on this new Def-Stan which it is proposed will come into use in 1989. The purpose of this new Def-Stan is to provide a standard for the development of software for use in safety critical systems. Within such systems the use of software to provide safety critical functions has an inherent problem of the unpredictability of software, in particular, the difficulty in testing software thoroughly. In a non-critical system this can be problem enough; in a safety critical system it is obviously a severe problem. The MOD approach is to use formal, mathematical methods supported by tools such as Malpas and SPADE (see Chapter 8). Although there is still some way to go before industry accepts the content of 00-55 there is little doubt that its general principles will come into being.

5.3.6 ISO 9001 (1987)

This is the European Quality standard published in 1987. It is a similar series to the BS 5750 System. Thus:

ISO 9001—BS 5750 Pt 1
ISO 9002—BS 5750 Pt 2
ISO 9003—BS 5750 Pt 3

Whilst ISO's Technical Committee (ISO/TC 176) has tapped the combined experience of its member countries in developing the new 9000 series of standards, the influence of the British experience and leadership in using BS 5750 (1979) is most evident. This can be seen by the close correspondence of many parts of ISO 9000 to parts of BS 5750 (1979) and the similar specification of models for three levels of quality systems. Most importantly, however, these new ISO standards represent improvements in terms of more comprehensive and strengthened requirements, improved readability and practical use for both the supplier and customer.

5.4 CURRENT SOFTWARE STANDARDS AND GUIDELINES

There are many standards and guidelines which address the documentation and development of software. The following sections describe the major documents currently in use.

5.4.1 HSE Document: Programmable Electronic Systems in Safety Related Applications (UK)

In 1981 the HSE issued a booklet, *Microprocessors in Industry,* which broadly addressed the problem of microprocessors in plant applications. This led, in 1984, to the drafting and subsequent public comment on a document originally called *Guidance on the Safe Use of Programmable Electronic Systems.* The guidelines were published in June 1987.

It has been recognised that, due to the wide spectrum of programmable electronic system (PES) applications, further second-tier documents, covering guidance on the development of specific applications, should follow. This should ultimately lead to simpler guidance and a more consistent approach to specific applications.

The guidelines are aimed at giving generic guidance on optimising the integrity of programmable equipment wherever it is used to provide a safety system. If the safety features are adequately satisfied by non-programmable equipment then the document does not apply, although the principles, in fact, apply equally well to any software or hardware system.

The document is in two volumes:

(1) An Introductory Guide.
(2) General Technical Guidelines.

Volume 2 considers the assessment strategy with respect to three characteristics:

(a) The configuration.
(b) The hardware reliability.
(c) The system integrity, which includes the quality of design and implementation of the hardware and software.

It addresses a number of basic system configurations involving both totally programmable and mixtures of programmable and non-programmable equipment. Three principles are given for assessing the

acceptability of a configuration. In summary they state:

(1) The combined number of programmable and non-programmable safety systems shall not be less than the number of conventional systems traditionally used.
(2) No single hardware failure in any PES shall cause a dangerous mode.
(3) No single software failure should cause a dangerous mode.

Figure 5.2(a) illustrates a single PES protecting some part of a process. Figure 5.2(b) shows two alternative protection systems where one is programmable and the other is hardwired. This arrangement is regarded as satisfactory provided that no feature of the software can inhibit the protection function of the hardwired backup. Figure 5.2(c) shows another duplicated protection system where both systems consist of PESs. In this case diversity of software is urged such that the two channels consist of separately designed and coded software. This

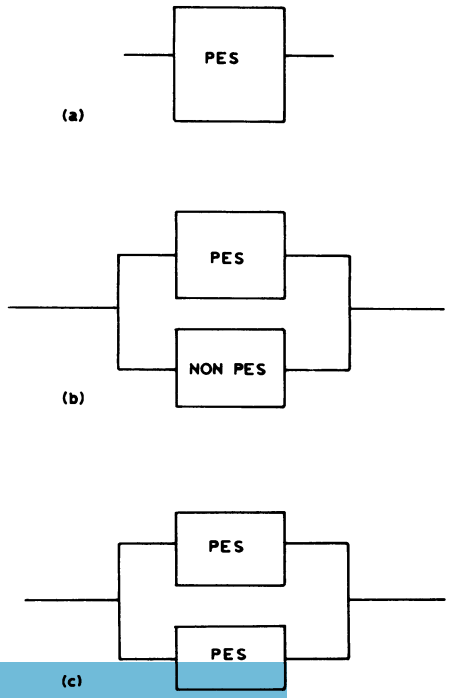


Fig. 5.2.

technique for achieving fault tolerance is by no means a total defence against common-cause failure and the reasons for this are discussed in Chapter 10.

Other configurations are described and discussed covering both control and protection systems. Many design and environmental considerations are listed, such as the features described in Chapter 10.

It also addresses software quality and describes methods of hardware safety and reliability analysis such as fault tree and failure mode analysis, as well as offering 25 pages of checklist questions covering similar items to the checklists in this book. The integrity assessment requires that the relevant questions be addressed.

As discussed in Section 4.9, the use of checklists carries both advantages and disadvantages:

(a) They provide an aide-memoire so that essential features are not overlooked.
(b) They should not be used slavishly but as a menu from which to select the pertinent areas for the task in hand and should therefore be used only by experienced auditors.
(c) They constrain lateral thinking by giving the impression of being exhaustive.

5.4.2 IEE: Guidelines for the Documentation of Software in Industrial Computer Systems (UK)

This document was prepared by the Institution of Electrical Engineers (IEE) Computing Standards Sub-committee and published in 1985. It is a fairly thorough treatment of the major documents required for software requirements and design documentation.

Section 1—'Introduction to the guidelines'—draws attention to the range of software system types, covering:

Fixed program systems (e.g. event loggers).
Limited variability systems (e.g. PLCs).
Full variability systems (e.g. real time control systems using a high level language).

Throughout the guidelines the type and size of each document applicable to each of the above types are addressed.

The software life-cycle is briefly introduced together with the associated areas of documentation.

Section 2—'User requirements specification'—provides a thorough template for writing requirements specifications using the traditional

free expression. (Chapter 6 will cover the future trends in formal specification languages.) This section includes:

A description of the purpose of a requirements specification.
Its objectives and structure and the need for unambiguous measurable requirements.
Operating requirements (e.g. consoles, inputs, outputs, graphics, alarms, safety, security, timing, printing).
Interface requirements (e.g. electrical, logical, human, digital/analogue, series/parallel).
Environmental requirements.
Attributes (e.g. availability, repair times, documentation, adaptability).

Section 3—'Functional specification'—provides an overview of the specification which follows from the user requirements specification and describes how the system will meet these requirements. The outline covers:

A description of the purposes of a functional specification.
Its structure.
System functions (e.g. control functions, malfunction response, data to be stored, operating procedures, safety and security).
System interfaces (e.g. human, data, real time, I/O, communications).
System attributes (e.g. reliability configuration, maintenance and support facilities).
Design and Test constraints (e.g. language, test methods, quality plans, design tools to be used).

The essential difference between the user requirements and functional specifications is that the latter describe features and parameters whereas the former state more generalised requirements.

Section 4—'Software system specification'—covers the hierarchy of design documents including a mention of the module level. Modules of code are referred to as programs. The section describes the specification in the following hierarchy and gives a description of each level.

System structure. The breakdown into subsystems.
Subsystem structure. The breakdown of subsystems into program modules.

Data. Inputs, Outputs and data bases.
Functions and relationships. Functional features and information flow.

The module level of code is not emphasised as strongly as in Chapter 4 of this book.

Section 5—'System acceptance testing'—describes the means by which the supplier demonstrates that the requirements of the functional specification have been met. The following documents are described:

Test philosophy. The overall scope, range and types of test.
Test plan. A schedule of activities.
Test specifications. Detail of each test.
Test logs. A record of all events.
Test summary. A summary of test failures.
Commissioning report. An overall summary including any modifications carried out.
Certificate of acceptance. A formal contractual acceptance document.

Section 6—Post-installation documentation—outlines the operating and maintenance documents but also includes development documentation (drawings, listings, etc.) and general contractual documents (licences, warranty, ownership, etc.). The quantity of post-installation documentation will vary considerably between fixed program, limited and full variability systems.

Section 7 addresses the purpose and implementation of configuration control.

Section 8 provides a glossary and bibliography.

5.4.3 EEA: Guide to the Quality Assurance of Software (UK)

The EEA (Electronic Engineering Association, 8 Leicester Street, London WC2H 7BN, UK) document was issued in 1978 and was one of the earliest guides to give a thorough coverage of software quality.

Six pages describe the activities, quality controls and documentation for each of the following life-cycle activities:

Design definition.
Planning.
Design implementation.
Design evaluation and test.
Post-design support.

There follow about 20 pages of detailed checklists addressing each of the life-cycle activities, including the production of a quality plan and the subcontracting of software design.

5.4.4 EEA: Establishing a Quality Assurance Function for Software (UK)

This is a companion document to the *Guide to the Quality Assurance of Software* described above in Section 5.4.3 and was published in 1981. It is in six sections, covering:

The need for software quality.
Software quality functions:
  Standards, planning, tasks, etc.
Responsibilities:
  For managers, designers, quality staff, etc.
Personnel:
  Recruitment, training, motivation, etc.
Practical aspects:
  e.g. standards, tools and techniques, simulations.
Cost effectiveness.

There are two Appendices:

The software life-cycle.
A compendium of standards.

5.4.5 EEA: Software Configuration Management (UK)

Also published by EEA, this guide followed in 1983 and seeks to highlight the differences between hardware and software configuration management. It is quite compact (15 pages) and consists of:

Introduction.
Principle of configuration management:
  Defines specification boundaries.
  Covers changes and controls.
  Covers subcontract control.
  Covers management plans.
Application to projects:
  Defines items controlled at various stages of the design cycle.
Automated systems for configuration control:
  Features.
  Advantages and disadvantages.

5.4.6 EEA: A Guide to the Successful Start-Up of a Software Project (UK)

Another EEA guide, published in 1985, which addresses the start-up activities of a software project and is intended for managers with little experience in such projects. It includes:

Introduction.
Initial considerations:
  Guidance on information needed, sources of information and information to be generated.
Customer requirements—Review:
  Reminds one of areas to be considered.
Constraint criteria:
  Introduces the idea.
Project planning:
  Lists some methods.
Project structure and responsibilities:
  Guidance on programmes and schedules.
Configuration management:
  Refers to the EEA guide (Section 5.4.5).
Documentation standards:
  Mentions the need for standards.

5.4.7 Ministry of Defence MASCOT (UK)

MASCOT stands for Modular Approach to Software Construction Operation and Test and is the MOD-preferred design method for Real Time Software development. It was developed between 1971 and 1975 at the Royal Signals and Radar Establishment (RSRE) and culminated in an official handbook in 1980. Further information can be obtained from Computer Applications Division, RSRE, Malvern, Worcestershire WR14 3PS, UK.

MASCOT is more than a guideline to software production. It is a development methodology consisting of:

A standard graphical design method.
A suite of construction software.
An executive program for controlling module interactions.

It is therefore also addressed in Chapter 6 (Section 6.6.7), where various requirements methodologies are described.

Mascot was originally developed to assist in the design and development of Coral 66 based systems. Recently however the development of Mascot 3 has extended the capability of Mascot and also facilitates more readily the use of Mascot in an Ada based environment.

5.4.8 Ministry of Defence JSP188: Requirements for the Documentation of Software in Military Operational Real-Time Computer Systems (UK)

This is a Joint Services Publication (of 1980) which, in a similar manner to the IEE document (Section 5.4.2), describes the hierarchy of documents necessary to implement software maintenance. The definitions are somewhat different, however, and can be summarised as follows:

Level 1. This is a 'functional specification' level which includes:

System block diagram.
Software task and global data diagrams.
Text.
Timing details.

Level 2. This describes the tasks structure and defines data areas and the interconnection between parts of the system.

Level 3. This describes the internal functional structure of each process by means of flow diagrams.

Level 4. This is the module level where basic coded units are documented and coded. The module specifications and definitions (as described in Chapter 4) occur at this level.

5.4.9 IEEE Software Engineering Standards (USA)

The Institution of Electrical and Electronics Engineers (IEEE) have developed, over a number of years, an extensive set of standards to enhance communication between software engineers and to provide guidance on the types, formats and content of software documents as well as on the activities in the software life-cycle. The standards, whilst being detailed, are nevertheless more generic in nature in that many organisations would need to tailor them to their needs. That having been said they still represent one of the best set of standards available and are strongly recommended.

The standards available at the time of writing are:

729—1983 Glossary of Software Engineering Terminology
730—1984 Software Quality Assurance Plans
754—1985 Binary Floating-Point Arithmetic
828—1983 Software Configuration Management Plans
829—1983 Software Test Documentation
830—1984 Software Requirements Specification
854—1987 Radix & Format Independent Floating-Point Arithmetic
983—1985 Software Quality Assurance Plan
990—1987 Ada as a Program Design Language
1002—1987 Taxonomy for Software Engineering Standards
1003.1—1988 Std Portable Operating System Interface for Computer Environment
1008—1987 Software Unit Testing
1012—1986 Software Verification & Validation Plans
1016—1987 Software Design Descriptions

The above only represent the software and software related standards. There are many more which cover subjects such as LAN's, Backplanes, etc.

5.4.10 ElektronikCentralen: Standards and Regulations for Software Approval and Certification (Denmark)

This was published in 1984 and reviews the software quality problem with respect to safety critical systems (as does the HSE document, Section 5.4.1). Sources of failures and defences against them are described in some detail. The contents are:

Introduction.
Requirements to software in critical applications.
  Critical applications.
Microprocessor characteristics.
Software quality features.
Attributes for safety.
  High reliability design.
  Safeguards and handling.
  Wilful misuse.
  Fault dictionaries.
  Fault correction.

Fail safe.
Fail operational.
Measurement of safety attributes.
Software quality (as described in Chapter 4 of this book).

5.4.11 Guidelines for the Nordic Factory Inspectorates

ElektronikCentralen has prepared a report which will form the basic material for guidelines to improve the safety and integrity of systems using microprocessors. They propose safety assessments which follow the general format of the CEC collaborative project (see Section 11.5.5) and cover the following seven steps:

Defining the system boundary.
Hazard analysis.
Specifying the safety requirement.
Identifying safety critical subsystems.
Analysing the safety critical subsystems.
Certification criteria.
Change management.

The guidelines contain design recommendations and checklists.

5.4.12 TUV Handbook: Microcomputer in der Sicherheitstechnik (Germany)

This was written to meet the need for a set of requirements to cover microcomputerised safety devices and gives advice to the developers of safety critical microcomputer systems. It contains:

'a catalogue of system layouts/structures and safety measures for microcomputer control systems, and shows the way to select those safety measures which will ensure that, in a given situation, the requirements of the relevant legislative provisions will be complied with. There are in this connection a number of possible options from which the designer or manufacturer can choose, in accordance with the operating conditions which he has specified for his product, so as to provide the optimum solutions for the problems in hand.'

The book classifies five categories of safety applications and provides examples and standards for each. It is not, as such, a national standard, but will doubtless provide the main input to any future DIN/VDE standard.

5.4.13 EWICS TC7 Documents

In Section 11.5.4 the activities of this EEC-funded workshop on industrial real time computer systems are described. The output consists of:

Book 1 published in 1988 which includes six papers:

1. Development of Safety Related Software. October 1981
2. Hardware of Safe Computer Systems. June 1982
3. Guidelines for Verification and Validation of Safety Related Software. June 1983
4. Guidelines for Documentation of Safety Related Computer Systems. October 1984
5. Techniques for Verification and Validation of Safety Related Software. January 1985
6. System Requirements Specification for Safety Related Systems. January 1985

Book 2 to be published in 1989 to include:

1. System Integrity.
2. Software Quality Assurance and Metrics.
3. Design for System Safety.
4. System Reliability and Safety Assessment.

Book 3 to be published in 1989 on the techniques for safety assessment and design of industrial computer systems. It will summarise safety analysis, fault avoidance, fault and failure detection and containment.

5.4.14 CEC Collaborative Project

Section 11.5.5 describes this European-funded project and the intention to publish a set of guidelines and also a wide-ranging review of documents in the PES area. The guidelines will contain:

A guidelines framework for assessing PESs.
A future strategy for developing the guidelines.
An assessment example.
A tabulation of techniques.
A review of electromagnetic capability requirements.
A glossary.

5.4.15 US Department of Defense Standard 2167: Defense System for Software Development

This is a 90-page document (4 June, 1985) which is difficult to summarise in a few lines. It contains much detail concerning the methods described in this book and has several useful diagrams of the software development cycle. The overall sections are:

Definitions.
General requirements:
Involving the software design cycle, organisation, subcontract, etc.
Detailed requirements:
Two to three pages on each aspect of the life-cycle (requirements, preliminary design, detailed design, coding, testing, etc.).
Coding standards.

The standard has recently undergone a revision with some change of emphasis and additions. Perhaps the most interesting change is that contractors now have to produce a risk management plan in which their strategy for the management and assessment of risk is related to the specification, design and development of software.

5.4.16 IECCA: Guide to the Management of Software-Based Systems for Defence, 3rd Edition

The Inter-Establishment Committee on Computer Applications was set up in 1968 and is an MOD body. Its purpose is to address real time computing and computer-based systems and, in particular, the evaluation of CORAL 66 and Ada compilers as well as the development of MASCOT. It also considers the production of advice, recommendations and guidelines.

This guide concentrates on real time systems and covers the main items in the design-cycle. It also addresses procurement policies:

System life-cycle:
Outlines the cycle and the main activities.
Potential problems and risks:
Escalating requirements, delivery, cost, maintenance.
Management techniques:
Feasibility, specifications, validation and test.
MOD Policy for real time systems:
Hardware, software, administration, future.
Impact of procurement policies:
National, EEC, NATO.

The guide may be obtained from The Secretary, IECCA, RSRE, Ministry of Defence, St Andrews Road, Great Malvern, Worcs. WR14 1LL, UK.

5.4.17 I Gas E: SR15, The Use of Programmable Electronic Systems in Safety Related Applications in the Gas Industry

In response to the HSE Guidelines (5.4.1) the Institution of Gas Engineers plan to publish, in 1989, a second tier guidance document. It aims not to repeat the guidance given in 5.4.1 but to provide additional specific guidance for safety related systems in the gas industry. Specific applications sections cover:

Burners and combustion.
Domestic appliances.
Supply pressure and flow control.
Fire and gas and shut-down systems.
Plant and grid management systems.
Distribution holder control.
Miscellaneous.

5.4.18 EEMUA: Safety Related Programmable Electronic Systems

The Engineering Equipment and Materials Users Association, also in response to the HSE Guidelines (5.4.1), have published a second tier document entitled 'Safety Related Programmable Electronic Systems'. It covers:

Relevant standards.
Definition of terms.
Assessment of application requirements.
Qualitative assessment of safety system applications.
General design consideration.
Design specification.
Changes and modifications.
Design-environmental aspects.
Testing and commissioning.
Operation and maintenance.

5.4.19 STARTS: The STARTS Guide

The STARTS (Software Tools for Application to Real Time Systems) guide was prepared under sponsorship of the DTI and coordinated by

the National Computing Centre. The emphasis is on tools and methods which are available for the management and control of software development. The preparation of the guide was undertaken by teams drawn from industry and is able to draw on their experience and use of the tools and methods. There are five categories within the guide which is now in its 2nd edition.

Project management.
Configuration management.
Project support environments.
Requirements definition and design.
Verification, validation and testing.

For each of the above headings the range of currently available tools and methods is discussed and described with an assessment given against common criteria. In addition there are three other publications:

The STARTS Purchasers' Handbook—Published by the STARTS Purchasers Group of major public and private sector purchasers of large real-time systems. It harmonises their procurement practices and software engineering requirements to be placed on suppliers. The Handbook outlines best practice in specifying, purchasing, and maintaining real-time systems and indicates the level of software engineering which purchasers should expect from their suppliers.

STARTS Debrief Reports—Practical reports by users documenting their experience with specific software methods and tools contained in the first edition of the STARTS Guide.

A short video programme—Introduces the STARTS approach to senior management and others new to software engineering.

5.4.20 Some Other Documents

EQD Guide for Software Quality Assurance, MOD Procurement Executive, 1977.

An early attempt to give guidance on the subject.

JPL Publication 78-53. *Standard Practices for the Implementation of Computer Software,* NASA.

A description of software management functions.

MIL-STD-1679. *Weapon Systems Software Development.*

This outlines the minimum requirements for software development.

MIL-S-52779A. *Software Quality Assurance Program Requirements.*

This is a US military guide equivalent to the above EQD document. It is compatible with AQAP 13.

Software Engineering Standard (ESA).

A European Space Agency document which describes the full software development life-cycle.

British Standard 4058. *Data Processing Flow Chart Symbols, Rules and Conventions.*

British Standard 5476. *Specification for Program Network Charts.*

British Standard 5887. *Code of Practice for Testing of Computer Based Systems.*

British Standard 3527. *Glossary of Terms Used in Data Processing.*

British Standard 5515. *Code of Practice for the Documentation of Computer Based Systems.*

UK Defence Standard 05-67. *Guide to Quality Assurance in Design.*

One section deals with software quality.

5.5 SYSTEMS FOR THE FUTURE

Design requires a far more structured and integrated set of rules, procedures and validation methods than are provided by the traditional hardware quality techniques. As a result, systems and standards will continue to develop and will address new methods and tools as they become available.

Looking ahead, the three features which will make quality systems more effective in preventing and removing faults are:

Automation of controls. The control of documents and configuration management, along with other clerical procedures, will reduce the probability of faults being produced by use of the wrong issues of media or documents. A computer clerical system which will not proceed unless all the required documents, including inspection and audit reports, are produced will make it very difficult to omit essential documents.

Automation of specification and design. At present, requirements specifications are written in ordinary English language. This has the advantage that any nuance of expression may be attempted but carries

the associated disadvantage of ambiguity. Currently, development is in progress on a range of more formal structured expression languages which will restrict the writer to a higher level English language expression set. Mathematical methods can then be used to verify the specification. This verification process will be automated into a type of 'macro-compiler' which will drastically reduce the more difficult faults, namely those arising due to subtle ambiguities and omissions in the specification. Chapter 6 will expand on this area.

Automated review of code. A range of test tools, known as static analysers, applies many tests to the source code by means of automatic suites of software. Whereas the requirements languages will improve the specification activity, these analysers will identify the faults which are created during design and coding. They are described in Chapter 8.

5.5.1 Paperless Design

The ideal for many years has been the idea of maintaining the whole gamut of software, specification, design and development documentation in paperless form, i.e., in some automated system. Recently such systems have begun to proliferate and are known by the acronym CASE, standing for Computer Aided Software Engineering.

These tools enable development of software to take place at a graphics workstation and supply, more or less, all the tools necessary to develop software, including real-time systems. The main achievement so far of CASE tools has been the enabling of methodical development of software through databases.

The overall process of CASE development starts with analysis of the problem. Software analysis involves turning, broadly stated, ambiguous requirements into detailed and consistent software specifications. The most popular method of achieving this is to use Yourdon-De Marco diagramming methods which have been extended to real-time and embedded systems using the Hatley and Ward–Mellor methods, although these methods have not yet been widely adopted. Design is achieved through further use of data-flow diagrams, structure charts and other diagramming methods which provide good graphical representation of the system. CASE tools should not just be automated diagramming tools but should also provide static checking to verify that analysis and design rules are obeyed and that specifications and design are consistent and complete. Part 3 looks at some of the methods and tools in more detail.

PART 3

Software Quality Engineering— An Ideal Approach

The next five chapters address the ideal software design approach in each of the stages of the design cycle from requirements to test.

Chapter 6

An Engineering Approach to Defining Requirements

6.1 ENGINEER VERSUS PROGRAMMER

Anyone with mathematical aptitude and a little training can write a computer program. The proliferation of computing in both home and school is sufficient illustration of that fact. However, a program which meets the requirements of easy use and maintenance demands quite separate skills from those required for mere coding.

This need is often not appreciated by software managers who, themselves, may well have risen from the ranks of these programmers. Promotion is frequently on the basis of rewarded good or fast programming despite the unfortunate fact that the skills required for software management are not the same as those being regarded.

The software engineer, as distinct from the programmer, will be concerned with:

Establishing the user requirements.
Verifying the user requirements.
Assessing and choosing the software design methods for the project.
Structuring and specifying tests.
Planning and implementing design reviews.
Choosing software tools (e.g. validators, compilers, test beds).
Criticising methods at the end of the life-cycle.
Deriving job satisfaction from team rather than personal achievement.

Recalling Chapters 3, 4 and 5, we looked at quality disciplines and controls which are applied over the programming activities largely in order to remove faults. The software engineer, however, will embrace these controls within himself and thereby generate better-quality

software in the first place. This does not remove the need for the software quality function in the design process but it does provide an environment conducive to the production of fault-free design. The benefits which follow from less schedule slippage and fewer failures during test and integration will lead, in turn, to:

Better team motivation.
Milestone dates being met.
Smoother and hence more effective testing.
Lower costs.
Fewer changes, hence fewer subsequent faults.

The software engineer will be conversant with automated requirements expression languages and automated review tools such as static analysers. Thus, traditional flowcharting/coding skills will become much less significant. They will be replaced by more management-oriented skills including:

Top-down decomposition of the requirement into a hierarchical design.
Knowledge of automated tools.
Ability to plan and control design reviews and tests.
Ability to negotiate and interpret user requirements.

6.2 A NEW LOOK AT THE LIFE-CYCLE

This shift in emphasis from programming towards the concept of software engineering puts a new focus on the software life-cycle. Significant factors are:

(a) The emphasis is shifted from correction to prevention.
(b) The focus is on the requirements specification.
(c) Formal mathematical techniques in specifying the requirements and in carrying out the design permit more complete, correct and unambiguous code.
(d) The design process is longer and involves more frequent and detailed design reviews.
(e) Test requirements become more embracing since they are evolved from the very beginning of the design process.
(f) There is better visibility to the design and code, hence diagnosis and field maintenance are smoother.

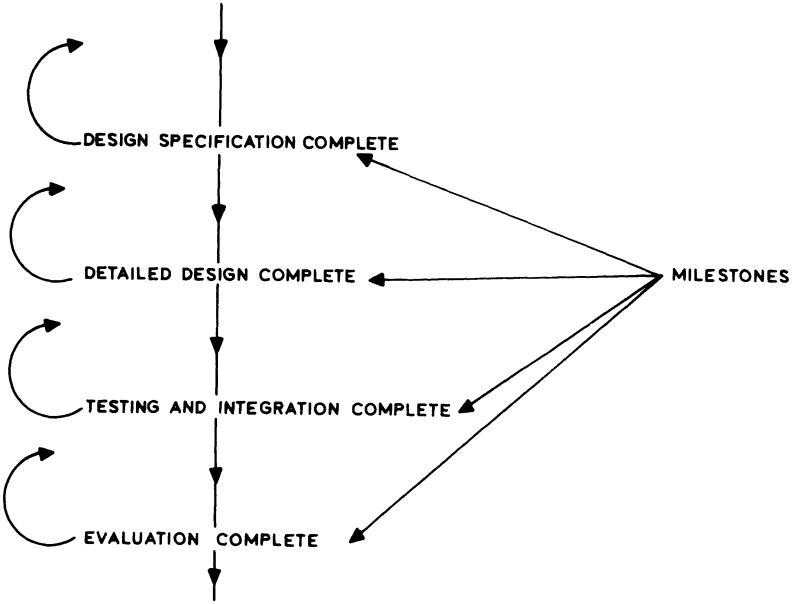


Fig. 6.1.

The main aim is to prevent or remove faults as early as possible in the cycle and thus minimise man-hours. Figure 6.1 shows the familiar design cycle but emphasises two aspects:

(1) Feedback is provided at each stage of design. The emphasis is on making each stage of the design fault free before progressing to the next. The software engineer or enlightened software manager will recognise the need to resist pressures to save time on review in order to carry on with further design. It will be understood that this will only cost more time in the long run and that by then the position will *not* be recoverable by expending extra effort.

(2) There are four milestones for formal review. These formal milestones do not obviate the need for individual design reviews after each separate work package. Design review of each module will be carried out at supervisor level with the engineer and his colleagues.

6.3 CURRENT STATE OF THE ART

Regrettably, current practice usually emphasises test results and test methods. In contrast to the software engineering approach, the emphasis is later in the life-cycle, resulting in larger and more expensive modification (Figs 1.4 and 1.5, Chapter 1). It leans more towards correction than to prevention and fails to recognise the benefits of the philosophy and techniques which occupy the remainder of this book.

The current emphasis placed on dynamic testing is difficult to justify since there is no strict methodology behind it other than the concept of 'full coverage' testing. However, complex systems defeat this attempt at completeness in testing. In many cases whole portions of code are opaque to the testing process.

Some automated techniques are in current use, mostly in the area of test (see Chapter 8). They improve the speed and efficiency of the debug process but still emphasise correction rather than prevention. Similarly, the two static analysers currently in use (Chapter 8) remove faults already committed in the code.

The breakthrough in software quality engineering requires a fundamental change to the initial process of describing the requirements. Frequently the actual problem to be solved is not clearly understood, as a result of which one takes refuge in describing the system in terms of its architecture, that is to say in terms of memory, peripherals, compilers, communications, etc. However, this pre-empts the design process, which is where such features should be specified.

At the requirements stage it is a 'functional architecture' which should be defined. The functional architecture of a problem is by no means the same as the system architecture employed to solve it.

There is an increasing awareness that validation and test aids are not enough to ensure software quality and that improved requirements specification techniques are required. There is much development in this area and the remainder of this chapter will describe the situation.

6.4 FORMAL VERSUS FREE EXPRESSION

It has already been stated that the original requirements specification is a major source of software failures. Clearly, if the programmer has been directed to solve the wrong problem, or if the requirements are

incomplete, ambiguous, or not understood, then even error-free design will still result in system failures as perceived in use. The potential for creating faults in the requirements specification arises largely from the fact that they are written in freely expressed English language. On one hand this permits a comprehensive description but, on the other, provides a vehicle for ambiguity and lack of clarity.

Formal requirements languages are fairly new, stemming from the middle 1970s, and in most cases are still under development along with the software tools which accompany them. They constrain the writer to the use of mathematically precise methods and expressions and, as a result, protect against ambiguity. Because they involve mathematical rules it is possible to verify a requirements specification by formal analysis methods. These 'proofs' are highly theoretical and are, mostly, carried out manually. In the future this process will certainly become automated and indeed some tools already exist.

6.5 EXPRESSING REQUIREMENTS—SPECIFICATION TECHNIQUES

The fact is that system requirements are highly complex, interactive and often ill-defined. There is no avoiding this problem and it must be faced that inadequacies in requirements definition lead to faulty design. Hence a requirements specification must embrace the following descriptions which together comprise the total requirement.

(a) *Why* the system is needed in technical, economic, maintenance and operating terms.
(b) *What* functions the system needs to fulfil. This does not involve *how* they are actually performed since that is part of the design.
(c) *Conditions* which place limitations on the design.

In an earlier paragraph (Section 6.3) we referred to a functional architecture. Requirements specifications seek to communicate this and the design process to implement it by means of a system architecture. The approach is therefore more abstract, which unfortunately deters many from considering the methods available. They need not be any harder to understand.

The difficulty inherent in specification is the language. If plain

English is used to express the specification then it is necessary to embrace not only the problem to be defined but the imprecisions and ambiguities of the language. Pitfalls include:

Dangling ELSE:
e.g. A must equal B or C. No mention of what happens if not.
Ambiguity of reference:
e.g. Add X to Y. This must be positive. What? X, Y or their sum?
Ambiguous words:
e.g. 'usually', 'quickly'.
Ambiguous logic.

Plain language provides the scope for free expression and allows the author a high degree of originality but the fact remains that the number of interpretations of the requirement may then be large.

The alternative is to constrain the use of language to some well-defined set of constructs and to force the writer to use only that language. This will largely remove the ambiguity but will tend to increase the volume of text required to describe a given problem. Furthermore, the language may be constrained so that a number of features are difficult or even impossible to describe, in which case one particular language may not always be appropriate for a given application. Nevertheless, the level of precision achievable with formal specification languages far outweighs their drawbacks and their use within software development will substantially increase in the next few years. A number of major organisations have committed themselves to their use even though this has required a substantial training effort. One serious impact of their use is that all levels of the organisation, from programmer to senior management, must understand the technique.

The organisations which have done this have found great benefits, not the least of which is that all levels of personnel within the organisation are able to understand and discuss the various types of document generated, be they detailed design documents or high level requirements specifications.

One very important aspect of introducing such methods is to ensure the availability of any support tools which are needed. For example, a language syntax checker will enable specifications to be developed on a computer, thus speeding the preparation of documents.

6.6 AVAILABLE SPECIFICATION LANGUAGES AND DESIGN METHODOLOGIES

A number of methodologies are described in the following pages. *The first six are specification languages* and Sections 6.6.7 to 6.6.13 describe *design methodologies.*

SPECIFICATION LANGUAGES

6.6.1 IORL (Input/Output Requirements Language)

The language was developed by Teledyne Brown to allow enforcement of a rigorous methodology for system development with the requirement that it should be easy to use. In particular, it was aimed at engineers so that they could express system performance characteristics and algorithms. IORL is a graphics and tabular specification language. The highest level is the Schematic Block Diagram (SBD). SBDs are rectangular boxes that identify all the principal system components and the data interfaces which connect them. In IORL the designer must maintain a distinct difference between control flow and data flow. SBDs are decomposed into other SBDs until decomposition is no longer feasible. Each SBD represents a different document. The Predefined Process Diagram (PPD) is used to depict the detailed flow logic of a single predefined process. It is used to improve the readability of the specification, to allow identification of dependent components and to permit the specification to be presented in a hierarchical manner.

A CASE tool called Tags is now available which uses the IORL requirement language to express software specifications. IORL uses blocks and icons to create flow and timing diagrams. Parameter tables accompany the diagrams. These specifications can be executed to simulate the real-time operation of the software being modelled.

6.6.2 CORE (COntrolled Requirements Expression)

This is an analysis method originated by systems designers and BAe, consisting of a set of formal techniques for gathering and structuring requirements information. CORE is a graphical technique whose features have been derived from other widely used specification methods. The notation can be used both for requirements and for design. The CORE method consists of a number of steps, each of which must be completed for each level of decomposition. These

stages are:

(1) Problem definition.
(2) Viewpoint analysis.
(3) Tabular collection.
(4) Data structure diagrams.
(5) Isolated viewpoint action diagrams.
(6) Combined viewpoint action diagrams.
(7) Non-functional requirements.
(8) System constraints.
(9) Completion.

The technique has been in use for some time but needs automated support if used for large systems.

6.6.3 VDM (Vienna Development Methodology)

This stems from development carried out at IBM in Vienna by Cliff Jones. VDM is a rigorous, mathematically based method which allows specification to be carried out in a mathematical form. VDM provides a formal notation and a variety of reasoning techniques suitable for most applications.

VDM is a model based approach in which specifications are explicit system models constructed out of either abstract or concrete primitives. The model base approach is well established and has been under development for many years.

VDM is based on the use of mathematical abstractions such as sets and mappings. Its original use was for the formal definition of programming languages. It has been subsequently developed and applied to a wider set of applications such as operating systems and databases.

VDM provides more than just a specification language. Besides a formal notation a method should supply rules and procedures to be followed in the various stages of system development.

VDM is model based in that descriptions of systems are given as models. The constituents of these models are data objects which represent the inputs, outputs and the internal state of the system, and operations and functions which manipulate the data. VDM encourages the top-down approach by supporting abstraction at the uppermost levels of description. At the uppermost level of specification an abstract model is given which captures only the system concepts necessary to explain the required functions of the system. Data objects

are then specified using very abstract mathematical data types and the operations and functions that manipulate them are specified implicitly or constructively using recursive functions.

A number of types are built into VDM such as *Int,* the set of integers, *Nat0,* the set of non-negative integers and *Nat* the set of positive integers. New sets of scalars can be introduced using type definitions, for example, *Colour* = {Red, Green, Blue}. Types can then be manipulated in a number of ways. Thus the declaration *Distance* = *Nat0* introduces a new name for the type *Nat0* which would be more meaningful. By the use of types, operations, etc. it is thus possible to build up a specification which provides a formal definition of the system we intend to build.

6.6.4 Z

The Z notation is a language for expressing formal specifications of computing systems. Like VDM it is a model based approach. It is also based on typed set theory and has the 'schema' as one of its key features. This consists of a number of named objects with a relationship specified by axioms. Z provides notations for defining schemas and later combining them in various ways so that large specifications can be built up in stages.

As an example of a schema, consider a database consisting of people's names against each of which is stored a telephone number. The state-space of the schema is described by a schema called DBtelephone:

DBtelephone

$$known : \mathbf{P}\, NAME$$
$$phone : NAME \rightarrow PHONE$$

$$known = \mathrm{dom}\, phone$$

Two components of this schema are the set of *known* of names known to the database and the partial function *phone* which records the telephone number against certain names.

As an example the following is a possible state of the system:

known = {Wood, Smith, Elsevier}
phone = {Wood → 01-123-1234, Smith → 01-234-2345, Elsevier → 01-345-3456}

From this state-space we can then go on to describe events which might happen.

Schemas can be used to describe all aspects of a system: the states it can occupy, the transitions it can make and, as we transform the specification into design, the relationship between one view of a state and another. Like VDM, Z thus provides us with a formal means of describing a system.

6.6.5 OBJ

OBJ is a language for writing and testing algebraic program specifications. It has seen development by teams in the USA and in Great Britain with executable versions being implemented. The basic motivation for OBJ is that experience has shown that it is difficult to produce specifications which are error free even for relatively small systems. OBJ provides facilities to assist the development of correct specifications. First it allows objects to be defined which allow the specifications to be broken down to a size which can be grasped conceptually. Second, facilities are provided for testing these small specifications and also to examine their interconnection. Other features are its strong typing, the systematic use of error conditions and of syntactic and semantic consistency checks. In some respects OBJ can be viewed as a programming language with inefficient execution. Objects in OBJ distinguish three kinds of operator: those used in normal situations, those used in error situations and those in recovery situations.

6.6.6 SREM (Software Requirements Engineering Methodology)

This was developed in the mid-1970s in the United States and its application, to date, has been largely in military control and data systems. It is similar to PSL/PSA (Section 6.6.12) and incorporates a 'stimulus/response' facility which permits automatic simulation to be carried out which generates feedback information on the characteristics and performance of the system being defined. The language and graphics are:

RSL (Requirements Statement Language).
R-nets (a graphical method).

Figure 6.2 illustrates the R-nets method.

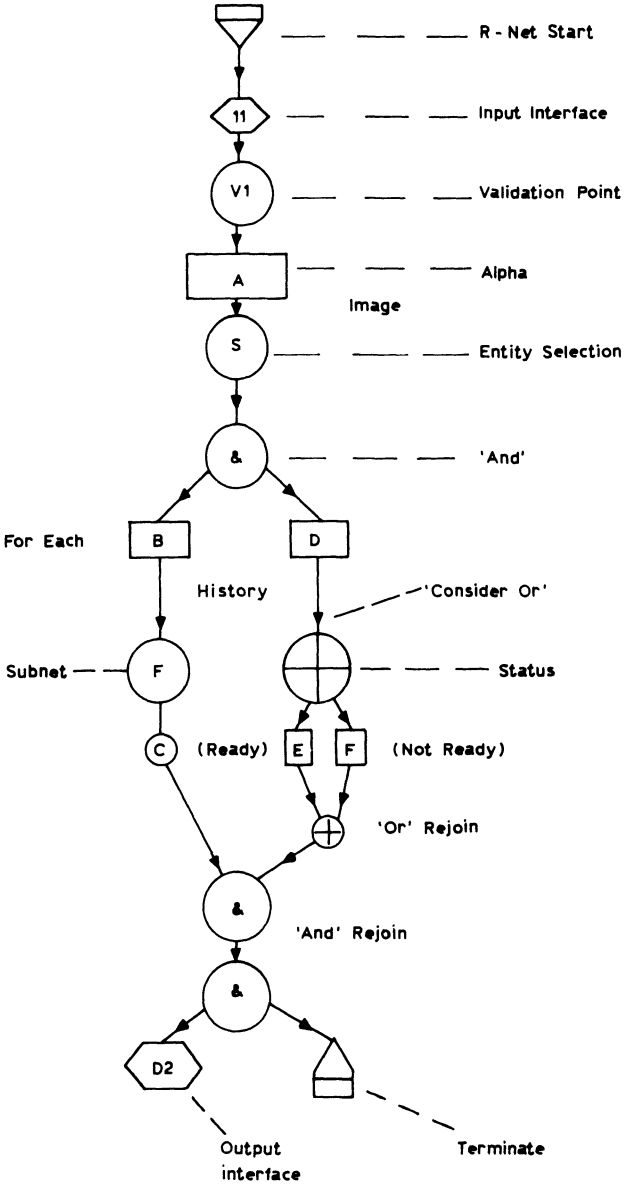


Fig. 6.2. An example of an R-nets diagram.

6.6.7 MASCOT (Modular Approach to Software Construction, Operation and Test)

This is a method (also discussed in Section 5.4.7) for the design and implementation of real time systems. The method is widely used in the UK defence and avionics industries and originated from work done at RSRE. MASCOT is characterised by the use of a graphical data-flow network as the medium for expressing software structure. It is combined with a systematic development method which ensures that this structure is accurately reflected in the resultant software. Inter-process communication is handled by a special type of design element which encapsulates the shared data storage and the access mechanisms which implement the synchronising actions necessary to preserve the integrity of the shared data. All the parallel processing knowledge of the system is thus isolated from the purely algorithmic concerns dealt with in the processes. A MASCOT application is tested and operated in a standard context which provides a set of run-time executive-level facilities for such purposes as process scheduling and synchronisation.

6.6.8 SSADM (Structured Systems Analysis and Design Methodology)

This grew out of work by the UK Government's Central Computer and Telecommunications Agency and Learmonth, Burchett Management Services, to devise a standard software design methodology applicable to the broad scope of work performed by various government departments.

SSADM is not a collection of techniques but a well-defined step-by-step approach. Rather than concentrating on one approach, such as functional analysis, SSADM regards functions and data with equal importance. SSADM is divided into six stages:

1. *Analysis.* Construct a logical model of the system.
2. *Specification of requirements.* Document the problems of the current system and the requirements of the new system.
3. *Selection of system option.* Identify and document the operational requirements of the new system.
4. *Logical data design.* Complete a detailed logical data design.
5. *Logical process design.* Complete a set of detailed logical process designs.

6. *Physical design.* Translate the logical design into database or file descriptions and the logical process designs into program specifications.

The primary techniques used in the six stages, but not in all, are data-flow diagrams, entity models, entity life histories, data normalisation, process outlines and physical design control. A standard set of forms to assist in developing systems also exists. One CASE toolset which attempts to cover a large part of SSADM is available from LBMS and is called LSDM.

6.6.9 JSD (Jackson System Development)

The Jackson methodology is a method for specifying and implementing computer systems. Within JSD there is a distinction between specification and implementation. The development procedure has six steps, of which four are concerned with creating a specification of the required system and two with its implementation.

JSD approaches the problem via the idea of a model and its relation to function. First an abstract description of the reality is written and then one determines how that abstract description can be realised in a computer model. In JSD the abstract description is created in two steps, and its realisation in a third step which is the development procedure.

It is not until the fourth step that system functions are considered. The techniques used in implementation have a common theme in that they allow the developer to make decisions, when the system is built, that might not otherwise have been made until it is run.

6.6.10 SADT (Structured Analysis and Design Technique—Ross)

This has been in use since 1974 and is a general-purpose modelling technique which can be used to describe a range of problems not necessarily confined to computer systems. It is a graphical language involving top-down decomposition of complex problems into easily perceivable elements. SADT consists of three aspects:

(1) The graphical language which communicates the requirement (*actigrams*).

(2) Methods which decompose the problem so that the graphical language can be used to describe it.

(3) Management and human factors rules which guide and control the above methods and diagrams.

The result is an ordered set of interrelated diagrams where each diagram must contain three to six boxes and occupy a single page. The number of diagrams will depend on the complexity of the problem and they will be connected by upwards and downwards referencing in a logical hierarchical structure. A top-down overview diagram will précis the total problem.

Diagrams consist only of boxes, arrows and text. Arrows represent relationships between boxes and should not be confused with data-flow or sequence arrows in other types of diagram. Figure 6.3 shows (a) an overview structure and (b) a single diagram (actigram).

6.6.11 SSA (Structured System Analysis—De Marco)

This also dates from the 1970s and is finding much application in data processing. Top-down and graphical techniques make it fairly similar to SADT but it incorporates some additional data base features. These

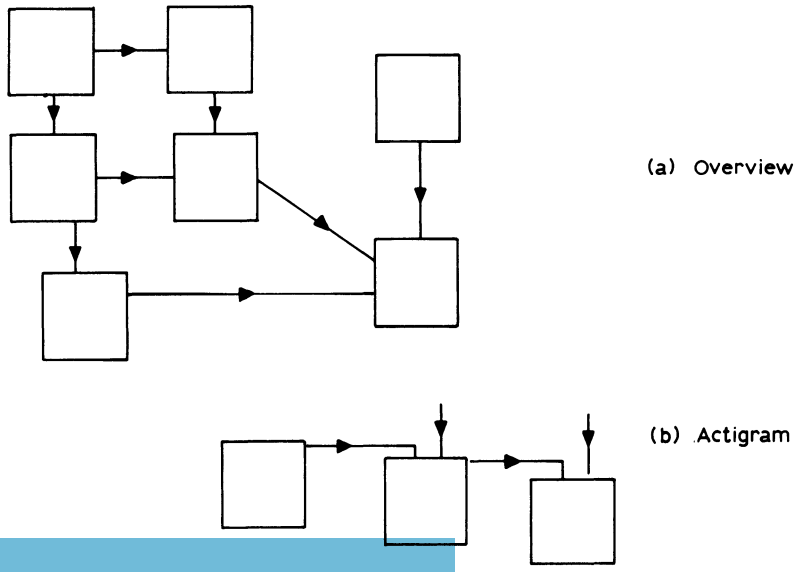


Fig. 6.3.

are:

Data flow diagrams.
Data dictionaries.
Process logic diagrams.
Data structures.

A standard data dictionary is used to define the various data elements in the requirements. The diagrammatic logic for describing the problem includes decision trees and decision tables. SSA is, hence, applicable to data processing systems and real time problems with substantial data flow and manipulation.

6.6.12 PSL/PSA (Problem Statement Language/Analyser)

This is a language originating in the mid-1970s. It was developed specifically for expressing requirements specifications. It is automated and can, to some extent, reveal gaps in the information flow or the existence of unused data. The three features are:

The information is in a computerised data base.
Processing is by computer.
The emphasis is on *what*, not *how*.

6.6.13 Petri-nets

This technique has evolved over the last ten years as a method for describing real time systems. The elements of a net are:

Token types which specify each type of information.
Places.
Transitions.
Arcs.
Initial marking.

The important feature is the translator which converts the Petri-net into a program structure such as Ada (see Chapter 9).

6.6.14 Object Oriented Design

In object oriented design, the software components are seen as objects rather than as functions. Each object has an associated set of permitted operations and objects communicate by passing a message where the message usually includes an instruction to activate a

particular function. OOD is founded on the principle of information hiding and on abstract data types and although it is relatively undeveloped at present it is likely to gain importance in parallel with the increase in use of the Ada language, Ada being founded on similar principles.

6.7 FUTURE TRENDS AND GOALS

One of the characteristics of the methodologies is that, in the past, they have been 'pencil and paper' techniques. More recently several of the techniques have begun to appear in automated form which allow the designer to manipulate diagrams on a workstation. These systems usually have, at their core, a data base or data dictionary which allows the designer to keep track of the various entities which he has created. The automation of design methodologies is a major step forward since it removes much of the drudgery in creating the diagrammatic requirements of each.

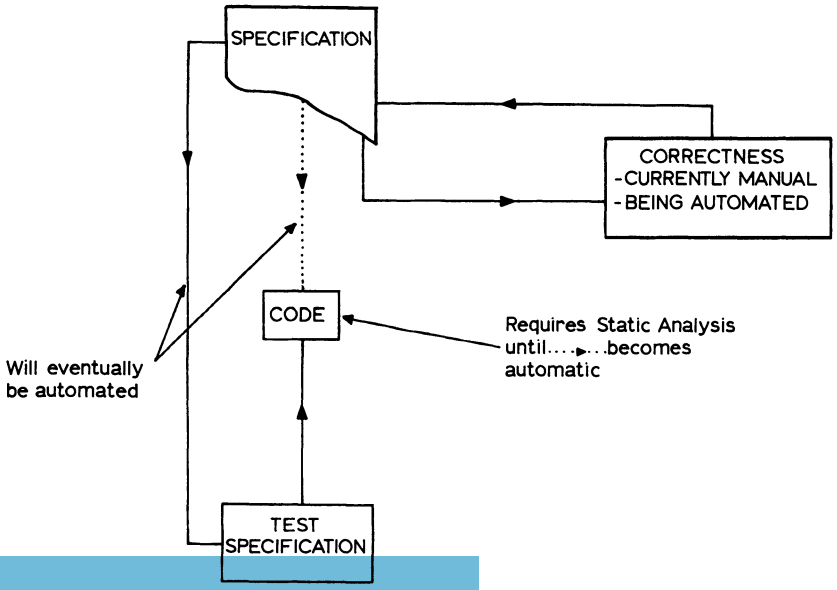


Fig. 6.4.

At the outset, specification techniques have been based on computer systems since it was recognised early that the high level of checking necessary in specifications could only be performed in that way.

The main tasks confronting the software industry today are specification techniques and the 'wrapping up' of the whole life-cycle into a largely automated system so that self-checking for consistency, completeness and so on can be left to the computer. A number of projects are running which will achieve this goal in the near future. Figure 6.4 illustrates this view of the design-cycle.

A great deal of work is also in progress on a variety of analysis systems which provide some degree of proof of correctness.

Chapter 7

Putting Design into an Engineering Context

7.1 VERIFICATION AND VALIDATION

Verification and validation are often confused and it is worthwhile discussing them again in the context of Fig. 1.2 of Chapter 1. There are two ways of looking at the definition of verification and validation. The more formal definitions are:

Verification: The process of ensuring that the result of a particular phase meets the requirements of the previous phase.

Validation: The process of ensuring that the results of the whole project meet the original requirements.

However, shorter definitions are sometimes used, one in particular by Boehm:

Verification: 'Have we built the product right?'

Validation: 'Have we built the right product?'

In the first case the emphasis is on the product for its own sake and clearly implies a process of continuous checking throughout development. Validation, on the other hand, requires the existence of a completed product in order that one may ask 'Have we met the customer's requirements?' Contrast this with the verification question 'Have we met the requirements of the previous phase?' (of the design).

7.2 THE DESIGN PROCESS

Let us take a closer look at the design-cycle, with particular attention to the need for verification at each stage. Figure 7.1 is a breakdown of the design stages following from the user requirement specification.

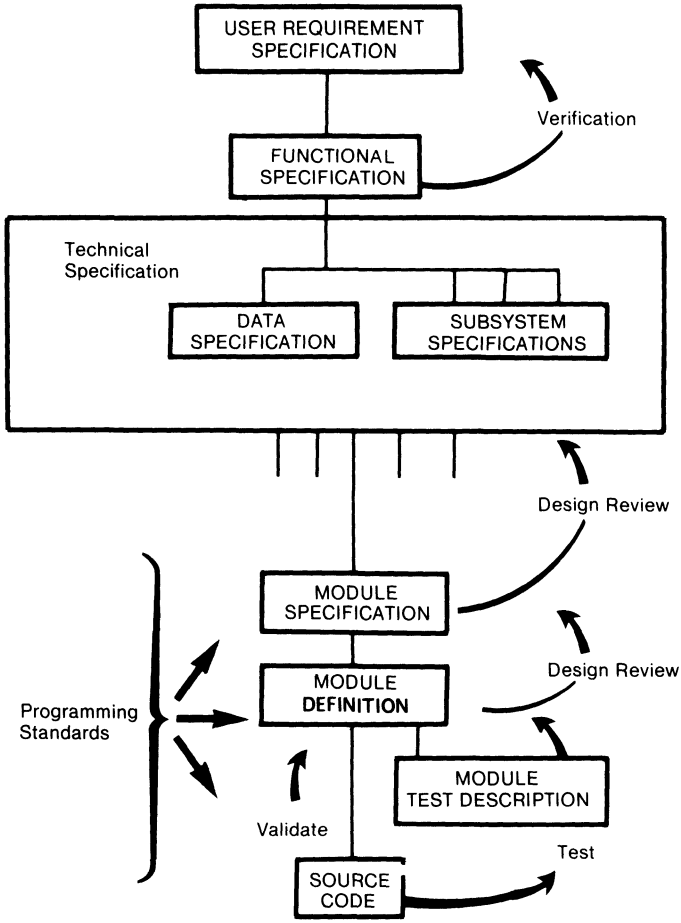


Fig. 7.1. The design-cycle.

It may seem strange that the requirements phase is included under the heading of design-cycle. There is a great deal of evidence that keeping design out of the requirements phase achieves a clearer specification of 'what is required' rather than allowing pre-conception of 'how to implement' to cloud the issue. Nevertheless it often happens that the implementation phase reveals features of the system which result in changes to the requirements. Thus to exclude the requirements phase from the design-cycle would be unrealistic.

Many different ways of viewing the design process have evolved

using a variety of different ways of 'viewing' the system. Some of these 'views' have already been described in Chapter 6. Normally the first stage of the design process is to identify functions, namely those aspects which are externally apparent to the system user.

This is clearly a difficult stage since the system developer, from the requirements, has to decide on the way in which the system will appear to the user. To decide, from requirements, all the operational characteristics is at best extremely difficult. Having achieved that aim, however, the system designer has a firm foundation on which to base his design decisions in developing a system which responds in the manner required. By moulding functions and data he is able to decompose the system to the point at which the programmer can perceive all the functions required at a particular module and thus will be able to control its consturction more efficiently. Iteration back through previous stages allows the design to be refined and then validation provides the confidence that the system produced is correct.

7.3 PROGRAMMING STANDARDS

In Section 4.5 the use of programming standards was introduced by describing structured programming and outlining some types of design diagram. The use of programming standards in design projects, although on the increase, is by no means as common as it should be. Programming standards range from a simple statement of layout and presentation of code to sophisticated guides covering many aspects of design and coding. Ideally, in large development projects, there should be a Company programming standard which is used as a template for the production of project-specific standards. Thus, at the beginning of each project, the software manager prepares a set of documentation and programming standards. These might consist of the following standards.

7.3.1 Module Specification Standard

It will describe the purpose of a module specification by delineating the need for a subset of requirements which are to be performed by that module.

It will state the areas of performance criteria which the designer must address when preparing the specification (e.g. response times, arithmetical accuracy, processing priority, data flow to and from other modules, degraded performance criteria under overload conditions).

It will state the need to specify the module interfaces and the data bases to be accessed.

Depending on the project, and the high level language to be used, pseudo code, decision tables, flowcharts and state diagrams may be used and the standard will discuss the techniques to be employed.

It will also require the designer to include, in the module specification, an estimate of the quantity of code and, if critical, the execution time.

Requirements for project-specific documentation numbering, relative to other specifications, will be included, as will formats for layout and legibility of the specification.

7.3.2 Module Definition (Documentation and Code Package) Standard

This consists of the package of work papers which actually constitute the software module. It includes:

The module specification.
The listing of code.
The test definition.
The test results.
Notes.

At a lower level of detail the standard will list the contents of the module definition package and the requirements for outward referencing to other specifications. It will require the following to be included:

(a) Files belonging to the module.
(b) Files accessed by the module.
(c) Complementary explanation to enhance the code. This should not detract from the requirement that code should contain adequate comment.
(d) Diagrams to describe data flow, flowcharts, etc., as applicable.
(e) Timing calculations (e.g. execution, disc access, CPU time).
(f) Notes of any compiler problems.
(g) Error-handling philosophy.
(h) Test requirements (e.g. harnesses, simulators, hardware).
(i) Inspection/walkthrough records.
(j) Modifications.

Again it will state the documentation numbering conventions for the project and will give rules for format and legibility.

The test description is part of the module definition package and will

include a requirement for the following to be described:

The item to be tested.
The tests to be carried out.
The hardware required.
The test results.
A summary of the results.
A summary of the errors.
A summary of any modifications.
Diagnostic details as they arise from the debug process.

7.3.3 Software Coding Standard

In addition to the module definition standard, a software programming standard will extend to the actual production of code. Thus, constraints and guidance are extended to the very bottom of the design hierarchy.

In the ideal case of a project specific standard the following will be included:

(a) A description of the computing environment including the operating system.
(b) The language and compiler (including version number).
(c) Details of compiler evaluation (see Chapter 9).
(d) Conditions under which the use of machine code is permitted (e.g. time-critical functions).
(e) Editing facilities.
(f) Rules for limiting the use of globals.
(g) The precise meaning of such terms as module, subsystem and routine and their functional boundaries.
(h) Header contents (e.g. name, date, issue, author, amendment history, calling sequence of module, synopsis of module, input and output parameters, data and globals accessed, messages passed to and received by the module, etc.).
(i) General advice on coding strategy.
(j) Rules for layout of the block structure and conventions for use of upper and lower case type (see Fig. 7.2). This includes the use of spaces, indented nesting, pagination, etc.
(k) Rules for the use of specific commands (e.g. limit on use of GOTO, use of simple IF conditions, absence of dangling THEN or ELSE commands).

(l) Rules within loops (e.g. entry, exit, forbidden commands within the loop).

(m) Code template examples for regularly used sections of code (see Fig. 7.3).

(n) Algebraic rules (e.g. the ambiguous A/B*C rather than A/(B*C)).

(o) Naming conventions referenced to the technical specification.

```
BEGIN
WHILE NOT KEY DO
        IF ENTRY ALLOWED THEN
        BEGIN
                FOR i: = 1 to n do
                CHECK (ACCESS, i, RESULT);
                IF RESULT DOES NOT = TRUE THEN
                ERROR EXIT ELSE
                SET ACCESS;
        END;
        IF ENTRY DENIED THEN
        BEGIN
                REPORT ENTRY (ACCESS-CODE, CODE);
                RAISE ALARM (CODE);
                IF TIME > 18 THEN
                BEGIN
                        REMOTE DIAL (CODE);
                        LOCK ALL;
                END;
        END;
END-DO;
```

Fig. 7.2. Sample of block structured code.

```
Package QUEUE is
    type Q is limited private
    procedure ADD (PQ: in out Q; X: in INTEGER);
    procedure REMOVE (PQ: in out Q; X: out INTEGER);
    function EMPTY (PQ: in Q) return BOOLEAN:
private
        Q-S: constant: = 100;
        type INTV is
            array (INTEGER range < >) of INTEGER;
        type Q is record
            Q-VEL: INTV (1 . . Q-S);
            FRONT: INTEGER range 0 . . Q-S: = 0;
            BACK: INTEGER range 0 . . Q-S: = 0;
        end record;
end QUEUE:
```

Fig. 7.3. A code template. This template has an Ada package header which makes it available for use as a template outside the package.

(p) Advice on commenting and spacing as an integral part of the code which should describe rather than reproduce it.

(q) Rules for subroutines relating to single entry and exit at top and bottom.

(r) Specific rules on the use of GOTO, such as obtaining project authorisation, forward branching, commented destination.

(s) Error return conventions.

(t) Language facilities not to be used.

(u) Exception handling rules.

7.4 DESIGN REVIEW—OBTAINING VISIBILITY

We have emphasised, many times, the problem of visibility due to the intangibility of software. It is well proven that design reviews are capable of identifying a substantial number of faults, thereby improving the quality of software. The features of a design review should therefore be geared to:

(a) Making the design visible.

(b) Providing a means of tracing the requirement through the specifications and design.

(c) Measuring the functions against the requirement.

(d) Identifying faults.

In other words, design review must provide a feedback loop verifying each stage where it is applied in order to judge the adequacy and completeness of the design and thus provide confidence to proceed to the next level of design (Fig. 7.4).

This process cannot be left to chance or to the whim of designers, so there is a need for these design reviews to be a formal Company requirement in order to demonstrate commitment. Procedures should identify:

(1) Which design stages will be subject to review.

(2) The participants and the person with overall responsibility.

(3) Details of records to be kept and rules for the control of follow-up actions.

(4) Rules for review of follow-up actions.

(5) Checklists for guidance—see the end of this chapter for a sample.

(6) Preparations to be made in advance of each design review (documentation, etc.).

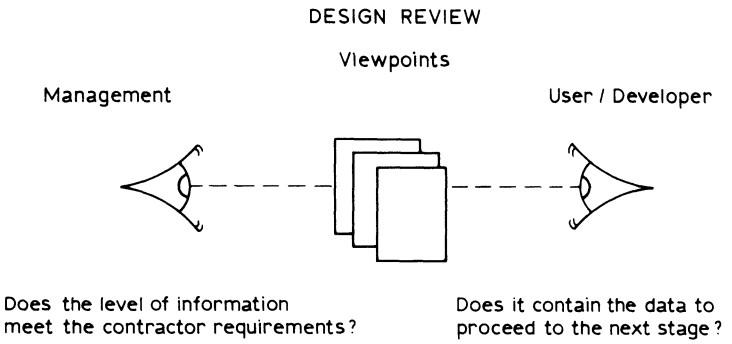


Fig. 7.4

It is important that the details of design reviews are adequately recorded. For example, the designer's development notebook provides a means of recording the faults found and the results of the modifications which followed. Additionally, some form of minutes will be necessary as a record of who was involved and what actions were agreed beyond the technical details recorded in the notebooks. Without such records the quality audit activity (Section 11.3) will have no evidence from which to measure the effectiveness of the design review process.

In Fig. 7.1, the loops shown as 'Design Review', 'Validate' and 'Test' are all reviews of the product against specific criteria which may consist of one or more of the following:

Conformance to documentation and programming standards.
Overall feasibility.
Technical conformance to a previous stage of specification or requirement.
Efficiency of the design in the sense of testability, size, timing, etc.

There are two levels of review.

At the module level the less formal review may involve only members of the design team and might be implemented as an ongoing project activity. A log will be kept either by way of the notebook or by adding notes to the module definition papers. Persons involved in this type of review include peers of the designer and other members of the team.

At a more formal level, planned design reviews at specific milestones will involve persons from outside as well as inside the design

team. These will be scheduled as major project activities at defined points in the development cycle as, for example:

After production of the functional specification.
After production of defined module definitions.
After coding and test of defined modules.
At various milestones during test and integration.

This type of review will be more objective but, by its nature, requires more preparation and hence involves more time and cost. Persons involved include peers, team members, members of other teams, and external experts. The functions of the various participants in a software design review are:

Chairman. Calls and conducts the review and issues reports.
Product designer. Presents the design. Carries out calculations and provides justifications.
Independent designer. Gives objective views on the design.
Consultant. Gives objective views on the design and on costs.
Customer (optional). Gives opinions as to the degree to which the design meets the requirements.
Software quality engineer. Ensures that the test requirements can be met.
Field. Ensures operating and maintenance requirements are met.

Where design reviews involve coded modules, various techniques for examining and comparing the code with the requirement can be applied. These are described in the next section and include:

Peer review.
Code inspection.
Structured walkthrough.

7.5 REVIEWS, INSPECTIONS AND WALKTHROUGHS

These three techniques, which can be considered together, provide a very powerful trio which can efficiently and economically be applied to any software development programme. They provide:

Low level verification.
Checks against standards.
Enhanced communication between team members.
Improved quality.
Reduced compile/debug iterations.

7.5.1 Reviews

Reviews form one of the most effective means of producing error-free software. Whilst the term 'review' may seem weak, properly applied they can detect at specification and design, faults which, if left until later in the life-cycle, will cost many times more to correct. At requirements or overall design level a review might involve many members of a project team. They should, ideally, be drawn from as many different aspects of the project as is possible, not just from the design team but also from test, quality assurance (QA) and management. In this way as many viewpoints as possible are gathered. A typical review life-cycle is the following. The author or authors ensure that the document is acceptable for review. If so, then it is circulated for comment. It is essential at this stage that those commenting on the document are given both the time necessary for them to provide comments and are provided with the terms of reference of the review. This ensures that the specific or general aspects which are required to be reviewed are given the attention necessary. The author then considers the comments prior to a review meeting. It is useful to classify them in some way, e.g., as accepted, not accepted, etc. In this way those accepted comments need not be discussed at the review meeting. Those comments not accepted are discussed with a view to resolving them. It is important that only the issues raised are resolved and that no attempt is made to re-design the system.

The review process can thus be an effective method of finding and correcting faults and also ensuring that problems do not get into the implementation.

7.5.2 Inspections

An inspection is a more formal method of review with the restriction that the aim is to find faults not to rectify them within the meeting. The technique was originally devised by Fagan at IBM and has proved to be an effective technique for the design, code and test phases. Provided the inspection is done carefully and data collected correctly it is possible to gather trend data from inspections which yield useful management information. It is important that inspections are managed properly in the sense that moderators who control the inspection process and inspectors who participate in the inspection are trained and know what is to be expected. It is too easy to allow an inspection to become a review and thus dilute the benefits of an inspection without achieving those of a review.

7.5.3 Walkthroughs

There are several types of walkthrough, some applying only to code, others to design. At the simplest level a walkthrough might perform a simple verification of code and also check adherence to standards. At a more complicated level it might be applied to a design in the sense that the design is 'walked-through' to test its adequacy. One of the most effective methods is that proposed by Yourdon related to the structured methodology bearing his name. As with all the techniques

		AB/089	*BL1*	*BL2*	*BL3*	*BL4*	*BL5*	*BL6*
Requirements Spec.		100	1·0	2·0	2·0	2·0	2·0	2·0
Functional Spec		200	1·0	2·0	2·1	2·1	2·1	2·1
Hardware Tech Spec		300		1·0	2·0	2·0	2·0	2·0
Software Tech Spec		400		1·0	2·0	2·0	2·0	2·0
Subsystems—	SUPYS	410				1·0	2·0	2·0
	INIT	420				1·0	2·0	2·0
	FULLCHECK	430				1·0	2·0	2·0
	DOFOREVER	440				1·0	2·0	2·0
Modules—	ADDRESS	441				1·0	2·0	2·0
	DETDIAG	442				1·0	2·0	2·0
	DETFIRE	443				1·0	2·0	2·0
	CLOCK	444				1·0	2·0	2·0
	DISPSTAT	445				1·0	2·0	2·0
	CHKKEY	446				1·0	2·0	2·0
	REPORT	447				1·0	2·0	2·0
Quality Plan		900	1·0	2·0	2·0	2·0	2·0	2·0
TestSpecs—	MOTHER Bd	911					1·0	2·0
	CPU Bd	912					1·0	2·0
	I/O	913					1·0	2·0
	COMMS	914					1·0	2·0
	PSU	915					1·0	2·0
	I/O CPU	921					1·0	2·0
	COMMS/CPU	922					1·0	2·0
	FUNCTIONAL	931					1·0	2·0
	I/O LOAD	932					1·0	2·0
	MARGINAL	933					1·0	2·0
	MISUSE	934					1·0	2·0
	ENVIRONMENT	935					1·0	2·0
EPROM—XYZ							1·0	—
PROM—ABC							—	1·0

Fig. 7.5. Document Master Index.

in this section it is important that complete records are retained for future reference. It is often found that in later development stages it is necessary to refer back to records of walkthroughs to try and track a fault.

7.6 Configuration Management

Although an overview of configuration management has already been given in Section 4.4 it is worth introducing it at this point in order that its context in design is appreciated.

When developing the design of any system it is vital that a baseline is set at various stages in order that stability is introduced into the process of design. It is otherwise too easy to allow the creative element in the design process to take over and all control over design is lost. Configuration management plays a vital role in this because it allows control to be exercised over these various baselines and the changes which are made to them. Whilst it is possible to perform this task manually it is better done automatically. Several tools are available which assist in the management of not just source and object code but also documents and information relating to them. The essence of these tools is that they provide for limited access to the objects under control and thus maintain control of the items under configuration control. In this way complete visibility of any changes to baselines, e.g., design baselines, are maintained.

Figure 7.5 shows a typical Master Index which describes the baselines (up to Baseline 6) of the case study in Chapter 14.

7.7 FORMAL VERIFICATION

The idea of formal verification goes back many years to the early attempts at correctness proving in programs. The principle is to show that a program as written implies its specification.

Practical tools for formal verification have only recently become available. The appearance of MALPAS and SPADE (see Section 8.7) now provides developers with the means of formal static analysis and hence formal verification. The principle behind both tools is that of directed graphs and regular algebras. Basically a program is first modelled in terms of an intermediate language. This intermediate language is strongly typed (see Chapter 9) with simple constructs. The program, thus modelled, is first represented as a directed graph and

then, after reduction to a minimal form, as a regular algebraic expression. Once in this form rigorous mathematical theory can be used to reveal information about the structure and function of the program.

There have been many attempts at formal verification of programs, most notably in the avionic field. In the case of software controlling aircraft systems it is clearly vital that the greatest confidence is established in the software. For this reason the developers of avionic systems have paid particular attention to the methods available for thorough software testing and, in the USA, work has been carried out on formal verification of flight software.

The whole area of software in safety-critical systems is one which is receiving particular attention and current trends are tending towards greater use of formal verification via such tools as MALPAS and SPADE.

CHECKLIST 7.1: DESIGN REVIEW

(1) Are all the operating cases and environments which the system will meet specified?
(2) Are all safety constraints identified and specified?
(3) How will system performance be perceived by individuals who come into contact with it?
(4) How will the system react to low probability events or failures in its environment?
(5) Is the system identified as modules and are their requirements listed?
(6) Has a failure mode analysis of each module been carried out?
(7) Have all the relevant documents been made available for the review?
(8) Do standards exist for the numbering and naming of documents such that adequate design and configuration control can be exercised?
(9) Is the software requirement adequately expressed through a specification?
(10) Is the software specification complete, consistent, unambiguous and perceivable?

(11) Is there traceability of the requirements through the specifications?
(12) What would be the effect of a major change in the system requirements, interfaces, hardware or their availability?
(13) Have software quality auditing procedures been included in the quality plan?
(14) What automatic software analysis aids are to be used in the reviews and tests?
(15) Have internal and external system interfaces been defined?
(16) Are there specific design areas (e.g. processor speed, new algorithms, security) which create new difficulties?
(17) Will the design be easily built—is the integration plan feasible?
(18) Does the system integrity depend upon diversity and how is this implemented?
(19) Is progress to plan or is there evidence of repeated slippage?
(20) Have adequate budgets for CPU, memory, I/O channels, timing, etc., been established in the light of the system response requirements?
(21) Is the software design adequately expanded into functional and operational flows and data definitions?
(22) Are macros, tables and templates adequately defined?
(23) Are any design features proving resource-critical?
(24) Are any automatic validation tools being used? If so does the predicted performance meet the requirements?
(25) Is there adequate provision for instrumentation for data gathering, debugging, integration and maintenance?
(26) Have all approved specification changes been integrated into the design?
(27) Have all operator interfaces been defined and are they appropriate to the operator capabilities and the manuals?
(28) Is subcontracted software proving adequate?
(29) Are there adequate provisions for software storage and security, including both media and documents?
(30) Does the test and integration specification map completely on to the requirements? Are there suitable cross-reference matrices for this purpose?
(31) Has adequate testing been carried out in the light of changes?
(32) Has there been analysis and interpretation of errors?
(33) Have outstanding actions from previous reviews been actioned?

CHECKLIST 7.2: CODE INSPECTIONS AND WALKTHROUGHS

This checklist can be applied to each module of code inspected. Alternatively, evidence of vendor code inspection to similar criteria will be sought.

(1) Are all constants defined?
(2) Are all unique values explicitly tested on input parameters?
(3) Are values stored after they are calculated?
(4) Are all checked defaults explicitly tested on input parameters?
(5) If character strings are created are they complete? Are all delimiters shown?
(6) If a parameter has many unique values, are they all checked?
(7) Are registers restored on exits from interrupts?
(8) Should any register's contents be retained when re-using that register?
(9) Are all incremental counts properly initialised (0 or 1)?
(10) Are absolute addresses avoided where there should be symbolics as, for example, in the use of an exact location instead of memory mapping?
(11) Are internal variable names unique or confusing if concatenated?
(12) Are all blocks of code necessary or are they extraneous (e.g. test code)?
(13) Are there combinations of input parameters which could cause a malfunction?
(14) Can interrupts cause data corruption?
(15) Is there adequate commentary in the listings?
(16) Are there time or cycle limitations placed on infinite loops?
(17) What is the program response to unacceptable inputs?
(18) Are data structures protected, or can they be accessed from many points?
(19) In accessing arrays, does the program start at the first element and finish at the last?
(20) Are empty character strings treated?
(21) Is the zero case taken into account in calculations?
(22) Do all loops terminate and if so, how?

Chapter 8

A Structured Approach to Static and Dynamic Testing

8.1 LIMITATIONS OF TEST

The main limitation to software testing is the inability to foresee all the combinations of external conditions and logic states in the program. If we could know the answers before writing the test procedures then the problem would not exist. No test can ever prove a practical piece of software to be error-free. In fact, the largest program to be so proved was 1600 lines and the activity involved a three-year Ph.D. Timing-related faults prove the most difficult to reveal. Alas, this is the real world and therefore the problem of test planning must be addressed as thoroughly as possible.

Practical test limitations include:

(a) The range of test conditions and loads which can be applied. It may only be possible to apply environmental conditions individually or in a limited number of combinations. This limitation may also apply to the loading of inputs which will depend on the test equipment and simulators provided.
(b) The range of operator actions which can be foreseen and planned for in the test procedure. These should include misuse tests but they will be limited by the imagination of the test writer.
(c) The extent of the operating instructions.
(d) The choice of language and compiler which will have an effect on the difficulties actually experienced during testing.

Management constraints of time and cost will also limit the test effectiveness. Since testing, however soon it commences, is the last

activity before despatch, and since schedules always slip, then it will be the activity most likely to be curtailed in favour of delivery.

Be most suspicious of repeated slippage during a test programme. It is invariably a symptom that each test procedure is revealing a large number of bugs which require extensive rework, that is to say redesign. It is unlikely that each test procedure will have been rerun, in which case additional faults are likely to exist, partly as a consequence of the rework, and partly since they may have been masked by those faults which were revealed.

The practice of pouring in additional manpower to recover the schedule is ineffective. In fact, the further division of labour will be most likely to slow down the project.

8.2 AN OVERVIEW OF TEST STRATEGY

Test is usually thought of as the activity of checking the various functions whilst executing the code. In fact that is only one aspect of test and is known as dynamic testing. There are four levels of test:

8.2.1 Code Inspection and Walkthrough

This is the weakest level of test and has already been dealt with in Section 7.5. It is subjective and relies on the application of past experience to the review of code. Experience may be consolidated, continuously updated and then formalised by the production of fault dictionaries and checklists.

8.2.2 Symbolic Evaluation

A path is followed through the code and the effect of various inputs on the corresponding outputs is tested. This is a low level parametric test which validates the performance of the code in respect of particular variables rather than addressing the functional specification.

8.2.3 Static Analysis

These are tests which do not actually execute the program but examine the logic and paths within it. To some extent a compiler performs a static test by checking for syntax errors and undeclared variables.

Static analysers examine the code algebraically without the use of actual input and output data values. The tools described in Section 8.3 fall into this category. They are extremely powerful since they permit theoretical checking of the code for all ranges of input conditions. To some extent, therefore, the limitations of dynamic testing, where all possible paths cannot be checked, can be overcome.

8.2.4 Dynamic Analysis

In this type of testing 'live' data is used to exercise the inputs and outputs and thus test the actual performance of the code.

These are tests which rely on executing the program either with simulated or real inputs. These include all forms of functional program testing such as:

Stress tests. Stress tests (sometimes called endurance tests) are black box tests which impose a range of abnormal and illegal input conditions so as to stress the capabilities of the software. Input data volume and rate, processing time, utilisation of data and memory are all tested beyond the design capability. The length and depth of the test will depend on the complexity of the product.

Environmental tests. These will also include tests under conditions of electromagnetic interference. Software systems are particularly prone to data corruption resulting from both mains- and air-borne interference and thorough functional tests are needed. Only real functional tests, using the finished hardware, are valid since the effects are hardware-dependent although it is the software which is affected. Some would argue that electromagnetic interference is a 'hardware' phenomenon but it is better mentioned here than ignored.

8.3 STATIC ANALYSERS

Static analysis packages are tests which assess the structure of a program. They operate at source code level and verify that the program is correct against its specification.

Furthermore they validate the program algebraically. Take the following example:

```
  BEGIN
  INTEGER A, B, C, D, E
  A: = 0
NEXT: INPUT C;
  IF C < 0 THEN GOTO EXIT:
  B: = B + C
  D: = B/A
  GOTO NEXT:
  PRINT B, D;
EXIT: END;
```

A static analyser will detect that:

(a) D is not initialised before use.
(b) E is never used.
(c) A is zero and is used as a divisor.
(d) PRINT B, D; command is never used because of its preceding statement.

A number of analysis suites are currently being developed. The most complete at the time of writing are MALPAS (MALvern Program Analysis Suite) and SPADE (Southampton Program Analysis Development Environment). A further package, TESTBED, uses static analysis to enable more 'intelligent' dynamic testing to be performed.

8.3.1 MALPAS

MALPAS was developed by RSRE at Malvern and is currently supplied by Rex Thompson and Partners, Farnham, Surrey. Figure 8.1 is a graphical representation of its structure.

MALPAS consists of six main analysers, each of which investigates different aspects of a program. The analysers may be run separately but in practice they are used together and run sequentially so that each successive analyser gives more detailed information about the program. The main analysers are:

Intermediate language. A translator which models the source code into a form which can be analysed by MALPAS. The appropriate translation is needed to apply MALPAS to the source code in question.

Control flow analyser. This identifies all possible starts and ends, unreachable code and 'black holes'. It gives an initial 'feel' for the quality of the program. If this is not good then there is every likelihood that subsequent analysis will not result in a good program.

Data use analyser. This identifies all the inputs and outputs and checks that data is not being incorrectly handled (e.g. read before it has been written).

Information flow analyser. Outputs are analysed to describe which inputs they depend on (e.g. output Z depends on inputs A, B, C).

Partial program generator. This extracts subprograms which cater for particular variables of interest. This helps to reduce complexity. These subprograms can then be submitted to the semantic analyser.

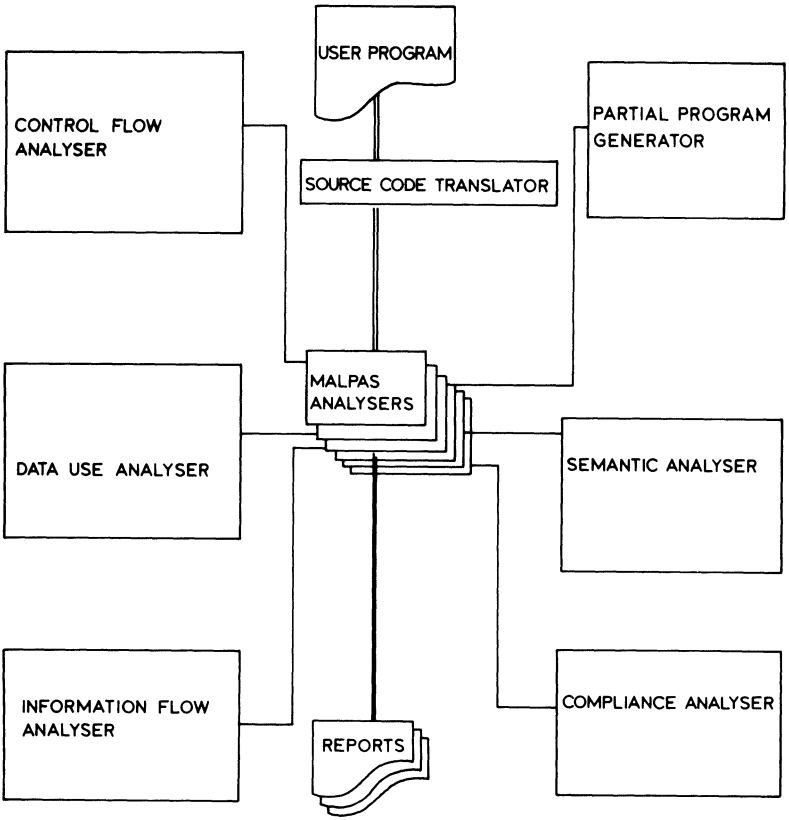


Fig. 8.1. MALPAS.

Semantic analyser. This provides the functional relationships between variables (e.g. P depends on A * Bsq + C). It identifies what the program is doing for each path and thus provides a means of assessing whether the program meets the specification.

Compliance analyser. This takes the output from the semantic analyser and compares it with an embedded specification. For example, if X must be in the range −3 to +90, the analyser tests to see if the condition is met, in the program.

MALPAS is used on software source code but prior to analysis it is necessary to translate the source code into an intermediate language (IL) which MALPAS can read. This obviates the need for a different version of MALPAS to be written for each programming language. The translation of the source code into IL is possible for virtually any

high or low level language and may be performed either by hand or by use of an automatic translator, a number of which are in existence for a variety of high and low level languages. At the time of writing, CORAL 66, Pascal, PL/M-86, 8086 and MAVIS (for VIPER see Section 10.6) translators are available.

The following example is reproduced by kind permission of RTP Software Ltd, 'Newnhams', West Street, Farnham, Surrey. It is more instructive than a theoretical description of each of the analysers.

MALPAS example

This example is a short program intended to control the water cooling system of a motor car fitted with a thermostat and an electric fan. The MALPAS outputs are listed at the end of this Chapter and are numbered MALPAS 1 to MALPAS 14. The specification for the program is:

'*The thermostat aperture should be closed at a water temperature of 83°C* ('*tempclose*') *and fully open at 96°C* ('*tempopen*') *and should open linearly between the two temperatures.*

The fan should turn on at 95°C ('*tempon*') *and off at 86°C* ('*tempoff*'). *It is intended that the program should positively switch the fan on or off in order to prevent partial switching* (*e.g. negative voltages*) *which may damage the unit.*'

The program has been coded in CORAL 66 and a listing of the source code is shown in MALPAS 1. Although the program is valid CORAL, it is not intended as an example of good coding practice and the unstructured use of GOTO statements within such software is not recommended if it can be avoided. A procedure has been introduced into this example to illustrate how MALPAS deals with procedures.

Intermediate language translation

CORAL 66 can be automatically translated into the MALPAS intermediate language (IL) by using a CORAL 66 translator and the IL version of the code appears in MALPAS 2.

The IL code consists of two sections, the first of these containing declarations and the second section being the actual program which appears after the MAIN statement.

Variables and constants are declared in a similar manner to CORAL 66 and, if required, original functions or new operators may also be defined before the main section of the code. The declarations section

also contains a PROCSPEC for each procedure called within the program. Each PROCSPEC contains a list of parameters passed to and from the procedure and an optional DERIVES list which defines, in a simple form, the relationship between procedure inputs and outputs. Hence, when MALPAS analyses the program, it analyses the procedure separately (using the procedure body) but substitutes only the DERIVES list relationships into the code whenever the procedure is called instead of expanding out the procedure each time. The advantage of this approach is firstly that the analysis of the program is considerably simplified if the procedure is long. Secondly, top-down or bottom-up analyses of programs can be performed by representing each module or level of the program by a series of procedures so that the code can then be analysed in a single stage.

The IL code in the program section is fairly straightforward and each line consists of an optional line number, the IL code and an optional comment (in square brackets). Within the IL code variables appear in lower case and language words such as IF and GOTO are written in upper case. With the introduction of MALPAS Release 4.0, the IL now incorporates a range of high level language constructs such as loops, blocks and nested and cascaded conditionals. These constructs considerably ease translation from high level languages whilst still permitting straightforward translation from low level languages. Procedure bodies usually appear after the main program section if the user wishes them to be analysed. In this example the procedure 'hotorcold' has been included in the program section; however, since it is only two lines long, the subsequent analysis of the procedure body is extremely trivial and is not discussed in the rest of this note.

The first action of the MALPAS IL Reader is to label the beginning and end of the program and number the lines of code that do not already possess line numbers. As can be seen in MALPAS 3, these new numbers appear preceded with a # sign. The IL Reader also produces a set of optional statistics relating to the program, detailing items such as numbers of nodes and variables. However, for simplicity these are not shown here. A useful feature of the IL Reader is the derivation of a procedure call-graph which identifies the names and locations of all procedures called within the program. In this example the call-graph is simple (MALPAS 4) since there is only one procedure but on more complex programs this feature can be of enormous benefit in providing an overview of the calling structure of the program.

Control flow analyser

The control flow analyser examines the structure of the program to identify all program entry and exit points, all loops and any multiple entry or exit points that they contain. Ideally, in high integrity software, both the program and all loops should have single entry and exit points so that if MALPAS identifies multiple entries and exits during code development the programmer will be alerted to such undesirable features which can then be removed. The Control Flow Analyser also reveals more serious errors within the program such as unreachable code, false entry points or dynamic halts. The analyser reveals the graphical structure of the code by representing the program as a series of arcs between nodes and it then performs a number of reductions on this graph until the program is finally reduced to its simplest form.

Due to the use of the GOTO statements in this example the structure of the code is relatively complex for such a short program and after the first stage of reduction (ONE–ONE) there are still eleven nodes present (MALPAS 5). From the descriptions of the nodes and their successors given here by MALPAS, it is possible to construct the node graph depicted in MALPAS 6. Although this graph is not the complex 'spaghetti' that can be achieved in programs by the use of unstructured programming, one would probably wish to rewrite the code to simplify it if one discovered a complex structure such as this through using MALPAS during code development. Nevertheless, despite the complex structure, one can see that the example program has a single entry and exit point, no loops and no seriously undesirable features.

Data use analyser

The data use analyser describes how data is used within the program and from this one can check, for example, that all output variables are written as intended and that input variables are correctly read. This may be of particular benefit on a large program for checking the lists of specified inputs and outputs.

The analyser itemises a number of different categories of data usage (see MALPAS 7) and these can be interpreted with the knowledge of the specific program to ensure that the data is used correctly. In this example, category 'R' shows that the water temperature ('temp') is read every time that the program is executed (as one would hope!), whilst category 'I' reveals that the status of the fan ('fan') is read only

on some of the paths through the code. More importantly, although categories 'U' and 'V' show that both the thermostat aperture and the fan switching are written on some paths through the code, it can be seen from category 'W' that only the aperture is updated every time the program runs. Hence this is an early indication that the program does not do exactly as intended since the specification required the fan to be written (i.e. positively switched on or off) every time that the program is executed.

This program does not have any data in the 'written twice without an intervening read' category ('A'); however, the identification of such data usage is often useful for revealing variables that may be updated to different values on different paths, variables that have been initialised before being written, or may simply indicate inefficient use of variables.

Information flow analyser

The information flow analyser identifies all the input variables upon which each output variable depends and provides an initial check that the program outputs are dependent upon the correct input values. The analyser also shows the conditional statements upon which each output variable depends and this is useful for determining whether the program can be partitioned into sub-programs by the partial programmer to enable semantic analysis of particular output variables.

In MALPAS 8 one can see that the output variable 'aperture' depends on the input variables that one would expect, these being the water temperature ('temp') and the four constants that define both the thermostat open and closed states and the temperatures at which these occur. However, the other output variable 'fan', as well as being dependent on the variables that one would expect from the specification, is also shown as being dependent on the supposedly unrelated temperatures at which the thermostat opens and closes ('tempopen' and 'tempclose') and therefore one is alerted to a potential error in the program.

MALPAS 9 shows the conditional nodes upon which each of the two outputs depends and it can be seen that, whilst the variable 'fan' is dependent upon all five conditional nodes (and hence on all paths through the program), the variable 'aperture' is dependent on only two of the conditional nodes. Therefore, if 'aperture' is the only program output variable of interest, one can perform partial programming for that particular variable and this is described below.

Path assessor

The results from the path assessor appear after the information flow analsis and MALPAS 10 shows that there are only six syntactically possible paths through this program. This is not of great importance in this example, especially as the number of paths can quite easily be calculated from MALPAS 6. However, the path assessor is of great use for large programs where, if it is shown that there are many hundreds of paths through the code, one is made aware that partial programming will be necessary to reduce program complexity and hence reduce the amount of semantic analysis to manageable proportions.

Semantic analyser

The semantic analyser is, with the exception of the compliance analyser, the most powerful of the MALPAS analysers and it describes the functional relationship between program inputs and outputs for each semantically possible path through the program. Hence, for the whole range of program input variables, it will reveal exactly what the program does in all circumstances. Once MALPAS has identified all the semantically possible paths through a program, this information can then be used to direct dynamic testing of the software during system validation.

The analyser presents the results for each path through the program in terms of a predicate, defining the conditions upon which that path depends, followed by a set of actions or relationships that apply under the defined input conditions. For this particular example, it can be seen in MALPAS 11 that there are five sets of these predicate-action pairs, indicating that there are five semantically possible paths through the code. This is less than the number of syntactically possible paths identified by the path assessor because the data values within the program lead to one of the syntactically possible paths being impossible to access.

An immediate point to note in MALPAS 11 is that, unlike the results from the other MALPAS analysers, the constant variable names have been replaced by their declared integer values. This simplifies the interpretation of the results in this example but MALPAS does allow the constant names to be retained if required. It should also be noted that the statements between the MAP and ENDMAP statements are not sequential but represent parallel assignments. Furthermore the variables on the left-hand side are the output

variables whilst those on the right-hand side to which they are related are all input variables.

The first three predicate-action pairs in MALPAS 11 are straightforward and one can easily see that they indicate that the program does what the specification stated that it should do for the given sets of conditions.

However, close inspection of the fourth predicate-action pair shows the first error in the program, since the predicate states that at 95 degrees the fan remains off if it is already off when the program is executed, yet the specification requires the fan to be turned on at 95 degrees. A more significant error is revealed by the next predicate which states that the fan remains on between 83 and 87 degrees, instead of being switched off at 86 degrees as required, and hence the fan is turned off 3 degrees lower than it should be.

The reason for these errors is apparent if one refers back to the information flow analysis (MALPAS 8), which revealed the incorrect dependence of the fan state on the thermostat opening and closing temperatures. Furthermore, the error first identified by the data use analyser, of the fan being written only on some paths through the code, is also apparent in these last two predicate-action pairs since the fan is not written at all on these paths.

The clear and precise description of the program provided by the semantic analyser is invaluable for verifying the program code against its specification, either during code development or as part of a final assessment or certification exercise. This analyser is particularly useful for detecting subtle coding errors, either from the direct relationship between inputs and outputs, or from the predicates that determine those relationships, as for example has been revealed here where the predicate relationship required was 'less than' and the program was shown to implement 'less than or equal to'. Although such errors may be small, the effects that they have on system performance could be catastrophic.

Another major benefit of this analyser is that it often reveals semantically possible paths through the code that the programmer was unaware of and may not have discovered despite extensive testing. In such cases it may be that the software will do something unexpected over a small range of input conditions and, whilst such input values may be outside the design range for the system with only a very remote chance that the conditions will be encountered in practice, it is of great benefit for MALPAS to reveal the existence of such conditions and to identify what happens in those circumstances.

Partial programmer

The partial programmer can be invoked when the program outputs of interest are dependent only on a subset of the total paths through the code and hence the semantic analysis for those particular variables is considerably reduced compared with the semantic analysis for the entire program. In this example the partial programmer has been run for the single variable 'aperture' with the result that there are only three predicate-action pairs from which (MALPAS 12) it is immediately obvious that the thermostat behaves exactly as specified.

Compliance analyser

The last and most recent MALPAS analyser is the compliance analyser which has been developed by RSRE Malvern using as yet unpublished mathematical theory. This analyser is intended to verify software by automatically comparing a program with its specification and explicitly identifying any differences between the two.

In order to do this the program specification has to be expressed in a simple mathematical form and embedded in the IL program header. Although this sounds difficult, it is in fact surprisingly easy to do for most software specifications and the specification written for the example program is shown embedded in the IL text in MALPAS 13.

The specification consists of a PRE statement, which defines the overall range of values that the input variables may take, and a POST statement, which defines the required relationship between program inputs and outputs. In this example an arbitrary range for the water temperature, of −50 to +120 degrees, has been specified in the PRE statement.

The POST statement, like the semantic analysis results, takes the form of a predicate followed by a definition of the functional relationship required between program inputs and outputs. The first line of the POST statement in MALPAS 13 is therefore interpreted as "If 'temp' is less than or equal to 'tempclose' then 'aperture' = 'closed' " and hence it can be seen that the POST statement is simply a mathematical form of the specification quoted above in words.

The compliance analyser identifies any difference between the program and its embedded specification as a 'threat'. If the program implies or meets its specification then the 'threat' is declared false or, if MALPAS is unable to reduce the threat expression to such a simple form, is represented as a logical expression which the programmer can inspect to ensure that it is false. Alternatively, if there is a difference

between the program and its specification then the threat will appear as a logical expression which can be true.

In MALPAS 14 the compliance analyser identifies two threat conditions associated with the example program. The first of these conditions occurs when the water temperature is 95 degrees and the fan is off; we have already seen from the semantic analysis that the program does not meet its specification at this point since the specification requires the fan to be on whilst in the program the fan remains off. Similarly the second threat condition is the second error identified from the semantic analysis, namely that between 83 and 87 degrees the fan is not switched off, as required in the specification.

Conclusions

The above paragraphs illustrate how MALPAS reveals the correctness of software code by describing the program in a form that can be directly compared with the program's specification. If errors are detected at a late stage of software development, the cost of correcting the errors and subsequently re-assessing the software can be large and hence the major benefits of MALPAS are to be realised if it is used throughout the design and development phase as well as over the rest of the software life-cycle.

This is particularly true for the compliance analyser, since the translation of an existing specification into a mathematical form may be very time consuming if performed at a late development stage, especially if the original specification was imprecise and incomplete. If, however, MALPAS is used from the outset of a software design and development programme then the specification can be initially defined in the precise form required by MALPAS. The specification may even be embedded in the original source code as comments in such a way as to be interpreted by the automatic source-code-to-IL translator so that the specification will appear as pre- and post-conditions in the intermediate language text. It is also probable that writing the specification in a precise mathematical way will lead to clearer and less ambiguous specifications.

MALPAS may also be used for the verification of software design if a formal methodology is used. The intermediate language incorporates a number of features making it suitable for software design, such as the ability to define abstract data types and then subsequently refine the definitions. This allows the design to be written either in IL or a formal design language / methodology, such as VDM or JSD and then

translated into IL. Once the design is represented in IL then it can be verified using the semantic and compliance analysers. A number of papers have been written outlining details of the techniques involved in using MALPAS in this way.

The use of MALPAS during software development encourages modular, well structured software, partly because any complex path structure and inefficient use of data will be identified by the control flow and data use analysers but also because semantic analysis of large multi-path segments of code can produce large amounts of output which may be time consuming to assess. Although one can use the semantic output reduction techniques, such as partial programming, that were developed for using MALPAS as a final assessment tool on multi-path code, it is obviously preferable if the software is sufficiently modular to permit easy interpretation of the analyser outputs. MALPAS is also particularly suited for analysis of modular software because, as has been mentioned above, it allows modules, levels and subroutines to be analysed separately and then represented by procedures before the analysis of the next higher level of the code.

There are obviously costs associated with the use of MALPAS on software projects; although it is relatively quick and simple to run, it can be time consuming, initially at least, to interpret the results produced. However, once the analyst is familiar with the software being investigated and has experience of applying the separate MALPAS analysers then the process is considerably quickened. There is also the benefit when using MALPAS that the need for other verification and validation techniques, including testing, is reduced. Since MALPAS is a static technique, it will always be necessary to carry out some dynamic testing to investigate real-time aspects, although the need for this is considerably reduced if one already has confidence in the correctness of the code before testing starts. At present MALPAS is able to perform limited modelling of some real time aspects of programs (for example it is possible to use it to determine processor timings of different paths through the code), and in future developments it is intended to extend its real-time applicability. Nevertheless the static analysis of real-time software is of great benefit for the verification of such software (indeed the majority of the use of MALPAS has been on real-time systems) and by performing this at an early stage in the development of the system, it should reduce life-cycle costs and result in correct, more reliable software.

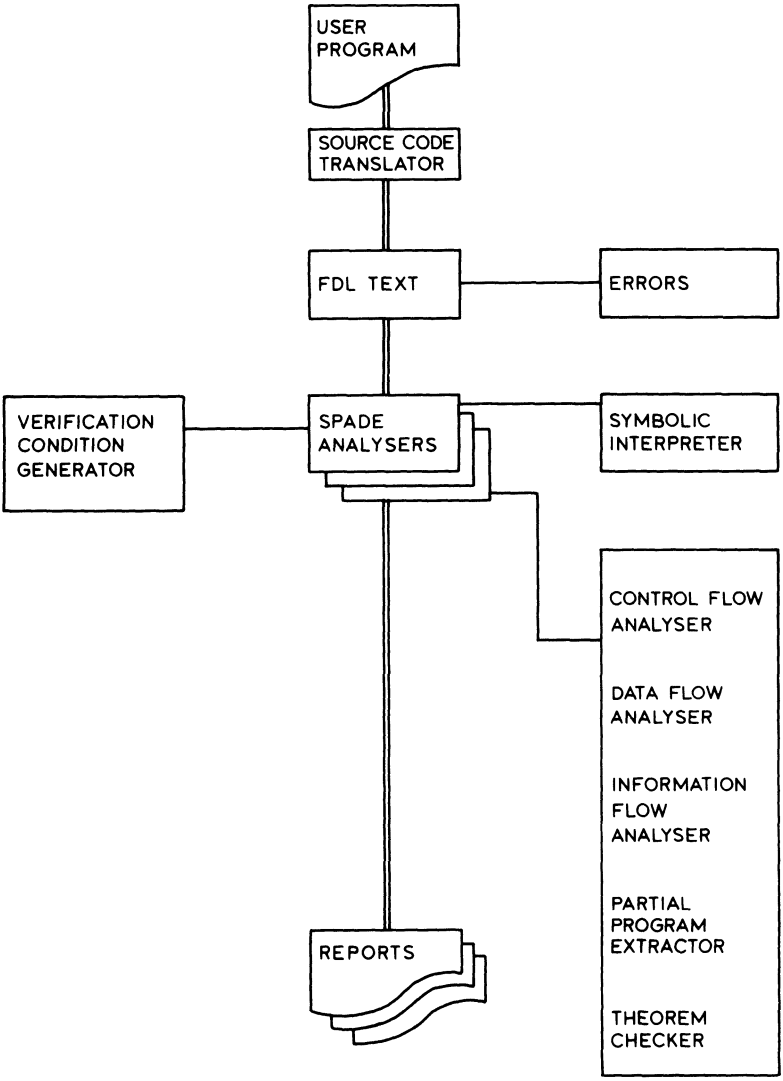


Fig. 8.2. SPADE.

8.3.2 SPADE

SPADE was developed at Southampton University and is now available from Program Validation Ltd, Southampton, UK. It also involves an intermediate language known as FDL (Functional Description Language). An FDL model of a program can be constructed automatically, using a translator, or by hand. FDL is not a programming language but a method of describing a program in terms of states and transitions. Figure 8.2 represents the structure of SPADE.

This automatic translation is available for Pascal and INTEL 8080 with a Modula 2 version under construction. An Ada subset, called SPARK, is also available. This is a limited subset of Ada intended for the implementation of safety critical software. SPADE can currently be run on DEC VAX machines (including Micro Vax II and 11/730) under the VMS (Virtual Memory System) operating system.

SPADE consists of a number of 'flow analysers' and 'semantic analysers' as follows:

Control flow analyser. This addresses the control flow and loop structure of the program and reports on unused code, multiple entry and similar faults.

Data flow analyser. This seeks undefined data variables and unused definitions.

Information flow analyser. This constructs the relationships between inputs and outputs and checks their consistency with the specification. It will, as a result, indicate ineffective statements and variables.

Partial program extractor. Taking a specified variable, the partial program extractor identifies just those statements which affect that value. These are then assembled as a 'partial program'.

Verification condition generator. This describes the path functions and the conditions under which they are executed. Verification can thus be carried out if a specification is provided.

Symbolic interpreter. The text is executed and paths are identified. Data-flow errors, check statements and run time tests are included.

Theorem checker. This proofchecker (written in PROLOG) permits verification at an arithmetic/logical level.

8.3.3 TESTBED (LDRA)

Formally known as LDRA, TESTBED was developed by Liverpool Data Research Associates and is now marketed by Program Analysers Ltd of Newbury, Berkshire, UK.

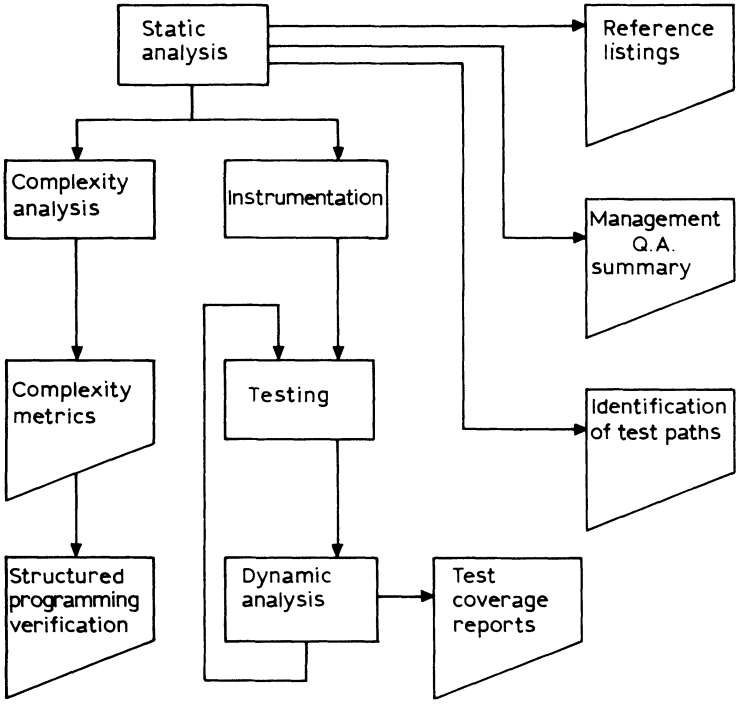


Fig. 8.3. TESTBED.

TESTBED involves dynamic as well as static analysis and is therefore also included in Section 8.4. It is available for CORAL 66, Pascal, FORTRAN, COBOL, PL/1, C, PL/M86 and Ada and will run on UNIX, IBM, VAX, SUN and GOULD systems.

TESTBED identifies program paths as a number of path fragments during static analysis of control flow. It generates reference listings and management quality assurance summaries. The user may then subject the software to complexity analysis which will produce complexity metrics and structured programming verification reports. Figure 8.3 gives an outline of the TESTBED structure. The dynamic testing aspects are described in Section 8.4.

8.4 DYNAMIC TESTING

8.4.1 Test Levels

Module tests

These are stand-alone tests of individual modules not yet integrated into the system. Hardware will probably not be available at the stage

when module tests are being carried out. Data inputs are therefore simulated manually or by means of test drivers with results and outputs being displayed on the VDU and printer.

Integration tests

This is a most important phase of testing. *Bottom-up* integration is the traditional approach whereby modules are tested together and a pyramid is built up. Only then is it possible to test the system as a complete set of functions. An alternative approach is the *top-down* method which involves building a set of simulation drivers, known as *stubs,* in order to provide a complete artificial system. This was also described in Section 4.7. Modules, and groups of modules, can then be tested as they become available. This approach has several advantages:

(a) A single test specification can apply throughout.
(b) All possible interactions of a module to the remainder of the system are likely to be tested.
(c) Complex system timing faults are likely to be revealed at an earlier stage.

There are many problems which arise during the process of integrating a system.

Diagnosis of the causes of failure, during integration and test, can be complicated by the fact that faults can interact to give symptoms quite different from those which they would produce as individual bugs. As a result, the rework after a particular test may not be adequate although it might seem that the problem has been rectified.

Modules, or subsystems, may perform correctly when tested alone but fail in combination. Unless this is foreseen during the writing of the integration test procedures, hidden faults can remain.

The time required for development and production of the various test harnesses may be underestimated. In that case integration tests will commence with incomplete test facilities and the result will be harder diagnosis of faults and incomplete testing.

Failures may relate to loading, timing and speed features of the inputs, outputs and data.

If the interfaces between modules and between the system and the tester are not fully understood, tests will not be conclusive and diagnosis will be harder.

The ease of testability of software is clearly a function of the code itself and it would seem obvious that this must be designed in. The most powerful method of achieving this is to provide built-in self-test routines in the code. It is an area which may well provide an application for expert systems.

System tests

The purpose of system testing is to utilise known input stimuli and data to check that the outputs conform to the established specification. These performance tests include:

Functional tests. To test the defined performance.

Environmental tests at stress limits. It is often pointed out that software is not affected by temperature or humidity. This is true but, nevertheless, the changes in electrical characteristics which result can alter timing features and thereby cause faults.

Misuse tests. Take account of the fact that products will be used outside their stipulated range of operating conditions.

Maintainability tests. To demonstrate ease of diagnosis and repair under simulated fault conditions.

Various techniques and test tools are employed and these will be outlined in the next two sections.

Production tests

The above test levels constitute qualification tests because they 'qualify' the design against the specification. Production tests, on the other hand, are repetitive and merely confirm that the build of production items has not changed.

8.4.2 Dynamic Test Tools

These test tools include:

Test drivers. These are used to input data into a module under test and to receive the resulting data output for checking. They are simulators which can mimic unavailable items of hardware or software.

Test beds. More sophisticated than the simple driver described above, they can display simultaneously the source code alongside the executing program. They show the values of variables and indicate

those portions of code affected. Current packages include:

(a) FPE (FORTRAN Programming Environment) from SOFTOOL (USA) which caters for FORTRAN programs.
(b) COBOL Animator from Microfocus which caters for COBOL programs.
(c) A CORAL test bed from Software Sciences Ltd.

Emulators. 'Intelligent' communications analysers having programmable stimulus and response facilities are used to emulate parts of the system (including their responses) not yet developed. In this way the software is tested as if it were surrounded by a real system.

Assertion checkers. Insert code statements (probes) and flag the results.

Path testers. Similar to the static testers but require the code to be executed.

Mutation analysers. Seed intentional errors into the code to test the fault tolerance of the system.

Symbolic execution tools. Execute the arithmetic and logic with symbolic variables following calculus rules. It could be argued that this is a static test since no execution is involved, the functions being tested symbolically.

The TESTBED tool was introduced in Section 8.3.3 since it contains a module of static analysis. It is also a dynamic analysis tool. It produces *test effectiveness ratios* that define how well a program has been tested. Heavily used as well as unused statements are highlighted. The source code is compiled, linked and executed with appropriate sets of test data under control of a menu system. Upon termination of the user program an additional file is generated by the instrumentation. Dynamic analysis is invoked to interrogate this file to generate reports on the effectiveness of the data simulation. This dynamic operation can be repeated, with selection of alternate sets of data, until a satisfactory test effectiveness is obtained.

8.5 TEST MANAGEMENT

Software testing is gradually undergoing a transition from being a 'black art' to becoming a science. In other words, test methods based on undocumented experience and subjective judgements are being replaced by formally designed tests.

One essential requirement is the existence of a test manager, who must be appointed at the beginning of the project. The whole structure and philosophy of the tests are dependent on the requirements specification and on the system configuration. The test plan and an outline of test methods must therefore begin to evolve, along with the hierarchy of documents, right from the beginning. Lead times for the procurement of simulators, test hardware, environmental facilities, etc., are long, and this adds greater emphasis to the need for test management at the earliest possible stage.

The first document, as regards testing, will be the test strategy. This might be a separate document or may form part of the quality plan. It will outline the various levels of test (described in the remainder of this chapter) and how these will build up to a final functional system test. Broad details of the test hardware and software which will be needed to carry out the various integration, simulation, loading and other tests will also be given. This, and the subsequent test documents, should not be produced by the designers but by a separate test authority.

Other essential documents will follow as the design proceeds. These include:

Test specifications. One for each separate test activity of which there may be several dozen. It will describe the functions to be tested and the test method to be employed, including the range of values which will be covered. The test equipment required is also described here.

Test procedures. The actual test instructions down to the details of connecting test equipment and the actual inputs to be applied and their sequence. A good test procedure should contain the anticipated result of each test and a record sheet on which to record the results.

Test records. There should be a test record for every test and, as mentioned above, this may be a part of the test procedure document.

Test utilities specification. Both the hardware and computer facilities needed for all the tests are specified here. Any test software (i.e. programs which provide test data so as to simulate modules or hardware items not yet designed) is also described here.

Test reports. There should be a test report for each test or group of tests. The main benefit is that the report provides a medium for recording any actions which arise as a result of the faults revealed. The actions, having been formally recorded, can then be reviewed until they are discharged.

CHECKLIST 8.1: TEST AND INTEGRATION

(1) Are there written requirements for testing subcontracted or proprietary software?
(2) Are there test plans/schedules/specifications and are they written in parallel with the design?
(3) Is there a build-up of integration and testing (e.g. module test followed by subsystem test followed by system test)?
(4) Is there evidence of test reporting and remedial action?
(5) Is there evidence of thorough environmental testing?
(6) Is there a defect-recording procedure in active use?
(7) Do test schedules permit adequate time for testing?
(8) Is a simulation possible on a larger configuration and, if so, is it planned?
(9) Are software prototypes envisaged for use in demonstrating the system concepts to the user? If so, at what stage will they be produced and what will they cover?
(10) Is the test facility a deliverable item?
(11) Is the test software under build state control?
(12) Can the test facility demonstrate all operational modes including behaviour under degradation conditions?
(13) Is the test hardware and its configuration thoroughly defined?
(14) Is there evidence of repeated slip in the test programme?
(15) To what extent are all the paths in the program checked?
(16) Does the overall design of the tests attempt to prove that the system behaves correctly for improbable real time events (e.g. misuse tests)?
(17) Does the test address what the system should not do as well as what it should?
(18) Is there evidence of software changes being implemented to circumvent hardware defects?
(19) Are 'power up' and 'power fail' tests included?

MALPAS EXAMPLE

MALPAS 1

```
'CORAL' CAR;

'SEGMENT' COOLING
'BEGIN'
    'DEFINE' TEMPCLOSE "83";
    'DEFINE' TEMPOPEN "96";
    'DEFINE' TEMPON "95";
    'DEFINE' TEMPOFF "86";
    'DEFINE' CLOSED "0";
    'DEFINE' FULLYOPEN "100";
    'DEFINE' OFF "0";
    'DEFINE' ON "1";

    'INTEGER' TEMP, FAN, APERTURE;

    'PROCEDURE' HOTORCOLD ('VALUE' 'INTEGER' OPENSHUT, ONOFF;
                                    'LOCATION' 'INTEGER' APERTURE, FAN);
    'BEGIN'
        APERTURE := OPENSHUT;
            FAN := ONOFF
    'END';

    'IF' TEMP <= TEMPCLOSE 'THEN'
     HOTORCOLD (CLOSED, OFF, APERTURE, FAN)
    'ELSE'
    'IF' TEMP >= TEMPOPEN 'THEN'
     HOTORCOLD (FULLYOPEN, ON, APERTURE, FAN)
    'ELSE'
    'BEGIN' APERTURE := FULLYOPEN *
                        ((TEMP - TEMPCLOSE)/(TEMPOPEN - TEMPCLOSE));
        'IF' TEMP > TEMPON 'THEN' 'GOTO' L10;
        'IF' FAN <> OFF 'THEN'
        'BEGIN'
                'IF' TEMP > TEMPOFF 'THEN' 'GOTO' L10
        'END';
        'GOTO' L20;
L10:            FAN := ON;
L20:
    'END'
'END'
'FINISH'
```

MALPAS 2

```
TITLE carcooling;

CONST tempclose : integer = 83 ;
CONST tempopen : integer = 96 ;
CONST tempon : integer = 95 ;
CONST tempoff : integer = 86 ;
CONST closed : integer = 0 ;
CONST fullyopen : integer = 100 ;
CONST off : integer = 0 ;
CONST on : integer = 1 ;


PROCSPEC hotorcold (IN openshut, onoff : integer
                    OUT aperture, fan : integer)
DERIVES aperture AS openshut, fan AS onoff ;

MAIN

VAR temp, fan, aperture : integer;

    IF temp <= tempclose THEN
        hotorcold( closed, off, aperture, fan)

    ELSIF temp >= tempopen THEN
        hotorcold( fullyopen, on, aperture, fan)

        ELSE aperture := fullyopen *
                         ((temp - tempclose) / (tempopen - tempclose));
            IF temp > tempon THEN GOTO 33 ENDIF;
            IF fan /= off THEN
                IF temp > tempoff THEN GOTO 33 ENDIF;
            ENDIF;

            GOTO 35;
33: [ L10: ]
            fan := on;
35; [ L20: ] ;
    ENDIF;

ENDMAIN

PROC hotorcold;
        aperture := openshut;
        fan := onoff;
ENDPROC;

FINISH
```

MALPAS 3

```
[1]                 TITLE carcooling;
[2]
[3]                 CONST tempclose : integer = 83 ;
[4]                 CONST TEMPOPEN : integer = 96 ;
[5]                 CONST tempon : integer = 95 ;
[6]                 CONST tempoff : integer = 86 ;
[7]                 CONST closed : integer = 0 ;
[8]                 CONST fullyopen : integer = 100 ;
[9]                 CONST off : integer = 0 ;
[10]                CONST on : integer = 1 ;
[11]
[12]
[13]                PROCSPEC hotorcold (IN openshut, onoff : integer
[14]                                    OUT aperture, fan : integer)
[15]                DERIVES aperture AS openshut, fan AS onoff ;
[16]
[17]                MAIN
[18]
[19]                VAR temp, fan, aperture : integer;
[20]
[21]          #1:       IF temp <= tempclose THEN
[22]          #3:           hotorcold( closed, off, aperture, fan)
[23]
[24]          #4:       ELSIF temp >= tempopen THEN
[25]          #5:           hotorcold( fullyopen, on, aperture, fan)
[26]
[27]                        ELSE
[27]          #6:   aperture := fullyopen *
[28]                            ((temp - tempclose) / (tempopen - tempclose));
[29]          #7:                       IF temp > tempon THEN
[29]          #9:   GOTO 33
[29]          #8:   ENDIF;
[30]         #10:                       IF fan /= off THEN
[31]         #12:                           IF temp > tempoff THEN
[31]         #14:   GOTO 33
[31]         #13:   ENDIF;
[32]         #15:   [SKIP]
[32]         #11:                       ENDIF;
[33]
[34]         #16:                       GOTO 35;
[35]
[36]                33:   [ 110: ]      fan := on;
[37]                35:   [ 120: ] [SKIP] ;
[38]         #17:   [SKIP]
[38]          #2:       ENDIF;
[39]
[40]         #18:   [SKIP]
[40]       #STOP:   [SKIP]
[40]        #END:   ENDMAIN
```

```
[41]
[42]                PROC hotorcold;
[43]          #1:       aperture := openshut;
[44]          #2:       fan := onoff;
[45]          #3:   [SKIP]
[46]        #END:   ENDPROC;
```

MALPAS 4

Call Graph

Section:	Calls:
mainsection	hotorcold
hotorcold	—

MALPAS 5

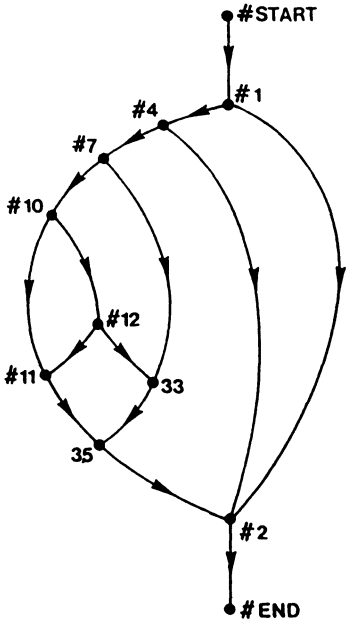
Recovering graph

After ONE–ONE, 12 nodes reduced

No selfloops reduced

Node id	No of predecessors	Successor nodes	
#START	0	#1	
33	2	35	
35	2	#2	
#1	1	#2	#4
#2	3	#END	
#4	1	#2	#7
#7	1	33	#10
#10	1	#11	#12
#11	2	35	
#12	1	33	#11
#END	1		

MALPAS 6



MALPAS 7

Key:

H = Data read and not subsequently written on some path between the nodes
I = Data read and not previously written on some path between the nodes
A = Data written twice with no intervening read on some path between the nodes
U = Data written and not subsequently read on some path between the nodes
V = Data written and not previously read on some path between the nodes
R = Data read on all paths between the nodes
W = Data written on all paths between the nodes
E = Data read on some path between the nodes
L = Data written on some path between the nodes

After HECHT (from ONE–ONE)

From node	To node	Data-use expression
#START	#END	H : fan temp
		I : fan temp
		U : aperture fan
		V : aperture fan
		R : temp
		W : aperture
		E : fan temp
		L : aperture fan

Summary of Possible Errors

Data never used	: None
Data that may be written twice with no intervening read	: None
INs never read	: None
INs that may be read but whose initial values are never read	: None
INs that may be read but would subsequently be written and not used	: None
INs always written	: None
INs that may be written	: None
INs used as local workspace whose final values may not be used	: None
INOUTs never written	: None

INOUTs always written but whose initial values are never read	: None
OUTs never written	: None
OUTs that may be written but only after being read when undefined	: None
OUTs that may be read when undefined	: None
OUTs that may not be written on some paths	: None
VARs never read	: aperture
VARs never written	: temp
VARs that may be read but would subsequently be written and not used	: None
VARs that may be written but only after being read when undefined	: None
VARs that may be read when undefined	: fan temp
VARs whose final values may not be used	: aperture fan

MALPAS 8

Information Flow

After HECHT (from ONE–ONE)

From Node	To Node	Identifier	may depend on identifier(s)
#START	#END	fan	: fan off on temp tempclose tempoff tempon tempopen
		aperture	: closed fullyopen temp tempclose tempopen

MALPAS 9

Identifier	may depend on conditional node(s)				
fan	#12	#10	#7	#4	#1
aperture	#4	#1			

MALPAS 10

Path Assessor Output

After HECHT (from ONE–ONE)

From Node	To Node	Number of paths
#START	#END	6

MALPAS 11

Semantic analysis

After HECHT (from ONE–ONE)

From node : #START
To node : #END

```
IF temp <= 83

THEN MAP
     fan := 0;

     aperture := 0
ENDMAP ENDIF
```

```
IF temp >= 96

THEN MAP
     fan := 1;

     aperture := 100
ENDMAP ENDIF
```

```
IF temp >= 87 AND temp <= 95 AND fan /= 0

THEN MAP
     fan := 1;

     aperture := 100 * ((temp - 83) / 13)
ENDMAP ENDIF
```

```
IF temp >= 84 AND temp <= 95 AND fan = 0

THEN MAP
     aperture := 100 * ((temp - 83) / 13)
ENDMAP ENDIF
```

```
IF temp >= 84 AND temp <= 86 AND fan /= 0

THEN MAP
     aperture := 100 * ((temp - 83) / 13)
ENDMAP ENDIF
```

MALPAS 12

Semantic analysis

Recovering graph and extracting the partial program
for the following output variables:

aperture

After ONE–ONE

From node : #START
To node : #END

IF temp <= 83

THEN MAP
aperture := 0
ENDMAP ENDIF

IF temp >= 96

THEN MAP
aperture := 100
ENDMAP ENDIF

IF temp >= 84 AND temp <= 95

THEN MAP
aperture := 100 * ((temp - 83) / 13)
ENDMAP ENDIF

MALPAS 13

```
TITLE car_cooling;

CONST tempclose : integer = 83;
CONST tempopen : integer = 96;
CONST tempon : integer = 95;
CONST tempoff : integer = 86;
CONST closed : integer = 0;
CONST fullyopen : integer = 100;
CONST off : integer = 0;
CONST on : integer = 1;

PROCSPEC hotorcold (IN openshut, onoff: integer
                    OUT aperture, fan : integer)
DERIVES   aperture AS openshut, fan AS onoff
POST      aperture = 'openshut AND fan = 'onoff;

MAINSPEC (INOUT temp : integer
          OUT fan, aperture : integer)
PRE       temp >= -50 AND temp <= 120
POST      (( temp <= tempclose ) - -> ( aperture = closed ))
  AND (( temp > tempclose AND temp < tempopen )
  - -> (aperture = fullyopen * ((temp - tempclose)/(tempopen - tempclose)) ))
  AND (( temp >= tempopen ) - -> ( aperture = fullyopen ))
  AND (( temp <= tempoff ) - -> ( fan = off ))
  AND (( temp > tempoff AND temp < tempon AND fan /= on ) - -> ( fan = off ))
  AND (( temp > tempoff AND temp < tempon AND fan /= off ) - -> ( fan = on ))
  AND (( temp >= tempon ) - -> (fan = on ));

MAIN

IF temp <= tempclose THEN
    hotorcold ( closed, off, aperture, fan)

ELSIF temp >= tempopen THEN
    hotorcold ( fullyopen, on, aperture, fan)

ELSE  aperture := fullyopen *
                  ((temp - tempclose)/(tempopen - tempclose));
          IF temp > tempon THEN GOTO 33 ENDIF;
          IF fan /= off THEN
                IF temp > tempoff THEN GOTO 33 ENDIF;
          ENDIF;

          GOTO 35;

33: [ L10: ] fan := on;
35: [ L20: ] ;
ENDIF;

ENDMAIN
```

```
PROC hotorcold;
    aperture := openshut;
    fan := onoff;
ENDPROC;

FINISH
```

MALPAS 14

All procedure calls are consistent with the required pre-conditions.

Compliance analysis

After HECHT (from ONE–ONE)

From node : #START
To node : #END

threat := temp = 95 AND fan = 0 OR temp >= 84 AND temp <= 86 AND fan /= 0

Chapter 9

Languages and Their Importance

9.1 PROGRAMMING LANGUAGE—THE COMMUNICATION MEDIUM

The process of translating a design into some particular programming language is more often than not regarded as 'programming'. Traditionally it has certainly been the case that the development of software-based systems has tended to emphasise the coding phase at the expense of the rest of the life-cycle. The reason for this is understandable. The programmer likes communicating with the computer and his means of doing this is via some mutually understood language. Strictly speaking, of course, this is not true since the original source text, generated by a programmer, has to go through various stages of translation before the computer can 'understand' what it is being asked to do. Even for interpretative languages this is the case. However, it still remains that the programming language is seen as the creative medium through which the programmer expresses his interpretation of some design, be it expressed in plain English or some formal methodology as discussed in Chapter 6.

The number of different programming languages is vast. Even within the description 'programming language' we must differentiate between various types. The category of language with which most programmers are familiar is known as imperative (or procedural) language. Included in this group are FORTRAN, Pascal, and Ada. When one programs in these languages one 'prescribes' the manner in which the computer is to go about solving the problem. That is to say, one explicitly specifies the control flow necessary to carry out a given computation. The other category of language is known as applicative (or declarative) and describes/declares the logical structure of a

problem in that it specifies what kind of solution is being sought. These two language categories tend to be used in different problem areas but there is now much interest in the use of declarative languages such as LISP and PROLOG, over a wide range of problems.

Apart from an array of different programming languages, the areas of application tend to be divided into two real world categories: real time and non-real time. The term 'real time' can be used to describe any information processing system which must respond to external events or stimuli within some period of time. For example, an automatic teller machine, outside a bank, is part of a real time system since it must respond to customer demands for information and money and must interface with the up-to-date data within the bank. On the other hand, a piece of software which performs a payroll calculation does not fall into the domain of real time since it has only to produce a list of monthly or weekly salaries and the appropriate deductions while updating a payroll data base. The only constraint is that it must complete the operation in less than one week.

The application areas where one tends to find real time systems are mainly of an 'industrial' type, e.g. process control, plant control, communications, medical, military. Because of this need for a response to external stimuli a computer language that is used to develop a real time system must have certain additional attributes over languages which are used for non-real time development. Languages such as FORTRAN and COBOL are not real time languages. That is to say they do not have the necessary constructs which allow one to respond

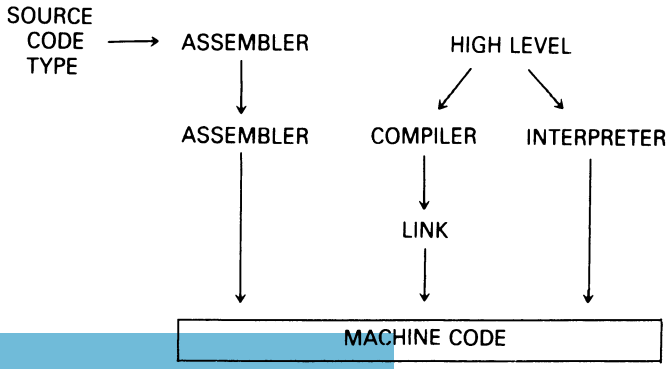


Fig. 9.1.

to those external stimuli. It must be stated, however, that certain manufacturers of compilers do add-on this facility to allow the language to cope with real time situations. The main point is, however, that such languages were not designed with real time applications in mind and thus will not be efficient when used with add-on facilities. Only relatively recently have genuine real time languages been designed and produced, the best-known being Ada. Modula 2 also falls into this category.

The choice of language for a particular application is a difficult one and is often decided as the result of factors outside the immediate constraints that one might be considering. For example, it may have been decided that because of a harsh environment only one type of computer hardware is suitable. If that particular machine has only, say, FORTRAN available, then the choice of language is narrowed to one. Thus both system and software designers must adapt their designs such that the use of FORTRAN can achieve the design aims. With the

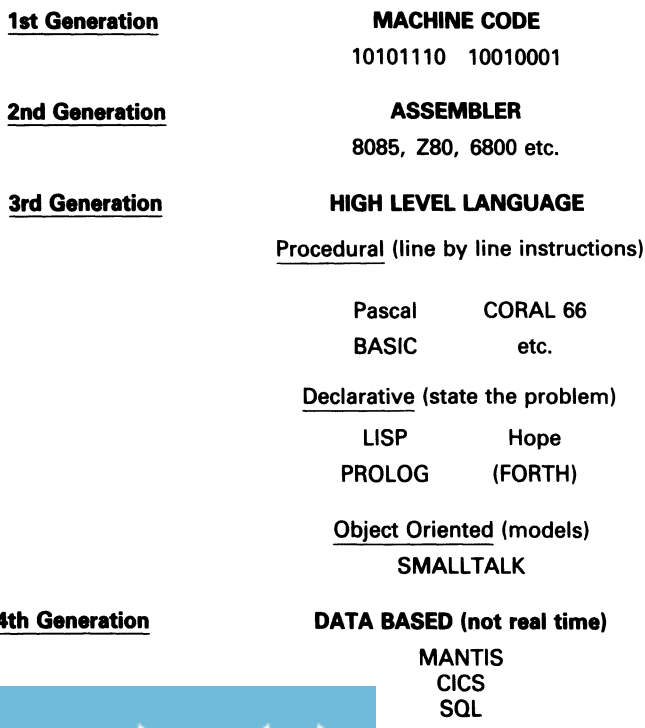


Fig. 9.2.

recent adoption, by various government agencies, of 'standard' languages (e.g. validated Pascal and Ada) the problem has been reversed to that of 'here is the language, which compatible system is suitable?' This leads to consideration of what is required of a real time language.

Figure 9.1 reminds us of a number of terms and places them in perspective. Figure 9.2 shows the four basic groups into which languages can be grouped. This chapter elaborates.

9.2 THE REQUIREMENTS OF REAL TIME LANGUAGES

Because of the nature of such systems, and their inherent complexity, there are certain additional criteria that must be addressed. Not all of these criteria are applicable to all languages, but all of them apply to real time languages.

9.2.1 Simplicity

Perhaps the overriding requirement for any language is that it be inherently simple. Achieving this leads to a number of advantages when one considers training, maintainability and portability. A good example of such a language is OCCAM developed for the transputer produced by INMOS. OCCAM was developed for a specific need but was designed with simplicity as the uppermost requirement.

9.2.2 Security

A secure computer language is one which is able to deal with errors made during programming or which occur at run time. Generally real time software needs to operate as reliably as possible and this places an added constraint that the language intrinsically allows one to create reliable programs. The first stage is clearly that errors made during programming be detected. The cost of correcting such errors at this stage is obviously less than during testing or, worse still, during service. As an example, if a language is strongly typed this helps prevent erroneous use of variables in expressions and forces the programmer to think more clearly about the way data is to be handled and transformed. The second stage is the detection of run time errors. These are faults which were not detected during compilation. Ada, for example, provides what it calls exception handling. That is, if some error condition occurs outside the predicted scope of the program, in other words an exception, facilities are provided to deal with the condition which might otherwise lead to failure of the program.

9.2.3 Adaptability

The language must allow the programmer sufficient flexibility to deal with the external environment since, in real time systems, it is most often the case that exotic peripherals need to be embraced by the system. Having to resort to machine code greatly jeopardises the integrity of the system and the language should, ideally, be adaptable enough to allow the sort of operations which are necessary.

9.2.4 Readability

One often-overlooked aspect of language design is the provision of constructs and facilities which allow the programmer to produce an easily read, and thus easily understood, program. Readability takes on great importance when modification is necessary by a programmer who may not have been involved in the original development. A language such as FORTRAN does not enforce readability on the programmer, whereas Pascal lends itself to better structuring and thus enhanced readability by virtue of its block structure.

9.2.5 Portability

The idea of language portability is one which has been around for a long time but its realisation in an actual language has taken many years to effect. There is inevitably a compromise here since, very often, a language implementation is mapped strongly on to the underlying hardware, thus making machine portability almost impossible. However, by separating out implementation-dependent aspects it is possible to attain portability and thus amortise the cost of the system development over a number of different hardware environments. An example of this is the ISO definition of Pascal which has allowed developers of Pascal-based systems to be confident of the portability between systems which support this standard.

9.2.6 Efficiency

Efficiency in a language has a number of distinct and often conflicting aspects. Until the recent fall in the cost of hardware, one of the most important considerations was the efficient use of available memory. It was often necessary to go to extraordinary lengths to squeeze a program into the available memory, often with the consequence that some of the requirements listed above could not be met. A further aspect of efficiency is to provide a language which can achieve the execution speed necessary to react to the stimuli being monitored.

With the relative decrease in hardware costs such considerations are becoming secondary and thus the onus of maintaining efficient programs falls on the language designer rather than the programmer.

9.3 PROGRAM STRUCTURES

The use of top-down design methods leads to considering the necessity of providing some means of mapping that design on to the language. That is, the programming language must be capable of representing a successive refinement of the design and thereby continuing the top-down process. Perhaps the most important construct which begins this is the module. A module is a collection of objects and their operators which is encapsulated in such a way that access from outside the module is controlled.

Within this module the language must provide some basic forms of control statement as, for example, the conditional statement:

IF *statement* THEN *statement* ELSE *statement*

or more specifically:

IF x = 0 THEN M := M + 1 ELSE RETURN

Another example is the WHILE statement:

WHILE *statement* DO *statement*

Such statements allow one to construct efficient and readable programs. The design of control statements is still a contentious area. Much attention has been paid to the use of the GOTO statement with various arguments presented as to its desirability or otherwise.

Once one is satisfied with the basic building blocks it is possible to consider the way in which they can be assembled within a module. One common approach is to assemble the program actions into some sort of block structure, perhaps delineated by the use of BEGIN . . . END. This makes it possible to define the scope of variables, which helps to improve efficiency in the program. Procedures and functions may be employed which allow the computation of frequently repeated tasks. Names can be given to these tasks such that they can be repeated as often as is necessary. Procedures also allow a better representation of the top-down structures. A function is a special form of procedure which has the specific task of computing a single value.

9.4 CONCURRENCY

Perhaps one of the most important facilities of a real time system is its ability apparently to execute several actions simultaneously. Once called multi-tasking, this involves the ability to execute several tasks or processes simultaneously. This can be achieved either by dividing up the time available that a CPU spends on each task or, in the case of true multiprocessing, by providing a multiple processing architecture with the appropriate synchronisation mechanism.

One obvious difficulty of concurrent systems is that some or all of the executing tasks may well be dependent on each other, so mechanisms have to be devised to enable the system to ensure that one particular task is completed before another starts. The solution to this, provided by Ada, is the so-called 'rendezvous' which provides a means by which two tasks may communicate. A different approach to this is to use a message-passing mechanism in which all task interaction is via the transmission of messages. A common approach within high level languages involves the introduction of a construct called the buffer. For example, a declaration of the type:

Mail-Box: Buffer [12] of *Message*;

which would create a mail-box buffer which can store up to 12 messages.

There are various other mechanisms which a real time language requires. One already mentioned is exception handling. In the case of a serious error it may be necessary to abort a process:

Abort (P) where (P) is an executing task.

Another necessary facility in a real time system is a clock for delaying a task or for waiting until some specified time. Also, it is useful to have some form of time-out mechanism.

9.5 DESIGN OF LANGUAGES

One of the main topics for consideration when designing a real time language is that of data typing. This is the feature, in a language, whereby each variable has to be bound explicitly to a specified data type. For example, volume and length could be derived from a mass

variable but a length value could not be assigned to a volume variable. It also means that the Js and Ks often assigned as loop counters have to be initially declared as integer types.

This affects the security aspects of the language and its flexibility. The readability will also be strongly affected by the language typing system. Whilst weak typing will allow more flexibility, for example manipulating bits, it is inherently insecure and wherever possible strong typing should be designed in with additional language features to make up for any possible loss in flexibility.

Program structuring can be divided into two levels. At the basic level, control structures are needed to specify the sequence in which basic program actions are executed. At the higher level, structures are needed to group sets of selected actions into single units. Together these two levels provide a mechanism of structured programming.

In multi-program systems the up-to-date approach is to introduce specific constructs for task specification, task communication and synchronisation into the language itself. This leads to much higher security as greater checking can be done when compiling. It does, however, put greater emphasis on production of higher-quality compilers.

Low level or assembler programming cannot always be tackled in a high level language; indeed, it may be desirable to avoid doing so, unless a high level language can allow bit-level manipulation, it will always be necessary to resort to low level programming. This need tends to be recognised and thus mechanisms are usually provided to facilitate the production of device drivers and the like.

One useful feature is that of separate compilation. Having to re-compile all programs each time a fault is corrected is both tedious and time-consuming.

Other features which need to be considered are the initialisation of variables and the features of input and output. It can be the case that a programmer may omit to initialise a variable before use and it is often considered desirable to make a compile time decision to initialise all variables. Within real time systems, the provision of input/output facilities is particularly difficult. Since each system is likely to be specific to a location, or a system in which it is embedded, such facilities will tend to become implementation-specific. One approach is to provide high level I/O facilities and leave it to the system implementor to provide the low level, system-specific part.

9.6 FUTURE LANGUAGES

So far only imperative languages have been discussed. Declarative languages, which include relational languages (e.g. PROLOG) and functional languages (e.g. Hope) are a new approach to the computer solution of problems. They are problem-oriented and are currently somewhat inefficient computationally. However, the advent of fifth-generation hardware will see this problem diminish. Perhaps the best known declarative language is PROLOG which stands for PROgramming in LOGic. Unlike other languages, which are formed out of functions, a PROLOG program is made up of a sequence of relations or assertions and rules about a subject. These form a data base of information about a subject that can be queried or added to. For example some assertions are:

is—functional (Hope)
is—logic (PROLOG)

An example of a rule is:

x is—declarative if (either x is functional or x is logic)

Functional languages manipulate functions rather than data and combine primitive functions to form a final function, the program. This program is then applied to the input data to produce the output. One consequence of this approach is that no variables are required. An example of a functional programming language is Hope (named after Hope Park in Edinburgh). Each function is represented by a set of equations that together will provide a result for the whole range of functional arguments. A program is simply a hierarchy of these functions together with a simple invocation of the highest level function. Hope allows the programmer to define specific (or polymorphic) data types that are checked by the compiler. These types allow for the creation of functions that can be applied to more than one type of data, for example a routine that can set numbers, characters, strings or records.

As an example of Hope one can define 'max' in the following way. Like Pascal, Hope is strongly typed. The function definition comes in two parts. First is the declaration and then one or more recursion equations. First one declares the argument and result types:

```
dec max : num# num→ num
```

dec, is a reserved word signalling a declaration. Here the two numbers are arguments which return a single number as a result.

The next part of the declaration gives the types of the arguments. Integers are of the predefined type, num. Read # as 'and a' and read → as 'yields'. Max only needs one recursion equation to define it:

```
--- max(x, y) <= if x > y then x else y
```

Read the symbol --- as 'the value of' and <= as 'is defined as'. A simple program using this max could be:

```
max(10, 20) + max(1, max(2, 3))
```

This would yield the result:

```
23: num
```

Existing functions can be used to define new ones. The following is the Hope version of max of 3:

```
dec max of 3: num# num# num → num
--- max of 3 (x, y, z) <= max(x, max(y, z))
```

Logic and functional languages are likely to lead to the solution of a much larger class of problems in the near future, not just in so-called expert systems but over a much wider field of application.

9.7 COMPILER EVALUATION

The greater emphasis placed on the use of high level languages, together with efforts to standardise such languages, has led to the production of validation suites for languages.

The use of high level language, whilst conferring many benefits on the development and maintenance of software, has one important drawback. This is that the software developer becomes dependent on the skill and accuracy of the compiler writer to generate machine code which represents his actual intentions. Also, because of the demands of portability, it is important that software need not be rewritten purely because of transfer from one machine to another. Since the development of COBOL, the need for language definition standards has been recognised and, more recently, these have been developed for Pascal, Modula 2 and Ada. Not only are these validation suites useful for the developer but they enable the end-user to specify their

use and thus be assured of a higher-quality product. The idea behind the validation suites is not just to exercise a language for conformance to a standard, but to provide deviance checks. Some suites look at quality and performance factors (e.g. speed of compilation). The development of such validation suites is difficult and they need to be under constant review and extension in order to be effective and meet the needs of users.

A measure of the interest which is being shown in such suites can be judged by the fact that no Ada compiler can be so-named unless it successfully passes the validation suite tests. The US government now requires that all Pascal development done on government projects uses a validated Pascal compiler.

Where it is intended to use either a language for which no formal standards exist or a language with no validated compiler, then it is essential that as full as possible an appraisal of the compiler and language be carried out, as soon as possible. Once implementation has begun it is too late to change course and restart the project. It can be argued quite strongly that if a hardware manufacturer chooses to ignore a language standard, then doubt should be cast on his ability to provide and maintain the services necessary for a software development.

9.8 CURRENT LANGUAGES

With standardisation efforts prominent and the development and use of Ada gaining momentum, it is worth while considering the languages currently available and their areas of application.

9.8.1 Procedural Languages

Ada

Ada grew out of the realisation that, in defence software development in the United States, many different languages were in use. No standard existed for defence work, particularly for embedded real time systems. The language was selected in 1979 and a draft ANSI standard produced in 1980. The 1983 version is proposed as an ISO standard and compilers are available for a number of host machines including the DEC VAX, although many more are in preparation. Current

compiler problems include:

Unreliability
Quality of the object code
Inefficient tasking
Slow compiling.

Improvements will follow but it is likely that Ada compilers will always consume more computing power than others. Nevertheless this will be far offset by gains in programmer productivity.

In particular, Ada addresses the needs of real time systems, specifically from the point of view of security. Ada is a strongly typed language with facilities for 'information-hiding' and is strongly biased towards top-down implementation. Some critics have viewed the language as over-complex thereby making it difficult to verify.

The verification problem has to some extent been addressed by the production of Ada sub-sets, however there has as yet been no agreement from the Department of Defence that such Ada sub-sets can exist with the label 'Ada'. The avionics industry in particular has identified many shortcomings in Ada which the use of a sub-set would overcome. However in the areas of performance, where speed of operation is paramount, assembler is still likely to be used.

Pascal

Developed from Algol in the 1960s and 1970s, and originally intended as a language for teaching purposes, Pascal has grown in recent years to become a full development language. It was designed to teach a top-down approach to system design and programming, whereby nested subroutines reflect a successive subdivision of the system into subsystems and modules. Hence, the term 'block-structured language' is often used. It is now one of the most popular languages, particularly for microcomputer programming.

The introduction of an ISO/BSI standard in 1983 and the development of a Pascal validation suite has given the language a major boost. It must be recognised, however, that it still has many deficiencies and it is not unusual to find subsets of the language in use so as to avoid the difficult features. Processors for the ISO standard are available for most systems.

Modula 2

Modula 2 grew out of the work done by N. Wirth on Pascal and is intended by him to correct some of the problems identified in Pascal,

and also to achieve the same goals as Ada but within a simpler framework. All Modula 2 programs are made up of separately compilable modules, each of which therefore contains details of the external and internal objects. It is thus a highly readable language.

With a standard for the language being imminent, Modula 2 is likely to be used by many real time software developers who either want to tread the path to Ada carefully or view Modula 2 as sufficient for their needs. A draft standard is anticipated shortly.

C

The C language has come to prominence in tandem with the greater use of the UNIX operating system. UNIX is almost entirely written in C and thus is seen by many as the language to use for writing programs to run on UNIX. C is best described as a system language which allows access to features which are usually only visible to assembler languages. C is a structured language with a slightly unusual syntax and is much favoured by programmers. However, C, like Pascal, has many deficient and unsafe features and its use must be constrained. It is thus not recommended for safety critical software.

FORTRAN 77

FORTRAN now dates back nearly 30 years and yet is still in widespread use. Originally a completely unstructured language, the '77' version introduced various structured concepts. FORTRAN is most widely used for scientific/engineering applications, although it has been used successfully in other fields (e.g. financial applications).

CORAL 66

This has been the preferred language of the UK MOD for real time embedded systems for many years and has been widely used for industrial and commercial control systems. It is due to be replaced by Ada in 1987. CORAL 66 will not disappear but Ada will be preferred.

COBOL

COBOL still dominates the commercial applications field. It was originated by the US DOD. Attention was paid to good data management and fast I/O.

BASIC

Beginners' All-purpose Symbolic Instruction Code is very widespread and is available on all small computers. It has become the universal

language of engineers and others who wish to do their own programming. Many versions exist, all of which permit instructions to be input for immediate response. Its disadvantage is that it is not a structured language and one is free to commit all possible coding errors.

ALGOL 60

This was the first truly structured language and was developed to provide all the now accepted language structures necessary for the construction of good programs. CORAL 66 in particular was heavily based on ALGOL 60.

APL

Standing for A Programming Language, this IBM developed language uses special symbols and keyboards to provide a very powerful language for modelling.

PL/1

Another IBM language which was originally developed to be the 'complete' language. Somewhat like ALGOL it is still used in IBM systems. PL/M is the microprocessor version.

9.8.2 Declarative Languages

PROLOG

The appearance of PROLOG in the last few years has been mainly connected with the growth in interest in expert systems. PROLOG is seen as a good vehicle for the development of such systems although its sphere of application is in fact much wider. As higher-performance computers appear PROLOG is likely to see even wider use in the computing community.

LISP

This was developed by John McCarthy at MIT in the late 1950s and is a highly specialised language for the area of artificial intelligence (IT). Since declarative languages are concerned with difficult ill-defined problems, the task of the programmer is not simply coding a solution but rather of exploring a problem and its possible solutions. LISP was thus designed as an interactive language, and a program interacts with the programmer to support experimentation with new ideas. Any LISP

implementation is supplied with aids for program development such as editors.

Hope
This was described in 9.6.

FORTH
This was designed by Charles Moore in the 1960s. FORTH is difficult to classify into these Chapter sections. It is a little like LISP but also contains features of assembler. The price associated with high level languages is the relative inefficiency of the compilers which need to take account of many hardware architectures. Assemblers are, on the other hand, specific to a processor and can be more efficient. FORTH steers a middle course between these extremes.

A key feature of FORTH is the stack which, unlike most high level applications, is controlled directly by the programmer. This language offers a high degree of interaction for the programmer together with fast execution speeds.

9.8.3 Object Oriented Languages
This is another group of third generation languages and includes one called SMALLTALK.

9.8.4 Fourth Generation Languages
Fourth generation languages evolved through a need for commercial programming environments which would achieve greater productivity. Most fourth generation languages are associated with specific data bases although the emerging standard SQL (Structured Query Language) is now available with most databases. Other well known languages include MANTIS, CICS and FOCUS.

Chapter 10

Aspects of Fault Tolerance in Software Design

A frequent misconception is that the elimination of errors during design is the sole factor in achieving quality software. This is too simplistic an assumption. In practice all software is likely to contain residual faults, albeit at very low levels, after extensive debug and quality activities. Compare, therefore, two safety systems, one of which has some low level of unknown residual faults and another which has double the number. Assume that the code having the greater number of faults has been carefully structured and limited in its ability to access variables and code in such a way as to restrict the propagation of errors. Assume, also, that there are a number of error check routines in the code which enable the program to reinitialise at known acceptable values when an error is detected. If, in addition to these features, the safety system and its software are designed in such a way that individual failures do not cause total loss of function, then it will at least continue to offer a degraded level of protection. An example of the latter would be a fire protection system which measures more than one parameter (i.e. UV light, smoke, rate of temperature rise). The interpretation of each type of input, and the generation of the executive output action, could be dealt with by separate parts of the hardware and software. This fault-tolerant type of design offers a far higher level of integrity than the system which, despite containing fewer faults, implies worse consequences in the event of failure.

There are many hardware design and system configuration features which have a direct influence on software reliability. The available strategies for achieving fault tolerance are described in the following sections which are summarised in Fig. 10.1.

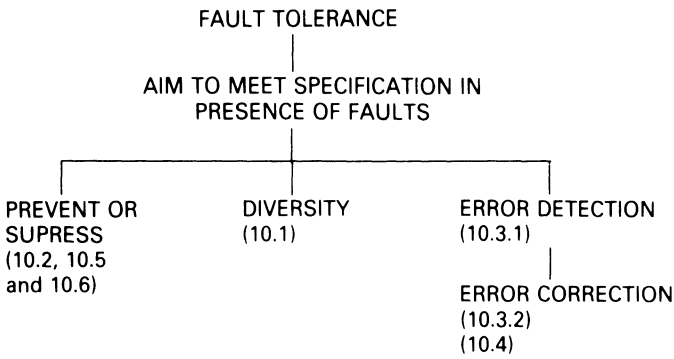


Fig. 10.1.

10.1 REDUNDANCY, DIVERSE SOFTWARE AND COMMON-CAUSE FAILURE

The most frequently used technique for improving system reliability is redundancy. Simple reliability mathematics leads to the conclusion that this will achieve several orders of improvement whereas, in practice, the situation is not as simple. Figure 10.2 shows a triplicated software system with some output being voted in order to achieve two-out-of-three redundancy. That is to say, any two correct inputs to the voter will result in a correct output. Hence, one of the triplicated software equipments may fail without detriment to the system function. Assume that the hardware failure rate of each equipment is 100 per million hours (approximately one failure per annum) and that a failed equipment is out of service for, on average, ten hours. The conventional formula for system failure rate is shown in Fig. 10.2 and suggests an improvement of two orders of magnitude. Consider, however, that for every 100 failures of a single equipment, just one is of such a nature that it occurs in all three. It is necessary, then, to add a series element to our reliability block diagram as shown. The failure rate associated with it is estimated as 1% of three times the individual failure rate. The effect, of course, is to swamp the effect of the redundancy. This phenomenon is known as common-cause failure.

Clearly, any software fault in the presence of simple hardware redundancy will represent a common-cause failure since the identical program will have been installed in each equipment. Common-cause failure is thus a problem in complex software systems. Recent surveys have shown that the number of common-cause failures due to software

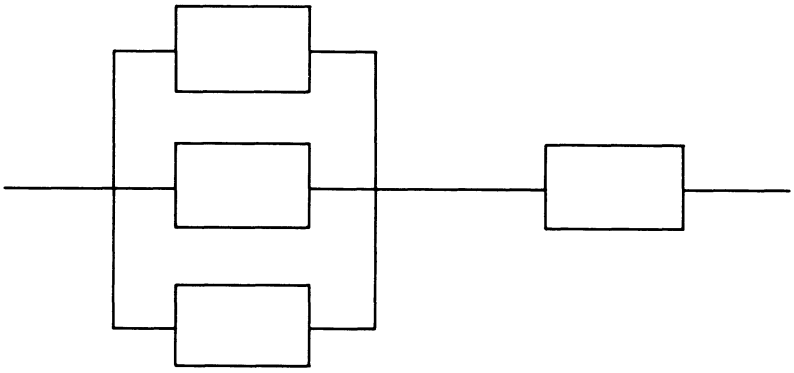


Fig. 10.2.

in real time redundant systems is generally greater than for any other group. Some typical redundant arrangements are:

Two channels with a comparator. If the two channels do not agree, then processing cannot proceed since there is no determination of which channel is healthy. An alarm or shut-down condition could then be initiated. In this case the overall reliability may be less than for a single channel since twice the number of failures will be likely. On the other hand, safety is enhanced since, for a hazardous failure mode, both channels need to fail.

Two channels with self-test. Each channel is subject to a periodic self-test initiated within the software. The most realistic type of test involves temporarily disabling a channel's output and then injecting a simulated signal to the input. In this way the function can be verified. If a single channel fails the self-test it can be declared unhealthy. Depending on the safety philosophy, the system can either revert to single-channel operation and inform the operator that there is reduced integrity or suspend processing. The self-test has to be thorough since error states which escape detection will cause system failures.

Three channels with two-out-of-three voting. An expensive alternative which compares three solutions and can permit one out of the three to become unhealthy. This solution can increase both safety and reliability due to the two-out-of-three redundancy which applies to both hazardous and spurious failure modes.

One approach to the common cause failure problem involves *software diversity,* which is a particular form of redundancy involving the design and coding of separate software for each of the replicated channels.

This is sometimes called *N version programming* where two or even three separate designs are implemented. Where there are two or more channels, circuitry and/or software has to be designed in order to decide which is the healthy channel in the event of an error. With three channels, a two-out-of-three voting arrangement can be employed.

Since diverse software is obtained by carrying out separate design and coding activities for the replicated channels, this will involve one or more of the following features:

Different algorithms.
Different storage media.
Different CPU architectures.
Different supply voltages.
Different CPU clock rates.
Different languages

Software diversity is extremely expensive and is unlikely to be applied except in highly hazardous applications such as nuclear or aerospace equipment. It is not, in any case, a total solution to the problem. We have already seen that a significant source of software faults is the activity of stating the requirements specification. As a result identical faults can propagate through each of the 'separate' design activities and manifest themselves in all the channels. Furthermore, even if the requirements were 'perfectly' stated, the designers in each team will surely have learned their profession via similar means. They will have accumulated the same thought processes and design techniques and read the same books. It is hardly surprising then that they implement the requirements using very similar code. Recent studies of this technique have revealed that the incidence of software common-cause failures is reduced but certainly not eradicated by this expensive dual (or triple) processing method.

10.2 ERROR PREVENTION

A number of design features involving both hardware circuitry and software techniques can contribute substantially to the system re-

liability. Some of these features prevent faults from occurring in the first place and others improve the fault tolerance of the system so as to render them non-critical.

10.2.1 Electromagnetic Interference (emi)

Electromagnetic interference is a significant problem with programmable systems. Screening and buffering techniques are useful and varied tests should be carried out. These should include passing interrupted current through loops close to the equipment and electrostatic discharge on to equipment surfaces.

Power supplies should be designed to resist mains-borne interference and transient spikes. In addition, the program should cater for power-fail recovery routines in order that the processing can recover from short-duration losses of power. The use of separate, preferably diverse, power supplies for each channel is essential, otherwise the power supply failure rate will swamp any improvement which may have been gained from redundancy.

Tests for resistance to emi include:

(a) Variation of supply frequency.
(b) Variation of supply voltage.
(c) Supply interruption (up to 500 ms).
(d) Spikes on supply (in kV orders).
(e) Electrostatic discharge on to surfaces (*c* 20 kV).
(f) Electromagnetic radiation (VHF to UHF up to 10 V/m).
(g) Ionising radiation.
(h) Electromagnetic radiation from high-voltage spikes in close-proximity cable.

10.2.2 Hardware Design and Architecture

Faults in one programmable device should not be capable of affecting another. Protection by means of buffers at inputs and outputs is desirable since it prevents faulty ports from pulling other devices into an incorrect state and helps to contain the error, thus minimising its effect and aiding diagnosis.

The input and output ports of solid-state devices usually fail to a permanent high or low condition. Circuit design can achieve better reliability, from a hazard point of view, by ensuring that failures are to a fail-safe condition. For example a comparator circuit should be designed to fail to a 'no comparison' state which is then detected as an error.

Memory should have adequate spare capacity to cater for expansion and overload eventualities. In the same way processor speeds and bus sizes should take account of future requirements.

Experience indicates that a large proportion of failures, from system integration and test onwards, result from timing problems of interaction between the software and the hardware equipment which it is controlling. Generous timing tolerances can be consciously specified during both hardware design and software writing so as to minimise this problem. It is not possible, of course, to foresee all the possible combinations of real time inputs and software states and it will still be necessary to uncover many of these faults by extensive and imaginative test procedures.

Also, graphics and human interfaces will have a significant influence on system reliability since they will influence the responses made by operators to various system states.

The previous paragraphs address some main features of design tolerance but the checklists at the end of this chapter suggest some additional areas for review.

10.3 ERROR IDENTIFICATION AND CORRECTION

Whereas the previous section dealt with design features which prevent the generation or propagation of errors, this section deals with their timely identification and correction.

The timely display of fault and error codes is a powerful aid to both reliability and maintainability because it helps the operator to recover from situations which, although not total system failures, might otherwise propagate in such a way.

10.3.1 Error Detection

This is achieved by a number of techniques.

Watchdog timer techniques involve using the processor clock to monitor outputs to verify that they are not stuck in one state. Unless it receives a reset within a predetermined period it will halt the processing. Many watchdog timers only require resetting within a loosely defined time and the result is rarely fail-safe. The technique will pick up a significant number of faults providing that the design of the watchdog is adequate, which implies an adequate understanding of the likely failure modes in the first place. The timer itself can be

periodically reset, thereby adding an element of sequence checking to its capability. It can also cater for endless loops by ensuring that defined points can only occur between two watchdog signals.

A large proportion of the hardware in a programmable system consists of memory. It is possible to check the state of a memory by writing a known bit into each location and then checking that it can be reread. This walking bit technique will minimise software corruption faults by flagging up failed memory. In the case of ROMs checksum techniques are needed which will verify the contents (which should not change) against predetermined checksum values.

The relay runner technique involves the setting up of a control variable which is incremented by known amounts at defined stages in the program. Its value, at any stage of execution, is thus predictable and if the value is incorrect there is evidence that an incorrect path has been implemented since the last satisfactory check of the variable.

Another technique is the use of code which effectively carries out the function of built-in test equipment. More sophisticated than the above-mentioned watchdog, it examines the state of the program and makes diagnostic judgements (designed in by the programmer) which enable instructions either in the form of codes or English language to be output to the user.

10.3.2 Error Correction

An extension of this philosophy is to provide code which can, having diagnosed that an error has been generated, correct the value or values in store so as to effect a recovery. The simplest example of error detection software is the parity digit, whereby an additional digit is included with some value. The binary value of the parity digit is set according to the value of the sum of the digits comprising the variable in question. If additional redundant information is provided it is also possible to deduce the bits which are in error and an algorithm can be designed which corrects them. More sophisticated checksum techniques have been developed from this idea.

Error detection and correction is thus a form of redundancy, not through hardware replication but by the use of additional code. The recovery philosophy is based on checking the acceptability of specific results (variables, data, outputs) and, if one of the tests fails, moving to an alternative path which will enable the program to recover. It is clearly not realistic to check the result of every step (instruction) in a program. However, the modular nature of the design will enable a realistic apportionment of the checks across the program functions.

Each check determines (by using an ELSE command) the next instruction. In this way the program is steered either to a recovery routine or to the 'no fault' sequence. The recovery sequence must contain some form of acceptance test to verify if the program has regained a satisfactory state. There are seven main approaches to such error recovery:

(1) *Reinitialisation.* This involves resetting the system to a known acceptable state and reinitialising the processing. Variables are reset to predetermined or known good values. The difficulty is deriving sufficient information to recreate the state immediately prior to the error. Clearly, remembering the state of all relevant variables prior to entering each block is far too inefficient. The program must jump to a predetermined position where known values and states apply.

(2) *Alternate path.* In the case of mathematical routines a second path, or try, can be provided. If the result of a particular calculation is not satisfactory then an alternative calculation can be performed.

(3) *Recovery blocks.* These consist of blocks consisting of an acceptance test for the calculation with one or more alternate routines which are used if the test fails.

(4) *Exception handling.* This consists of identifying conditions which are defined as exceptional and which require additional code to carry out the processing.

(5) *Memorising executed cases.* Here a record is made during certification of code of the allowed paths for program execution. This information is stored in some way and when the program is executed in its real environment the actual execution is compared with this allowed set. If an uncertified path is executed then safety action is taken.

(6) *Error correcting codes.* Here we try to detect and correct errors in sensitive information by using different types of code, e.g. Hamming codes. (See Section 10.4.)

(7) *Manual recovery.* In the event of failure, control of the system is returned to an operator who attempts to control the system rather than leaving it to automatic means.

10.4 DATA COMMUNICATIONS

The data communications medium between parts of the system or between systems in different locations is a source of bit error. Parity

and checksum techniques are used to detect and correct these. A data bit error rate of 1 E-6 means that one binary bit in 1 000 000 will be corrupted. By sending checksum codes far fewer than 1 in 1 000 000 messages will be corrupted.

For example, a 112-bit message may contain 96 bits of data and 16 bits of coded information derived from the other 96. A comparison of those 96 bits with the 16, after their receipt at the other end of a communications link, permits error correction to take place. The simplest method is for the software at the receiving end to request a retransmission of the message until the checksum computes correctly. In this way only 1 in 2 to the power 16 (65 536) of corrupted messages will propagate undetected.

10.5 GRACEFUL DEGRADATION AND RECOVERY

The whole design philosophy should take account of the need to operate in degraded modes. This can only be achieved at the requirements level where functional diversity can be specified, levels of system function can be defined and the operating requirements grouped into categories.

The software design decomposition should attempt to minimise the routes of communication between groups of modules so that errors are discouraged from propagating through the system. In this way errors are more likely to be confined to single functions or, at most, groups of functions. The system may then be able to provide service, albeit at a degraded level, by means of other functions. This is of particular importance in software systems controlling hazardous processes.

The incidence of mains-borne interference has already been mentioned. It is not uncommon for mains power to disappear for short intervals. Power fail recovery routines are part of the software and can enable a system to resume normal operation without failing. These must be designed in and thoroughly proved during system test.

10.6 HIGH INTEGRITY SYSTEMS

The achievement of high integrity in systems is of particular interest in the military field where hardware together with its embedded software

is required to survive extremely harsh environments and still exhibit high levels of reliability. In the early 1980s the UK Ministry of Defence began a research programme at the Royal Signals and Radar Research Establishment (RSRE) into developing microprocessors for such applications. The result has been the VIPER (Verifiable Integrated Processor for Enhanced Reliability) chip.

The root causes of failure in commercial microprocessors are:

—Imprecise specification of what is required.
—Inadequate verification of the gate-level design against that specification.

To overcome these problems the designers of VIPER have used formal mathematical techniques both to specify and subsequently to verify their microprocessor. By doing this, RSRE have produced a microprocessor of proven capability. As has already been pointed out however, hardware is only as reliable as the software which drives it. In order to achieve the combining of high integrity hardware with high integrity software a new computer language called Newspeak is being developed which has, as its goal, the production of safe and reliable programs.

The chip has been designed to form the main component of a self-checking computer module. Such a module would incorporate two VIPER chips, one operating as the active processor, able to read and write to the data bus, and another (monitor) processor which would only be permitted to read from the data bus. It incorporates a bank of comparison logic so that if illegal operations are attempted the processor stops and remains in that state until RESET is asserted. Also any discrepancy between the two processors can be quickly detected and execution halted. Details of the cause of any halt in execution are recorded in an on-chip diagnostic register and, provided that the detected error is non-critical to the module's performance, the processors can be restarted.

At present, programs for VIPER are written in a language called VISTA which is a structured assembler. Programs written in VISTA can be translated into MALPAS and SPADE (see Section 8.3.1) intermediate languages and thus subjected to static analysis. The commercial production of VIPER and the Newspeak compiler will herald a new era in high integrity system design and production.

CHECKLIST 10.1: DESIGN FEATURES

General

(1) Is there evidence that the following are taken into consideration:
  (a) Electrical protection (main-, air-borne);
  (b) Power supplies and filters;
  (c) Opto isolation, buffers;
  (d) Earthing;
  (e) Battery back-up;
  (f) Choice of processors;
  (g) Use of language;
  (h) Rating of I/O devices;
  (i) Redundancy (dual programming);
  (j) Data communications;
  (k) Man/machine interface;
  (l) Layout of hardware;
  (m) Hardware configuration (e.g. multidrops);
  (n) Watchdog timers;
  (o) RAM checks;
  (p) Error confinement;
  (q) Error detection and recovery?

Overall Software Design

(1) Were estimates of size and timing carried out?
(2) Are the timing criteria of the system defined where possible?
(3) Is the system secure?
(4) Is it testable as a whole and as subsystems or modules?
(5) Are there standard interfaces for:
  (a) Data transfer;
  (b) Peripheral interconnections;
  (c) Man/machine communication?
(6) Is the system designed in such a way that it can be progressively built up and tested?
(7) Have hardware and software monitoring facilities been provided? Is DMA a design requirement? If so, is the previous function affected?
(8) Have the controls over the data files been stated?
(9) Is use made of error check software?
(10) Have the checks on data loss and how to recover the loss been defined?
(11) What is the traffic of data on links and the speed of links? Will the links handle the traffic in the required time?

Will they give the required response rates?
(12) Can the internal data highways meet their loading and timing requirements?
(13) Will the system detect and tolerate operator error?
(14) To what degree can it survive abuse?
(15) To what degree will it prevent hardware or program errors damaging or losing data?
(16) Will it reconstruct any records that may be lost?
(17) Are there facilities for recording system state in the event of failure?
(18) Have acceptable degraded facilities been defined?
(19) Is there a capability to recover from random jumps resulting from interference?

Fault Tolerance

(1) Are detailed statements of reliability, degradation and recovery requirements of the system stated?
(2) Are there any special characteristics or failure modes of the hardware that the software must protect against or avoid?
(3) Are there syntax and protocol checking algorithms?
(4) Are interfaces defined such that illegal actions do not corrupt the system or lock up the interface?
(5) Are all data files listed? (There should be a separate list.)

Hardware Aspects

(1) Will the user configuration support the software?
(2) Is a development configuration required? If so, does it need extra facilities?
(3) Is there enough storage and is it of the required access time and degree of permanence?
(4) Are the processors sufficiently powerful and are there enough?
(5) Are the I/O devices sufficiently tolerant to misuse?
(6) Are the following adequate:
    (a) Electrical protection (mains and emi);
    (b) Power supplies and filters;
    (c) Earthing?
(7) Is memory storage adequate for foreseeable expansion requirements?
(8) Are data-link lengths likely to cause timing problems?

PART 4

New Management for Software Design

These last three chapters address various management tools including scheduling, forecasting and quality measurement. Section 5 is an exercise which will assist the reader in seeing the application of many of the techniques described. The book ends with a comprehensive glossary.

Chapter 11

Software Project Management

A number of management aspects need to be considered, such as estimating methods, audit procedures, recent national quality programmes and so on. This chapter provides an overview of these areas and their relevance.

11.1 USE OF AUTOMATED TOOLS

The difficulty of software project management is evident from the frequency with which software development projects flounder and overrun. This is, of course, partly due to a problem which has already been mentioned, that of changing requirements. However, there are a number of contributing factors. Perhaps the most important of these is the difficulty of measuring the level of completeness of any particular software task.

At the design level progress is viewed as a simple task since design is presented in tangible form, i.e. documents. However, when the design is translated into code the degree of completeness is less evident since the only point at which it is judged is the test. There are intermediate stages, such as compiling of the code, but the amount of information provided from this is very small. It is therefore evident that some measure is required in order to enable the project manager to determine just how far a package is on the road to completion.

Whilst not the total answer, the use of automated tools does provide a useful reference point as well as repeatability. As already mentioned, the use of automated systems for specification and design provides greater visibility to the system and also allows more thorough checking for completeness and consistency.

At the design level the information available from such systems can prove useful in identifying areas where effort needs to be concentrated rather than having to treat the whole system equally.

Once code has been provided, the main validation difficulties arise. There are a variety of steps which can be taken to minimise these difficulties. Firstly a development environment can be used to monitor changes. At a higher level the use of tools such as static analysers, coverage analysers and symbolic evaluation all provide valuable information about the program. The main point is that automated methods provide an independent assessment of the development rather than relying on subjective views which invariably lead to argument.

11.2 THE NEW APPROACH TO SOFTWARE QUALITY

The traditional approach to organising software quality takes on one of two structures.

The first is that of an organisational umbrella which provides a corporate quality service to the Company. The second is to attach to a project or development a separate dedicated quality function purely involved in that project activity and nothing else.

Both arrangements can make use of the established quality system and both may require project specific procedures or standards to be developed to cope with out-of-scope areas which might arise. The main reason for structuring quality systems in this project fashion is to provide a reporting mechanism which bypasses the organisational structure and allows rapid problem reporting to a higher level of management than would otherwise be the case. This, in theory, allows problems to be resolved more rapidly.

Whilst most organisations find the approach effective, the result is to put the quality function into a box and isolate it from the areas where it can be most effective. The result is a view of quality as part of the 'production process' whereby it is bolted on in the same way as other components. This reduces the level of quality finally achieved in the product.

It applies equally well to hardware and software that a typical project organisation separates the quality function and leads to the idea that responsibility for quality matters lies elsewhere. The reality, of course, is that each member of the development team has an effect on quality and it is in this area that improvements can be made.

One of the main problems of establishing a quality system is that the traditional approach tends to erect barriers between the development team and the quality function. Quality must be accepted as part of the working practice and only then will its level be raised. This is largely a matter of motivation in that the average programmer or designer only sees the task in narrow terms. He will view the task of programming and designing within a localised framework rather than in terms of the whole project. Only when the team members look at problems, and their solutions, in overall terms will their perception of quality and the means of incorporating it into the product be achieved.

This is, in part, an educational problem in that most software developers are never asked to consider the quality aspects of development, let alone how the development methods should be modified in order to assure built-in quality.

One possible approach is to introduce into the project the concept of overall quality involvement. This involves the idea of quality circles, now popular in manufacturing industries. The idea is to make all team members responsible for quality and thus distribute the function throughout the project rather than identify a single responsibility. By involving the team in this way, their perception and appreciation of quality are enhanced. In some respects this is an extension of the design review idea where a group discusses, in some formal way, the design as it stands at a particular stage. By extending the principle to all activities (including testing), greater emphasis can be focused on quality-related aspects. The general level of motivation towards achieving better quality is improved since team members then have a far greater understanding of the reasons for it and the benefits which will result.

11.3 SETTING UP AN AUDIT

Audit is a worthwhile activity and should be used both within an organisation and as a method of controlling vendors.

11.3.1 Objectives of the Audit

(a) To establish, by reviewing the techniques described in this book, that there is adequate control over the software design process.

(b) To establish that standards are being used for the documentation and production of the software.
(c) To assess, aided by the checklists, the comprehensiveness of the controls and the integrity of the product.
(d) To seek evidence that the standards are being applied and periodically reviewed.
(e) To establish that there are adequate controls over test and integration.
(f) To establish that there is a real capability, on the part of the team, to implement the requirements into a system.
(g) To establish that strict configuration controls exist.
(h) To establish that there is control over bought-in software.

11.3.2 Planning the Audit

It can be seen, from the contents of this book, that the number of potential questions which could be addressed is vast and, for that reason, it is essential to plan the strategy of an audit. Since it will not be possible to address every feature that affects software quality it will be prudent to select a sample based on the critical features of the product and any known problem areas experienced by the designer. The following information must be available in order to formulate a plan:

(a) *Vendor details.* The range of products and services. A summary of any approvals which he has from other customers or authorities. The management and quality structure. The names of executives responsible for design and quality. The main customers. The number of employees in hardware and software design, test, quality, etc. The existence of any standards such as BS 5750.

(b) *Product details.* The requirements specification. A functional summary and environmental details. The range of hardware with type and quantity of memory. The language used. The current stage of development.

(c) *Audit team and schedule.* A list of persons involved with their areas of responsibility. In determining the skill requirements, the program language and equipment type are relevant.

(d) *Documentation summary.* A complete list of specifications such as are described in Chapters 4 and 5. Prior knowledge of the documents and their structure will allow more time to concentrate on evaluating their use and the state of the project. A list of documents

should be prepared and a perspective of the hierarchy obtained by preparing a large chart such as was shown in Chapter 4. Any documentation standards or guidelines should be studied thoroughly.

(e) *Checklists*. A copy of the appropriate checklists from the previous chapters. These can be marked up to indicate the sample of questions which have been decided upon for the audit. In Chapter 4, one of the checklists is a specific sheet designed for each module. Sufficient of these for the number of modules to be audited should be available.

11.3.3 Implementing the Audit

Include the vendor's quality organisation in the documents and insist on a quality manual and quality plan. Look for evidence that these are not simply for show. There is a tendency at present to pay lip-service to software quality and often procedures are more advocated than practical. Look for hard evidence of their application.

The first aim should be to review the documents vertically to establish that requirements are fully and correctly reflected down through the specifications to the code modules. Module definitions should be audited for conformance to coding standards, layout, cross-referencing and functional performance. It will probably be necessary to take samples, in which case:

(a) Establish how long is to be spent on the activity and thus how many modules are to be audited.
(b) Allow adequate time for study of the requirements and functional specifications. This should not be skimped in order to include a few more modules.
(c) Choose a sample of modules having regard for the critical functional areas of the equipment. The sample need not be random.
(d) Choose a sample of changenotes and trace each through the system.

Review with the project manager the areas which you intend to audit and examine the schedule for timing of design reviews and tests. The audit is unlikely to be a single activity but spread over the various design-cycle activities. In the early stages the specifications can be audited. Later the coded modules can be examined, followed by the design review and inspection/walkthrough activities and eventually test and integration.

In an audit which extends over more than one day it is a good idea to present the problems daily to the vendor for discussion. For example, each day's findings could be copied to the vendor at 4.00 pm and jointly reviewed at 10.00 am the next morning. In this way timely remedial action can be initiated on the spot.

The sequence should be:

(1) Plan.
(2) Establish schedule of activities with vendor.
(3) Prepare checklists, specifications and standards.
(4) Audit.
(5) Initiate remedial action with vendor.
(6) Prepare audit report.

An important feature is that each and every deficiency should be written down and agreed, at the time, by everyone involved. All deficiencies must be based on factual evidence.

11.3.4 The Audit Report

Where the audit is spread over a long period then interim reports should be prepared after each visit. At the end of the audit a full report is required consisting of:

(a) Persons involved and their roles.
(b) Each of the checklists.
(c) Written report on each audit item.
(d) List of deficiencies and remedial action agreed.
(e) Actions taken and modifications which resulted.
(f) Summaries of design reviews.
(g) An overview and recommendations (no more than one page).

11.4 ESTIMATING

11.4.1 Seeking Metrics

The process of developing software is not just about coding—it involves documentation, design, programming, file building, testing, training, quality assurance and, above all, management. The development of software has been characterised by cost overruns and schedule slippage. Since the level of investment in software is increasing all the

time, managers need the answers to key questions such as 'How much will it cost to develop?', 'How long will it take and with how many staff?', 'What levels of productivity can be expected?' All of these questions, in principle, can be answered with a single quantity. The question is—can we provide such a number, or *metric,* for software?

A reasonable approach might be to study the many software projects in the past, gather data and thus set up models which, in theory, should permit the prediction of the parameters required and thus answer the above question with some degree of accuracy. The costs, manpower levels, durations and so on could thus be predicted.

A starting-point might be, for example, the number of source code statements generated per man-month. Every manager knows, however, that just because an average programmer generates ten lines per day, ten programmers will not necessarily generate 45 000 lines in two years. These are, of course, statistical averages and it must not be assumed that one particular programmer will perform in that way. The single productivity figure does not take into account the complexity of the given task, the spread of abilities in the team, interface problems and so on. Also, as has been said, software is not simply the generation of code. Clearly the amount of data and the number of parameters in these models must be very large in order even to approximate realistic values. How do the actual techniques already generated cope?

11.4.2 Actual Methods

The major current software packages which attempt this scheduling model are:

SLIM.
GECOMO.
PRICE S.
ARTEMIS.

The attraction of better control of projects via support tools is clear. Several attempts at providing them have been made.

SLIM (Software Life-Cycle Management) provides cost, time, personnel and machine estimates for developing computer software. Its accuracy has been validated using over a thousand different projects, in both the USA and Europe and covering a wide range of applications. The basis of SLIM is the work by Laurance Putnam of Quantitative Software Management Inc. It can be used from the

beginning of a project and provides information on productivity, minimum cost and time forecasts, risk profiles and manpower estimates. The question arises, 'How accurate are the values provided by such models?'

A number of useful studies and user surveys have revealed quite a variation in response. Whilst some projects have certainly benefited from the use of these tools, some users have reported as much as a 50% discrepancy in the values produced. It seems that in spite of quite large data bases and complicated parametric models it is still possible to produce projects which are able to lie outside the boundaries of the models being used.

Perhaps the best point made about these tools is that, used carefully throughout a project, they provide guidance on how various project parameters are behaving. Certainly on larger projects (i.e. over 100 000 source code statements) some benefit is likely from the tools and from the disciplines needed to collect and input the necessary data.

11.5 NEW SOFTWARE QUALITY PROGRAMMES

Awareness of the need for methods of generating quality software is increasing. This has resulted in a number of programmes being set up for the purpose of researching into methods and spreading information about them. The major programmes, worldwide, are described here.

11.5.1 The Alvey Programme

This is a UK programme, set up in 1983, to research software engineering as well as intelligent knowledge-based systems and VLSI. The Software Engineering part has the specific objective of promoting high quality and cost-effective software design by addressing:

Formal methods.
Software reliability.
Associated metrics.
Use of knowledge-based systems.

The major project is the development of two integrated project support environments. These are ASPECT and ECLIPSE. Another large project involves attempts to establish quality metrics for quantifying software reliability. The Alvey programme involves active

meetings of members which are successful in promoting awareness of techniques by an interchange of information and views.

Unfortunately funding for an extension to this programme has been reduced and it seems likely that much of the research instigated will be cut short. An Alvey II which is much more specific may evolve or a research effort with greater collaboration with other EC countries are likely future directions.

11.5.2 STARTS

This a UK DTI-funded programme. The letters stand for Software Tools for Application to Real Time Systems. Its objective is stated as 'The development and implementation of a national strategy for improving software engineering'. An important feature is the *STARTS Guide* (1984) which provides advice on choosing software tools and methods. It is intended to encourage software producers to adopt tools and methods likely to improve the quality and reliability of their products and is shortly to be updated with a new section on project support environments. The existing guide addresses:

Project control.
Requirements specification.
The design process.
Verification, validation and testing.
Version and configuration control.

Reports have also been published on SLIM, PRICE S, ARTEMIS, VDM, SOFCHIP, JSD, SDL, SAFRA, CORE and Z. Information can be obtained from NCC Ltd, 11 New Fetter Lane, London EC4A 1PU.

The two STARTS programmes—now known as Real-Time STARTS and IT-STARTS—run in parallel, but there is significant cross-fertilisation between the two as well as common technical concerns.

Whilst the approach of the two programmes is different in detail (because they address different communities), both are aimed at users *and* suppliers. They achieve their aim of accelerating the use of methods and tools:

'by providing guidance on best practice to in-house developers and external suppliers along with information about, and assessments of, the best available software engineering methods and tools;
in Real-Time STARTS, by promoting co-ordinated and constructive

demand from purchasers of systems for their suppliers to use the best software engineering practice;
in IT-STARTS, by involving the user more productively in the development process and by increasing awareness amongst senior management.'

A rapidly expanding programme of publications, projects, and events—including reference guides, handbooks, case studies, training, seminars, conferences, and user groups—ensures that the STARTS message is communicated to the widest possible audience.

Both programmes are supported and co-ordinated on behalf of the Department of Trade and Industry by Secretariats within the Software Engineering Division of The National Computing Centre.

11.5.3 ESPRIT Programme

The European Strategic Programme for Research and Development in Information Technology was set up by the EEC in 1984. It supports projects spanning industry, university and research organisations, covering:

Advanced microelectronics.
Software technology.
Advanced information processing.
Office systems.
Computer integrated manufactured office systems.

The software technology projects address all stages of the design-cycle and are concerned with the development of tools and methods, in particular:

Integrated programming support environments.
Formal design methods.
Fast prototyping.
Reusability of software 'components'.

A major project in this subgroup is REQUEST which deals with 'measures for software quality and the production of a database'.

11.5.4 EWICS TC7

The European Workshop on Industrial Computer Systems was set up as an EEC-funded group of committees of which TC7 is now the main activity. TC7 addresses the safety, security and reliability of industrial

real time computers. The work depends on the voluntary and active participation of the members and the main activity was the production of four guidelines on:

(a) Systems integrity:
Safety and reliability in the working environment.
(b) Software quality and metrics:
Identification of software metrics correlating with the safety features.
(c) Design for system safety:
Design- and fault-tolerant techniques.
(d) Reliability and safety assessment:
Methods of assessing design integrity.

11.5.5 CEC Collaborative Project

This European programme, from 1979, was set up for the 'assessment, architecture and performance of industrial PESs with particular reference to robotic safety'. The participating organisations were:

UK: HSE, NCSR.
Germany: BIA, IPA.
Denmark: EC, AT.
France: INRS.

The emphasis was on industrial robots, NC tools and automated warehousing and the seven objectives were:

(1) To collect safety reliability and classification data on PESs.
(2) To create a data bank.
(3) To collect and assess current guidelines.
(4) To identify areas for work.
(5) To formulate a framework for guidelines and for their review and development.
(6) To promote guidelines.
(7) To hold a seminar.

The PES3 symposium in Guernsey in May 1986 was the fulfilment of the final objective. No specific guidelines have been published at the time of writing externally to the project although draft documents exist. The data collection activity did not involve recording elapsed times: hence the failure information is qualitative.

11.5.6 SEI

In the USA, the Software Engineering Institute is funded to encourage the transition of modern software design methods from R & D into practical industrial and commercial use. The emphasis is on the production of suites of software tools for use in the design-cycle and projects are encouraged to that end. Some major projects include:

(a) Assessing existing programming environments.
(b) Software rights and licensing.
(c) Education and training courses and workshops.
(d) Integrating existing tools.
(e) Evaluating Ada environments.

11.5.7 MCC Programme

The Microelectronic and Computer Technology Corporation is funded by US industry for the purpose of carrying out R & D into computer technology. One of its four areas of interest is the software technology programme, whose aim is twofold:

(1) To develop methods for software quality and productivity.
(2) To transfer these technologies into the sponsoring organisations.

11.5.8 SPC

The Software Productivity Consortium is largely funded by US defence contractors. Its main area of interest is the high cost of embedded software in mission-critical systems. As a result research is focused on:

(a) Software reusability.
(b) Prototypes.
(c) Knowledge engineering for systems development.

11.5.9 STARS

Not to be confused with STARTS, STARS (Software Technology for Acceptable, Reliable Systems) is a US Department of Defense initiative to enhance software technology. The goal of the STARS programme is to improve productivity whilst achieving greater system reliability and adaptability. The DOD Ada development provides an initial focus for the development of a common shareable software base. The STARS programme broadens the scope of attention to the environment in which software is conceived and evolved.

11.5.10 JSEP

The Joint Software Engineering Programme, based in Singapore, currently concentrates on computer aided software development in respect of:

(a) CAD workstation to permit paperless design.
(b) Processing of natural languages.

Other areas of research are:

(c) Software development methodologies.
(d) Software metrics and quality.

11.5.11 SIGMA

This is a Japanese government-funded project to set up a national network of computer-linked bureaux, large users and software houses to the available software design support tools. Included are activities for developing software tools.

11.5.12 SPP

The Software Plant Project is a Brazilian programme for aiding the industrial production of software. It includes a system of proven programs.

11.5.13 RACE

A recent EEC programme, Research and development in Advanced Communications technologies in Europe, addresses 'common standards for all forms of electronic communications in Europe'.

11.6 SOFTWARE SECURITY

11.6.1 Security Against Data Theft

There are two ways of preventing unauthorised people from reading confidential or sensitive information:

—by denying them access.
—by making the files unreadable to them.

As a first line of defence, one can lock floppy discs away in a safe at the end of each day, but with large files, or a large number of smaller files, this can be very inconvenient. Then again there are programs

where repeated disc access would make this very slow. There are devices such as keyboard locks or post fitted locks which may be used to deter most attempts at prying but these do not equal the ultimate security of locking data in a safe.

The second method is to make files unreadable to unauthorised people by the use of specialised security packages which encrypt files before writing to disc, and decode them after reading from disc, and before passing them to the user program. These tools may be purely software or include add-in hardware cards for the PC. To use one's files a password may be required which is then used in a hashing algorithm to code or decode the files. Alternatively a dongle, or electronic key is plugged into the card and supplies the necessary encription code.

It is important to remember that security measures are only as good as one's commitment to use them properly. Passwords are not popular and a very short list includes the majority in use. Common names and dates continue to dominate that list.

Essential factors are:

—Commitment from management and staff.
—A hierarchy of responsibility to ensure that appropriate and adequate measures are taken.
—A training programme for all staff.

11.6.2 Security Against Data Loss

There are various ways of protecting against losing data. The main defence is to take copies of all files on a regular basis. Programs are not usually copy protected and many software houses recommend that backup working copies are made. It is important to:

—Follow their instructions.
—Label discs carefully.
—Store backup and master discs carefully in different locations.
—Record the serial number of the machine on which each copy of the programs is loaded. Do NOT make unauthorised copies. In the United States some of the larger software houses have started to take large corporations to court for breach of copyright, primarily to deter others.

Personal data files are harder to protect since they will change and will need constant copying to keep them current. One must make a

cost judgment as to the value of data against the value of time taken in protecting it, and take into account the probability of data loss. The most common loss of data often occurs when working on a spreadsheet and loading another without saving the first.

There are programs which will overcome this—one called BOOKMARK regularly saves files to disc either after an elapse of time, or after a preset number of keystrikes. This will also protect against unexpected power loss. After losing a large spreadsheet, with several hours work, most people will soon get into the habit of regularly saving their work, and in most packages it is harder to lose work without making a definite decision to exit without saving and, fortunately, power failures are relatively infrequent.

If a hard disc is corrupted, or files become erased, or a machine fails then one wishes one had backed up the hard disc. There are tape streamer systems at a cost of around £500 which will back up an entire 40M byte hard disc in a single operation. Moving down the list, there are programs specifically for backing up files. The most well known and widely used is probably 'FASTBACK' which saves the files to floppy disc, and can either copy an entire disc, or selective directories. It is fast and efficient and at the end of the operation states how long it spent copying files, and how long one took changing discs. Usually one spends more time than it does.

'XTREE', at around £35, is another popular package which includes back up routines as well as its more widely known file handling. One feature is its control of file attributes which means an archive bit can be set off after copying each file, but this will be reset when the file is next written to. By using this, one can do a selective backup on just those files previously written to. The time spent in saving files to disc can become tedious, so it is important not to copy more files than necessary. If work files and program files are lumped together, life becomes much more difficult.

11.6.3 Viruses

Much has been written in the press about computer viruses. Many articles have been rather too alarmist but there is nevertheless cause for caution and a need for awareness rather than panic on the subject. So called 'viruses' are rogue programs designed to do some form of damage to a computer or communications system. One known type of PC virus consists of extra code embedded in the COMMAND. COM file which writes itself into every copy of this file that it comes across.

If a floppy disc is 'infected' in this way and a program from it run on hard disc the virus embeds itself into the COMMAND. COM on that hard disc. Putting another floppy disc in the PC causes its COMMAND. COM to become 'infected'. One such virus waits until it has replicated itself a number of times before doing any damage. It then starts a malevolent process, such as destroying all the files on the disc.

Viruses can be embedded in any program, such as a popular utility. They can do anything intended from the sort of malicious destruction of data described above to something totally innocuous such as putting a random dot on the screen. As soon as a particular virus is discovered, someone will write a program to detect and 'kill' it. Virus programmers are therefore continually looking for newer ways to avoid detection.

There is no cure for viruses but sensible defences include:

—Being careful where one gets software from. Some 'shareware' and software downloaded from bulletin boards have been known to be infected.
—Not using illegal copies of copyright software.
—Backing up hard disc data and keeping several copies.

Viruses are, however, rare and there is a hundred times better chance of losing data through one's own mistakes.

11.7 SOFTWARE SAFETY AND LIABILITY

With the introduction of new laws on product liability and the inclusion of software within such laws, an additional burden is placed upon the software developer not only to ensure that his software functions as well as is possible but also will not provide cause for future litigation. As yet no test case has occurred to provide a precedent but it is likely that with the increasing use of software within critical systems, such as medical equipment, nuclear installations, automotive and transportation systems, etc. a case is likely, within the near future, which will involve a software 'bug' causing some form of disaster.

There are a number of points arising from this new legislation which must be considered carefully by software developers. The first and perhaps most pertinent point is whether to remain in software

development! If a company, say, views the risks of litigation as too great then this should be a serious consideration. It might involve the decision to move out of software development altogether or to subcontract to specialist software houses the development of the critical aspects of a system. Many companies have already taken such decisions and given the level of damages awarded and the cost of liability insurance, it is a decision likely to be taken by others.

If a company decides to remain within an area involving the development of critical software then it must look carefully at all aspects of management and control of software development. Many of the techniques and methods listed in this book must be considered, particularly formal methods for requirement specification since this is the area in which most problems lie. Also the use of formal tools for static analysis of software should be viewed more seriously, since the benefit of such tools has largely been under-rated. One danger is to adopt one of the many emerging CASE tools and expect them to solve all the problems inherent in software development. This is not so, since such tools have to fit into a framework of standards and procedures with good management appreciation of the needs of a particular project. Many of the tools available are very much 'horses for courses' and must be looked at carefully in the context of the system to be developed.

Thus the development of software in general is becoming more difficult from all points of view. The use of any methods which might achieve greater reliability and quality has become a priority.

Chapter 12

Quality—Can it be Measured?

This chapter looks at some of the aspects of software quality which will receive increasing attention in the near future,

12.1 BY THE SYSTEM DESIGNER

It is a frequent claim that 'We produce high-quality products' whereas in practice it is not easy to investigate or justify the statement. Quality is usually perceived as a relative concept. There is no absolute standard for quality: claims are based relative to some perceived norm or level against which its features are compared. If there is no reference standard the task of measurement becomes well-nigh impossible and since the term is needed for the whole range of commercial, industrial and domestic goods the problem is indeed daunting.

Quality is usually defined by relating it to conformance to specification. That is, if a product meets its specification then it is deemed to be of acceptable quality (whatever that may imply). However, consider the following circumstances which might apply to any product. A computer program is written which conforms to specification but which contains many GOTO statements, multiple entry loops, little commenting and so on. Is it good, because it meets the specification, or is it of poor quality for the reasons given above? It could be argued that the specification should have called for particular programming standards to have been used. In reality, therefore, it is the 'quality' of the specification and of the subsequent design which determines the product quality.

It is in this area that the greatest influence can be made by the customer specifying such features as programming standards. A

problem remains since human judgement is still involved in determining some arbitrary measure of quality.

In software design there is a tendency towards formal methods (Chapter 6) which provide greater visibility and traceability and which can ultimately lead to some formal metric relating to quality. Currently most quantitative methods involve applying metrics relating to certain complexity measures (number of paths, etc.). An alternative is to examine the outputs of a quality system and consider the number of software errors detected. These are still only rough estimates of quality because the fact that, in Project A, twice as many errors were detected as in Project B does not make it twice as 'bad'. Project A's staff might have been more effective in their integration and testing and will thus encounter fewer in-service failures. It is thus not simple to define a single quality metric.

A model of the whole life-cycle is necessary which will embrace the design-cycle and thus provide the system designer with a means of studying the design methods and outputs and of quantifying them.

12.2 BY THE BUYER

Ultimately it is the purchaser of the software who must judge, in some way, the quality of the system. Whether this is in an arbitrary manner or by some qualitative route may not involve the designer but, in the end, it is the buyer who will choose.

Currently most quality systems involve a requirement for vendor appraisal. This involves examining the controls and codes of practice used by the software developer. One major limitation here is the quality of the staff carrying out the audit. A good auditor, who is knowledgeable about software practice, will discover far more than a good traditional quality engineer. Specific software subject knowledge is important when dealing with this type of contract. The only way of extending the audit is to use sophisticated tools actually to analyse the software. The static analysers described in Chapter 8 are one example of recent additional tools.

12.3 BY MEANS OF METRICS

Traditionally, the features of software which relate to its quality are identified and evaluated qualitatively. The whole of the current

software quality approach, described in Part 2, relies on judgements of such qualitative characteristics.

However, it has been a theme of this book that the path towards better quality software involves a more formal and objective approach to the process of design. Such a trend involves the identification of more measurable features.

Quantitative measures of quality are referred to as software metrics. Various measures of size, complexity, structure and programming resources are identified and an attempt is made to model the software quality by means of these variables. For example, the relationship might be established between complexity metrics and errors or coding time.

The two main variables of interest, for which metrics are sought, are:

(a) *Errors.* The number of code errors or faults per line of code. Note that we talk of errors per line rather than per hour.

(b) *Coding time.* The total time taken to specify, code, review and test modules.

A key area of interest must therefore be the validation of the metrics against actual working systems. Preferably historical project data should be used in order to provide objective evidence of the relationships. However, this data is seldom collected.

Major limitations arise from:

(a) *The use of only single dependent variables.* A simple model might address the relationship between, say, error rate and a particular complexity measure. This approach fails to recognise the interrelationships and trade-offs between parameters. One designer might achieve error-free code by patient documentation and review of his code, whereas another might prepare code faster and expend effort on debug and test. The resulting error rate might well be the same although the metrics are not.

(b) *The use of only a single independent variable.* In other words only one metric is chosen to model the software performance.

(c) *The assumption of linearity.* This assumption has often led to the rejection of a metric as a valid indicator of software quality. More sophisticated models may be necessary.

(d) *Misuse.* Each metric describes a particular feature of the software. Judging only on the number of lines of source code would

lead to the erroneous conclusion that the use of assembly language is more productive than high level language.

The various metrics fall into four broad groups:

(1) *Code metrics*. These address the more easily quantified measurements at the code level. They are, however, not available until well into the design-cycle and hence do not provide the complete answer. They include:

Lines of code:
The number of lines.
Cyclometric complexity:
A measure of path and execution complexity.

(2) *Structure metrics*. These address the higher levels of design and include:

Information complexity:
A measure involving data flow and data relationships between components.
Invocation complexity.
Review complexity.
Stability.

(3) *Hybrid metrics*. Hybrid metrics are modifications of structure metrics weighted according to various code-related measurements. Three hybrid metrics are:

Weighted information flow.
Weighted review complexity.
Weighted stability measure.

(4) *Programmer-related metrics*.

Number of changes.
Years of programming experience.
Mix of programming experience.
Number of pages of documents.

Given that correlations can be established between the variables of interest and the metrics, then the problem lies in the repeatability of the model. It is tempting to assume, having observed a historical connection between error and coding time, that a repeatable model has been established. This is potentially dangerous, as has been found

in the case of the somewhat similar hardware failure rate regression models.

It is credible that the number of factors which influence software failure rate and coding time may not all have been identified. In any case interrelationships of metrics must alter with such intangible variables as personality or project organisation.

One benefit of even an imprecise model is that those modules being more likely to contain the larger numbers of errors can be identified and additional review effort applied to them.

On the positive side, this work is in its early stages and many successful predictions are claimed. Only the continued collection and analysis of field data will enable these regression models to be established.

12.4 BY FAILURE DISTRIBUTION MODELLING

Several statistical models have been developed for the purpose of estimating reliability. Those described in this section are unstructured models in that they treat the software as a black box and attempt to model reliability against time. General opinion seems to be that no one model is better than the others at predicting reliability. The basis for such models is the assumption that failures are statistically distributed at random. This is not necessarily supported, since they occur only as a result of a specific path being executed coinciding with a program fault.

The problem, which is lightheartedly illustrated in Fig. 12.1, is that the data (Monday–Wednesday) may contain the information to predict a few days ahead but not necessarily to the time when the very low failure rates of interest apply.

The models are, however, useful as management tools in monitoring the effectiveness and the progress of test. They are not predictive during design but only extrapolate empirical data. No attempt is made to compare the models here but a brief description is given of each.

12.4.1 Jelinski Moranda

This model, based on analysis of NASA and US Navy data, assumes that failure rate is proportional to the current error content of a program, that the remaining faults are equally likely to occur and that additional faults are not introduced into the system.

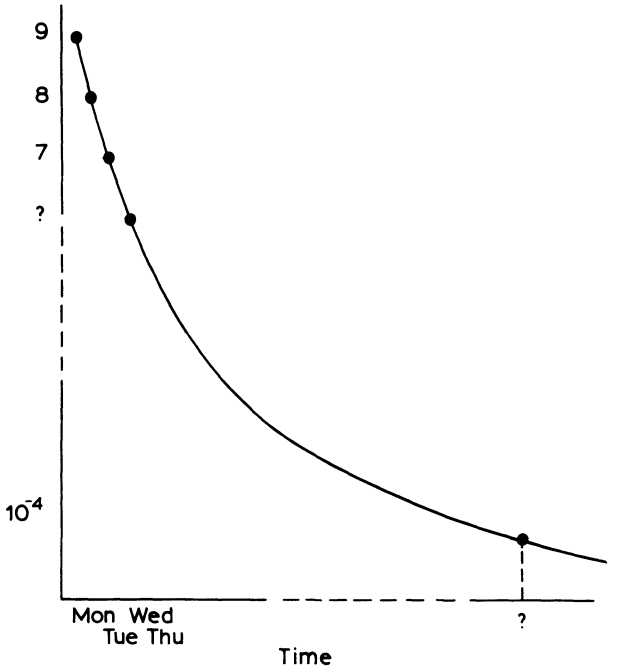


Fig. 12.1. Software failures.

12.4.2 Musa

John Musa's model is based on program execution time and assumes that:

(1) Errors are mutually independent and occur at a constant rate.
(2) There is a representative mix of instructions and functions.
(3) The MTBF is greater than the execution times.
(4) Errors are removed when revealed.

12.4.3 Littlewood and Verral

This is a Bayesian growth model which estimates the time to the next failure by inference from the previous failures. An exponential distribution is assumed such that reliability increases, with time, as each failure is revealed and corrected.

12.4.4 Shooman

In this case empirical data from previous, similar, projects is used to provide the values of the constants (e.g. errors per instruction) for a

model. Again, a fixed number of errors, which decrease as failures occur, is assumed. The model involves an error correction rate parameter which, in practice, will vary with the manpower levels available to the project.

12.4.5 Schneidewind

Another empirically based model which involves collecting data, identifying distributions of failure and applying those distributions to the parameters of the project in question. Exponential and Weibull models are utilised as appropriate according to the way in which the failure rate changes.

12.4.6 Brown and Lipow

This involves defining the possible combinations of inputs to a program in order to model the reliability based on test results on a known number of input combinations. Since the complexity of software programs implies very large numbers of combinations, it is necessary to define subsets to which the model can be applied.

12.4.7 Seeding and Tagging

A number of researchers have applied seeding and tagging techniques to the problem of estimating failure populations. The method involves the injection of known faults into the software. The success rate of debugging the known faults is used to predict the total population of failures by applying the ratio of successful diagnoses to the revealed non-seeded failures. For the method to be successful one has to assume that the seeded failures are of the same type as the unknown failures and thus equally likely to be revealed.

12.5 THE PROBLEM OF CERTIFICATION

The development of computer systems with embedded software has one inherent uncertainty. That is the impossibility of proving a program's correctness. Whilst it may not be a disaster if a payroll system fails (unless it belongs to your employer), the failure of an aircraft flight control system is obviously of great importance and raises the question of certification.

The purpose of certification is to ensure that a system has been developed and implemented in accordance with accepted practice and,

as a consequence, is unlikely to fail and cause loss or injury. With hardware systems the certification problem is simpler but, for software, difficulties arise from ambiguities, such as what constitutes accepted practice.

At present the closest measure of acceptability is that a company operates a strict quality system. Clearly this does not guarantee a conforming or 'quality' product, but it is unlikely that more precise forms of certification will be available in the short term.

In the medium term only the use of the formal techniques (Chapter 7) and automated test methods (Chapter 8) will provide this confidence.

12.6 FAILURE DATA ACQUISITION

In order to construct meaningful metrics it is important to have sufficient failure data to enable the models to be derived. The manner of data collection is important but unfortunately there is still very little interest shown in collecting detailed information on the performance and failures of systems. Furthermore, the metrics and modelling methods discussed cannot be validated, let alone generated, without adequate field failure data. Ideally, the following data is required:

(1) A description of the running conditions of the program.
(2) Date and time of the start of the run.
(3) Date and time of the incident/failure.
(4) Date and time of restart.
(5) Date and time of the normal termination had there been no failure.
(6) Effect of failure on system performance.
(7) Data and I/O load and any environmental factors.
(8) Detailed narrative of the incident.

Clearly this quality of data is only available from a highly formalised and controlled maintenance reporting system. It costs money to collect that level of information and, therefore, is only likely to be generated in a project where management are suitably enlightened as to its benefits in terms of reliability improvement and future maintenance planning.

In many cases it is only possible to count the number of failures which have occurred in a given time or a given number of tests. Under

these circumstances the calculation of metrics is highly unlikely. There still remains the problem that we are 'hung up' on the concept of time, which has no real relevance. We should be attempting to clock the number of new patterns or routes used within the system since the rate of change of these is the important factor.

12.7 BENEFITS AND DRAWBACKS OF ASSESSING SOFTWARE

12.7.1 Integrity Assessment

Software integrity assessments are more and more frequently required as the application of programmable systems finds wider applications. In the case of safety-related systems the first step is to establish the boundary of the safety system and to define the failure modes to be considered. A qualitative assessment is then carried out by reviewing the quality-related features described in this book. The HSE document, described in Section 5.4.1, provides one particular basis for assessment particularly for applications where failure results in hazard.

There are, however, both benefits and drawbacks in carrying out these assessments.

12.7.2 Benefits

Confidence. By identifying and removing specific faults and by observing features of good design, confidence is established in the system.

Feedback. Design faults are fed back and corrected at an early stage in the design cycle, thus minimising the cost of failure.

Liability. In cases of fatality or personal injury the trend is towards absolute liability irrespective of negligence. The only defence, if EEC proposals are implemented, will be that of best endeavours having regard to the technology. Whilst not a total defence, software quality and integrity studies must surely make a positive contribution.

12.7.3 Drawbacks

Error rate versus fault tolerance. At the beginning of Chapter 10 a comparison was drawn between a low error rate and the benefits of a fault-tolerant design. It is important that assessments do not concentrate on error prevention alone.

Relevance of parameters. Although the many aspects of software quality, described in Chapters 4 to 10, contribute to low error rate and to fault tolerance they are not a guarantee of no failures. There will be parameters which are not known and therefore not addressed in the assessment.

Lack of metrics. The problem concerning metrics and certification was addressed in Sections 12.3 and 12.5. Since there is, as yet, no satisfactory simple way to quantify the 'quality' of a piece of software, assessments remain largely qualitative.

False confidence. The lack of precision in assessing software has been stated. Nevertheless, there is a tendency to assume, merely because a system has been subject to audit and assessment, that it is therefore free of faults. This is clearly not so but remains a drawback associated with assessment.

Chapter 13

The Role of the Software Engineer

13.1 WHAT IS NEEDED

The development of the computer industry over the last 30 years has been both rapid and characterised by change. In the Preface the major changes were summarised and this book has dealt with the effects of the new techniques involved. One factor that has remained almost constant, however, is the industry's reliance on the computer programmer. This reliance stems from the fact that computers are of no use without programs, or sequences of instructions, to execute. Thus, the programmer has found himself at the centre of the stage and, more often than not, is criticised for the frequent delays encountered during software development. A question worth raising is 'Is the programmer to blame?'

The short answer is probably 'No'. The task of programming a computer, whether directly in machine code or in a high level language, is one of problem solving. For many years, and perhaps still, computer programmers have been drawn from a variety of disciplines and often trained from scratch with no specific educational requirements having been identified.

The average data processing department has its sprinkling of graduates from various disciplines, and post 'A'-level students, as well as others with no formal qualifications at all. The skill which unites them is the ability to interpret problems in a way compatible with the computer. Clearly a mathematical background is useful in programming the solution to a differential equation and an accounting background is useful when writing a sales ledger system. Nevertheless, the primary ability lies in translating an understanding of the problem into a new language—the programming language. Thus it would seem

a simple matter to provide the computer industry with the type of personnel it needs. This unfortunately is not the reality of the situation.

One of the major problems facing the computer industry, including the current drive into 'information technology' and its applications, is the severe shortage of suitably 'experienced' rather than 'qualified' staff. The difference between 'experienced' and 'qualified' arises because the computer industry in particular is guilty of being very experience-specific when seeking staff. A cursory glance through the job advertisements will quickly reveal an emphasis, from every employer, on experience of specific languages and machines. This is due partly to the large variety of hardware, but in the main to a familiarity with this type of staffing philosophy. One solution would be for universities and polytechnics to provide more computer scientists but, unfortunately, it is here that another problem arises. The problem is one, as we have already seen, to which the industry is particularly prone—that of defining what is required. It seems that the problem of imprecise requirements (Chapter 6) is not confined to system design but also flavours the specifying of personnel skills.

In the very early days of the business, the 'programmer' was more than likely a mathematician who also needed to know the intricacies of the hardware which he was attempting to program in low level machine code. Gradually, higher level languages appeared and the need for intricate hardware knowledge lessened. Ability to code, debug and test rapidly became predominant and there was a tendency to document the design after the event. Recognition of the problems inherent over the whole life-cycle have sharpened the view that formally qualified software engineers are needed. These engineers require formal instruction in the aspects of software development described in this book and the skills include:

(a) *Discrete mathematics*. To tackle more formal specification methods and to use formal verification techniques.

(b) *Formal design methods*. To provide traceability and maintenance information within the design.

(c) *Static and dynamic test methods*. To verify, during design, the system being developed.

(d) *Microelectronics*. To aid system understanding.

(e) *Social issues*. So that social, safety and liability implications of the impact of complete systems are understood.

Until a fully qualified software engineer, with the appropriate knowledge and skills, makes his appearance, the computer industry will remain experience-specific and the quality of computer software will remain low. The software engineer should become the driving force in determining standards for his own profession and thereby establish criteria for classification.

It is only through this process that a profession, in the accepted sense, will be established. That the academic world seems currently incapable of satisfying this need is clear since only few establishments produce graduates of the type described. An initiative is needed to develop the basic standards upon which to base the professional software engineer. This requires that the academic bodies, the engineering institutions, industry and government all agree and implement the appropriate educational policy. Only through a more professional class of computer staff will the industry advance. The software engineer is not the only new professional needed. Systems engineers and quality engineers must also be trained with the necessary skills to contribute to the life-cycle in order to cope with the complex systems of the future.

13.2 STRUCTURED TRAINING FOR A STRUCTURED DISCIPLINE

The need has been stated for a new type of software engineer and Part 3 of this book has described the structured and formal techniques, with their automated tools, which the programmer/engineer must acquire.

The design-cycle has been presented as a formal and logical progression of requirements and design phases with appropriate feedback loops for verification and testing at each stage.

It seems self-evident, therefore, that software engineering training should be similarly structured and should proceed in the same sequence of activities as the design-cycle. In this way the discipline of the design-cycle will be instilled into the engineer at the same time as encouraging an orderly decomposition and stepwise refinement of the problem.

Three features of the training should be identified and defined at the planning stage when structuring a software training course. These are:

(1) *Knowledge.* This may be defined in terms of areas such as those listed in Section 13.1.

(2) *Skills*. These should be defined in behavioural terms. In other words each objective should describe a set of functions which the trainee will be able to perform as a result of the training.

(3) *Attitudes*. This is the hardest area to define: it refers to the preconditioning which governs an engineer's response to the software tasks.

The knowledge and skills can be related to the design-cycle and should enable the engineer to perceive and control the software problem. The principles of decomposition, modelling and structuring code then follow, supported by appropriate mathematical and support tools. The design cycle thus provides the areas for training:

Requirements:
- Understanding the difference between *what* and *how* (Chapter 3).
- Handling requirements languages (Chapter 6).

Design:
- Formal documentation and readability (Chapters 4, 7).
- Programming standards (Chapters 4, 7).
- Language familiarity (Chapter 9).
- Fault tolerance (Chapter 10).

Review and test:
- Design review (Chapter 7).
- Code inspection and walkthrough (Chapter 7).
- Test strategy (Chapter 8).
- Static analysis (Chapter 8).
- Audit and subcontract (Chapter 11 and Exercise Chapter 14).

Structured training has frequently been shown to produce software engineers/programmers whose performance is better in terms of errors by as much as an order of magnitude as well as their being better at meeting schedules. The shift is from error detection to error prevention, and programmers, now *engineers,* acquire a sense of professionalism and of being in control.

13.3 THE IMPORTANCE OF THE WORKING ENVIRONMENT

One of the major reasons for the inadequacy of the software models discussed in Sections 12.3 and 12.4 is the large variation in performance criteria for programmers. For this reason cost, schedule and reliability predictions are well-nigh impossible.

It has been suggested that high- and low-performance groups tend to cluster in specific organisations. Whether this is true or not there is little conclusive information which might allow one to identify the reasons for any one individual's performance. A number of likely factors have, nevertheless, been discussed:

(a) *Peer effects.* In one controlled experiment randomly paired programmers carried out a particular software design task. It was observed that, whereas the performance across the group varied by a factor of 5.6:1, the variation between the two members of each pair was only 1.2:1.

(b) *Speed.* The same study indicated that speed did not detract from quality. In fact the median-speed programmers produced the most defects whereas, surprisingly, both the faster and slower programmers generated fewer.

(c) *Workplace.* It is fairly commonplace for programmers to criticise their workplace in terms of:

Noise.
Privacy.
Interruption.
Ambiance.

In general 50% are critical of each factor and tend to be pessimistic about the prospects of improvement.

Studies show that the better performers correlate with the better perceived working environments. It is not clear whether the environment produces the improvement or if the better performers, by virtue of their performance, acquire the better conditions. Experiments have been conducted where moves to better conditions resulted in twofold improvements in performance. However, as Hawthorn showed in the 1950s, industrial performance is usually improved by any change, be it for better or worse, since it is the fact of the change, rather than its nature, which stimulates performance. There is clearly much scope for study in this area in order to identify 'programmer metrics', or features, which correlate with performance.

Nevertheless, performance and good environment are, for whatever reason, positively correlated and there is little excuse for not seeking the potential twofold improvement in quality and output which could result from addressing this much neglected area.

PART 5

Exercise

The following exercise presents a hierarchy of documents representing the requirements and the design of a detection system. It is seeded with errors (mostly deliberate) and some guidance is given to the reader in identifying them. Substantial contributions were made to this exercise by Dr Paul Banks and Mr John Dixon.

Chapter 14

Software System Design Exercise— Addressable Detection System

This exercise is in the form of a set of documents covering the design-cycle of a simple system for the detection and annunciation of fire in an enclosed building. The design contains deliberate errors, omissions and ambiguities, some of which are obvious and others which will require more careful study to reveal them. The problem should be treated as a whole rather than as individual elements since the errors will often only be revealed by comparing requirements between documents. The documents include:

—A Requirements Specification.
—A Functional Specification.
—A Software Specification.
—Module and Sub-module Specifications.
—A Quality Plan.
—A Test Specification.

In Chapter 4, the need for a traceable document hierarchy was emphasised. Figure 14.1 is an outline of the document structure for this design. Some of the document numbers and titles have been deliberately omitted. An initial scan of the documents in this section will provide the information needed to complete the chart. It is strongly recommended that this is attempted before proceeding with the exercise.

The requirements and design documents which follow should be studied and compared with each other. The tutor's discussion notes near the end of this section describe some of the omissions, ambiguities and errors in order to guide the reader. The remaining errors, including the not so deliberate, we leave to you.

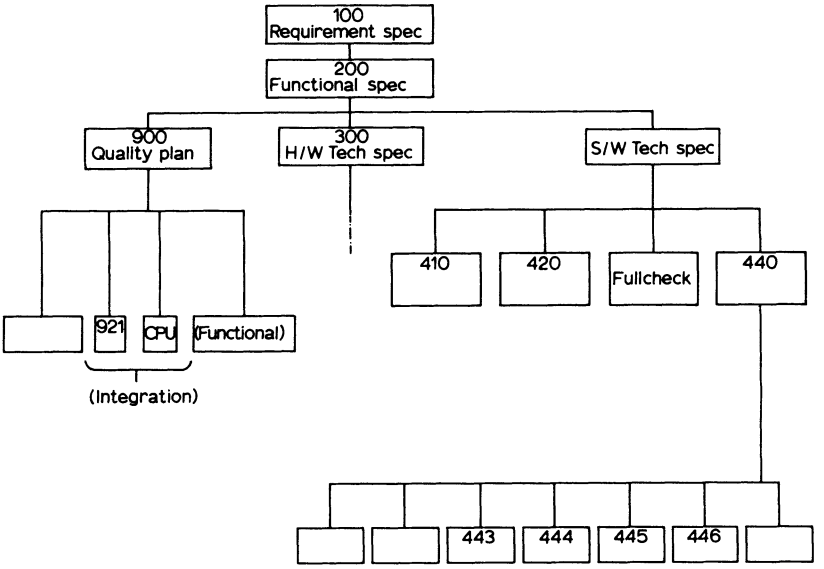


Fig. 14.1. Documentation hierarchy.

At the end of this section a revised functional specification (Issue 2.0) is given. Although by no means a model solution, it takes into account some of the errors in Issue 1.0.

PD352/100
Issue 1.0
1.1.88

REQUIREMENTS SPECIFICATION— ADDRESSABLE DETECTION SYSTEM

1.0 OVERVIEW

A programmable detection system is required to provide both automatic executive actions and annunciation to the user. It is used to control fire water sprinklers and plant shutdown equipment. The equipment is required for use in enclosed spaces.

2.0 OBJECTIVES

2.1 The system is protecting a hazardous plant and thus must provide continuous monitoring of inputs and must always fail safe.

2.2 To provide visual and audible annunciation of detectors.

2.3 To provide automatic initiation of executive outputs.

3.0 OPERATING REQUIREMENTS

3.1 Switch on and switch off must not cause spurious actions.

3.2 Hardware failures of the equipment must automatically be detected, diagnosed and displayed to the operator.

3.3 The system must be capable of tolerating up to 24 hours mains failure and respond to mains recovery without loss of function.

3.4 The system must be capable of periodic automatic self test whereby the effect of simulated input signals is verified without loss of function.

3.5 The equipment is to operate on 220/240 V mains 50 Hz.

4.0 FUNCTIONAL REQUIREMENTS

4.1 Inputs must respond to digital and analogue transducers.

4.2 Inputs must be uniquely identifiable to the system.

4.3 Outputs must consist of:

(a) Signals to sprinkler systems.
(b) Volt-free output loops.
(c) Data to a display panel giving input status.

4.4 Maximum I/O capacities are:

(a) 80 digital inputs.
(b) 80 analogue inputs.
(c) 50, volt free outputs.
(d) 50, 24 V outputs.
(e) Provision for hardwired and RS232 interface to a graphics or mimic facility.

4.5 Ability to group inputs into definable zones.

4.6 A duplicated logic unit with the ability to initiate outputs on the basis of combinations and comparisons of the inputs of each zone.

4.7 Ability to respond to the following stimuli:

(a) Ultra violet light (25 ms duration).
(b) Smoke (hydrocarbon fires).
(c) Infra red light.
(d) Temperature.
(e) Rate of temperature rise.

4.8 All applications software shall be implemented using a block structured language having an ISO standard version.

5.0 ENVIRONMENT

5.1 Ground fixed enclosed buildings.

5.2 Ambient temperature range −10°C to 35°C.

5.3 Humidity to 95%.

5.4 Near proximity of hydrocarbon process plant.

6.0 OTHER REQUIREMENTS

6.1 The equipment must be repairable on line without total loss of function. A repair time objective of 1 hour is required.

6.2 The system must be able to accommodate future extensions either by an increase of I/O capacity or by interworking duplicate or higher numbers of equipments.

6.3 The probability of failure to respond to a valid input, on demand, shall not exceed 10^{-6}.

6.4 The incidence of spurious action shall not exceed one incident per annum.

6.5 A facility for logging events.

6.6 The following standards and guidelines will be applied during design and manufacture.

(a) BS 5760.
(b) BS 5750 (Part 1) 1987.
(c) STARTS Purchasers Handbook.
(d) HSE Document—Programmable Electronic Systems in Safety Related Applications 1987.

6.7 The equipment shall fit into an area 7 m × 3·5 m with a ceiling height of 2 m.

6.8 The weight of the equipment including power supplies shall not exceed 500 Kg.

PD352/200
Issue 1.0
1.1.88

FUNCTIONAL SPECIFICATION— ADDRESSABLE DETECTION SYSTEM

This document in no way represents an example document for use. Warning! Contains deliberate errors. For training purposes only.

1.0 INTRODUCTION

1.1 The proposed system will perform the following functions:

(a) Monitor the fire detectors to interrogate their status at one second intervals each.
(b) Initiation of built-in test mode at system start up and at operator request.
(c) Display status of each detector on a colour graphics screen.
(d) Triggering of audible alarm in the event of fire detection.
(e) Triggering of fire sprinkler systems and outputs to other control systems.
(f) Self diagnosis of the computer system at start up to detect faults.
(g) Printouts of system events.
(h) The equipment shall be 220/240 V 50 Hz mains operated.

1.2 The hardware configuration is as shown in Fig. 1.2.

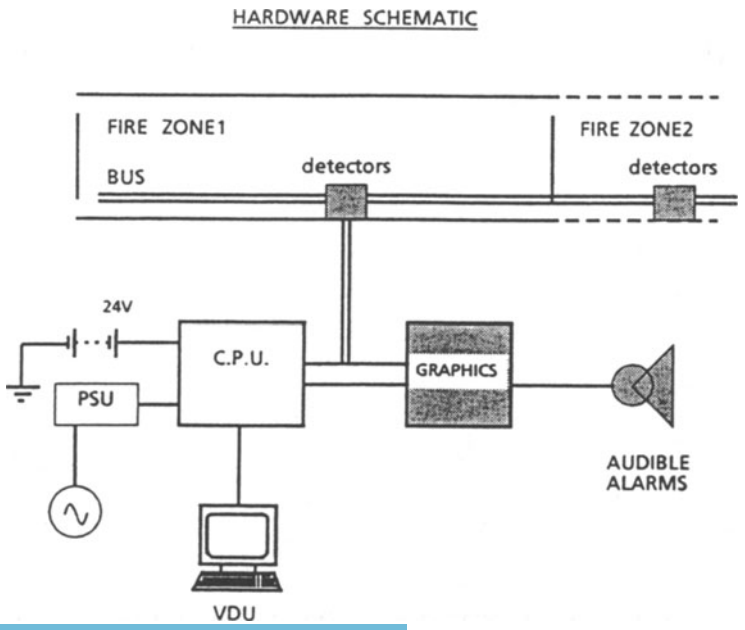


Fig. 1.2. Hardware schematic.

2.0 FACILITIES AND FUNCTIONS

2.1 The system will allow connection of up to 127 detectors each of which is separately addressable.

2.2 A graphical display will provide a visual mimic of the status of each detector with a 'picture' of its location.

2.3 The audible alarm must be sounded when any event occurs.

2.4 A VDU will duplicate the information displayed by the mimic.

2.5 In the event of failure of the CPU the computer system will revert to the use of hardwired circuitry.

2.6 The system will provide a start up mode of operation which will perform system checks and then go into normal operation. A system shut down mode will also be provided.

2.7 During normal operation all events and operator commands will be logged on a disc.

3.0 SYSTEM OPERATION

3.1 Operator Inputs

The system will be operated via the computer system keyboard. Commands may be entered, singly, at the keyboard. The set of commands is:

DIAG —Performs auto diagnosis on the computer system.
DEST(⟨n⟩) —Initiates detector self test where n denotes the detector.
STAT(⟨n⟩) —Forces a report of all detectors' status or, if n is specified, a particular detector.
CANC —Shuts down audible alarm.
LOG —Prints all status changes since last system restart.
MIMC —VDU reverts to zone mimic (see 3.3).

It will not be necessary for the operator to input data except where

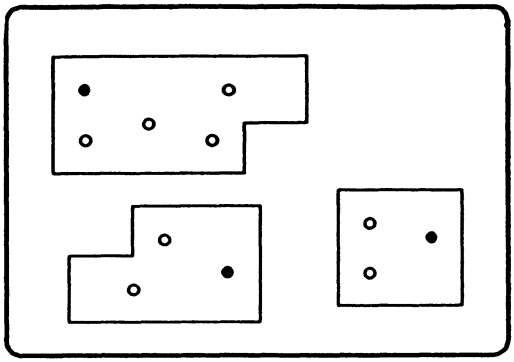


Fig. 3.2.

required by a command. This will enhance the security and integrity of the system.

At switch on, the system will require the entry of a password known only to the operator. Password validation will be allowed only three times within a minute and then an enforced 20 minute delay will begin.

3.2 VDU Format

The detectors will be polled sequentially and the VDU will normally display information in the form:

Time	*Detector*	*Status*
xx:xx:xx	n	OK

Upon the command MIMC (see 3.1) this will be replaced with a zonal mimic (see Fig. 3.2).

Activated detectors will be indicated by a change of colour and intermittent flashing.

3.3 MIMIC

The graphics panel will consist of groups of LED displays representing the fire zones and their individual detectors.

3.4 Log File Layout

The log file, held on disc, will contain all command and response information and also the result of system start up. Each message and

response will be stored sequentially in the form:

⟨TIME, COMMAND, RESPONSE⟩ ⟨TIME, STATUS CHANGE⟩ .

3.5 Detection Logic

There are two levels of detection logic.

(a) A non-executive level which actuates the audible alarm and updates the VDU and mimic. This is normally triggered by the activation of a single detector.
However the detection level shall be programmable by the user according to his requirements.

(b) An executive level which is normally used for the activation of outputs associated with the fire suppression systems. A typical arrangement would be the voting of two inputs out of n.

3.6 Executive Outputs

These are triggered in response to the appropriate detection or manual inputs as pre-programmed during installation by the user. The executive outputs will consist of volt-free loops capable of switching 24 v solenoids and of carrying 50 mA. Both make and break conditions shall be available as the outputs.

4.0 DESIGN, DEVELOPMENT AND TEST

4.1 Hardware

(a) A standard 16 bit micro-computer will be chosen for this system implementation using the MS-DOS operating system. Hardware peripherals will be as specified in PD352/300.

(b) Suitable I/O units will be used to interface each detector with the binary bus to enable individual detector addressing.

(c) Eurorack equipment practice will be used.

4.2 Software

(a) The Pascal language will be used for the coding of all software except where it is necessary to interface to hardware, where assembly code will be used.

(b) A validated compiler will be used.

(c) A standard real time operating system will be employed which

has disc file facilities embedded. No special tailoring of the package is expected.

(d) The software developed for this system will be validated when the system is completely coded. In this way the testing will be more efficient and the problems more quickly resolved. For this reason configuration control will not be applied until this stage is completed.

4.3 Test

Since this is a small system only a minimal project plan will be produced.

5.0 OPERATION AND MAINTENANCE

5.1 The system will be attended by an operator who will be conversant with the output facilities. He or she will be capable of using the input commands.

5.2 Printed board changes will involve system re-start.

5.3 Detector changes will be possible without disenabling the system.

5.4 Removal of a detector shall not result in any alarm state but must be indicated at the operator station.

PD352/300
Issue 1.0
1.1.88

HARDWARE TECHNICAL SPECIFICATION—ADDRESSABLE DETECTION SYSTEM

This document in no way represents an example document for use. Warning! Contains deliberate errors. For training purposes only.

This document is not used in the case study

PD/400
Issue 1.0
1.1.88

SOFTWARE TECHNICAL SPECIFICATION—ADDRESSABLE DETECTION SYSTEM

1.0 OVERALL SOFTWARE DESIGN

1.1 Strategy

The software shall be designed using top-down techniques and coding using structured programming. The major programming language will be ANSI Pascal except where assembler code shall be used for the purpose of speed requirements.

Documentation will include use of an in-house pseudo-code to describe the top levels of the design, with successive levels decomposed into pseudo-code. The functional specification is described in document PD352/200 (Functional Specification).

1.2 Design and Development Factors

A standard real time operating system will be employed which has disc file facilities embedded. No special tailoring of the package is expected to be necessary.

2.0 DETAILED SOFTWARE DESCRIPTION

2.1 Top Level

The software will be structured so that after initialisation of the system the main body of the program will run continuously until shut down.

The modules defined within this document will each be the subject of a module design document. Test and QA are described elsewhere.

2.2 Data Flow (Logical Design)

The data flow within the system is described by Fig. 1.

2.3 Software Systems Structure

The overall systems structure is shown as Fig. 2.

2.4 Control Flow

2.4.1 The overall systems flow chart (flow control)

This is shown in Fig. 3.

2.4.2 The top level of design (pseudo code)

The top level of design is shown, expressed in pseudo code, covering the overall top level systems flow chart.

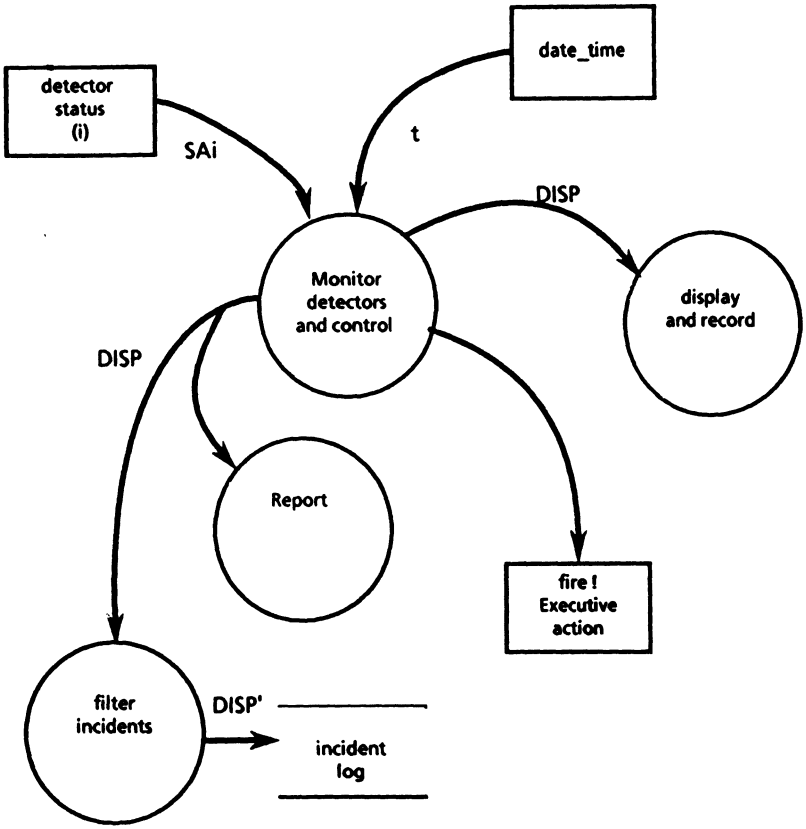


Fig. 1.

The overall top level pseudo code will have the following structure:

```
Supervisory module;
  Begin
    Initialise system;
      Perform diagnostic checks;
        Do Forever;
        For i = 1 to n (where n is the maximum number of detectors)
        Begin
        check status (i)
        check detector (i)
        set report
      End For
        End Do
End Supervisory module;
```

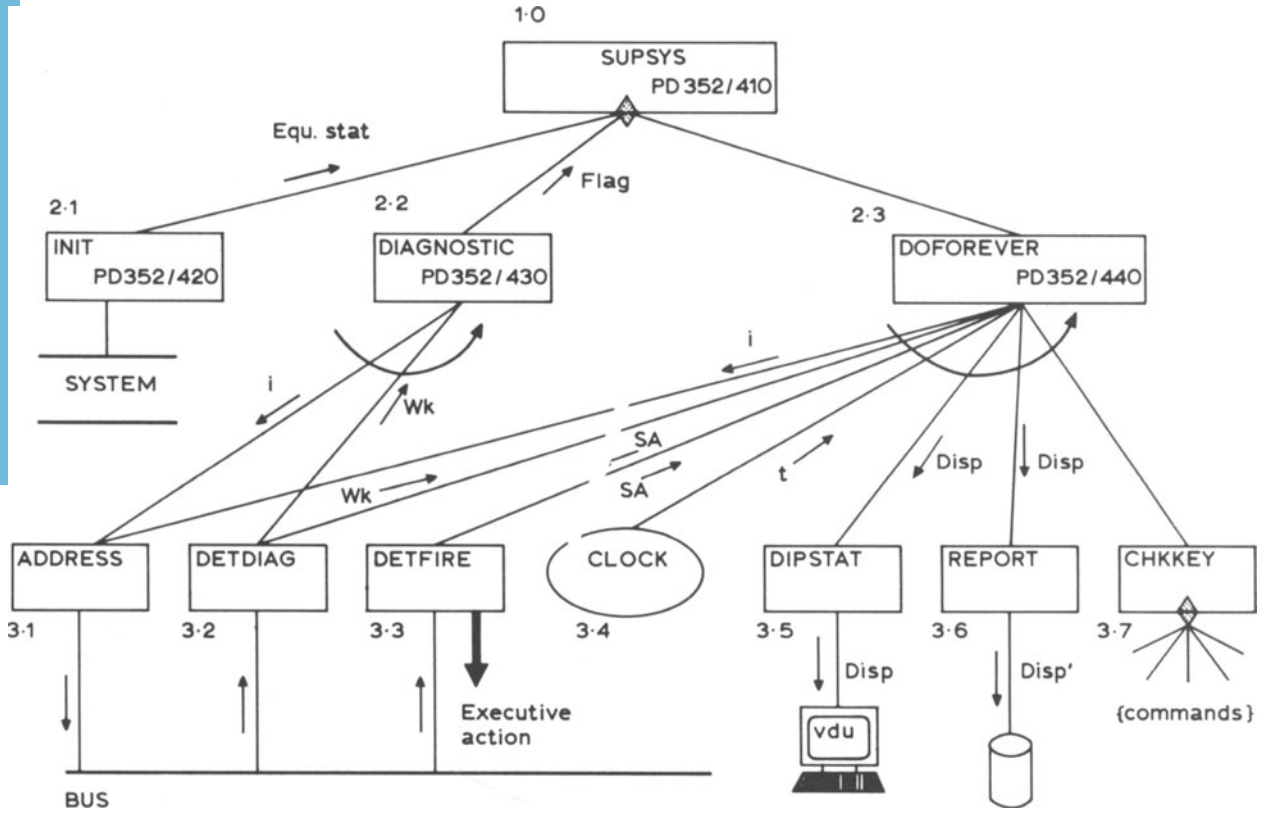


Fig. 2. Overall system structure chart. Note information in data dictionary (2.5).

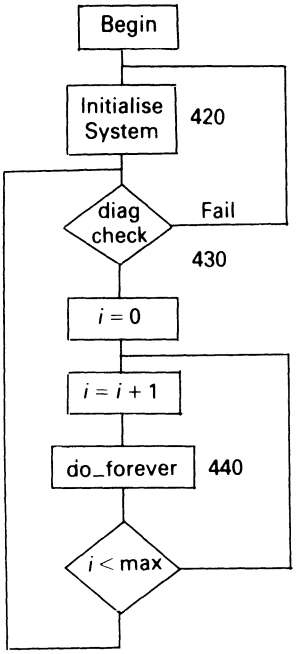


Fig. 3.

2.5 Data Dictionary

Item	*Type*	*Dimension*	*Description*
i	Integer Scalar	—	Control counter for detector sampling $1 \leq i \leq$ DMAX
Flag	Boolean Scalar	—	True if detectors working, else false
WK	Boolean Scalar	—	True if detector[i] in working order, else false
EQU-STAT	{Text Arrays Boolean arrays}	(to be decided) (to be decided)	Message about outcome of initialisation. Logical Flags for outcome of various activities of initialisation.
t	Text Array	12	Date and time, i.e. DDMMYYhhmmss
SA	Boolean Array	≥[DMAX]	Status array for detector fire status. For [i] ≤ [DMAX] then SA_i = TRUE ⇒ Fire detected else SA_i = False
DISP	Data Construct		[SA*i*, *t*, *i*]

3.0 SOFTWARE SYSTEM DESCRIPTION

The overall software shall be modular in construction, each module will be testable and address one particular subsystem of the complete software description. The major subsystems are given below with their corresponding documentation number.

3.1 Supervisory System (PD352/410)

The purpose of this module is to amalgamate all lower levels of software within a cohesive whole.

3.2 Initialise (PD352/420)

The purpose of this module is to perform various checks at system start up:
These functions are:

—Run diagnostics on main computer.
—Set up VDU and Graphics.
—Inform the operator of system state.

3.3 Diagnostic Check (PD352/430)

The purpose of the diagnostic check module is to check the initial status of the detectors (a) before full operation and (b) at periodic intervals during operation.

3.4 Do Forever Module (PD352/440)

This loop is the principal mode of operation, it polls the detectors in sequence and reports on their status. Each pass will do the following:

—Address detector.
—Check detector operation.
—Acquire status on fire detection.
—Display status.
—Report.
—Check keyboard status.
—Execute Command.

The commands are:

DIAG —performs auto diagnosis on the computer system.
DEST⟨n⟩—Initiates detector self-test where n denotes the detector.
STAT⟨n⟩—Forces a report of all detectors status or if specified, a particular detector.
CANC —Shut down audible alarm.
LOG —Prints all status changes since last system restart.

PD352/440
Issue 1.0
1.1.88

SOFTWARE TECHNICAL SPECIFICATION—ADDRESSABLE DETECTION SYSTEM

DO FOREVER MODULE—VERSION 001

1.0 MODULE DEFINITION

This loop is the principle mode of operation since it polls the detectors in sequence and reports on their condition. Each diversion through the loop will address the following sub-modules, each sub-module shown below is the subject of a software design document (written in pseudo-code for ease of understanding).

1.1 Address the Detector (PD352/441)

That is unless otherwise directed each detector will be addressed sequentially starting at detector 1 and progressing to detector n.

1.2 Check Detector (PD352/442)

Module will perform a basic diagnostic check on the addressed detector.

1.3 Acquire status on fire detection (PD352/443)

Fire status will be reported on a is fire detected basis. Polling will take place between detectors in the same zone as will voting.

1.4 Display Status (PD352/444)

This software module will indicate the presence of the detector and its status will be reported on the VDU.

1.5 Report (PD352/445)

This software module will record all status changes onto the disk drive.

1.6 Check Keyboard (PD352/446)

This module will check the keyboard for admissible input.

1.7 Execute Command (PD352/447)

This module will, on receipt of a significant input from the keyboard, execute a given command.
The commands are:

DIAG —Performs auto diagnosis on the computer system.
DEST⟨n⟩—Initiates detector self-test where n denotes the detector.
STAT⟨n⟩—Forces a report of all detectors status or, if specified, a particular detector.

CANC —Shut down audible alarm.
LOG —Prints all status changes since last system restart.

2.1 System Flowchart for Do Forever Loop

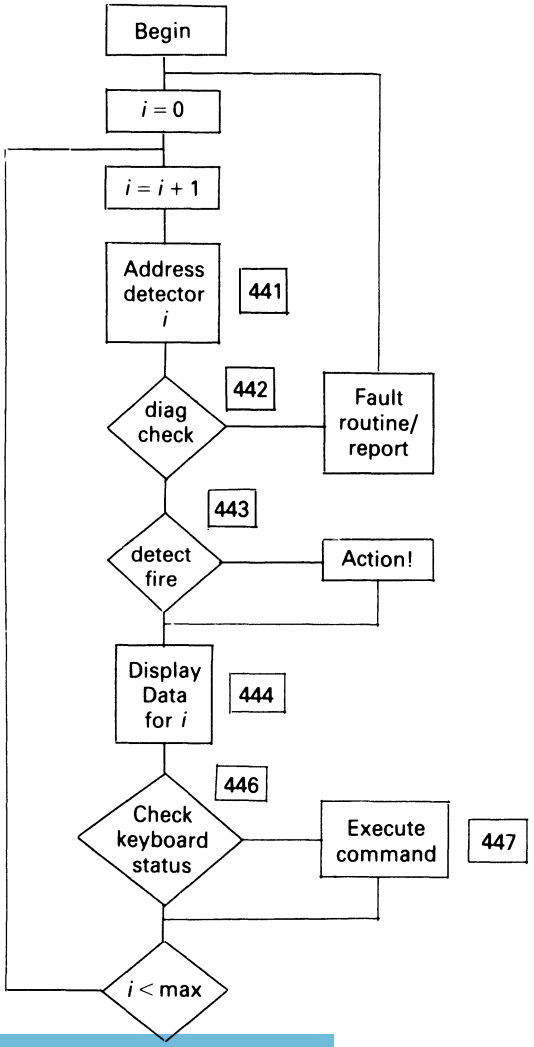


Fig. 1.

2.2 Do Forever Module Code

```
Begin {Main loop}
        Begin
    Address Detector (i);
        Begin
                    Set address detector;
        Check detector;
                  Begin
                                Get detector data;
            if detector data ? OK then;
            Set fault (i);
            End (Check detector);

            Detector status;
        Begin
                Get detector status;
            if detector status ?OK then;

                Set alarm (i);
                    Begin;
                        Set status (i) = T,
                End (Set alarm);
            End (Detector status);

        Display data (i);
            Begin
                Set Graphics region (i, status, fault);
        End (Display data (i))

                    Report data (i);
            Begin
                    Send output data to log;
            Send output to log;
Send output to printer;
            End (Report data (i))

            Check Keyboard entry;
            Begin
                    If command type then;
                    Begin
                                Do case of command;
    DIAG:                       Perform diagnostics;
                                Report results;
    DEST:           Perform detector self test;
                                Report results;
    STAT:           Report all detector status;
    CANC:           Cancel audible alarm;
    LOG:            Print all status changes;
    MIMC:           Perform route to mimic;
                    End (If)
            End (Check Keyboard entry)
End {Main loop}
```

PD352/443
Issue 1.0
1.1.88

SOFTWARE TECHNICAL SPECIFICATION—ADDRESSABLE DETECTION SYSTEM

DO FOREVER MODULE VERSION 001

DETFIRE SUBMODULE VERSION 001

SUBMODULE DEFINITIONS

1.0 PURPOSE

The module will interrogate the detector status line returning status:

{FIRE / NO FIRE}

for the given detector.
And by a voting system involving detectors in the same zone, signals the detection of a systems status of fire.

(a) Any one detector; status = Fire, will provoke an audible alarm.
(b) Any two or more detectors; status = Fire, in same zone will provide systems status equivalent to Fire, and provoke Fire executive action and will provide an audible siren. Fire executive action is the subject of another document and not addressed here.

2.0 PSEUDO CODE DETECTION STATUS

```
Definition of Terms
Poll Detector (i) = Detector Status (i)
                  = TRUE      FIRE
                    FALSE     OK

Begin
    Poll Detector (i)
    If detector Status (i) = TRUE Then
        Set Audible Alarm
        Do for all Other Detectors in same Zone
            If Detector Status (i) = TRUE, for i = i, Then
                Set Executive Action
End If
        End Do
    End If
End (     )
```

Normally at this point in the documentation the module would be implemented in programming language.

PD352/900
Issue 1.0
1.1.88

QUALITY PLAN—
ADDRESSABLE DETECTION SYSTEM

CONTENTS

.
.
.
.
.
.
.

3.0 TEST STRATEGY

3.1 Strategy

The development testing will consist of a bottom up strategy. Printed Board Assemblies (PBAs) will initially be tested alone. This will be followed by integration tests involving groups of PBAs. Functional tests will then be carried out on the complete system. These will include I/O load tests and misuse tests. Finally an environmental test will be applied.

3.2 PBA Module Tests

Each PBA will be subject to a stand alone functional test on the programmable GENRAD XXXXX tester with simulated inputs programmed as appropriate. The PBA test specs are:

PD352/911	MOTHER BOARD
PD352/912	CPU BOARD
PD352/913	5 CCT I/O BOARD
PD352/914	COMMUNICATIONS BOARD
PD352/915	PSU

3.3 Integration Tests

The purpose of these tests is to establish that boards can communicate and function together. The integration test specs are:

PD352/921	I/O–CPU INTEGRATION
PD352/922	COMMS–CPU INTEGRATION

3.4 Functional Tests

The functional test specs are:

PD352/931	SYSTEM FUNCTIONAL TEST
PD352/932	I/O LOAD TEST
PD352/933	SYSTEM MARGINAL TESTS
PD352/934	MISUSE TESTS
PD352/935	ENVIRONMENTAL TEST

PD352/921
Issue 1.0
1.1.88

I/O–CPU INTEGRATION TEST SPECIFICATION—ADDRESSABLE DETECTION SYSTEM

1.0 OBJECTIVE

To establish that input stimuli give rise to correct outputs whilst controlled by the CPU board.

2.0 TEST HARDWARE AND SOFTWARE

2.1 I/O PBA with simulated inputs for three fire detectors and one manual call point (MCP).

2.2 I/O PBA with simulated alarm and executive outputs.

2.3 CPU PBA with application software including cause and effect logic for the simulated fire area.

2.4 Test PBA (PD352/921/001) to simulate the communications PBA responses and thus prevent communications failure errors which would otherwise shut down the system.

2.5 Communications analyser (XXXXXXXXXX) with connections to the CPU data bus.

3.0 TEST PROCEDURE

3.1 Start Up

Connect power and switch on.
Observe that no outputs are activated.

3.2 Detector Sampling

Set up and connect the communications analyser to the CPU address data bus.
Observe that the three detectors and MCP addresses are polled at no less than 1 s intervals.

3.3 Detector 1 Out of 3 Alarm

Simulate one detector input.
Observe simulated audible alarm output.

Repeat the test for No 2 detector.
Repeat the test for No 3 detector.

3.4 Manual Call Point Executive Action
Simulate the MCP input.
Observe simulated executive action output.

3.5 Detector 2 Out of 3 Executive Action
Simulate Detectors 1 and 2 input.
Observe simulated executive output.
Repeat for Detectors 1 and 3.
Repeat for Detectors 2 and 3.
Repeat for all three detectors.
Repeat for MCP and all three detectors.

TUTOR'S DISCUSSION

Requirements Specification PD352/100

1. There are a number of important items missing from this specification. The following are a few examples:

—Extendability (6.2) should be quantified.
—Maintenance scenario (e.g., frequency, type of on line repair).
—Is equipment stand alone or connected to other control systems?
—Several terms are not defined.

Attempt to list additional omissions.

2. There are many ambiguities, for example:

—Fail Safe (2.1). Spurious action (6.4) is not necessarily safe.
—Simulated (3.4) can involve a number of methods.
—What sort of auto-test?

Attempt to list additional ambiguities.

3. Specifying a PES (Programmable Electronic System) in (1.0) is a design implementation—not a requirement. There are at least two other design decisions which should not be specified at this level. Attempt to identify them.

Functional Specification PD352/200 (Issue 1.0)

1. There are a number of items missing from this specification. The following are a few examples:

—The test functions (1.1b) are not defined.
—The heat, smoke, UV requirements are not brought forward from the requirements specification.
—Mains failure recovery is not addressed.
—Fig. 1.2 shows no output ports on the processor.

Attempt to list additional omissions.

2. There are many ambiguities, for example:

—'Mimic' and 'Graphics' are not defined.
—'Self diagnosis' (1.1f) is not clear.
—(2.1) should include the word sequentially.

Attempt to list additional omissions.

3. The requirements specification calls for auto-test (RS 3.4) but the FS calls for operator initiated test.

4. The functional specification is still too early in the hierarchy to specify design details such as PES and Disc.

4. MS-DOS is not a real time operating system.

Attempt to identify other errors.

Hardware Technical Specification PD352/300

In order to keep the exercise within reasonable bounds, a hardware specification is not given. It is where the details of PES type (erroneously mentioned above) should be found.

Software Technical Specification PD352/400

1. There are a number of items missing from this document. The following are a few examples:

—No arrows in Fig. 3.
—No watchdog facility.
—No power down sequence provision.
—MIMC missing in 3.4.

Attempt to list additional omissions.

2. There are ambiguities, for example:

—There is a mixture of graphical methods. THIS IS DELIBERATE IN ORDER TO ILLUSTRATE SOME OF THE POSSIBLE DESIGN METHODOLOGIES.
—The meanings of 'various' (3.2), 'cohesive' (3.1) etc.

3. The DO FOREVER does not return to the diagnostic in 2.4.

4. Assembler is called for in (1.1) for speed whereas the functional specification (4.2) justifies it on the basis of interfaces.

Attempt to identify other errors.

DO FOREVER Module PD352/440

1. Examples of items omitted are:

   —MIMC command on page 1.
   —No arrows on Fig. 1.

2. There is no voting carried out in the module.

3. If the nth detector is faulty (Fig. 1) then the $n+1$th is never reached.

4. The seven modules in 1.0 do not map those in PD352/400.

Attempt to identify other errors.

DETFIRE Sub-module PD352/443

1. There is no export of status to the voting module in the 443 subsystem.

2. Detector status implies 'fire' or 'no fire'. Other states such as 'under maintenance' and 'failed auto-test' need to be considered.

Quality Plan PD352/900

1. Only dynamic testing is specified. Reviews and static analysis are not addressed.

2. Test strategy only maps the hardware configuration.

Attempt to identify other shortcomings.

Test Procedure PD352/921

1. The tests do not specify the expected result. Thus, there can be no objective pass/fail criteria.

2. Tests do not specify what should NOT happen.

Attempt to find other shortcomings.

PD352/200
Issue 2.0
2.1.88

FUNCTIONAL SPECIFICATION— ADDRESSABLE DETECTION SYSTEM

1.0 INTRODUCTION

1.1 The proposed system will perform the following functions:

(a) Monitor each of the fire detectors at one second intervals to interrogate their output status.
(b) Initiation of built-in test mode at system start up during normal operation and at operator request. This involves auto test of detector and output loops and incorporates a diagnostic check of the equipment.
(c) Display status of each detector on a colour graphics screen.
(d) Triggering of audible alarm in the event of fire detection.
(e) Triggering of fire sprinkler systems and outputs to a shut down system.
(f) Self diagnosis of the computer system at start up to detect faults (see 3.7).
(g) Printouts of alarms.
(h) The equipment shall be 220/240 V 50 Hz mains operated.

1.2 The hardware configuration is as shown in Fig. 1.2.

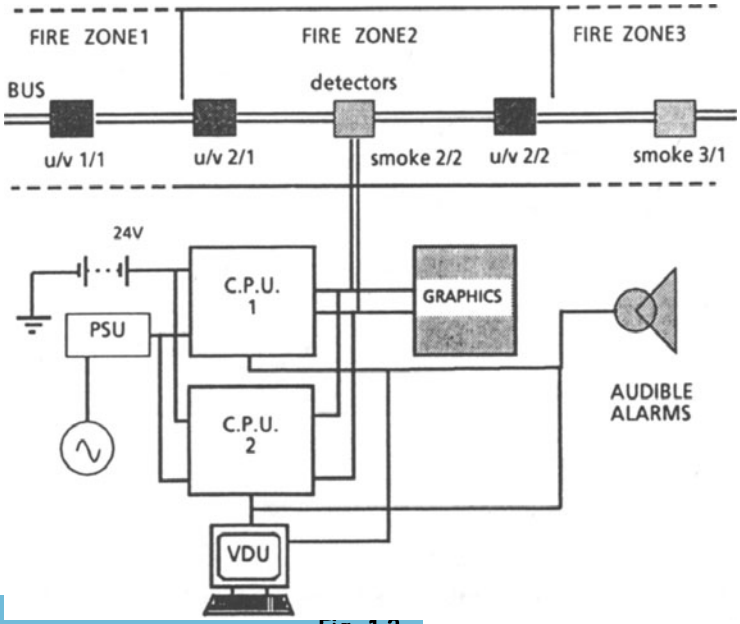


Fig. 1.2.

2.0 FACILITIES AND FUNCTIONS

2.1 The system will allow connection of up to 160 detectors each of which is separately addressable. There may be UV, smoke (hydrocarbon fires), infra red, temperature and rate of rise detectors.

2.2 A graphical display will provide a visual mimic of the status of each detector with a picture of its location. (See Fig. 3.2).

2.3 The audible alarm must be sounded when one or more detectors activates.

Alarms must differentiate between one detector or a zonal state of two or more.

2.4 A VDU will duplicate the information displayed by the mimic.

2.5 In the event of failure of the computer control. The system will revert to the use of hardwired circuitry.

2.6 The system will provide a start up mode of operation which will perform system checks and then go into normal operation. A system shut down mode will also be provided.

2.7 All events and operator commands will be logged on a permanent medium.

2.8 There shall be two alarm states

(a) Level indicating a single detector only.
(b) Multiple alarm determined by preset zonal voting arrangements, resulting in alarm and executive action.

3.0 SYSTEM OPERATION

3.1 Operator Commands

The system will be operated via the computer system keyboard. Commands may be entered, singly, at the keyboard. The set of

commands is:

DIAG —Performs auto diagnosis on the computer system.
DEST(⟨n⟩)—Initiates detector self test where n denotes the detector.
STAT(⟨n⟩)—Forces a report of all detectors' status or, if n is specified, a particular detector.
CANC —Shuts down audible alarm (protected by a key switch).
LOG —Prints all status changes since last system restart.
MIMC —VDU reverts to zone mimic (see 3.2).
TIME —Time and date change input.

No other data input will be accepted by the system.

Use of these commands will require the entry of a password known only to the operator. Password validation will be allowed only three times within a minute and then an enforced 20 minute delay will begin. A separate password will be required to enable the use of the CANC command.

3.2 VDU Format

The detectors will be polled sequentially and the VDU will normally display information in the form of Fig. 3.2.

Activated detectors will be indicated by a change of colour and flashing at 1 s intervals. Upon the command STAT this will be replaced

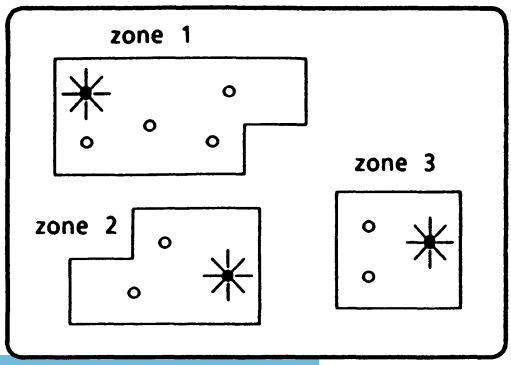


Fig. 3.2.

with a list of the states of the first 20 detectors in the form:

Time	*Detector*	*Status*
xx:xx:xx	No: Type:	OK
xx:xx:xx	No: Type:	ALARM

Subsequent commands will display blocks of 20 detectors.

3.3 MIMIC

The graphics panel will consist of groups of LED displays representing the fire zones and their individual detectors.

3.4 Log File Layout

The log file, will contain:

(i) all commands
(ii) response to commands
(iii) detector response
(iv) result of system start up.

in the form:

⟨TIME, COMMAND, RESPONSE⟩ ⟨TIME, STATUS CHANGE⟩ .

3.5 Detection Logic

There are two levels of detection logic, corresponding with states in 2.8(a) and 2.8(b).

(a) A non-executive level which actuates the audible alarm and updates the VDU and mimic. This is normally triggered by the activation of a single detector.
However the detection level shall be programmable by the user according to his requirements.
(b) An executive level which is normally used to activate outputs of the fire suppression systems (e.g. halon, fire pumps). A typical arrangement would be the voting of two inputs out of n.

3.6 Executive Outputs

These are triggered in response to the appropriate detector input or manual input as has been pre-programmed during installation by the user.

The executive outputs will consist of volt free loops capable of switching 24 V solenoids and of carrying 50 mA. Non-executive outputs comprise the VDU and graphics. Both make and break conditions shall be available as the output state.

3.7 Diagnostic Details

The auto test facility will perform the following checks.

(a) Simulated detector inputs whilst outputs disenabled.
(b) Simulated output signals whilst output function is disenabled.
(c) Cause and effect logic functional checks.
(d) Memory read/write checks.

4.0 DESIGN, DEVELOPMENT

4.1 Hardware

(a) A micro-computer will be chosen for this system implementation using an appropriate real time operating system. Hardware peripherals will be as specified in PD352/300.
(b) Suitable I/O units will be used to interface each detector with the binary bus to enable individual detector addressing.
(c) Eurorack equipment practice will be used.
(d) The PSU will consist of a battery float arrangement with capacity to provide a 24 h backup in the event of mains failure.

4.2 Software

(a) A high level language will be used for the coding of all software except where it is necessary to interface to hardware, where assembly code will be used.
(b) A validated compiler will be used.
(c) A standard real time operating system will be employed which has disc file facilities embedded. No special tailoring of the package is expected.

5.0 OPERATION AND MAINTENANCE

5.1 The system will be attended by an operator who will be conversant with the output facilities. He or she will be capable of using the input commands.

5.2 Printed board changes will involve re-start up of the system.

5.3 Detector changes will be possible without disenabling the system. Removal of a detector shall not result in any alarm state but must be indicated at the operator station.

Checklist Application Chart

This chart indicates the areas of application for each of the checklists in the book. Each checklist number indicates the chapter in which it can be found.

Activity	*Management (4.1)*	*Documentation (4.2, 4.3)*	*Design features (10.1)*	*Programming standards (4.5)*	*Design review (7.1)*	*Code inspection (7.2)*	*Test and integration (8.1)*	*Changes (4.4)*
Predesign	*							
Specification	*	*						
Overall design	*	*						*
Detail design	*	*	*					*
Coding	*	*	*	*	*	*		*
Test and integration	*	*	*		*		*	*
Subcontracted software	*	*	*	*	*	*	*	*

Glossary of Terms

The following is an explanation of the main terms used in software and systems engineering. There are many glossaries in the various documents described in Chapter 5 which address this subject and it could be argued that there is no need for yet another. However, this list attempts to combine the available glossaries in the authors' words and to offer a comprehensive coverage of the terms. It is split into groups of words, and the words in each group are in alphabetical order. The final section of the glossary gives the meanings of some common abbreviations. The groups are:

(A) Terms connected with failure.
(B) Terms connected with software.
(C) Terms connected with software systems and their hardware.
(D) Terms connected with procedures, management and documents.
(E) Terms connected with test.
(F) Common abbreviations.

(A) TERMS CONNECTED WITH FAILURE

Availability
What is usually referred to as availability is the steady availability. It is the proportion of time that the system is not in a failed state.

Bit Error Rate
The rate of incidence of random incorrect binary bits. This usually refers to the effect of corruption in a communication channel.

Bug
A slang expression for a software fault (see *Fault*).

Common-Cause Failure (Common Mode Failure)
Both terms refer to the coincident failure of two or more supposedly independent terms as the result of a single cause. This is especially relevant in systems incorporating redundancy where one event causes the coincident failure of two more normally independent channels.

Data Corruption
The introduction of an error by reason of some alteration of the software already resident in the system. This could arise from electrical, magnetic or ionising interference or from incorrect processing of a portion of the software.

Error
An error has occurred when the software in the system reaches an incorrect state—a bit or bits in the program or in data take incorrect values. This can arise as a result of outside interference or because of faults in the program. An error is caused by a *fault* and may propagate to become a *failure*. *Error recovery software* may thus prevent an error propagating.

Error Recovery Software
Sections of a program, involving redundant (*Parity*) bits or checksums, which can recognise and correct some bit errors.

Failure
Termination of the ability of an item (or system) to perform its specified task. In the case of software the presence of an *Error* is required in order for it to propagate to become a failure.

Failure Rate
The number of *Failures,* per unit time, of an item. Since software failures are path- rather than time-related this is not a particularly useful parameter except for hardware failures.

Fault
Faults may occur in hardware or in software. Whereas hardware faults are time-related, software faults are conditions which may lead to bit

Errors in the system. These faults may be ambiguities or omissions in the logic structure of the program or environmental/hardware conditions which can cause software corruption.

The presence of a software fault does not necessarily guarantee that an *Error* and *Failure* will ensue.

Fault Tolerance
Hardware and software features (discussed in Chapter 10) which render a software system less likely to *Fail* as a result of software *Faults*.

Graceful Degradation
A design feature whereby a system continues to operate, albeit at a reduced efficiency or with fewer functions available, in the presence of *Failures*.

Integrity
The ability of a system to perform its functions correctly when required. The term 'integrity' is usually associated with safety systems.

Intrinsic Safety
A degree of *Integrity* designed into a system in order to meet defined safety criteria.

Maintainability
The probability that a failed item (or system) will be restored to operational effectiveness within a specified time and when the repair action is carried out according to prescribed procedures. The parameter is often expressed by reference to a Mean Down Time (MDT) or Mean Time To Repair (MTTR).

Reliability
The probability that an item will perform a required function, under stated conditions, for a stated period of time. System reliability is often described by reference to parameters such as failure rate and mean time between failures. Since software reliability cannot easily be quantified, these terms are better applied to hardware alone.

(B) TERMS CONNECTED WITH SOFTWARE

Algorithm
A set of logical rules for the solution of a defined problem.

Alpha Numeric
A code or description consisting of both alphabetic and/or numerical characters.

Application Language
A problem-oriented language whose statements closely resemble the jargon or terminology of the particular problem type.

Application Software
The software written for a computer for the purpose of applying the computer functions to solve a particular problem. This is distinct from the resident operating software which is part of the computer system.

Assembler
A program for converting instructions, written in mnemonics, into binary machine code suitable to operate a computer or other programmable device.

Assembly Language
A low level language where the instructions are expressed as mnemonics and where each instruction corresponds to a simple computer function.

Basic Coded Unit (BCU)
Often referred to as a *Module,* a self-contained manageable element of a program which fulfils a specific simple function and is capable of being compiled and run. The BCU should be at a sufficiently simple level to permit its function to be described by a single sentence.

Baud
The unit of signal speed where 1 baud corresponds to 1 information bit per second. This is equivalent to the bit speed in a binary system where each bit can take either of two values (0 or 1). In multilevel signalling the baud rate will be higher than the bit rate.

Binary Coded Decimal
A binary notation whereby each decimal digit is represented by a four-bit binary number (e.g. 1001 0011 represents 93).

Bit
A single binary digit taking the value 0 or 1.

Code
Any set of characters representing program instructions or data in any computer language.

Code Template
A standard piece of proven code, for a particular function, which is used repetitively.

Compiler
A program which, in addition to being an *Assembler* generates more than one instruction for each statement, thereby permitting the use of a *High level language*. It consists of:

Lexical analyser (recognises the words);
Syntax analyser (recognises logical commands);
Semantic analyser (sorts out the meaning);
Code generator (generates the 0s and 1s).

Data Base
Any set of numerical or alphabetical data.

Data-flow Diagram
The next stage after the requirements specification involves data-flow analysis. The data-flow diagram is a graphic (usually flow) diagram showing data sources and the flow of data within a program.

Decision Table
The representation of a number of decision options showing the various outcomes. This is usually shown in matrix or tabular form.

Default Value
The value of a variable which will be assumed when no specific input is given.

Diversity
One form of diversity is said to exist when the redundancy in a system is not identical. In software terms this would apply if redundant channels had been separately programmed. The disadvantages of this are discussed in Section 10.1.

Another form of diversity exists when an alternative means is

available for a particular function to be performed despite the failure of the main function.

Dump
To transfer the contents of a store or memory to an external medium.

Global Data
A major named group of data which serves as a common base between various tasks in a program. It will be accessible to all modules.

High Level Language
A means of writing program instructions using symbols, each of which represent several program steps. High level languages do not reflect any particular machine structure and can be compiled to any computer for which a *Compiler* exists for that language.

Initialisation
The process whereby a system, usually at switch-on, is put into the correct software state to commence operation.

Interpreter
A type of *Compiler* which enables one instruction at a time to be checked and converted into machine code. This permits step-by-step programming at the VDU. Syntax errors are then announced as they occur.

Language
The convention of words, numerals, punctuation and grammar which enables programs to be written in a form comprehensive to a computer.

Listing
A printed list of the coded program instructions. A listing is usually of the *Source code.*

Machine Code
See *Object code.*

Metrics
Parameters for describing the structure, size and complexity of a program. Attempts are made to relate these metrics, by regression techniques, to software quality (see Chapter 12).

Mnemonic
Characters used to represent a particular instruction. Low level languages use mnemonics in their instructions.

Module
The basic testable unit of software (see also *Basic coded unit*).

Multi-tasking
The ability of a computer system to carry out more than one task, apparently simultaneously.

Object Code
The final machine code, probably the output from a *Compiler,* which the computer can understand. Programming directly in machine code is now extremely rare.

Operating System
The machine resident software which enables a computer to function. Without it, applications programs could not be loaded or run.

Parity
An additional bit or bits which are added to a segment of data or instruction to enable subsequent checking for error. The value of the parity bits is generated from the values of the bits which it is 'protecting'.

Pseudo Code
High level 'English' language statements which provide an intermediate level between the module specification and the computer language.

Procedure
An identifiable portion of a program which carries out a specific function. A procedure may be a module.

Program
A set of coded instructions which enable a computer to function. A program may consist of many modules and be written in assembly or high level language. Note the spelling 'program', whereas 'programme' is used to describe a schedule of tasks.

Routine
A frequently used piece of code which is resident inside a program.

Software
The term software covers all the instructions and data which are used to operate a computer or programmable system. It includes the operating system, compiler, applications software and test programs. The definition also embraces the documents (specifications, charts, listings, diagrams and manuals) which make up the system.

Source Code
The listing of a program in the language in which it was written.

Structured Programming
Well-defined and standardised programming techniques which result in greater visibility of the program and less complexity.

Syntax
Rules which govern the use and order of instructions in a language.

Task
A sequence of instructions which together carry out a specific function.

Translator
A program which transforms instructions from one language into another.

(C) TERMS CONNECTED WITH SOFTWARE SYSTEMS AND THEIR HARDWARE

Analogue/Digital Converter
A device which converts an analogue electrical value (voltage or current) into an equivalent binary coded form.

Applications Hardware
A special-purpose unit designed, as a peripheral to the computer, to carry out some specified function.

Asynchronous
A timing arrangement whereby each event is started as a result of the completion of preceding events rather than by some defined trigger.

Bus
A digital signal path (or highway).

Configuration
A complete description, at a point in time, of a product and the interrelationship of its parts. This includes the hardware, software and firmware and is a description of its characteristics. Both physical parts and performance are described.

Configuration Baseline
A specific reference point, in time, whereby the *Configuration* is described. All changes are then referred to that baseline.

Configuration Item
A collection of hardware and software which forms a part of the *Configuration.*

Digital/Analogue Converter
The opposite of an *Analogue/digital converter.*

Disc
See *Magnetic disc.*

Diversity
An attempt at fault tolerance whereby redundant units are separately designed and coded in order to reduce common mode software failures. See also Section B of this glossary.

Ergonomics
The study of man/machine interfaces in order to minimise human errors due to mental or physical fatigue.

Firmware
Any software which is resident on physical media (e.g. hardwired circuitry, EPROM, PROM, ROM, disc).

Interrupt
The suspension of processing due to a real time event. The program is so arranged that processing continues after the interrupt has been dealt with.

LSI—Large Scale Integration
The technology whereby very large numbers of circuit functions are provided on a single component. A programmable system may consist of one or more LSI devices.

Magnetic Disc
A rotating circular lamina with a magnetic coating which can record binary bits by means of magnetic storage.

Magnetic Tape
A tape with a magnetic coating which can record binary bits by means of magnetic storage.

Media
A collective term for the devices on which software programs are stored (e.g. PROM, EPROM, ROM, disc, tape).

Memory
Storage (usually binary) in a computer system.

Microprocessor
The central processing unit of a computer (usually contained on a single device) consisting of memory registers, an arithmetic and logic unit, program and instruction registers and some interface to the external world.

Modem
An acronym for MOdulator/DEModulator. It converts binary signals into frequency form for transmission over telecommunications channels and vice versa.

'N' Version Programming
See *Diversity*.

Peripheral
Any piece of equipment, apart from the computer achitecture of logic and memory, which provides input or output facilities.

PES—Programmable Electronic System
Any piece of equipment containing one or more components providing a computer architecture such that the functions are provided by a

program of logical instructions. The term is increasingly used in the context of the control, monitoring and protection of plant.

PLC—Programmable Logic Controller
A computer system with real time inputs and outputs. It is provided with a special-purpose problem-oriented language and is increasingly used in the control of plant and processes.

Programmable Device
Any piece of equipment containing one or more components providing a computer architecture with memory facilities.

Real Time System
A computer system (including PES, PLC) which operates in response to on-line inputs and stimuli.

Redundancy
The provision of more than one piece of equipment for achieving a particular function.

Active redundancy. All items operating prior to failure of one or more.
Standby redundancy. Replicated items do not operate until needed.

Safety Critical Software
Software whereby one or more failure modes can have hazardous consequences.

Safety-related System
One or more systems upon which the safety of a plant or process depends.

Synchronous
An arrangement whereby logical steps or processes in a program are only initiated as a result of a reference clock rather than as a result of preceding events being completed.

Terotechnology
An integrated approach to the overall optimisation of life-cycle costs and resources.

Watchdog
A part of the system (may be hardware or software) which monitors the state of the processor and signals when tasks are not completed within a prescribed time.

(D) TERMS CONNECTED WITH PROCEDURES, MANAGEMENT AND DOCUMENTS

Acceptance
The act whereby a user states to a supplier that the product is now satisfactory. This acceptance may be partial in that agreed outstanding modifications or rectifications are to be implemented.

Archiving
Storing programs, data and documents externally to the computer system either for security (in the event of corruption) or for later resumption of design.

Bureau
The facility where computer processing facilities and software packages are offered for hire.

Code Inspection
The design review activity wherein members of the design team, other than the programmer/designer, examine a module of code against standards in order to reveal faults.

Code Walkthrough
The design review activity wherein the programmer/designer participates in leading other members of the team through a module of code. They then attempt to discover faults by questioning.

Configuration Control
The discipline that ensures that all changes and modifications to the *Configuration baseline* (see Part C of this glossary) are controlled and recorded and that, as a result, documents and firmware conform to issue status.

Design Review
A formal comparison of the software and hardware with the specifications in order to establish conformance. Code inspections and walkthroughs are part of this process, which may be carried out at many stages in the design.

Feasibility Study
A preliminary study of some proposed solution to requirements in order to establish viability apropos of cost, schedule, function, reliability, etc.

Flowchart
A graphical representation of the logic and data flow which satisfies a specification. Normally coding would follow from the flowchart. Modern structured languages have much reduced the need for this technique.

Library
The formal documentation and software storage within an organisation.

Life-cycle
The complete series of activities from requirements specification, through design and test, to field use and modification.

Quality Assurance
The total range of activities which attempt to ensure that a finished product, both in design and manufacturing respects, meets the requirement.

Quality System
A formal set of procedures, methods and standards whereby the management of design and manufacture seeks to operate *Quality* (see Section 5.3).

Release
The issue of software after it has been formally validated by design review.

Requirement
A statement of the problem or function required by the user.

Specification
A document, at one of the levels in the hierarchy of design, which describes either requirements or solutions (see Chapters 4 and 5).

(E) TERMS CONNECTED WITH TEST

Acceptance Testing
Testing carried out specifically to demonstrate to the user that the requirements of the functional specification have been met.

Development System
Usually a VDU, keyboard and computer equipped with appropriate support software in order to test and debug programs.

Diagnostic Software
A program which assists in locating and explaining the causes of faults in software.

Emulation
A type of simulation whereby the simulator responds to all possible inputs as would the real item and generates all the corresponding outputs.

Endurance Test
Extreme testing whereby the software is subjected to abnormal and illegal inputs and conditions and saturation levels which stress the program capabilities in terms of data volume and rate, processing time, response time, utilisation of memory, etc.

Integration
The step-by-step process of testing where each module is tested alone and then in conjunction with others until the system is built up.

Load Test
See *Endurance test.*

Simulation
The representation, for test purposes, of a unit or system by hardware or by software in order to provide some or all inputs or responses.

Soak Test
A test where a system is submitted to prolonged periods of operation, often at elevated temperature and humidity.

Static Analyser
A suite of software which analyses code (see Section 8.7).

Stress Test
See *Endurance test.*

Test Driver
See *Test harness.*

Test Harness
Specially designed hardware and software used to replace parts of the system not yet developed and in order to permit testing of available modules to proceed.

Test Software
Any program used as a test aid.

Validation
The process of ensuring that the results of the whole project meet the original requirements.

Validator
A suite of programs which examines a computer program and determines if certain types of fault exist (see Section 8.7; see also *Static analyser*).

Verification
The process of ensuring that the result of a particular phase meets the requirements of the previous phase.

(F) COMMON ABBREVIATIONS

A/D	Analogue to Digital
APSE	Ada Project Support Environment
ASCII	American Standard Code for Information Interchange
AT	Arbejdstilsynet, Denmark

ATE	Automatic Test Equipment
BCD	Binary Coded Decimal
BCU	Basic Coded Unit
BIA	Berufsgenossenschaftliches Institut für Arbeitssicherheit, Germany
BS	British Standard
CAD	Computer-Aided Design
CCF	Common-Cause Failure
CCITT	Committé Consultative International pour Telegraph et Telecommunications
CMF	Common Mode Failure
CPU	Central Processing Unit
D/A	Digital to Analogue
EBDIC	Electronic Binary Decimal Information Code
EC	ElektronikCentralen, Denmark
EMA	Extended Memory Addressing
emi	Electromagnetic Interference
EPROM	Erasable Programmable Read Only Memory
HSE	Health and Safety Executive
INRS	Institut National de Recherche et de Sécurité, France
I/O	Input/Output
IPA	Fraunhöfer Institut für Produktionstechnik und Automatisierung, Germany
IPSE	Integrated Program Support Environment
ISO	International Standards Organisation
LCD	Liquid Crystal Diode
LED	Light-Emitting Diode
LSI	Large-Scale Integration
NCSR	National Centre of Systems Reliability
PERT	Programme Evaluation and Review Technique
PES	Programmable Electronic System
PLC	Programmable Logic Controller
PPD	Predefined Process Diagram
PROM	Programmable Read Only Memory
QC	Quality Control
RAM	Random Access Memory
RISC	Reduced Instruction Set Computers
ROM	Read Only Memory
SBD	Schematic Block Diagram
VDU	Visual Display Unit

Bibliography

1 BRITISH STANDARDS

British Standards Institute,
Sales Department,
Linford Wood,
Milton Keynes,
MK14 6LE.

BS 3527 Glossary of Terms used in Data Processing.
BS 4058 Data Processing Flow Chart Symbols, Rules and Conventions.
BS 4778 Glossary of Terms used in Quality Assurance (Including Reliability and Maintainability Terms).
BS 5345 Code of Practice for the Selection, Installation and Maintenance of Electrical Apparatus for Use in Potentially Explosive Atmospheres (Other than Mining Applications or Explosive Processing and Manufacture).
BS 5476 Specification for Program Network Charts.
BS 5515 Code of practice for the Documentation of Computer Based Systems.
BS 5783 Code of Practice for the Handling of Electrostatic Sensitive Devices.
BS 5887 Code of Practice for the Testing of Computer Based Systems.
BS 5905 Specification for Computer Programming Language—CORAL 66.
BS 6238 Code of Practice for Performance Monitoring of Computer Based Systems.
BS 6488 Code of Practice for Configuration Management of Computer Based Systems.

2 UK DEFENCE STANDARDS

Defence Standards are obtained from The Directorate of Standardisation, Ministry of Defence, Montrose House, 187 George Street, Glasgow G1 1YU, UK.

Def-Stan 00-13	Achievement of Testability in Electronic and Allied Equipment. Part 1 *Guide.* Part 2 *Production and Acceptance Testing.*
Def-Stan 00-14	Guide to the Defence Industry in the use of ATLAS.
Def-Stan 00-16/1	Guide to the Achievement of Quality in Software.
Def-Stan 00-17	Modular Approach to Software Construction Operation and Test (MASCOT).
Def-Stan 00–18	Avionic Data Transmission Interface System (five parts).
Def-Stan 00-19	The ASWE Serial Highway.
Def-Stan 00-21	M700 Computers.
Def-Stan 05-47	Computer On-Line Real-Time Application Language—CORAL 66—Specification for Compilers.
Def-Stan 05-57	Configuration Management—Requirements for Defence Equipment.
Def-Stan 05-67	Guide to Quality Assurance in Design (Section 12 of this guide deals with computer software and the quality assurance thereof).
IECCA	*A Guide to the Management of Software-Based Systems for Defence,* 3rd Edition (Available from the address given in Section 5.4.16).
EQD	*Guide for Software Quality Assurance.*
JSP 343	MOD Standard for Automatic Data Processing (this Standard is to be published in five volumes and to the best of our knowledge only Volume 1—*Documentation Standard Manual* is currently available).
JSP 188	*Specification and Requirements for the Documentation of Software in Operational Real-Time Computer Systems.*

3 US STANDARDS

DOD STD 2167	Defense System for Software Development.
MIL-S-52779A	US Military Specification—*Software Quality Assurance Program Requirements.*
MIL-HDBK-344	US Military Handbook—*Evaluation of a Contractor's Software Quality Assurance Program.*
MIL-STD-1750A	Military Standard: 16 Bit Computer Instruction Set Architecture.
Publication 78-53	*Standard Practices for the Implementation of Computer Software,* Jet Propulsion Laboratory, Pasadena, CA 91103, USA.
ADatP-2 (B)	*NATO Glossary of Automatic Data Processing (ADP) Terms and Definitions.*
IEEE 729	*Standard Glossary of Software Engineering Terminology.*
IEEE 830	*Guide to Software Requirements Specifications.*
IEEE 730 1984	Software Quality Assurance Plans.

IEEE 754 1985	Binary Floating-Point Arithmetic.
IEEE 828 1983	Software Configuration Management Plans.
IEEE 829 1983	Software Test Documentation.
IEEE 854 1987	Radix & Format independent floating-point arithmetic.
IEEE 983 1985	Software Quality Assurance Plan.
IEEE 990 1987	Ada as a Program Design language.
IEEE 1002 1987	Taxonomy for Software Engineering Standards.
IEEE 1003.1 1988	Std Portable Operating System Interface for Computer Environment.
IEEE 1008 1987	Software Unit Testing.
IEEE 1012 1986	Software Verification & Validation Plans.
IEEE 1016 1987	Software Design Descriptions.
IEEE P982	Draft Standard, Measures for Reliable Software.
IEEE 416	*Definition of Abbreviated Test Language for All Systems—ATLAS* (formerly ARINC 416).

4 OTHER STANDARDS AND GUIDELINES

Establishing a Quality Assurance Function for Software, Electronic Engineering Association Guide.

Software Configuration Management, Electronic Engineering Association Guide.

Quality Assurance of Software, Electronic Engineering Association Guide.

Code of Practice for the Avoidance of Electrical Interference in Electronic Instrumentation and Systems, J. H. Bull, ERA 75-31, available from ERA Technology Ltd, Cleeve Road, Leatherhead, Surrey KT22 7SA, UK.

Guidelines for the Documentation of Software in Industrial Computer Systems, The Institution of Electrical Engineers, Savoy Place, London, WC2R 0BL, UK, 1985.

Guidance on the Safe Use of Programmable Electronic Systems in Safety Related Applications, Health and Safety Executive, UK, June 1987.

Volume 1: An Introductory Guide.
Volume 2: General Technical Guidelines.

Guide to User Needs for Technical Documentation (*Engineering*), Engineering Equipment and Materials Users' Association Handbook No. 36, 1982.

Guide to the Engineering of Microprocessor Based Systems for Instrumentation and Control, Engineering Equipment and Materials Users' Association Handbook No. 38, 1981.

Contracts for the Acquisition and Utilisation of Computer Software for Industrial Control and Monitoring Systems, British Electrical and Allied Manufacturers' Association, Legal Department Publication No. 240, 1982.

Electromagnetic Compatibility for Industrial-Process Measurement and Control Equipment.

Part 1: General introduction. IEC Publication 801-1.
Part 2: Electrostatic discharge requirements. IEC. Publication 801-2.
Part 3: Radiated electro-magnetic-field requirements. IEC Publication 801-3.

5 BOOKS

Characteristics of Software Quality, Boehm *et al.,* North Holland, Amsterdam, 1978.

Documentation of Software Products, J. D. Lomax, National Computing Centre Publications, Manchester, 1977.

Elements of Programming Style, B. W. Kernighan and P. J. Plaugez, McGraw Hill, New York, 1978.

Penguin Dictionary of Computers, Anthony Chandor, Penguin, London, 1977.

Practical Reliability Engineering, 2nd edn, P. D. T. O'Connor, Wiley, Chichester, 1981.

Reliability and Maintainability in Perspective, 3rd edn, David J. Smith, Macmillan, London, 1988.

Safety and Reliability of Programmable Electronic Systems, B. K. Daniels, Elsevier Applied Science Publishers, London, 1986.

Software metrics, T. Gilb, Brookfield, London, 1982.

Software Requirements Specification and Testing, T. Anderson, Blackwell Scientific Publications, London, 1985.

Structured programming, Dahl, Dijkstra and Hoare, Academic Press, New York.

Index