Abstract:

A total of 64 references to papers, books and conference proceedings on the subject of software reliability have been selected. Each of these references is provided with an annotation consisting of a paragraph of commentary.

An earlier version of this bibliography appears in the INFOTECH State of the Art Report on Reliable Software (1977), for which it was originally prepared.

Reliable Software:
A Selective Annotated Bibliography

By
T. Anderson and S.K. Shrivastava

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 111
September, 1977

© 1977 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
ANDERSON, Thomas

Reliable Software: a selective annotated bibliography.
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1977.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 111).

SHRIVASTAVA, Santosh Kumar
UNIVERSITY OF NEWCASTLE UPON TYNE, Computing Laboratory.

Suggested classmarks (primary classmark underlined)
Library of Congress:
Dewey (17th):
U.D.C.: 001.6425016
681.322.06(016)

Suggested keywords

Abstract

A total of 64 references to papers, books and conference proceedings on the subject of software reliability have been selected. Each of these references is provided with an annotation consisting of a paragraph of commentary.

About the Author

Dr. Anderson is a Senior Research Associate at the Computing Laboratory of the University of Newcastle upon Tyne.

Dr. Shrivastava has been a Research Associate in the Computing Laboratory since August 1975. Prior to that he was with the Plessey Co. Ltd.
Introduction

Any bibliography on the topic of 'reliable software' has a rather wide subject area to cover since the majority of papers which are concerned with the problems of software are intended to alleviate, in some measure, those problems; as such they can usually be regarded as making a contribution to the reliability of software. In the broadest sense software reliability is measured by the extent to which the user of the software is satisfied with its behaviour. Largely because of the wide range of papers which address software issues and fall within this broad view of the subject area, this bibliography can be regarded as selective rather than critical. In selecting items, preference has usually been given to papers that are introductory rather than advanced, and to those providing an up-to-date survey of some relevant area.

The bibliography is divided into six sections. After the first section, which contains references to relevant conference proceedings, the remaining sections relate to various stages in the process of software production. Section 2 is concerned with papers determining what demands a user will impose on a system and, correspondingly, what software will be needed to respond to those demands. Section 3 covers some of the methodologies which, by virtue of the constraints they impose, can assist in the actual design and coding of the software. Section 4 refers to work on the certification or validation of software, which has the aim of confirming that the programs will, when executed, behave according to their specifications. Section 5 provides references to papers presenting techniques which are intended to ensure that acceptable behaviour is maintained, even in the presence of residual faults in the system. Papers included in the last section describe methods for estimating and measuring the reliability of software, based on probabilistic models utilising error data collected during system development.

The following reference is to a book which addresses the whole area of software reliability:


A basic conceptual framework is established in the first of four parts of this book. The second part considers how design techniques can enhance software reliability. Methodologies which aim to preclude the presence of faults in software are given prominence, but it is recognised that this aim is impossible to achieve in practice. Some consideration is therefore given to error detection and correction techniques. Part 3 provides a lengthy and thorough discussion of how software should be tested, and includes a chapter on debugging. The final part covers a number of additional topics of relevance to software reliability, namely: management techniques, programming languages, computer architecture, correctness proving, reliability modelling, and software support systems. As a comprehensive treatment of the many issues relating to the reliability of software this book can certainly be recommended.
1 Multiple Sources

Most of the entries in this section refer to the proceedings of recent conferences on topics relevant to reliable software. Each entry constitutes a source of a number (in some cases, a large number) of important papers in this area.


The reports on the two NATO Software Engineering conferences (held at Garmisch in 1968 and Rome in 1969 respectively) have been reprinted in this single volume. It includes transcripts of discussion among the participants and also working papers presented at the conferences. The first conference brought together theoretically oriented research workers and practical software producers, while the second eventually focused on the communication gap between them. Both reports can be read both for entertainment and enlightenment, the discussion sessions in particular containing many worthwhile observations.

(1.2) Rustin R. (ed.) Debugging Techniques in Large Systems. Prentice Hall, 1970

These books are based on the proceedings of conferences held at New York and Chapel Hill respectively. Together, they provide a comprehensive selection of papers in the areas of debugging and testing. The first volume includes a paper by Mills advocating the use of only if and while for control flow (placing blind reliance on Jacopini's well known result), a description of King's program verifier and an enjoyable overview from Schwartz. The second volume provides 23 papers, a nice quotation of Edward (sic) Dijkstra (p. 223) and a bibliography of about 375 items.


Most of the papers in this series of annual symposia are concerned with hardware reliability, and as such do not necessarily yield any immediate insight into the problems of software. However, they do provide a rich source of expertise in a related area. A few papers on software reliability have appeared, particularly in the 1975 proceedings (FTC-5).


Sessions at this conference covered reliability, testing, reliability modeling (over a third of the papers), management techniques and programming methodology. Papers by MacWilliams (real-time systems), Rowe et al. (distributed processing), Shooman
(reliability estimation) and Kernighan and Flaugher (programming style) have particular merit. The unrecorded panel sessions appear, on the basis of their titles, to have been the highlights of the conference.

(1.5) International Conference on Reliable Software, Los Angeles, April 1975. (SIGPLAN Notices 10, 6.)

The sequel to the above symposium was this particularly successful conference with many papers of a very high standard. Most aspects of software reliability are covered by this proceedings, which should definitely be consulted by anyone with an interest in the subject. Avoid those papers which appear to equate increasing reliability with adding on extra complexity.


Of these two conferences (held at Washington D.C. and San Francisco respectively), the second in particular turned out to be a major conference with over 100 presented papers, many of which are directly relevant to considerations of software reliability. Of particular interest are the sessions on specifying user requirements. Avoid those papers which appear to equate engineering with the application of a sledge-hammer. A selection of the papers has now appeared in IEEE Transactions on Software Engineering, and these are not included in the conference proceedings.

I Software Validation             Computing Surveys 8,3
II Fault-tolerant Software       Computing Surveys 8,4

The five papers included in these two special issues give interesting but rather narrow views on particular topics in the area of reliable software. Fosdick and Osterweil show how anomalous data flow behaviour can be detected, while Bantler and King describe one way of proving partial program correctness. The papers by Denning and by Linden are largely concerned with protection mechanisms in operating systems. Hecht describes applications of software fault-tolerance to real time systems in the aerospace industry.

(1.8) ACM Conference on Language Design for Reliable Software, Raleigh, March 1977. (SIGPLAN Notices 12, 3.)

A smaller, specialist conference as its name indicates. Specific languages considered in papers presented are Gypsy, Mesa, Euclid, Pascal, Alphard, CLU and PL/I. Other issues treated include protection, exception handling, type mechanisms and process interaction. Some of the papers are excellent. A selection of the papers will appear in the Communications of the ACM, and these are not included in the conference proceedings.
2 Requirements Definition and Engineering

The complexity of large systems is an inherent fact of life with which one must cope. Almost always, the users of a proposed system have imprecise notions of why and what they want out of the system. Requirements definition is a methodology which addresses this problem. The end product obtained by using this methodology is intended to be a precise functional specification of the entire system. Requirements engineering is the methodology that translates this specification into a set of precise data processing system requirements. The first five papers of this section are mainly concerned with the former methodology while the next three papers cover the latter. The last two papers describe specification techniques which are appropriate for the modules of a system.


In this paper Boehm states "Often, the difference between success and failure on a large software project lies in the consistency and completeness with which the system requirements have been specified and translated into design specifications. Yet, relatively little has been done to develop formal methods for requirements analysis." A case study is described where system requirements were first determined by discussions with users. A set of questions were used to draw out implied operational requirements. A formalism which uses a requirements/properties matrix is described to help check the consistency and completeness of the requirements.


An information system is defined to be any system used to provide information (including its processing) for whatever use that can be made of it. No assumption is made about the use of computers. It is therefore important to first define the information needs and then consider the problems of representing it (possibly on a computer). The author is well known as a developer of information systems theory which attempts to define user needs for data processing from the users' viewpoint rather than from that of a computer program. The author stresses the importance of separating the 'infological' task of defining information needed by users from the 'datalogical' task of representing this information in a way that makes efficient use of current technology.


All too often, design and implementation begin before the real needs and system functions are fully known. Requirements definition includes, but is not limited to, the problem analysis that yields a
functional specification - it is a careful assessment of the needs that a system is to fulfill. The authors describe their approach to requirements definition using the SADT (Structured Analysis and Design Technique) approach they have developed. SADT provides methods for (i) thinking about large complex problems; (ii) communication of ideas in clear precise notations; (iii) documentation of current results; and (iv) planning, managing and assessing progress of team efforts. The authors state that the SADT approach has been successfully applied to "a wide range of planning, analysis and design problems involving men, machines, software, hardware, databases, communications, procedures and finances".


The paper is concerned with Information Processing Systems which are built to aid the management of an organisation. The paper describes techniques by which the needs of complex organisations can be communicated. The distinction between stating information needs and developing processing procedures that are to be used to satisfy them is often blurred when general purpose programming languages, such as COBOL, are used for documenting information needs. The author presents a case for having a requirements specification language and states its desirable properties. A critical survey of currently available languages is presented. A brief description of the author's own language, FSL (Problem Statement Language), is given.


A computer-aided approach to requirements definition developed at the University of Michigan is presented. The paper describes a computer-aided documentation system for maintaining a data base containing all the basic data about the system. The language PSL is used to describe the proposed system. PSL is intended to be used in an interactive environment. The authors have developed a model of information systems, on which PSL is based. A software package (PSA - Problem Statement Analyzer) is used to record PSL information in the data base. The PSL/PSA approach gives the capability of describing information systems (manual or computerised, existing or proposed) and producing extensive hard copy documentation for use by analysts.

HIPO is an acronym for Hierarchy plus Input, Process and Output; it is a technique for stating user requirements for a data processing system. This paper presents a straightforward introduction to the technique; more information can be obtained from IBM document GC20-1851. The HIPO design process is an iterative top-down activity which produces a hierarchic functional breakdown of the system. The top box of the hierarchy describes the entire piece of software to be developed, in terms of a single function. Each level below is a more detailed functional description (this process is known as functional decomposition). The author illustrates the use of HIPO with the help of a simple 'inventory control' example.

Many advantages of HIPO are also stated.


A methodology for generating software requirements for a large real-time unmanned weapons system is described. First, the given data processing system requirements are formally stated in Requirements Statement Language, ESL (a Petri net based language). A Requirements Engineering and Validation System, REVS, is then used to check the completeness and consistency of the requirements stated in ESL. Thus, a specification of the processing to be performed is obtained which is independent of the architecture of any particular hardware or software system. This specification can then be used to select the appropriate hardware and subsequent software design to meet the requirements.


The authors describe their efforts to "devise an advanced engineering methodology that supports the design, development, validation and unambiguous communication of complete and consistent data processing subsystem performance requirements". A method of decomposing the system into 'decomposition elements' is given and software aids are provided for consistency checking. The major part of the paper is concerned with automating verification techniques for specifications. The software developer need then only be concerned with the validation of the generated code.

Two principles for the specification of a software module are stated in this widely referenced and influential paper. First, to provide a user of the module with all the information needed to use the module and nothing more. Second, to provide the implementor of the module with full information about the intended use of the module and nothing more; in particular, no information about the structure of calling programs should be conveyed. A specification technique based on these principles is presented, and illustrated with the help of many simple but realistic examples. The specification of error handling in modules is also discussed.


A critical review is presented of current approaches to formally specifying program modules. Six criteria are presented for evaluating specification methods: (i) formality; (ii) constructability; (iii) comprehensibility; (iv) minimality; (v) applicability; and (vi) extensibility. A program module is taken to mean a "program unit supporting a data abstraction". Specification techniques are classified as those using (i) some well established mathematical discipline (for instance, graph theory); (ii) state machine models; (iii) axiomatic descriptions; and (iv) algebraic definitions. These specification techniques are explained with the help of a simple stack example. It is pointed out that no one technique is superior to all the others. This paper forms a useful introduction to the specification of data abstractions, a topic on which there is much current research.
Programming Methodology

Almost any limitations imposed on the designing and writing of programs can be considered as a methodology. Preferably the limitations should be chosen as a way of precluding some of the many opportunities for erring open to a programmer. Such constraints range from methods of program design to the influence of the programming language in which the software is actually coded. Clearly, the design of programming languages constitutes an important area of investigation with respect to software reliability; five of the references (3.6 - 3.10) present a contribution to this area. The first seven elucidate general program design principles while items 3.11 and 3.12 discuss rather more specialised techniques. Item 3.13 considers how the computer can assist the programmer by providing specialist tools for software development (see also item 3.5). The last reference discusses the importance of adequate documentation for software.


The three sets of notes contained in this justly renowned book are all highly recommended; they apply structuring principles to program design with reference to both control and data. For an entertaining and detailed review of the book see Knuth's Stanford University Technical Report: STAN-CS-73-371, A Review of Structured Programming (1973). Knuth asserts "It is impossible to read [this book] without having it change your life".


Controversy over the \texttt{go to} statement was instigated by Dijkstra's well known letter in 1968. Knuth's lengthy paper was written with the goal of laying that controversy to rest. The two main sections discuss first the elimination and second the introduction of \texttt{go to} statements. Much of this material is pragmatic and deals with important efficiency considerations. Program manipulation systems are advocated so that well structured programs can be safely transformed into more efficient but possibly less structured versions. The iterative control construct of Zahn is promulgated as a means of avoiding the need for \texttt{go to} statements. The third and concluding section recognises that much of the earlier material addresses the wrong issue and redresses the balance by concentrating on more methodological questions of program structure. A valuable bibliography of 102 items is appended. (The paper appears in a special programming issue of Computing Surveys which is worth reading in its entirety.)

An aggressive and evangelical stance is taken on the benefits to be gained from structured programming (as seen by Mills). "The new reality is that you can write programs which are error free." However, the methodology advocated is over simple, requiring little more than the use of disciplined control structures. Of course, even this can constitute a significant advance over using completely unconstrained control mechanisms. The paper includes a well balanced discussion of the status to be accorded any mathematical proof (not only those of program correctness), namely that a proof is a repeatable experiment which is performed on the reader of the proof. If the reader is convinced then the experiment was successful.


The methodology advocated herein is based on data structuring. Flow diagrams are discarded and replaced by (data) structure diagrams. Program structure is then derived from a structure diagram using control constructs selected to make program and data correspond as closely as possible. Like Mills the approach is rather simplistic and like Mills (but less strident) there is an element of evangelism in the presentation - "We know that the program is correct, because we designed it correctly". Most of the examples are oriented towards commercial data processing, which must enhance the usefulness of the book to many practising programmers. It is written in a very readable, direct style and is often amusing (for instance, see the tale of Big Louie).


Both of these books are intended to demonstrate, largely by example, what constitute good programming practice. The first book presents a large number of programs (taken from published texts and written in either Fortran or PL/1) which are criticised and then rewritten to remedy their defects. A rule or general conclusion is then proposed for each case. Well over 50 such guidelines are summarised at the end of the book. The second book includes over 5000 lines of program written by the authors themselves in the language Mataf, which is a "rationalised" version of Fortran with modern control flow statements. As well as being used to demonstrate a methodology of programming, the programs themselves form a collection of tools to assist the programmer. Not surprisingly there are still errors in some of these programs (which have all been tested but are not claimed to be flawless). An epilogue suggests three principles - "keep it simple; build it in stages; let someone else do the hard part".
One of the central notions of the book is that of defining the semantics of a programming language by means of predicate transformers. The predicate transformer corresponding to a particular language mechanism can be used to derive the weakest pre-condition which will ensure that a given post-condition is established by the use of that mechanism. In this way the semantics of a 'mini-language' are specified. Included in the language are guarded commands, constraints on the scope and initialisation of variables, and a coherent treatment of array variables. The second half of the book provides programmed solutions to a sequence of 13 problems, in many cases including the design and development process which led to the solution. Like its subject matter, this book is difficult but this does not diminish its significance.

This book has - at least - three virtues. As an elementary introduction to programming it can be commended for its clarity and precision. The language used as a vehicle for this purpose is PASCAL, the design of which has influenced many subsequent language designs. Throughout the book, and specifically in the final chapter, the methodology of stepwise program development is adopted as a means of imparting structure to the finished program, structure which should be mirrored in the notation in which the program is expressed.

The most important and interesting aspect of the CLU programming language is the provision made for the use of abstraction by a programmer. A linguistic construct called a cluster can be used to implement abstract data types in such a way that the details of their implementation and representation are concealed. Only the required behavioural properties of objects of an abstract data type are made available for use. The paper concentrates on this data abstraction facility, but also discusses procedural and control abstraction mechanisms. As a control abstraction CLU provides an iterator which permits iteration over collections of any type of object. The CLU language, like many others, has been influenced by SIMULA, but shares with SIMULA the necessity of associating each operation with precisely one data type - a restriction which seems rather anomalous.

After discussing how a programming language should be designed to increase the reliability of software written in the language, the paper documents the results of an experiment performed to assess the effect on reliability of a number of language design decisions. Despite the small sample sizes for which measurements were taken (10 and 15 programmers) statistical tests indicated that several of the decisions significantly affected reliability. Although over 1200 errors were analysed, nearly all of the conclusions are based on less than 100 of these errors, indicating how expensive such empirical measurements are. Some useful references are given to studies of how errors are made while programming.


For light relief, here is a paper which has as an alternative title "How to Design Languages to make Programming as Difficult as Possible", and towards that end supplies 29 maxims for those of "evil heart and mind". Most of the suggested guidelines are given added piquancy by drawing on everyone's favourite languages for examples. The text is enriched by some very apposite quotations, and ends with this one from Ploen: "There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs".


To obtain the maximum benefit from a modular approach to the design and construction of large systems it is desirable that the interfaces between the modules should be simple and that the modules themselves should be as independent as possible. This paper examines two modular designs for an indexing system, and argues that the straightforward approach of using a module for each major step in the processing is markedly inferior to the use of modules which conceal design decisions from the rest of the system. However, efficient implementation is then necessary to minimise the cost of calls between modules, since the number of such calls will be greatly increased if the latter approach is adopted.

The methodological contribution of this paper is in its presentation of an approach to the construction of complex systems. A hierarchical design methodology and the notions of module and level are discussed and exemplified in the context of the design and implementation of a family of operating systems. Stress is placed on matching the structure of the documentation and specification of the system to that of the system design. Much of what is said can be applied to any large software system; the first half of the paper in particular rewards a careful reading.


The Programmer's Workbench is a highly successful program development system providing a set of programming tools which are independent of the systems on which the programs that are developed are to be used. The system is based on a network of small computers (actually PDP-11s located in Bell Laboratories, New Jersey); it is claimed that this organization is particularly suitable for a program development facility. It seems clear that the Programmer's Workbench has benefited greatly from being supported by the excellent UNIX operating system employed on the PDP-11s. This introductory paper gives an overview of the system and its development. Further details on particular aspects of the system (command language, documentation tools, a testing facility, a project management facility, and a user view) can be obtained from five companion papers in the same proceedings.


The case study demonstrates that clear and concise documentation can make a major contribution to the successful development of a software project. The project in question was to implement an operating system and an assembler for a minicomputer. Details of the documentation system used are presented; the design and development group regarded documentation aspects as an integral part of the project. The author concludes that "Documentation procedure helped in clarifying the minds of project participants, and in recording results and decisions. It made possible the management of the technical quality, the co-ordination of the work and to some extent the work's progress in time." A useful paper.
4 Software Certification

A program is certified by presenting a case that the behaviour of the program when it is executed will be that which is required of it. An attempt to certify a program can range from 'obviously it works' through 'it has been thoroughly tested' to 'it has been proved correct'. Arguments presented in a specification can be rigorous or nonrigorous, formal or informal, lucid or obscure, concise or verbose. Published material on certification (or validation) covers both correctness proving and testing. Both areas are also relevant in methodologies for producing reliable software; proving as a control and guide to program construction, and testing as a means of debugging. Of the 10 entries in this section, six relate to program proving, one warns that testing is still necessary, the next two discuss testing methods, while the last suggests how testing can become respectable as a certification technique.


An informal presentation of verification techniques (other than testing) principally in terms of the assertion methods. Pragmatic considerations are also discussed: whether large complex programs can be verified; the many assumptions made about the environment in which a program is executed; termination issues; and the difficulty of generating the necessary assertions. An outline is given of the work which has been done on automatic and interactive verification systems. Finally, an encouraging picture is painted of the current state-of-the-art in program verification. The paper is eminently readable and provides a list of 65 references.


A lengthy and extensive survey of techniques relevant to the problem of establishing program correctness. To some extent the treatment is biased towards the dream of mechanical program verification, considered by the authors to be "the only foolproof approach"—an opinion which seems overly optimistic. One consequence of this attitude to verification is the inclusion of a long chapter on mechanical theorem proving. Other chapters discuss the theory of program schemas, informal methods of proof, and the formal results of Manna. The paper asserts (correctly) that informal proofs are unreliable (but valuable), and predicts that verification will ultimately be a mechanised though highly interactive process. Although mostly readable the paper does contain a number of more technical sections. The bibliography provides 86 references.

In order to provide a logical basis for proofs of the properties of a program, Hoare presents axioms and rules of inference appropriate for a simple language containing integers, assignment, composition and iteration. The notations of this early paper have been utilised by many subsequent authors, primarily because of their directness and intuitive appeal. The paper also discusses the need for correctness proofs, and the application of an axiomatic approach to programming language design and definition. It is noted that "reliability can be purchased only at the price of simplicity".


A tutorial style of exposition is used to present a program proving technique (based on that of Nanna) which can be viewed as symbolic execution of a program, using induction to avoid indefinite looping. Example proofs are used to elaborate the method and to explain the difference between this approach and that of Floyd; namely that Floyd assertions are of the form "At I in the program a certain property always holds" whereas the presented technique uses assertions of the form "At I in the program a certain property will hold sometime". Situations where one or the other approach is simpler are considered. A fairly informal and easy to read paper.


A very clear and straightforward presentation of program correctness proving by means of symbolic execution using Floyd's form of induction for loops. As a result, termination issues are ignored and the paper concerns itself only with partial correctness. Symbolic execution trees are explicitly discussed, and the way in which induction is used to cut these trees down to finite size is explained (although the required assertions are produced by an "inductive genius"). A method is described for embedding the proof of a procedure as a lemma in the proof of the calling program, together with a splendid example of a difficulty with the parameter mechanism used (which demonstrates that a proof may not always prove what was intended).

Hoare's axiomatic proof method as applied to parallel programs is extended, principally to handle the removal of auxiliary variables (which do not influence the computations performed) from a program. The axioms given for parallel execution, critical sections and auxiliary variables, together with the normal sequential program axioms, are claimed to be complete in that any true formula \[ F \in S \{ \} \] can be established from them. Proof techniques are described for proving explicitly programmed mutual exclusion, deadlock avoidance and termination, all with reference to the ubiquitous dining philosophers example. An informal and readable paper.


Consideration is given to 12 published programs in which errors or inconsistencies have been discovered even though the programs had been constructed using one or more of the methodologies of formal specification, structured programming or program proving. The paper's intention is to provide a "more realistic insight into the mathematical foundations of these methodologies" and in so doing warns of human fallibility in mathematics as well as in programming. A cautionary tale, emphasising the continuing need for testing as well as the more recent methodologies.


The approach described is that of determining a set of test cases (that is, inputs) for a program such that every edge of the control flow diagram associated with the program is traversed at least once. This criterion is generally regarded as being the minimum that is needed. Clearly such a set is likely to provide more comprehensive testing than an equal number of test cases chosen at random. The paper suggests (dubiously) that statistical significance can be attached to the outcome of such tests. Two methods for finding a set of test cases are considered, one is empirical and the other is a more formal technique. Limitations of these methods are pointed out in some concluding remarks. An elementary level of presentation is adopted. (Some significance may attach to the fact that the defects of the algorithm of the first example program in the paper remain unremarked.)
The paper considers ways of automating the process of generating test data to exercise a given path in a program. Methods are advocated based on symbolic execution of the program. Difficulties are noted with respect to loops, arrays and module calls within the program. These methods lead to a set of potentially non-linear constraints on the input variables which must be solved to yield the required test data values. An iterative, trial and error method of solution is suggested. A prototype capable of generating test data automatically for FORTRAN programs has been implemented. Some weaknesses and inefficiencies of the implementation are mentioned.

This paper attempts to provide a start towards a sound theoretical basis for program testing. Definitions are given which ensure the result that: if a test data selection criterion is both reliable and valid, and a complete set of test data selected by that criterion executes successfully, then all executions will be successful. In practice it is the reliability of the test which is important. It is demonstrated that testing methods based on ensuring the exercising of all statements or control paths in a program are inadequate. Test data need to be devised so as to exercise paths under all relevant circumstances. Testing is discussed in relation to program proving. "Testing and proving are complementary methods for decreasing the likelihood of program failure. Both have an essential role in program development, though each is a fallible technique." An important paper.
5 Software Fault-tolerance and Related Topics

Three classes of papers have been grouped in this section, on the topics of: (i) error confinement; (ii) software fault-tolerance; and (iii) some working reliable systems. One of the essential properties a reliable system must possess is that of error confinement: the property of preventing an erroneous or corrupted module from damaging other modules. This directly relates to the protection of information in computer systems and the first five papers in this section are devoted to this subject. The next five papers deal with software fault-tolerance itself. This is a relatively new subject and not many papers are available. The paper by Davies deserves a special mention since it contains ideas which can be applied to any system - whether software or not. Finally, the last three papers describe reliable computing systems incorporating many of the ideas of the previous papers of this section.


Contains a concise and very readable account of protection in computer systems. The chapter on memory management describes the two well known protection schemes: access list based and capability based; included is a discussion on hardware features necessary to support these schemes. Later chapters describe user authentication mechanisms and file protection techniques. An additional benefit is that the book also contains details of file recovery techniques and methods of system restart after a failure. Highly recommended.


A very comprehensive survey of techniques for protecting computer-stored information from unauthorised use or modification. Eight design principles for designing a protection system are given: (i) economy of mechanism; (ii) failsafe defaults; (iii) complete mediation; (iv) open design; (v) separation of privilege; (vi) least privilege; (vii) least common mechanism; and (viii) psychological acceptability. Next, a capability based protection system is developed and its main advantages are discussed. The disadvantages of such a protection scheme (for instance, revocation of access is not easy) are also discussed. Another protection scheme - based on access control lists - is then developed and its advantages and disadvantages are described. The paper concludes by pointing out areas for further research. The paper contains a valuable list of references to papers on protection and related topics.

In this influential paper Lampson introduced the idea of an access matrix. If $A$ is an access matrix, then element $A[i,j]$ specifies the access (such as read, write) which domain $i$ has to object $j$. Rules are specified for adding or deleting entries from $A$. For example, a domain $d^1$ can remove access attributes from domain $d^2$ if it has ‘control’ access over $d^2$. An access matrix together with these rules represents an abstract model of protection which exhibits the properties of most protection mechanisms. Lampson then describes how the model can be implemented: if the access information is attached to domains then one has capability based protection system. On the other hand, if the access information is attached to objects, one gets the access list based protection system. A hybrid approach is also described.


Contains a critical account of the protection techniques used in the Multics system, which is a good example of an access list based protection system. With each segment (‘segment’ is synonymous with ‘file’ in Multics) is associated an open-ended list of names of users who are permitted to access this segment, together with the kind of access permitted. This list is called the access control list of a segment and a user is allowed to access a segment only if he is mentioned in the corresponding list. Such a check is carried out for every access, and hardware support is provided for speeding up this process. This hardware mechanism essentially consists of copying the access information from the access control list to the user’s segment descriptor. Multics also contains a hierarchical protection scheme (called the rings of protection) for the creation of protected subsystems. The paper contains a valuable discussion on weaknesses of the Multics protection mechanisms.


The protection mechanisms considered so far have been concerned with guaranteeing the protection of information and not with the flow of information. It has proved very difficult to guarantee that a process will not, intentionally or unintentionally, leak information to other processes. This paper develops an ‘information flow’ model which can be used to specify secure information flow requirements. Some existing security systems are described using the model. It is shown that practical systems will need both access control (characterised by Lampson’s access matrix model) and flow control to satisfy all security requirements.
This paper is mostly hardware oriented, but nevertheless, well worth reading. Two approaches to reliable system design are considered: fault-tolerance and fault-intolerance. In the first approach reliability is obtained by the use of protective redundancy for error detection and recovery while in the second approach reliability must be obtained by a priori elimination of the causes of unreliability. The paper contains a discussion on the types of faults and various redundancy techniques.

The paper presents a convincing case for incorporating fault-tolerance in software. It states "...all software faults result from design errors. The relative frequency of such errors reflects the much greater logical complexity of the typical software design compared to that of a typical hardware design". A method is proposed for dealing with software faults which consists of a program structure (known as a recovery block) for error detection and recovery. The essence of this scheme is that it provides a facility for a computation to backtrack to an earlier state if an error is detected, and then proceed again using a possibly different algorithm. The paper also discusses error recovery problems between interacting processes, and presents a methodology for constructing reliable complex systems. A recommended reading for everyone interested in software fault-tolerance.

A strong distinction is drawn between reliability and correctness, in that reliable software need not be correct (if failures occur but rarely), while correct software can be quite unreliable (if the user's expectations conflict with the software specifications). The paper presents three guidelines for reliable system design: (i) when specifying system modules, take into account the behaviour which is desired when perfect behaviour is not obtainable; (ii) when specifying interfaces between modules, specify not only what the interfacing elements should do in the normal case, but also which assumptions should be verified by run-time checks and what actions are required when an error is detected; and (iii) include in the interfaces conventions for informing affected modules about things that have gone wrong elsewhere in the system. Some examples are given to support the author's proposals. This is a thought provoking paper.

The paper demonstrates the importance of including fault-tolerance in real-time software. It states "At input and output, the real-time computer system interfaces with sensors, communication channels and control mechanisms are subjected to failure, noise and temporary abnormalities caused by the environment. Critical software is expected to cope with these conditions ..." The recovery block approach advanced in item 5.8 is stated to meet these requirements. An interesting example is given on the use of recovery blocks in an aircraft navigation program. The paper also develops a simple reliability model for fault-tolerant software, the application of which shows that a significant reduction in failure probability can be achieved.


In this important paper, Davies introduced the concept of sphere of control as a bound around a process for describing and solving, among other things, error recovery problems amongst processes exchanging information. When a process is inside a sphere of control, any effects produced by the process are not regarded as committed and the process can back out without affecting processes not in this sphere. On the other hand, once a process exits from a sphere of control, the effects produced by it are regarded as committed, so any recovery actions must be taken by an enclosing sphere of control (if any). Davies also describes how a sphere of control may be 'activated' and 'de-activated'. This paper contains ideas of major significance and requires careful reading.


Here is a good example of a working real-time system (Bell System's Traffic Service Position System) incorporating software fault-tolerance. The designers realized that a complex system will contain residual design errors, so extensive checks were incorporated for protecting data from corruption. Independent check programs (audits) were employed to monitor the system and undertake corrective actions whenever data corruption is suspected. The paper contains many illustrative examples to describe the audit techniques used. It was observed that from 10 to 100 errors per day were detected and corrected by audits in a typical operational system. When audits fail to provide recovery from errors, system restart is used; this was happening at the rate of about once every two months. This paper provides interesting reading.

An experimental, high integrity transaction oriented system is described. It consists of a set of asynchronous parallel processes whose job is to cyclically process transactions. Capability based protection is used for ensuring error confinement. Four versions of every file are kept: two read only versions (on secondary storage) for global, cold start recovery and two read-write versions for run-time recovery. If a process is unable to process a transaction, its state is reset to that just before the beginning of the cycle so that the transaction can be reprocessed; if the system crashes, then all of the processes are reset to the beginning of their respective cycles. These recovery actions are described in detail in the paper.


The paper describes an experimental relational database system and contains an interesting account of the recovery techniques used. A transaction is treated as a unit of consistency and recovery. A user can specify save points within a transaction. Integrity assertions about data are checked and enforced at the end of a transaction. If some assertion is not satisfied, the transaction is backed out to the nearest save point. The recovery information for a transaction is maintained in a time ordered list which contains information about each change to recoverable data. During a transaction recovery, special routines are used to undo all of the listed modifications, back to the most recently recorded point. Two system recovery mechanisms are also employed: one to deal with main memory corruption and the other to deal with disk storage corruption.
6 Software Reliability Modelling

It has been realised that, with the exception of trivial programs, software is seldom free from faults. Thus, despite intensive testing, the 'delivered product' is likely to contain residual design errors. Observation has revealed that if during the testing and integration phase of software development (when individually tested modules are combined to form a system which is then tested) sufficient data is collected about detected errors, time to correct errors, mean time between software errors etc. then predictions about reliability and project schedules can be made. The first five papers of this section describe current efforts to predict software reliability, while the last three papers present error data (and its analysis) collected during tests.


This is one of the earliest (and quite influential) papers describing an effort to predict software reliability using traditional reliability theory. The authors state that a 'constant failure rate' assumption can be applied to software error detection and hence software reliability at time t can be stated as \[ R(t) = \exp(-\lambda t) \] where \( \lambda \) is the failure rate. The failure rate can be assumed to be proportional to the residual errors in the software; the authors present a technique whereby error detection data collected during tests can be used to estimate the failure rate.


A very similar model to that of Jelinski and Miranda has been independently developed by Shooman. The model can be used to predict software reliability given the debugging effort involved in removing errors. The model is applicable to the last phase of software development: test and integration. The failure rate is assumed to be equal to \( CEr(t) \) where \( C \) is a constant (indicating 'instruction processing rate') and \( Er(t) \) is the number of residual errors after \( t \) months of debugging time. Shooman presents a technique for estimating \( C \) and \( Er(t) \) so that the reliability at time \( T \) for software that has undergone \( t \) months of debugging can be stated as:

\[ R(T,t) = \exp(-CEr(t)T). \]

In contrast to the previous two approaches, Schneidewind has taken an empirical approach and suggests a reliability prediction scheme based on fitting failure intervals with an appropriate reliability function. First, data is collected in the form of "time between software troubles". The nature of this distribution contains clues as to what distribution function (e.g. exponential, normal, Weibull) can be fitted over it. Schneidewind then describes how goodness of fit tests (such as Kolmogorov-Smirnov or Chi Square) can be used to check that the chosen distribution function appropriately represents the test data. Once the distribution is determined, reliability can be obtained by using the standard results from reliability theory. The feasibility of this approach was demonstrated by applying it to Naval Tactical Data System data; the details of this experiment are, however, not given.


This is one of the best papers on this subject. The author develops a model (called the execution time model) incorporating many of the features of the previous three papers; this model is then validated by applying it to four medium-sized software development projects. The model can be used, in advance of a project, to estimate the amount of testing (stated in terms of execution time) needed to achieve a specified reliability goal. The required execution time can be related to calendar time thus allowing project schedules to be developed. The author also presents a method (employing the maximum likelihood estimation technique) for estimating model parameters.


The model is developed to predict reliability and availability of large software systems (10^5 words of code or more) given the error occurrence rate and the error repair rate. A discrete-state, continuous-time Markov model is developed under the assumption that the system contains an unknown number of bugs, and that error detection and correction occur alternately and sequentially. Some extensions to the model are also suggested - e.g. inclusion of several categories of errors (critical, noncritical) - which are currently under investigation by the authors.
(6.6) Endres A. An Analysis of Errors and Their Causes in System Programs. IEEE Transactions on Software Engineering 1,2, pp. 140-149, June 1975.

The paper presents interesting data on errors in a system program (DOS/VSE release 28). The data was collected when the modules of the system were ready for test and integration. A somewhat surprising observation was that 85.1% of the errors could be corrected by changing only one module (per error). Surprising because changes to many modules could be needed to correct an error in an operating system where modules are highly interdependent. It was also found that 46% of the errors were due to incorrect understanding of problems to be solved and 38% of the errors were due to incorrect implementation of a given algorithm. This study thus shows that better programming techniques (better programming languages, more systematic programming) can only avoid at most half of the mistakes—the other half must be attacked by employing better tools for problem specification, such as specification languages.


The paper gives actual error and reliability data collected from a large telephone switching system. In effect, this data validates the Shooman model referenced earlier. The data was collected during the test and integration period of the system. Data collected included cumulative number of errors detected, time to correct each error and MTTF of the system. The form of reliability improvement (as the errors were corrected) closely resembles that predicted by Shooman. The authors describe in detail the selection of a testing environment to simulate the actual working conditions. The main conclusion of this work, as the authors point out, is that if sufficient data is collected during system testing, realistic software reliability estimation can be performed.


This paper demonstrates that if error data is collected during software module tests, estimation about the number of errors in similar modules and the time needed to correct them can be made. This data is thus useful in project planning and scheduling. The author observed a strong correlation between the number of errors detected and the nature of the program.