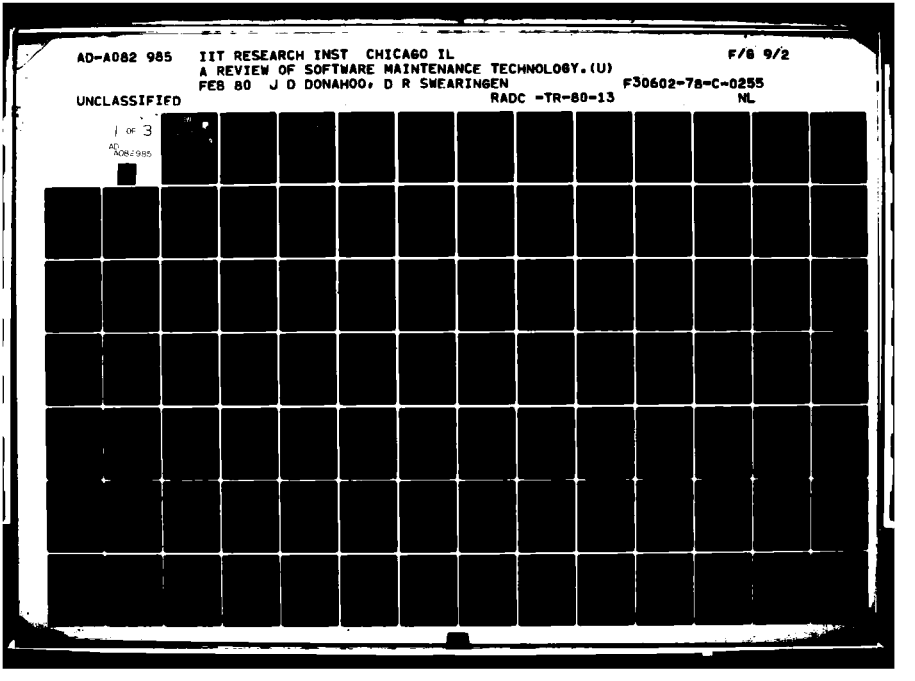
AD-A082 985 IIT RESEARCH INST CHICAGO IL F/G 9/2
A REVIEW OF SOFTWARE MAINTENANCE TECHNOLOGY.(U)
FEB 80 J D DONAHOO, D R SWEARINGEN F30602-78-C-0255
UNCLASSIFIED RADC -TR-80-13 NL

1 OF 3
AD A082985



LEVEL II

(12)

RADC-TR-80-13
Interim Report
February 1980

A REVIEW OF SOFTWARE MAINTENANCE TECHNOLOGY

ITT Research Institute

John D. Donahoo
Dorothy Swearingen

A REVIEW OF SOFTWARE MAINTENANCE TECHNOLOGY

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-80-13 has been reviewed and is approved for publication.

APPROVED: John Palaimo

JOHN PALAIMO
Project Engineer

APPROVED: Wendall C Bauman

WENDALL C. BAUMAN, Colonel, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-80-13	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): A REVIEW OF SOFTWARE MAINTENANCE TECHNOLOGY.	5. TYPE OF REPORT & PERIOD COVERED: Interim Report · Mar — Nov 79,
6. PERFORMING ORG. REPORT NUMBER: N/A	
7. AUTHOR(s): John D. Donahoo, Dorothy Swearingen	8. CONTRACT OR GRANT NUMBER(s): F30602-78-C-0255
9. PERFORMING ORGANIZATION NAME AND ADDRESS: IIT Research Institute, 10 West 3rd Street, Chicago IL 60616	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 63728F, 25280101
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: Feb 1980
	13. NUMBER OF PAGES: 221
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office): Same	15. SECURITY CLASS. (of this report): UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: John Palaimo (ISIS)

This work was performed under a subcontract by Computer Sciences Corporation, 6022 Technology Drive, Huntsville, AL 35807.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Maintenance
Software Maintenance Tools
Software Maintenance Techniques
Software Life Cycle Management
Software Verification and Validation
Software Failures
Software Testing
Software Modification

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this effort was to develop a comprehensive statement about software maintenance techniques and tools in use today. This report focuses on software maintenance technology as it is described and defined in open literature and technical reports. Material was selected based on its relevance to the subject of software maintenance and date it was published. Generally, only papers and articles published since 1974 and reports and books published since 1975 were selected.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

175350

FOREWORD

This technical report presents the results of a review of open literature and technical reports concerning the development and use of computer software maintenance techniques and tools. This effort was sponsored by the Information Sciences Division of the Rome Air Development Center (RADC) under contract number F30602-78-C-0255. J. Palaimo of RADC was the Project Engineer. Review research activities were managed by the RADC Data and Analysis Center for Software (DACS) in conjunction with its software engineering research program.

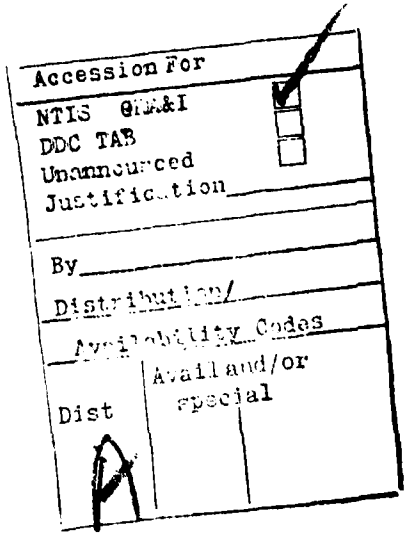
Approved by

Lorraine Duvall
DACS Program Manager
IIT Research Institute

Submitted by

John D. Donahoo
Computer Sciences Corporation

Dorothy R. Swearingen
Dorothy Swearingen
Computer Sciences Corporation



EXECUTIVE SUMMARY

At the present time computer software maintenance is receiving a great deal of attention; from data processing managers who see more and more of their resources committed to the support of operational software, from programmers and analysts who find themselves responsible for increasing volumes of program code, and from users who demand improved performance, expanded capabilities, and new products from existing systems. As an outgrowth of that attention the published literature is filled with reports and discussions of software maintenance related topics. There are new maintenance support systems, both automated and manual and expanded applications for the established technology. New software maintenance concepts and approaches are being presented. This report offers some insight into the wealth of information on these topics that exists in current articles, papers, reports, and books. No attempt has been made to evaluate the techniques and tools described in this report. They are presented through the medium of summary descriptions in a common format and they are correlated with maintenance activities using a correlation matrix. The set of techniques and tools described in this report is by no means complete. However, it is representative of software maintenance technology as it exists today. These techniques and tools have been selected because they are typical of applied concepts and approaches, and because they have been adequately documented or described in open literature.

A significant deficiency that exists in all the literature reviewed for this report is a lack of definitive information about technology performance in a maintenance environment. How well do these techniques and tools support maintenance of operational software systems? The technology is usually described statically in terms of attributes and processes. If application is discussed the information is typically general in nature with no specific reference to performance or effectiveness. This one aspect of software maintenance technology literature appears to be worthy of more attention and research.

CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

SECTION I
INTRODUCTION

1.1 REVIEW BACKGROUND/PURPOSE

The data processing community is now experiencing the effects of being associated with a maturing industry. The effects are more pronounced for the hardware segment of the industry and have been examined and reported in some detail. Identification and definition of these effects for the software segment is relatively tentative now, but an awareness of their presence is reflected in experiences and concerns described in many of the papers, reports, and articles reviewed in preparing this report. These experiences and concerns are conveyed through discussion of such diverse subjects as life cycle management and technological challenge, data processing terminology, software maintenance management and technology, computer program development, and others. In these documents the writers quite often cite the lack of a structured and universally applicable discipline for implementing and managing software across its life cycle, as a significant impediment to achieving full potential of that software. Their statements reflect a growing frustration with the fundamental problems facing the entire data processing industry today. One of those problems involves an ever increasing body of expensive and complex operational software that must be maintained in a viable state.

This report focuses on software maintenance technology as it is described and defined in open literature and technical reports. No attempt has been made to define a software management or maintenance discipline. That would have been far beyond the scope of this project. The purpose in conducting the technology review was to develop a comprehensive statement about the maintenance techniques and tools in use today and to describe how they support the activities associated with computer software maintenance. The computer program maintenance environment was

also examined and is described through discussion of topics related to implementation of software maintenance technology. It is hoped that this report might foster further research into software maintenance principles and ultimately lead to definition of a software engineering discipline encompassing all aspects of computer program development and maintenance.

One realization that comes early to anyone reviewing the literature of data processing is that the terminology is not consistent. It is, no doubt, simply another symptom of the rapid development and changes that are occuring in this industry. The language of data processing has not been given an opportunity to "catch its breath" as the technology continues its accelerating advances. For that reason, each writer must carefully define the key terms and phrases which he uses in developing his technical presentation. During the review of literature and reports for this report certain terms and phrases were common to the discussions of maintenance technology. In most cases definitions were given or meanings were obvious from the context. The most prominent of those terms and phrases are shown below, along with brief discussions of their usage and the definitions applied for this report. A complete glossary of terms to be found in this report are presented in Appendix A.

- Software (Computer Program) Maintenance. Some writers have expressed reservations about using the term maintenance with respect to computer programs. Among the reasons given is the concept that programs don't fail the way hardware fails (References 1 and 2). That is, software "parts" don't deteriorate or break, but functional failures occur because of existing states of the programs. Therefore, maintaining software is really only a changing or modifying process. Also the point is made that the term "maintenance" has a less than desirable connotation for most programmers (Reference 3). Maintaining computer programs is considered to be work at a lower skill level than program development and is thus to be avoided. However, the word maintenance seems to be too firmly entrenched in the literature on this subject to be replaced now. For this report a definition of software maintenance was followed which is consistent with that presented

by Boehm in his paper, Software Engineering (Reference 4). That is, "the process of modifying existing operational software while leaving its primary functions intact". In addition the definition is broadened to include software enhancement or extending the capabilities of the software to include new or modified functions.

- Software Engineering. In the decade since its introduction this term has assumed increasing significance as the importance of structured software development and maintenance methodologies has been established. Yet, without a common body of techniques and tools and standard procedures for implementing them, "software engineers" have been engineers in name only. The beginning of the revolution in thinking that must occur before software can be truly engineered is evident in literature on that subject. Researchers are addressing the unknowns in software reliability and maintainability, programmer productivity, program functional modularity, etc. Software engineering is generally defined as the body of knowledge which may be applied to the design, construction, and modification of computer programs (References 4, 5, and 6). That body of knowledge is limited at the present time, but it is rapidly being expanded.

- Software (Computer Program) Maintainability. The concept of developing a capability for identifying the level of maintainability of a computer program is fairly new. It is an important adjunct to the creation of a software engineering discipline which includes characteristics such as reliability, portability, testability, as well as maintainability. Quantitative measurement of the maintainability factor has been reported by at least one research team (Reference 7). Software maintainability is defined as a measure of the relative level of effort required to modify, update, or enhance a computer program.

1.2 REVIEW SCOPE

As stated in the previous section this report does not attempt to define or describe a discipline for software management or maintenance. The basis for this report was the technical papers, articles, and reports that are shown in the bibliography. The information in this document was derived from those references in order to present a review of software maintenance technology.

Research for compilation of the reference sources was conducted at the following facilities;

Redstone Scientific Information Center, Redstone Arsenal, Alabama.

University of Alabama in Huntsville Library, Huntsville, Alabama.

Data and Analysis Center for Software, Rome Air Development Center, New York.

Syracuse University Library, Syracuse, New York.

Material was selected based on its relevance to the subject of software maintenance and the date it was published. Generally, only papers and articles published since 1974 were selected. For reports and books a publication date of 1976 or later was observed. There were exceptions for material that was deemed to be particularly relevant or unique in content.

1.3 REPORT CONTENT

The balance of this report is organized by sections as follows:

Section II puts the software or program maintenance environment in perspective with an overview of that environment. The purpose of the overview is to provide necessary background information for an understanding of the maintenance technology discussion. Maintenance activities are discussed with respect to three bases of software maintenance as defined by Swanson (Reference 8). These bases motivate technology application, that is, corrective maintenance for software failures, adaptive maintenance for environment changes and perfective maintenance for software enhancements. Where possible the continuity of these maintenance activities is established, from their initial implementation early in the life cycle through their ultimate use in support of operations and maintenance phase requirements. In addition, software system and program maintenance engineering across the life cycle is discussed. These topics encompass consideration of computer

program maintenance continuity through the life cycle and specific maintenance engineering functions. Maintenance tasks are discussed from the related viewpoints of the automated data system manager and maintainer in administering maintenance. Finally, the concept of developing maintainable software is explored.

Section III discusses the technology of software maintenance as it is practiced today. From the research material a consistent classification of maintenance functional requirements is compiled. The intent is to create a framework within which the techniques and tools of software maintenance can be defined. These definitions are structured so that comparisons may be made among similar techniques and tools, and those that are complementary may be readily identified. A discussion of maintenance activities in the operations and maintenance phase is included next. Maintenance activities such as error identification/isolation, system/program redesign, test and integration, quality assurance, configuration management and others are identified. These activities are then associated with maintenance technology functions through the use of a technology/activity correlation matrix. The matrix qualifies each technology function by identifying its application base in a corrective, adaptive or perfective role.

Section IV contains descriptions of maintenance techniques and tools. Each technique or tool is discussed using a format that includes the category and characteristics, reference sources for the description, status of usage, description of the technique or tool, research findings from use and reports of actual usage in maintenance. A tool/technique application matrix which summarizes the characteristics of each technique and tool is shown.

Section V provides an assessment of the present state-of-the-art in software maintenance technology. This assessment is based on research data selected for the survey report. The assessment leads to a presentation of goals or objectives to be considered for further research into software maintenance technology.

SECTION II
SOFTWARE MAINTENANCE OVERVIEW

2.1 LIFE CYCLE MAINTENANCE

The term life cycle maintenance implies that computer programs must be maintained both before and after they are released to the user. This, of course, is true although during program development maintenance activities are generally not referred to as such. Not until the software is released to the user does maintenance become a recognized support function. However, it is useful to consider the implications of maintenance support for computer programs from their creation to their deactivation.

The program maintenance function is created with the initial lines of code that the programmer writes. As the program design is translated into form and function, the software internal organization is established. That structure, the details of which are often left to the discretion of the programmer, can directly influence future program and system maintenance requirements. The second line of program code written creates an implicit requirement to analyze and perhaps modify the first. Subsequent lines likewise impact all previous ones. Thus, program maintenance begins and the concept that the first line of code establishes the dimensions of later maintenance operations (Reference 5) should be of interest to software developers and users alike.

Accepting that a significant portion of operational maintenance is represented in extensions or revisions to the delivered software design, and that typically there are undiscovered flaws in all software when it is declared operational, then operators and maintenance support organizations have a big stake in the program development process. The literature suggests that all functions of operational maintenance are affected by the development process which produced the software. A precise relationship between factors influencing program development and subsequent

operational maintenance costs has not been established. If such a relationship exists and if it can be defined, it may be possible to control the spiraling costs of software maintenance. There are certain elements of the program development process that must be considered prime candidates for establishing this relationship. Among these are:

- Cost. Is a design to cost approach being used? Can system requirements be satisfied within program budget or must compromises be made in design and development?
- Performance. Are performance requirements completely and consistently specified? Can performance requirements be met efficiently through the design as defined? Must design or coding integrity be sacrificed in order to meet performance requirements?
- Schedule. Are development milestones met? Is program coding accelerated at the expense of adequate static analysis in order to meet production schedules? Is testing concluded based on the calendar rather than achievement of test goals?
- System Life Expectancy Forecast. How long will this system be maintained and operated? Does its operational life expectancy exceed the period of development by a predictable amount of time?
- Operational Maintenance Planning. Who will maintain the operational system – the developer, user or third party?
- Software Documentation. Is it complete, accurate, and comprehensible?

2.1.1 Life Cycle Engineering Research

Interest in establishing a more complete understanding of software engineering as a discipline has led to close examination of program development and maintenance. That examination has resulted in expression of certain ideas and concepts concerning the nature of the maintenance environment and the processes which influence program evolution. When viewed from the perspective of life cycle maintenance for software these concepts and ideas

create a potential for significant improvement in the effectiveness of applied software maintenance. Perhaps the most important contribution of these theories is the insight they offer into program creation and maintenance processes. The result is a greater understanding of how program maintenance can be structured and implemented as a unified function. Continuing research in this area should produce an even clearer picture of maintenance requirements and the technology that may be applied to satisfy them.

Illustrative of the research being done are the definition of program evolution dynamics by Belady and Lehman (References 9 and 10), system partitioning evaluation by Uhrig (Reference 11), and software performance analysis by Yau and Collofello (Reference 12).

2.1.1.1 Program Evolution Dynamics. Using a large program (OS/360) as a research vehicle Belady and Lehman studied its evolution after initial release. They examined data from each version or release of the program in order to isolate and characterize the interaction between management activities and the programming process. These data included system size measures, module counts, release dates, manpower and machine usage, and costs for each release.

As expressed by Lehman (Reference 10), "The most fundamental implication of our observations is the existence of deterministic, measurable regularity in the life cycle of an application program or of a software system". That regularity is formally expressed in the authors' three laws of Program Evolution Dynamics.

- Law of Continuing Change. A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.
- Law of Increasing Entropy. The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.

- Law of Statistically Smooth Growth. Growth trend measures of global system attributes may appear to be stochastic locally in time and space, but, statistically, they are cyclically self-regulating, with well-defined long-range trends.

In the authors' view these laws function to "direct, constrain, control, and thereby regulate and smooth, the long-term (program) growth and development patterns and rates".

2.1.1.2 System Partitioning Evaluation. In his definition of a life cycle model for evaluation of system partitioning, Uhrig (Reference 11) expresses an important concept relative to operations and maintenance phase activities. The evaluation model is proposed as a means for quantitative comparison of alternative system partitioning schemes. System partitioning refers to segmenting the system into areas of responsibility such as development, test, operations and maintenance and growth. Evaluation measures in the areas of cost, schedule, performance, and risk are taken across the system life cycle to provide input data for the model. In the words of the author, "A major feature of the model is its recognition of three major dependencies: (1) development productivity on the amount of coordination required among elements (areas of responsibility), (2) test cost on the number of elements in the test configuration, and (3) reliability and maintainability on the manner in which technologies are distributed throughout the system".

In his discussion of operations and maintenance dependencies Uhrig introduces the concept of the operations and maintenance phase as an abbreviated repetition of the preceding life cycle phases. That is to say, maintenance activities routinely encompass system definition, design, code, and test tasks. Thus, the operations and maintenance phase may be looked upon from the maintenance viewpoint as a microcosm of system development. As corrective, adaptive or perfective maintenance is performed on the software these component tasks are accomplished. This concept

provides fresh illumination of the software maintenance environment and perhaps a basis for new approaches to developing an understanding of its requirements.

2.1.1.3 Software Performance Analysis. In an interim report on their research into maintenance effects on software performance, Yau and Collofello (Reference 12) present a maintenance technique for predicting those effects. Maintenance changes implemented on large, complex software systems can disturb prior states of functional and performance equilibrium. Functional disturbances existing either before or after the maintenance can be dealt with by implementing conventional techniques. Yau and Collofello propose a specialized approach for dealing with disturbances to the performance characteristics of a software system. They have defined and analyzed "mechanisms for the propagation of performance changes, performance attributes, and critical software sections". From these results they have developed the maintenance techniques. Much of the detailed information and research data supporting development of this technique remains to be published in a second report.

The purpose of their research is to develop a set of criteria which maintenance personnel may use in selecting optimum software modifications. Obviously, maintenance personnel must consider both functional and performance impact when implementing a repair or update modification to software. The technique proposed in the interim report supports determination of performance effects of software modifications, system retest after maintenance changes and prediction of worst-case effects of proposed changes. According to the authors this technique contributes to a software engineering approach to maintenance.

This report identifies eight candidate mechanisms by which software modifications cause performance changes to be propagated throughout a software system. They are:

- Parallel Execution. Modifications affect a module's capability to be executed in parallel with other modules.
- Shared Resources. Modifications disrupt timing among modules sharing resources.
- Interprocess Communication. Modifications disrupt timing of message transmission between modules.
- Called Modules. Modifications affect the performance of a module that is called by other modules.
- Shared Data Structures. Modifications alter the storage and retrieval times for data or cause saturation of the data structure to be used by multiple modules.
- Sensitivity to the Rate of Input. Modifications change input data rate leading to saturation and overflow of data structures or interruptions in processing.
- Abstractions. Modifications to modules using abstractions cause "hidden" performance changes.
- Execution Priorities. Modifications disrupt the calling sequence of modules or priority allocation.

In summary, the maintenance technique is implemented in two phases. Phase one consists of program analysis and data base production. Phase two is applied during the maintenance process using data from phase one. In outline form the steps contained in the two phases are as follows:

Phase I

Step 1. Decompose program performance requirements into key performance attributes.

Step 2. Determine propagation mechanisms present in the program.

Step 3. Identify critical sections of the program.

Step 4. Identify performance dependency relationships.

Phase II

Step 1. Identify critical sections to be affected by maintenance activity.

Step 2. Determine corresponding performance attributes affected by maintenance activity.

Step 3. Identify all performance attributes affected by changes to performance attributes in previous step.

Step 4. Identify performance requirements affected by the maintenance activity.

2.1.2 Life Cycle Maintenance Categories

In his paper "The Dimensions of Maintenance", Swanson (Reference 8) presents a discussion of a new typology for application software maintenance. He develops maintenance categories or bases, as he calls them, in a preliminary step to development of a candidate set of maintenance performance measures. These measures are proposed as elements of a maintenance data base to be established. This data base is to function as a repository of maintenance measures which will be used in research to assess the dimensions of software maintenance. Once these dimensions are known for any data processing environment then performance criteria can be established and used to promote improved maintenance management.

The implication of those maintenance bases to this review of maintenance techniques and tools is that they provide a reasonable framework within which to discuss maintenance technology application. The bases represent a commonsense approach to defining types of maintenance performed and they encompass the entire spectrum of software repair activities.

A description of the bases is presented in summary form in a table taken from the Swanson paper and shown below.

TABLE 2-1. SUMMARY - BASES OF SOFTWARE MAINTENANCE

A. CORRECTIVE

1. Processing Failure
2. Performance Failure
3. Implementation Failure

TABLE 2-1. SUMMARY - BASES OF SOFTWARE MAINTENANCE (CONCLUDED)

B. ADAPTIVE
   1. Change in Data Environment
   2. Change in Processing Environment

C. PERFECTIVE
   1. Processing Inefficiency
   2. Performance Enhancement
   3. Maintainability

2.2 MAINTENANCE ENGINEERING

Given the definition of software engineering presented in the introduction to this report, it might be of interest to refine that definition to include specialized subdisciplines. Maintenance engineering could be one of these subdisciplines.

The term maintenance engineering implies the existence of an organized body of scientific and technical information that may be applied to maintaining computer software systems. Unfortunately, state-of-the-art development is such that a true software maintenance engineering discipline does not exist. Understanding and general agreement within the data processing industry on the definition of software maintenance tasks has been achieved. A number of methodologies that may be used to improve maintenance for a variety of specialized software systems have been documented. Articles and papers, citing the rising volume and cost of operational maintenance for computer programs, call for recognition and admission of the significance of operational maintenance support today. It is apparent from a review of this literature that concern for maintenance requirements is rising and tentative, preliminary definition of maintenance engineering procedures is being attempted.

One of the first steps in the process of formalizing a maintenance engineering discipline is developing a clear understanding of computer program maintenance requirements. As stated earlier, program maintenance spans the tasks of correcting execution faults,

adapting for changed environment, perfecting to improve performance, and modifying for functional enhancement. Maintenance requirement definitions must encompass the tasks within the context of the computer program life cycle. Table 2-2 identifies a set of capability requirements which have been defined for embedded computer systems maintenance (Reference 13).

TABLE 2-2. SOFTWARE MAINTENANCE/MODIFICATION REQUIRED CAPABILITIES

- PROBLEM VERIFICATION
  - -REPRODUCE TROUBLE SITUATIONS
  - -VERIFY REPORTED SYMPTOMS
  - -IDENTIFY CAUSE: SOFTWARE, HARDWARE, INTERFACE
- DIAGNOSIS
  - -SYSTEM STATE SPECIFICATION/SEQUENCE CONTROL
  - -SOFTWARE/HARDWARE TEST POINTS ACCESS
  - -TEST DATA COLLECTION
  - -TEST DATA ANALYSIS
- REPROGRAMMING (NEW REQUIREMENTS OR SPECIFIC CORRECTION)
  - -SOURCE CODE MODIFICATION
  - -OBJECT CODE GENERATION
  - -SYSTEM RELOAD
- BASELINE VERIFICATION/REVERIFICATION
  - -SCENARIO CONTROL
  - -DATA COLLECTION
  - -DATA ANALYSIS

The requirements listed in this table could be considered as applicable to all classes of software systems. Stated in the broader context of maintenance requirements this list should also include:

- Configuration Management
  - -Program Revisions Control
  - -Baseline Configurations Documentation

Formal definition of a workable maintenance engineering discipline can only be achieved with identification of the technology and procedures supporting an integrated approach to satisfying

maintenance requirements. The compilation of typical techniques and tools along with the maintenance activities/technology correlation contained in this report provides a basis for further research into maintenance engineering. The material in this report will support data collection and analysis to identify maintenance engineering functions. This research should produce as a minimum the following:

- A minimum set of unique techniques and tools that satisfy maintenance requirements for all classes* of software systems.
- Standard procedures for application of the set of techniques and tools identified above.
- A standard set of metrics for specifying the degree of effectiveness of maintenance engineering procedures such as error removal rate, compilation ratio, fault identification rate, etc.
- A glossary of common maintenance engineering terms.

2.3 ADMINISTERING MAINTENANCE

Application of software maintenance technology is a direct result of joint efforts by the software system manager and maintainer. Each has unique functions to perform and a particular perspective on maintenance which governs his or her approach to the task of maintaining computer programs. Those functions and perspectives should be complementary to assure effective application of maintenance techniques and tools. Published literature on the subject of software maintenance application predominantly features examination of manager and management aspects of maintaining computer programs. Software maintainer concerns and responsibilities have not been accorded equal attention by

*Definition of software system classes, such as data base management, process control, operating system, etc., is a concept that must be formalized. Class differentiation in this context implies that there are varying maintenance requirements among the classes.

researchers and writers. Nevertheless, both the manager and maintainer participate jointly in the implementation of maintenance techniques and tools and both points of view are discussed here.

In his discussion of software management from the corporate level point of view , Cooper (Reference 14) includes summary descriptions of management obstacles and pitfalls. While the obstacles and pitfalls he describes address software management in general they are applicable to the specific concerns of maintenance management. The following comments focus on the specialized environment of software maintenance obstacles and pitfalls.

- Corporate decision makers' lack of computer related experience. This is a direct result of the relative newness of the entire data processing industry. For a manager overseeing software maintenance this lack of experience is often demonstrated through impatience with system limitations and intolerance for the costs of system enhancements.

- Hardware orientation of software management mechanisms. Most directives and techniques for controlling the development and maintenance of software have been adopted from hardware engineering disciplines. Thus, quality assurance, reliability and maintainability, and configuration management procedures reflect an orientation toward tangible products. Their translation for use within the environment of intangible software components has not been a completely successful one. The manager of maintenance must judiciously apply these controls in ways that tend to make each application somewhat unique to the system on which they are used.

- Excessive concern for development of software with little consideration for life cycle costs. This has significant impact on the tasks of managing and maintaining software after development. Computer programs that are developed in the most expeditious, cost-effective way to meet performance standards are not necessarily maintainable. Often the development project manager must sacrifice software design features that are conducive to program maintainability in order to meet cost, schedule or performance requirements. This leaves the user with software that is costly to maintain.

- Increased software system complexity when developed or maintained as a result of efforts to introduce state-of-the-art design, expand requirements as defined or introduce assembly language routines. Complexity is not inherently bad for maintenance if introduced in moderation and if documentation is adequate. In today's data processing environment of expanded processing and storage capabilities there is less need than ever before for complex designs and elegant code. Considering the increasing costs of software development and maintenance it makes more sense to produce straightforward program logic and code.

- Contract "buy in" for acquisition of a software system. This situation affects maintenance only indirectly as a result of the effects of any cost cutting on the part of the developer. The impact of these constraints is similar to that described previously under excessive concern for software development.

- Risk, cost, and reliability estimating deficiencies. With the exception of reliability estimating these estimation techniques do not directly influence maintenance management. Accurate reliability estimation would greatly enhance the maintenance managers effectiveness in allocating resources for program maintenance.

- Absence of common software development or maintenance practices. This places managers at all levels in the awkward position of having to learn or relearn to "read" management control data from each new system. In part, the purpose of this report is to establish a basis for identification of common maintenance practices.

Successfully coping with the obstacles and pitfalls of maintenance management requires the skilled and disciplined exercise of management controls at all levels. Software maintenance management has in the past been a "seat of the pants" operation with on the job training as the primary learning medium. This is beginning to change as more research is conducted and greater understanding of the dynamics of the management environment is achieved.

The software maintenance manager does have a growing body of information and technology available to aid in directing and controlling maintenance tasks. Unfortunately, because general awareness of the significance of maintenance is coming late to

the data processing industry there are no standards established for maintenance management. Technology and concepts must be examined and evaluated on an individual basis. Consequently, it is still necessary for a manager to piece together any management program to be implemented based on professional experience and understanding of system requirements. One writer has proposed a set of broadly defined management tools for the software maintenance activity (Reference 13). These tools, encompassing all types of software maintenance, offer a framework for structuring a maintenance management program:

- A comprehensive system/software trouble reporting system.
- A complementary set of software-oriented test procedures for operating command use.
- A mechanism for controlling and tracking operational program revisions.
- A Software Configuration Control Board (SCCB) to review and authorize changes.

Data collected through an industry wide survey of managers of systems and programming departments has provided new insight into the problems associated with application software maintenance (Reference 15). In evaluating the maintenance function most respondents characterized maintenance as being more important than new system development. Also, when those surveyed were asked to rank problem areas of maintenance, the majority indicated that user requests for system enhancements and extensions comprised the most significant problem area. Thus, maintenance is perceived to be a software manager's most important responsibility and evolutionary modifications appear to dominate other maintenance activities.

What about the software maintainer? Does he or she view maintenance as an important or challenging task? Unfortunately, the maintainer has received scant attention by researchers and

writers. There is reason to believe, however, that program maintenance personnel typically do not consider their status or their assigned tasks in a very favorable light. Gundeman (Reference 16) states that, "Traditionally, program maintenance has been viewed as a second-class activity, with an admixture of on-the-job training for beginners and of low status assignments for the outcasts and the fallen". If this is a common perception of program maintenance work it is not difficult to understand why it would be shunned by most programmers.

A number of suggestions have been offered that relate to improving maintenance programmer motivation and the environment of maintenance programming. One of the most straightforward involves exchanging the term maintenance for production, creating "production programming" (Reference 3). As the writer points out this removes the connotation of unskilled labor that is attached to maintenance and provides a link to the concept of an engineering discipline, as in software engineering, of which production programming would be a part. Of course, it would be very difficult to implement a change such as this throughout the data processing industry. However, it is certainly worth considering. Another idea presented concerns developing a set of integrated maintenance procedures and a comprehensive supporting technology for the maintenance staff (Reference 14). These procedures and technology are to be system oriented and must be defined and in place before a software system enters the operations and maintenance phase. If this is done the costly and time consuming "ad hoc" approach to program maintenance can be avoided. The information contained in Sections III and IV of this report is offered as an initial step in creating this planned maintenance approach. Finally, the concept of maintainable software has been presented as an approach to improving the program maintenance environment. If computer programs can be designed with at least some consideration being given to those characteristics that promote their maintainability then the maintenance (production) programmer's job satisfaction will undoubtedly improve.

In his discussion of major aspects of software management, Daly (Reference 17) develops an approach to managing software development that results in early introduction of the maintenance programmer to any new software. During the testing phase, prior to acceptance, new or modified computer program operation is verified by both the chief programmer (responsible for development) and the maintenance programmer (responsible for maintenance after acceptance). The chief and maintenance programmers are responsible for integration test planning and each reports on testing progress to their respective line superiors. This provides an important cross-check on testing performance. In addition the maintenance programmer is responsible for assuring that both the program code and documentation meet established standards prior to acceptance. Later, a system test will be conducted by a team composed of chief and maintenance programmers to check the interoperation of all subsystems with the new software. Early involvement by the maintenance staff produces timely program performance feedback to the design staff which facilitates the software transition from development to operational status.

Assessing the psychological complexity of understanding and modifying computer programs (one measure of the level of difficulty experienced by a software maintainer) was the goal of research conducted by Curtis, et al (Reference 18). In the study, Halstead and McCabe complexity measures and program length were used as basis for correlation of test data from two experiments using a group of professional programmers as test subjects. In one experiment the subjects were asked to reconstruct a functional equivalent for each of three programs from memory with timed periods for study and reconstruction of each. The second experiment involved completion of a specified program modification by each subject with no time limitation for completion. The study results suggest that program length and McCabe's complexity

measure may be used to predict the level of difficulty in achieving program understanding. As reported by the researchers, "All three metrics (Halstead, McCabe, and program length) correlated with both the accuracy of the modification (in experiment 2) and the time to completion. Relationships in both experiments occurred primarily in unstructured rather than structured code, and in code with no comments. The metrics were also most predictive of performance for less experienced programmers. Thus, these metrics appear to assess psychological complexity primarily where programming practices do not provide assistance in understanding the code".

Finally, instituting a policy of scheduled maintenance gives a more structured environment in which to work. Scheduled maintenance complements system version production control in that all system modifications, enhancements and corrections, are implemented in batches on a scheduled basis. Of course, emergency maintenance is still performed on a priority basis, but routine changes are introduced according to a schedule. Lindhorst (Reference 19) cites several benefits to be derived from this approach:

- Consolidation of requests. Some efficiency can be achieved because multiple changes to the same program or module can be combined under one maintenance task.
- Programmer job enrichment. The maintenance schedule should provide an opportunity for selective programmer upgrade training or career broadening assignments.
- Forces user department to think more about the changes they are requesting. Delayed implementation of new capabilities will tend to filter out those changes that will be short lived, unimportant or both.
- Periodic application evaluation. Scheduled changes provide convenient milestones for consideration of the cost effectiveness of continuing the current system.
- Elimination of the "squeaky wheel syndrome". When users realize that change requests all receive equal consideration and implementation of the changes is

on a planned basis, there is less cause for attempting to pressure the maintenance staff.

- Programmer back-up. The maintenance staff manager has more latitude in assigning his personnel to tasks and can conduct crosstraining within the maintenance teams.

- Better planning. Long and short range staff planning can be more effectively accomplished when the workload can be predicted with a reasonable degree of accuracy.

- Data processing change requests are regarded as being as important as user requests. Under this type system it is possible to give both user and change requests fair consideration when planning for the next scheduled maintenance period.

There are shortcomings and problems with instituting scheduled maintenance. First, the concept must be approved and backed by senior management. Without this support adherence to the policy of planned implementation of data processing updates and changes cannot be enforced. Additionally, the changeover from old maintenance policies to scheduled maintenance can be traumatic for the organization.

2.4 MAINTAINABLE SOFTWARE

Computer programs may be designed and coded so that it is relatively easy to isolate and correct errors or to satisfy new requirements. Such software is said to have a high degree of maintainability. To be useful as a specification criterion the characteristic of maintainability should be quantifiable and measurable. By developing maintainability measures or metrics that can be applied to all computer programs a capability for establishing maintainability standards exists. These standards could be specified for any software system before it is acquired. Not only could they be specified, but as measurable standards they could be enforced. When applied, the concept of "designing in" software maintainability should result in decreasing software maintenance costs. There are a number of systems being developed or in use that can be applied

during software design to promote maintainability in the resultant computer programs. Among these are the following systems:

- ISDOS - a computer aided technique for requirements analysis which employs the Problem Statement Language and Problem Statement Analyzer
- R-Nets - a technique for organizing software processing paths
- HOS - an axiomatic approach to definition of a software system
- FACT - a diagramatic method for functional definition of software operational structure
- DECA - a technique for organizing, validating, and portraying the design of a software system
- SADT - a vehicle for structuring and documenting the software development process.

Development of maintainable software is considered a realistic and achievable goal. One aspect of research into attaining that goal involves definition of viable software metrics and procedures for their measurement. Gilb (Reference 20) addresses this question in his book Software Metrics. Two approaches to measuring maintainability of computer programs have been reviewed for this report.

2.4.1 Software Metrics Definition

As described by Walters and McCall in Reference 7 the definition of a set of software quality metrics ultimately leads to a capability "for quantitatively specifying the level of quality required in a software product". This approach was developed as a result of a study of software quality factors. The purpose of the study was to develop guidelines for objective software quality specification in system requirements documentation. The methodology presented in Reference 7 consists of the following steps:

- Determination of software quality factors whose combined values will represent a quality rating.
- Identification of criteria to be used in rating each quality factor.
- Definition of criteria metrics and a functional relationship among them for developing a quality factor rating.
- Validation of metrics and functions using existing software system historical data.
- Translation of study results into project management guidelines.

The software quality factors identified for the study represent the most commonly accepted and desirable characteristics of software. For application in this methodology they are grouped into three sets representing their orientation toward the functional areas of product revision, transition, and operation. This facilitates expression of factor ratings in terms of user interaction with the software product. The quality factors identified are maintainability, flexibility, testability, interoperability, reusability, portability, correctness, reliability, efficiency, integrity, and usability. This discussion will focus on the maintainability factor which is associated with product revision.

Maintainability criteria are established through expansion of the definition of maintainability into specific attributes which can be objectively measured. Those criteria are consistency, simplicity, modularity, self-descriptiveness, and conciseness. Some of these criteria are also criteria for the reliability quality factor. Shared criteria are used to describe factor interrelationships and occur between other quality factors. Once the criteria are identified they are linked with specific software life cycle phases both for application of criteria metrics and for indication of when they will affect software quality.

Definition of criterion metrics is based on two considerations; they must support quantitative measurement of the criterion

and they must be accessible through available software information. Two types of metrics are defined. One type is a value measure with a given range of values and the other is a binary measure of the existence or absence of some factor. Metric units are carefully chosen and expressed as a "ratio of actual occurrences to the possible number of occurrences". The set of values representing those metrics supporting the maintainability criteria becomes the input domain of a normalization function which produces the final maintainability rating. That function is derived by applying the principles of statistical inference to the metric values and establishing appropriate mathematical relationships among the values. In the case of maintainability the final quality rating is expressed in terms of the number of man-days required to correct a software error. Development of that function is to be accomplished as a result of further research and experimentation.

Validation of the normalization function is accomplished through an iterative process of comparing predicted quality ratings with actual ratings. The authors state that with more experience in applying these metrics and more data to support further refinement of the functions, confidence in their use as predictions of software quality will grow.

2.4.2 Design by Objectives

In his paper on the subject of controlling software maintainability, Gilb (Reference 21) asserts that maintainability can be designed into programs and systems. He offers as a methodology for accomplishing this a quantitative process which he has called Design by Objectives (DbO). For those familiar with management systems, DbO resembles Management By Objectives (MBO) in that identification of quantifiable and achievable goals are the focus for both methodologies. Also like MBO, DbO offers a structured and disciplined set of procedures for achieving the goal or goals. DbO is based upon specification of software attributes that are

accessible and measurable. Manipulation of those attribute values is accomplished through application of an integrated set of design specification tools. DbO may be applied during the development process to establish control of the attainment of any system quality goals. The cited paper discusses only an application of DbO to attain a desired degree of maintainability.

Creation of a DbO program begins with definition of a set of design goals. The goals must be quantifiable particularly at the subgoal level. The goals and subgoals form a system attribute specification. This specification is documented in matrix form using subgoals and descriptive parameters. The parameters establish quantifiable levels of achievement for each subgoal. These parameter or attribute values entered into the attribute/subgoal matrix represent degrees of attainment of the subgoals. Based on system characteristics and development requirements the subgoals are subjectively prioritized.

Next a function/attribute table is created. A list of system functions which are "of interest at some stage of design" and are quantifiable in some form is identified. These functions and the attributes from the system attribute specification are entered in a matrix form which will be the function/attribute table. The elements of this matrix contain symbols that reference a coded description list of techniques. When the referenced technique or techniques are applied the result should be that the particular function will have a satisfactory amount of the indicated attribute. In order to determine the total quality of a function the attribute qualities must be summed. According to the author the summing is intuitive at this point, but application of certain engineering principles may offer a more disciplined procedure for this.

The DbO methodology is proposed as an engineering oriented approach to achieving controlled quality levels in software. Through implementation of the function/attribute table, goal

directed use of resources and techniques is realized and contribution from the use of the techniques is quantified. DbO encourages a more disciplined examination of the software design process. It offers the software developer a greater potential for more efficient use of his resources in meeting design objectives.

SECTION III
SOFTWARE MAINTENANCE TECHNOLOGY

3.1 MAINTENANCE FUNCTIONS

As has been previously established, software maintenance may be categorized as corrective, perfective or adaptive. These bases of maintenance characterize the application of techniques and tools and are an important element of their definition. Whether the maintenance is corrective, adaptive or perfective, commitment of resources is required. In order to develop a viable maintenance program and a capability for committing those resources in an effective manner, management must understand the requirements of software maintenance and how the available technology supports those requirements.

Maintenance requirements may be expressed at the highest level in terms of the functions performed in maintaining software. An important part of this review of maintenance technology was consideration of how that technology is applied to satisfy maintenance requirements. Implicit in each technique or tool description is consideration for how that technology item supports the maintenance functions.

Definition of the functions of maintenance is taken from "Software Engineering" by Boehm (Reference 4) as follows:

- "Understanding the existing software: This implies the need for good documentation, good traceability between requirements and code and well-structured and well-formatted code."
- "Modifying the existing software: This implies the need for software. hardware, and data structures which are easy to expand and which minimize side effects of changes, plus easy-to-update documentation."
- "Revalidating the modified software: This implies the need for software structures which facilitate selective retest, and aids for making retest more thorough and efficient".

3.2 MAINTENANCE ACTIVITIES

Discussion of maintenance activities will focus on the operation and maintenance phase even though maintenance is not limited to that phase. Maintenance during software development tends to be unstructured and unrecorded. Current literature suggests that delivered software is no longer viewed as a finished product with correction and update considered exceptional tasks. Now, a more realistic approach to operation and maintenance phase requirements seems to prevail among users. That approach accepts the delivered software as having attained an acceptable level of operational performance, but realizes that a potential for undiscovered operations flaws exists and that changing operational requirements will force software updates. In short, all computer systems, hardware and software, exist in an operational environment that is naturally dynamic. If systems cannot exhibit some degree of structural and logical flexibility, then their operational utility will be limited and even their continued existence will be in doubt. Users will demand systems that are more responsive to their needs and more economical to maintain.

Operation and maintenance phase activities must support all functions of maintenance. Understanding, modifying, and revalidating computer programs involves activities that focus both on the correction of existing code and the development of new code or programs. For this reason it is appropriate to include in this phase activities that are usually thought of as development phase activities. In the correlation matrix that is presented in the following subparagraph these activities appear as matrix elements. They are used to specify the particular functional orientation of each technique or tool employed in the operation and maintenance phase. Those activities are:

- Requirements Analysis
- System/Program Redesign
- Code Production

- Test and Integration
- Documentation
- Error Identification/Isolation
- Quality Assurance
- Configuration Management

Figure 3-1 illustrates the relationship between maintenance functions and the above activities. In this figure activity support of a function or functions is indicated by line(s) connecting the activity and the function(s). To emphasize that together, functions and activities are invoked by the causes and choices which motivate software maintenance, the bases of software maintenance are included in the lower portion of the figure.

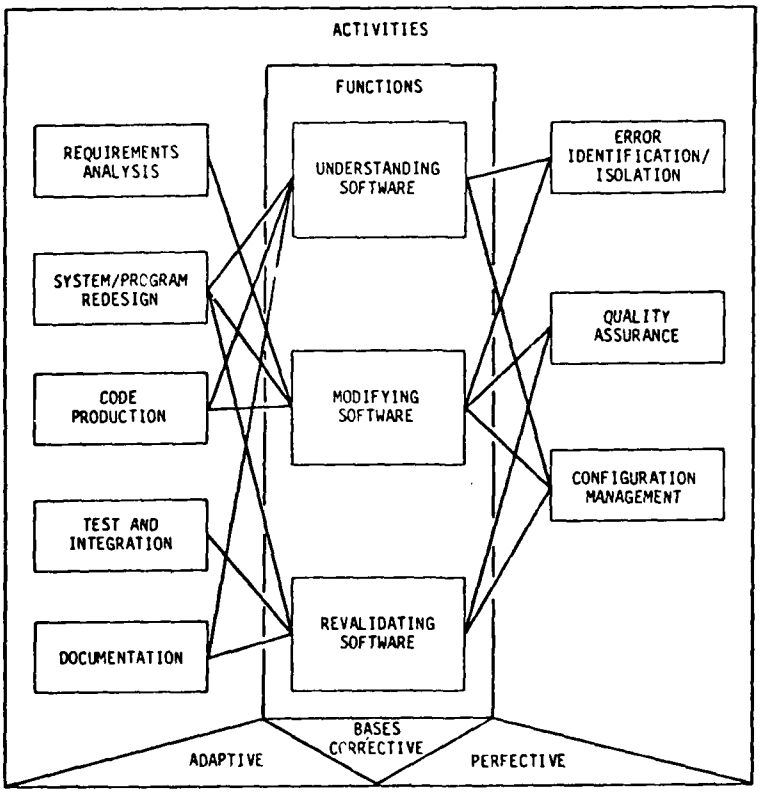


Figure 3-1. Functions - Activities Hierarchy

3.3 TECHNOLOGY/ACTIVITY MATRIX

As a preliminary step in the process of identifying and describing software maintenance tools and techniques, a set of maintenance technology functions has been defined. Each technology function, such as performance monitoring, static code analysis, path flow analysis, and test data generation is supported by one or more maintenance tools and techniques. These functions provide a logical link between the software maintenance requirements for understanding, modifying and revalidating computer programs and the description of how tools and techniques satisfy those requirements.

Correlation of maintenance technology functions with maintenance activities is depicted in Figure 3-2. The linkage between technology and activity is further refined by indicating on the correlation matrix the nature of the technology application in terms of bases of software maintenance (corrective, adaptive or perfective).

3.3.1 Maintenance Technology Categories

Maintenance technologies are grouped somewhat arbitrarily by primary function in order to avoid repetition of the same technology in several different categories. For example, static code analysis is grouped with verification and validation techniques but is also a valuable tool for redesign analysis. The complete application of each technology can be determined by reading across the matrix row for that technology. The specific tools or techniques which utilize each maintenance technology are shown in the Maintenance Tools and Techniques Applications Matrix, Figure 4-1.

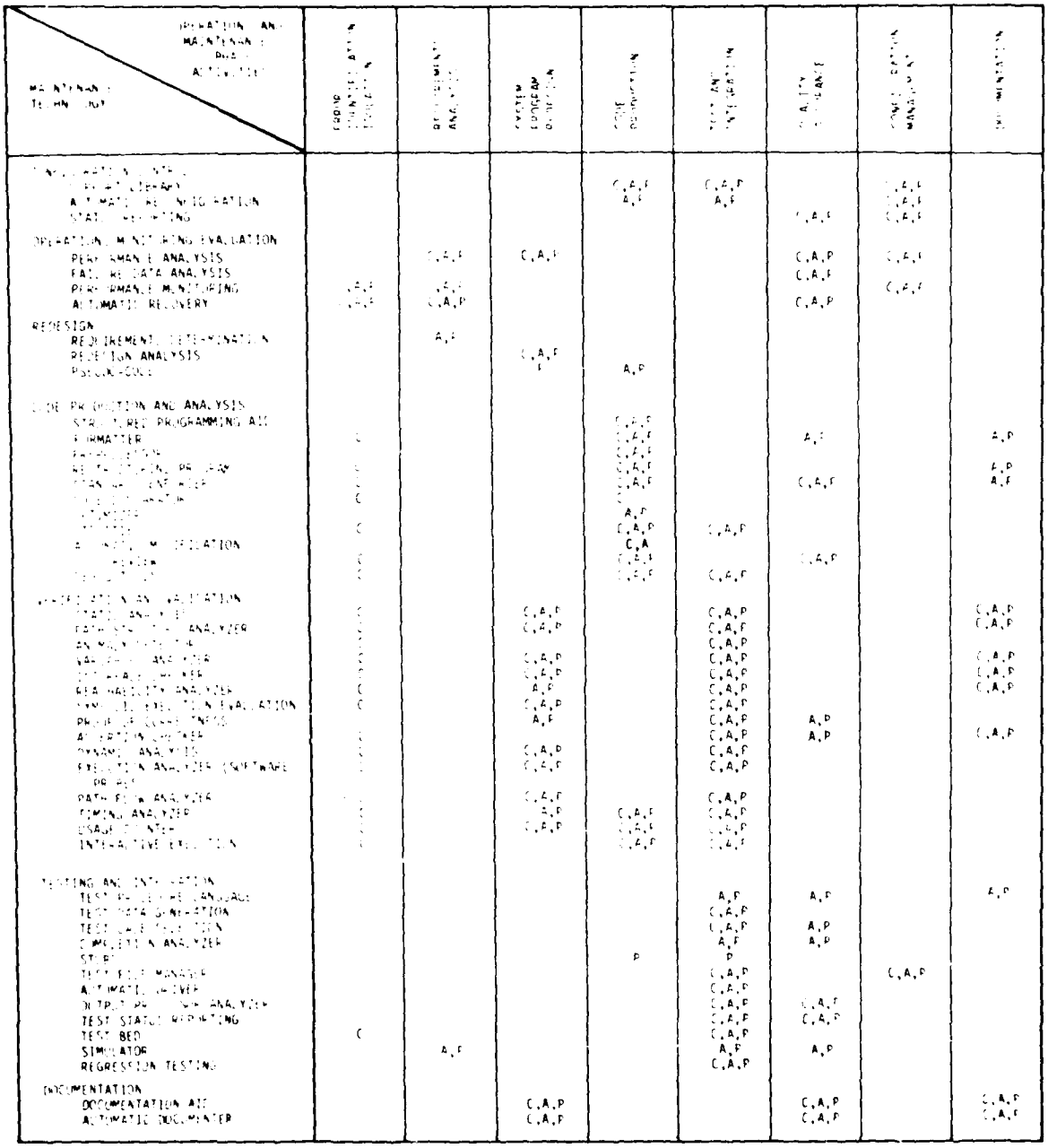


Figure 3-2. Maintenance Technology/Activity Correlation Matrix

3.3.2 Maintenance Technology Definitions

3.3.2.1 Configuration Control

a. Support Libraries

Libraries provide a constantly up-to-date representation of the system's source and object code and a past history of coding changes. Test data and test history may also be included. Libraries are useful in maintenance programming support by assuring that the modifications are made to the proper version of a program and by improving the visibility of the system for cross-checking. They can also be used as a centralized data base for such management functions as version control, test case maintenance, and project reporting.

Programs which implement support libraries are commercially available and include Applied Data Research's LIBRARIAN and International Business Machines' Program Production Library (PPL) (Reference 22).

b. Automatic Reconfiguration

This technology allows rapid reconfiguration, based on stimuli from the run-time environment, of a software system to reflect changes made to a number of its modules.

c. Status Reporting

An adequate data reporting, repository and control system provides the capability for project control by assuring that the status of all problems, deficiencies, changes and configuration data are reported to the responsible manager for analysis. These reports also assist in error isolation during the maintenance phase by providing information concerning the latest program change, such as date, time, statement numbers, person responsible, etc.

3.3.2.2 Operations Monitoring/Evaluation

a. Performance Analysis

This technology assists the analyst in determining existing systems performance characteristics for purposes of improving the performance or assessing the effect of proposed modifications. Performance analysis becomes necessary when a system is too complex to be informally or analytically understandable (Reference 23). Performance evaluation is concerned with the efficiency and effectiveness with which a computer system may perform. Performance goals are generally stated in terms of rates of work output, utilization of devices, or satisfaction of restraint conditions.

b. Failure Data Analysis

Failure data analysis is a general category used to designate technologies which measure or predict mean-time-between-failures (MTBF) or mean-time-to failure (MTTF) or otherwise quantify failure data for user analysis. The data is primarily of interest to managers for maintenance scheduling and quality assurance.

c. Performance Monitoring

Software monitors provide detailed statistics about systems performance during production, including core usage, queue lengths, program utilization, etc. The measurements can be used in tuning existing programs and resolving resource contention problems.

Hardware monitors obtain signals from the host computer system through probes attached directly to the computer's circuitry. The data is reduced to provide information about CPU utilization, channel activity, etc., which can be used to improve program and system performance.

Monitors can operate together or separately, either continuously or by sampling.

d. Automatic Fault Recovery

This experimental technology addresses the identification of logic faults as they occur during production runs and attempts to recover from the fault without halting the run by switching to backup files, using alternate program segments, etc. If automatic recovery cannot be accomplished, the run may be halted with appropriate diagnostic messages and the program made available for interactive debugging.

3.3.2.3 Redesign

a. Requirements Determination

The requirements determination technologies included in this report have been limited to technologies specifically identified in the literature as applicable to the maintenance phase. These technologies are usually directed toward the requirements for system enhancements and functional changes but they may also be needed to determine the requirements of the existing system. For example, the first step in the Yau and Collofello maintenance technique (Reference 12) is the decomposition of the existing system performance requirements into the key performance attributes.

b. Redesign Analysis

This technology assists the analyst in understanding the existing system, determining the portions of the system affected by proposed modifications, and in choosing between alternative approaches to the redesign.

c. Pseudo-Code

Pseudo-Code, or program design language, is a "pigdin" natural language with the syntax of a structured programming language. Pseudo-Code permits the quick construction of a rough outline of an entire problem solution, with more and more detail being added as needed, and an orderly transition into the actual programming language.

3.3.2.4 Code Production and Analysis

a. Structured Programming Aid

The general category "structured programming aid" is listed separately for ready identification in the Tools and Techniques Applications Matrix. The various types of aids are shown in individual categories, such as, language preprocessors, standards enforcers and pseudo-code.

b. Formatter

An automatic formatter can be used to improve the readability of a program for maintenance purposes, to assist the programmer in debugging program changes, and to aid in the production of documentation.

c. Preprocessor

A preprocessor is a computer program used to add capabilities to a system without changing the existing compiler or the system itself. Preprocessors which support structured constructs are a typical example.

Reifer (Reference 22) reports a list of well over 50 preprocessors used to extend FORTRAN for structured programming. A few representative preprocessors are described in this study for purposes of illustration.

d. Restructuring Program

A restructuring program converts an unstructured source-language program into its equivalent structured replacement. The structured program is easier to understand for maintenance purposes and structured coding itself is a recognized technique for modifying existing software (Reference 4).

e. Standards Enforcer or Auditor

A standards enforcer is a computer program used to automatically determine whether prescribed programming standards and practices for a specific program language have been followed.

Standards that can be checked for violations are program size, comments, structure, etc. The use of a standards enforcer "contributes directly toward the program's understandability and maintainability" (Reference 24).

f. Code Comparator

A comparator is used to compare two versions of the same computer program to establish identical configurations or to specifically identify changes in the source coding between the two versions (Reference 22). Boehm (Reference 4) mentions comparator programs as a tool for revalidating modified software. The use of a comparator can limit the scope of reverification that has to be performed on programs that have been modified.

g. Optimizer

An optimizer is used to improve processing efficiency by modifying code within individual modules or possibly the structure of the complete system. Optimization, in itself, is sometimes considered to be one form of system maintenance (Reference 12). Execution analysis tools characteristically provide valuable data for determining optimization requirements; however, an entry is shown in the "optimizer" column of the Tools and Techniques Applications Matrix only when the optimization function was specifically mentioned in the literature.

h. Utilities and Programming Support Tools

Utilities are computer programs employed to provide special services such as preparing card listings, creating load tapes, sorting, and plotting output results. The use of proven generalized programs reduces the probability of introducing new errors during maintenance. However, specially written utilities often reflect unique requirements and are redeveloped for every new application.

Programming Support Tools include compilers, assemblers, macro processors, linkage editors, loaders, etc., which are usually vendor-supplied.

When the use of a technology in this general category has been specifically mentioned in the literature concerning a tool or technique, the specific technology will be listed in the Tools and Techniques Applications Matrix.

i. Automatic Modification

This technology refers to methods for automatically changing a program and/or constructing different realizations of the same program.

j. Code Reading/Review

This technology refers to methods for reviewing program code, such as desk checking, structured walkthroughs and the reading of one programmer's code by another programmer. These techniques are an important complement to the use of automated tools in program maintenance.

k. Debug Tools

Debug tools are widely available, state-of-the-art tools for use in error isolation and determining the cause of errors. A complete range of debug tools are commonly available as vendor-supplied software. Finfer (Reference 25) states "there appears to be no new or unique (software) debugging tool concepts within the past 10-12 years" but "there is an estimated 10-year gap in the level of sophistication between the software support offered by manufacturers of large scale machines and that offered for minicomputers, a similar gap separates mini and microcomputers."

Debug tools specifically mentioned in the individual technology descriptions are named in the "debug tools" column of the Tools and Technology Applications Matrix. Standard debug tools useful in the maintenance phase include:

(1) Cross-Reference Program

The cross-reference generates information on system components in relation to other programs, macros, parameter names, etc. and provides a static description of data access. In addition

to their debug function, cross-references are useful in assessing the impact of program changes. In the Tools and Techniques Applications Matrix, cross-reference which identify data relationships are shown as "variables analyzers" and those identifying module relationships are shown as "interface checkers."

(2) Editor

The editor analyzes source programs for coding errors and extracts information that can be used for checking relationships between sections of code.

(3) Dumps/Displays

Dumps provide program/system status and selected data values as requested by the user. Dump/display techniques include:

- Post-mortem dumps which provide the static state of the instructions and data at a particular point in time when the program is not executing.
- Snapshot dumps which provide the dynamic state of data values while the program is executing.
- Breakpoint dumps which display the requested data when a certain location or condition is met.
- Programmed-in dumps which provide the dynamic state of data items as requested in print statements coded by the programmer.

Zelkowitz (Reference 5) points out that standard dumps are not very effective debug tools because they provide much data with little or no interpretation.

(4) Traces

Traces display state information based on the sequence of program operation during execution. Traces show the step-by-step operations of the execution, including traversed paths, transfers of control and selected memory contents.

(5) Computer Emulator

An emulator is programmed to interpretively execute the instruction set of the target computer on a host computer, enabling the execution of programs written for another system.

(6) Computer/Device Simulators

A simulator is programmed to simulate the target computer's instruction execution, the interfacing computers in a network system, terminals, or the functions of peripheral equipment. The simulator provides the system with inputs or responses that resemble those of the device being simulated. Simulation of microarchitecture on a host machine is a technique used in applying debug tools to microprograms.

(7) Breakpoint/Traps

These features are associated with the interruption of a program's execution for the initiation of a debug activity. Breakpoints are interruptions generated when certain program locations or statements are reached. Traps are interruptions generated when specific conditions are met, regardless of the location within the program.

(8) Checkpoint/Restart or Save/Restore

These tools allow the capability to save an image of execution status at any point during program operation. The saved data may subsequently be reloaded/restored to initiate processing from the checkpoint position.

(9) Interactive Modification

These tools provide a means for dynamically altering program states such as memory locations, data content, program instructions and execution flow. Interactive debugging systems represent a major advance in debugging methodology for program development; however, unless the system is specifically designed with interactive debugging facilities (See automatic recovery,

3.3.2.2.d), most problems encountered in the operational system must be debugged using post-mortem data or special runs reconstructed to display errors in the interactive environment.

(10) Graphic Output

Graphic display can be used to present decomposed versions of the software which can simplify the understanding of the program logic or graph-like lists can be produced on the printer.

3.3.2.5 Verification and Validation

a. Static Analysis

Static analysis is a technique for examining the program source code and analyzing the program structure without actually executing the code. Various types of static analyzers are listed separately in the Tools and Techniques Applications Matrix in order to identify the specific static analysis performed by each individual tool or technique. Static analysis can be applied to error isolation, program revalidation, documentation and redesign analysis in the maintenance phase. Yau and Collofello (Reference 12) note that "a complete static analysis of the implemented program is necessary for a more precise prediction of performance changes resulting from software modification."

b. Path Structure Analyzer

Path structure analyzers are computer programs used to examine source code and identify the program paths and control flow. Structural flaws, such as improper loop nesting, may also be identified during the analysis. Directed graph analysis is a commonly used technique for path analysis, with the nodes representing statements or sequences of statements and the edges representing program flow of control.

c. Anomaly Detector

Anomaly detectors search the program source code for syntactically correct but logically suspicious constructs which

are likely to cause problems when the program is executed. The detection programs identify potential error conditions not normally detected by compilers, such as the referencing of uninitialized variables.

d. Variables Analyzer

A variables analyzer is a tool or technique which determines the nature of the data structure or data flow of a program. A static variables analyzer provides information concerning the definition and use of the individual variables in the program. A dynamic variables analyzer provides information concerning the actual values of variables, such as maximum and minimum values, during execution of the program.

e. Interface Checker

Interface checkers are static analysis programs used to automatically identify program interfaces (such as called and calling modules or modules involved in interprocess communications) and to check the range and limits of the module parameters. Interface checkers are valuable tools for analyzing the impact of one module's modifications on other modules and for identifying the use of data abstractions (from subroutine calls, function calls and macros) for redesign analysis.

f. Reachability Analyzer

Reachability analyzers identify the specific program paths exercised in order to reach a specific module, subroutine or section of code within a system and may also identify unreachable modules and "dead" code.

g. Symbolic Execution/Evaluation

Symbolic execution of a program is carried out by assigning dummy symbolic values rather than actual numeric values to all or some of the input variables of a program. It provides a capability to express paths in terms of all necessary conditions to be satisfied in selecting the path. The output

of the symbolic execution, in its simplest form, is a formula which describes the program path and the computations used to derive the output value. The results may also be used to develop a minimum set of test cases. At the present time, symbolic execution techniques are being used experimentally in software research. Some characteristics of symbolic execution potentially applicable to the maintenance phase are:

- The evaluation of program paths assists the user to understand a program when the actions of the program are not immediately obvious from casual reading.
- Isolation of paths or modules provides the capability for user analysis before and after program modification. The entire program may also be evaluated before and after modifications to ensure that unmodified paths have not been affected.
- One symbolic execution may verify the program over a large subset of the input domain, encompassing a potentially infinite number of test runs with different input values.

A disadvantage of the symbolic execution technique is that the output symbolic representations can be very complex and may be too long to be meaningful.

h. Proof of Correctness

Proof of correctness refers to the techniques of proving programs are correct by means similar to those used in proving mathematical theorems. The axioms and theorems derived are used to establish the validity of program assertions; that is, that the program implementation satisfies the program specification. Research is being pursued in several approaches to correctness proving. However, Reifer and Trattner (Reference 22) observe that the proof of correctness approach will probably not be used operationally until automated aids are perfected which generate the verification conditions, do proof checking, formula simplification and editing, or interactively generate symbol transformations.

i. Assertion Checker

Assertion checking techniques allow the programmer to express his validation requirements in a way that reflects the program's intended function. Assertions concerning the value or condition of program variables are inserted at various points in the program code. The assertion checker program compares the assertions with the results derived by symbolic or dynamic analysis of the program.

The assertion checking technique addresses understanding the program behavior rather than proving mathematical properties. A side effect of assertion checking is the documentation of the program's critical requirements as stated by the assertions. This documentation enhances the understandability and maintainability of the program.

Assertion checking may also be applied to error detection activities, for example, dynamically checking critical parameters for range, value, and order violations based on the prescribed bounds of the assertions. As an error isolation methodology, assertion checking "is seen (Reference 25) to work best with well-constructed software, given an experienced analyst, and a feasible application problem in an accepted computing environment."

j. Dynamic Analysis

Dynamic analysis is a technique for monitoring the behavior of a program at run time and is dependent on the input data used to exercise the program. If the execution history is stored on a data base, a backtracking capability may also be provided. Various types of dynamic analyzers (such as path flow analyzers and execution analyzers) are listed separately in the Tools and Techniques Application Matrix so that the specific dynamic analysis performed by each tool or technique can be readily identified. Dynamic analysis is an important tool

for error isolation, redesign analysis, optimization and modification validation/testing in the maintenance phase and is useful in developing regression testing strategies.

k. Execution Analyzer (Software Probes)

Execution analyzers instrument the source code by generating and inserting counters at strategic points to provide measures of test effectiveness. Their basic function is to gather run time statistics, such as execution times and counts, which give insight into program behavior. Most of the tools provide a control language that allows the user to communicate with the tool and to instrument only selected code segments or options. A disadvantage of execution analyzers is that additional computer core and execution time is required for their use.

Timing analyzers and usage counters are listed as separate categories so that the type of execution analysis performed by the individual tool or technique can be easily identified from the Tools and Techniques Applications Matrix.

l. Path Flow Analyzer

A path flow analyzer records program paths followed through test executions and may also force execution of branches to determine program flow. This technique is suitable for detecting sequencing and control errors.

m. Timing Analyzer

Timing analyzers record the actual timing data related to program test execution, such as total execution time, time spent in each module, function, routine, subroutine, etc.

n. Usage Counter

Usage counters count the number of times that statements, modules, etc. are exercised during a test execution. Heavily used code is a candidate for manual optimization and chronically unexecuted code is often symptomatic of program logic

errors (Reference 26). The measurements are also useful for balancing test cases and detecting unexpected program path use (Reference 20).

o. Interactive Execution

During interactive execution, the user monitors the test/validation run from an interactive terminal and may select program paths (branches) or assign values to variables at various points in the program.

3.3.2.6 Testing and Integration

a. Test Procedure Language

A test procedure language is used to specify formal executable test cases to be applied to a program or program module. The language provides a basis for standardizing test specifications, documenting test activities and analyzing regression tests.

b. Test Data Generation

Test data generators produce test data to exercise the target program. The purpose is to relieve the user of the tedium of manually generating a large volume of data. The data may be created using statistical algorithms, random number generators, or test scenarios input by the user. The test data is generated in the format required by the program to be tested.

Tools which attempt to generate sample test data by program path analysis are shown in the Tools and Techniques Applications Matrix as aids to test data generation. The test data must be augmented by test data prepared by the user. The advantage of these tools is that the data is unbiased and may initiate the testing of program paths previously overlooked.

c. Test Case Selection

The usual goal of the test case selectors which are based on control structure analysis is to exercise all statements and/or all branches in a program. Techniques for selecting

a representative sample of test cases (or program paths) are the subject of much of the current research in program testing. In the maintenance phase, the objective is to determine the minimum selection of test cases to ensure that modifications are correct and that new errors have not been introduced.

d. Completion Analyzer

Completion analyzers provide data that shows how thoroughly the source code has been exercised during the testing, in relation to the testing goals that have been set.

e. Stubs

Stubs are used in top-down development to represent program modules or elements that have not yet been coded. In the object program, the stubs satisfy the control passing functions depicted in the tiered hierarchy and may also model the consumption of computer resources, such as memory space and processor time. In the maintenance phase, stubs can be used to analyze and test the interface for new modules or program segments before coding of the modules is complete.

f. Test File Manager

Test file managers are used to update and monitor test data and allow easy manipulation of generated test files. These tools are particularly useful in testing systems that are data base driven (Reference 24).

g. Automatic Driver

Automatic drivers are used to run tests in a controlled manner, including such functions as initiating input from test files, calling appropriate processing modules and monitoring processing times. The comparison of test results to predetermined results and test status reporting are sometimes included as additional functions of the driver. Automatic drivers facilitate test repetitions during regression testing and are useful in revalidating module interfaces after program modifications or the addition of new modules.

h. Output Processor/Analyzer

The test output processor is used to perform test output data reduction, formatting and printing. Test output analyzers perform a statistical analysis of the output or compare the test output to a predetermined set of values.

i. Test Status Reporting

Test status reporting includes management information such as identification of test cases, counts of test runs per case, number of successful runs, degree of test coverage achieved, etc.

j. Test Bed (Environment Simulator)

A software test bed is a computer program that simulates actual hardware and interfaces, independently from the application system, permitting evaluation of hardware/software interfaces, control of input/output, analysis of actual timing characteristics, and full test repeatability. The input data is usually precalculated and can be re-used any number of times.

The technique is applicable to adaptive maintenance and the central maintenance of systems which are operational in more than one environment.

k. Simulator (Function Simulator)

A simulator is a computerized model of a system (or process) used to conduct experiments for the purpose of understanding the behavior of the system or evaluating alternative strategies for operation of the system. The technique is used to study specific system characteristics, including performance, capabilities and constraints, over a period of time under a variety of conditions. Simulators are usually highly specialized and applicable only to the system for which they were developed (Reference 24) and may become expensive in terms of manpower and computer time (Reference 27).

Simulation techniques can be used during the maintenance phase to evaluate the effects of changes. These techniques help isolate some of the errors introduced during the change process and their side effects. Simulations can also be used to help predict how the system will react to modified configurations and alternative loads.

1. Regression Testing

Regression testing (retesting after a change) is an essential activity in the maintenance phase of the life cycle and poses a significant problem because of the high cost of retesting the entire software system (Reference 12). For this reason, regression testing is categorized separately in this report and tools and techniques which specifically address any aspect of regression testing can be immediately identified from the Tools and Techniques Applications Matrix.

3.3.2.7 Documentation

a. Documentation Aid

For this report, techniques that assist in the documentation process (according to statements by the user or developer) but do not actually produce documentation are categorized as documentation aids.

b. Automatic Documenter

Munson (Reference 1) states "one of the current problems with post-delivery maintenance of software is the quality of the documentation." Automatic, usable documentation for future maintenance is a very important maintenance tool. For this report, techniques that produce reports or listings which are usable as documentation without any further revision (according to the user or developer) are classified as automatic documenters. The referenced documentation may be a formal deliverable or an item recommended for inclusion in a "maintenance" workbook for reference in future modification/maintenance activities.

Static analysis tools characteristically produce documentation useful to program maintenance; however, entries are shown in the "documenter" column of the Tools and Maintenance Applications Matrix only when the documentation function of the tool was specifically mentioned in the literature.

SECTION IV
MAINTENANCE TOOLS AND TECHNIQUES

4.1 TOOLS/TECHNIQUES APPLICATIONS MATRIX

The Tools and Techniques Applications Matrix, Figure 4-1, summarizes the literature survey conducted for this report and shows the specific maintenance applications of each tool or technique. Techniques are defined as practices and procedures used in the development and maintenance of software systems. Tools are defined as computer programs which perform tasks which would be tedious or impractical to do manually. The individual tools or techniques shown in the rows of the matrix are described in detail in paragraph 4.2.

The columns of the matrix identify the maintenance applications of the tool or technique as specified in the literature. An entry of "U" indicates that this application of the tool or technique was discussed in the literature. The primary application of the tool or technique is identified with an asterisk (*). An entry of "M" indicates that the application was mentioned in reference to program modifications or software maintenance. General applications such as "static analysis" are shown for easy identification, as well as the specific application, such as "interface checker." An entry of "R" indicates that the user or developer has recommended that the tool or technique be expanded to include this application. An entry of "E" indicates that this application is partially implemented or experimental. An entry of "T" shows that the user or developer tried this application of the tool or technique but was not satisfied with the results. An entry of "A" shows that the tool or technique aids the user in performing the application manually but does not provide complete support for the application.

The additional applications that are possible for any given maintenance technology are shown in the Maintenance Technology/ Activity Correlation Matrix, Figure 3-2. For example, a tool may

APPLICATIONS / MAINTENANCE TOOLS AND TECHNIQUES	SOURCE	STATUS	LANGUAGE/ SYSTEM	CONFIGURATION CONTROL			OPERATIONS MONITORING/EVALUATION				REDESIGN			CODE PRODUCTION AND ANALYSIS										DEBUG TOOLS
				SUPPORT LIBRARY	AUTOMATIC RECONFIGURATION	STATUS REPORTING	PERFORMANCE ANALYSIS	FAILURE DATA ANALYSIS	PERFORMANCE MONITORING	AUTOMATIC RECOVERY	REQUIREMENTS DETERMINATION	REDESIGN ANALYSIS	PSEUDO-CODE	STRUCTURED PROGRAMMING AID	FORMATTER	PREPROCESSOR	RESTRUCTURING PROGRAM	STANDARDS ENFORCER	CODE COMPARATOR	OPTIMIZER	UTILITIES/PROGRAM SUPPORT	AUTOMATIC MODIFICATION	CODE READING/REVIEW	
ASSET 4.2.28	BCS	METHODOLOGY PROPOSED COMPONENTS BEING DEVELOPED	NOT SPECIFIED	U M		U M					U M	U* M	U M								S M			
ATDG 4.2.19	TRW	EXPERIMENTAL PROTOTYPE IMPLEMENTED	FORTRAN UNIVAC 1110 INTERACTIVE																					
ATTEST 4.2.3	CLARKE (U. OF MASS.)	EXPERI-MENTAL RESEARCH TOOL	ANSI FORTRAN																					
CASEGEN 4.2.33	RAMA-MOORTHY	PROTOTYPE IMPLEMENTED	FORTRAN																					
CCS 4.2.13	BELL LABS	OPERATIONAL	NO. 1 ESS LANGUAGE INDEPENDENT	U M*		U M					U M								U M		S M		U M	
CSPP 4.2.11	RADC	OPERATIONAL (MULTIPLE SITES)	COBOL											U*		U								
DATFLOW 4.2.39	ALLEN & COCKE	OPERATIONAL	FORTRAN																					
DAVE 4.2.20	UNIV. OF COLORADO	PROTOTYPE	FORTRAN IBM CDC															A						
DISSECT 4.2.4	NAT'L BUREAU OF STANDARDS	IN USE-RESEARCH TOOL	ANSI FORTRAN PDP-10																					
EFFIGY 4.2.1	KING (IBM)	EXPERI-MENTAL RESEARCH TOOL	PL/I SUBSET IBM 370																	R				TRACE, BREAK

U(USED) - THE TOOL OR TECHNIQUE USES THIS TECHNOLOGY

*INDICATES PRIMARY FUNCTION OF TECHNIQUE/TOOL

M(MAINTENANCE) - AN APPLICATION TO THE MAINTENANCE PHASE IS MENTIONED IN THE LITERATURE

R(RECOMMENDED) - THE USER OR DEVELOPER RECOMMENDS OR PLANS THE EXPANSION OF THE TOOL OR TECHNIQUE TO INCUDE THIS TECHNOLOGY

T(TRIED AND DISCARDED) - THE USER OR DEVELOPER TRIED THIS APPLICATION OF THE TOOL OR TECHNIQUE BUT WAS NOT SATISFIED WITH THE RESULTS

E(EX

CODE PRODUCTION AND ANALYSIS											VERIFICATION AND VALIDATION													TESTING AND INTEGRATION											DOCUMENTATION	
STRUCTURED PROGRAMMING AID	FORMATTER	PREPROCESSOR	RESTRUCTURING PROGRAM	STANDARDS ENFORCER	CODE COMPARATOR	OPTIMIZER	UTILITIES/PROGRAM SUPPORT	AUTOMATIC MODIFICATION	CODE READING/REVIEW	DEBUG TOOLS	STATIC ANALYSIS	PATH STRUCTURE ANALYZER	ANOMALY DETECTOR	VARIABLES ANALYZER	INTERFACE CHECKER	REACHABILITY ANALYZER	SYMBOLIC EXECUTION/ EVALUATION	PROOF OF CORRECTNESS	ASSERTION CHECKER	DYNAMIC ANALYSIS	EXECUTION ANALYZER (SW PROBES)	PATH FLOW ANALYZER	TIMING ANALYZER	USAGE COUNTER	INTERACTIVE EXECUTION	TEST PROCEDURE LANGUAGE	TEST DATA GENERATION	TEST CASE SELECTION	COMPLETION ANALYZER	STUBS	TEST FILE MANAGER	AUTOMATIC DRIVER	OUTPUT PROCESSOR/ANALYZER	TEST STATUS REPORTING	TEST BED	SIMULATOR	REGRESSION TESTING	DOCUMENTATION AID	AUTOMATIC DOCUMENTER
							S M				M	U					U M			M	U															U	
											U	U	U	U													A	U*									
											U	U					U										U*			U							
												)					U										U*	U									
			U M				S M			U M																										
U*		U																																		
											U	U		U*		U																				
				A							U	U	U*	U	U										R										PROG. DOC.	
											U	U					U*	T	U						T		A								TEST DOC.
						R				S TRACE, SAVE, BREAKPOINT	U	U					U*	E	U						U			R								

T(TRIED AND DISCARDED) - THE USER OR DEVELOPER TRIED THIS APPLICATION OF THE TOOL OR TECHNIQUE BUT WAS NOT SATISFIED WITH THE RESULTS

E(EXPERIMENTAL) - THIS APPLICATION OF THE TOOL OR TECHNIQUE IS PARTIALLY OR EXPERIMENTALLY IMPLEMENTED

S(STANDARD) - WHEN REFERRING TO THIS TECHNOLOGY (EITHER "UTILITIES" OR "DEBUG TOOLS"), SUCH TERMS AS "THE STANDARD SET" OR "THE USUAL ARRAY" WERE USED IN THE SOURCE ARTICLE. (SEE PARA. 3.3.2 FOR FURTHER DEFINITION)

A(AID) - THIS APPLICATION IS NOT AUTOMATED IN ITSELF, BUT THE TOOL OR TECHNIQUE PROVIDES INFORMATION TO ASSIST THE USER IN PERFORMING THE APPLICATION MANUALLY

Figure 4-1. Maintenance Tool & Techniques Applications Matrix

APPLICATIONS / MAINTENANCE TOOLS AND TECHNIQUES	SOURCE	STATUS	LANGUAGE/ SYSTEM	CONFIGURATION CONTROL			OPERATIONS MONITORING/EVALUATION				REDESIGN			CODE PRODUCTION AND ANALYSIS										
				SUPPORT LIBRARY	AUTOMATIC RECONFIGURATION	STATUS REPORTING	PERFORMANCE ANALYSIS	FAILURE DATA ANALYSIS	PERFORMANCE MONITORING	AUTOMATIC RECOVERY	REQUIREMENTS DETERMINATION	REDESIGN ANALYSIS	PSEUDO-CODE	STRUCTURED PROGRAMMING AID	FORMATTER	PREPROCESSOR	RESTRUCTURING PROGRAM	STANDARDS ENFORCER	CODE COMPARATOR	OPTIMIZER	UTILITIES/PROGRAM SUPPORT	AUTOMATIC MODIFICATION	CODE READING/REVIEW	DEBUG TOOLS
FACES 4.2.31	RAMAMOORTHY	OPERATIONAL	FORTRAN UNIVAC 1108 CDC 6400 IBM 360/65															U						CROSS REF
FADEBUG-I 4.2.14	FUJITSU LTD.	OPERATIONAL (*BEING* EXTENDED)	ASSEMBLER LANGUAGE FACOM 230-60																		S FACOM OS			
FAST 4.2.27	BROWNE AND JOHNSON	METHODOLOGY PROPOSED PROTOTYPE IMPLEMENTED	FORTRAN INTERACTIVE									U M												
FAVS 4.2.7	RADC	OPERATIONAL	FORTRAN HIS 6180														U							BREAKPOINT, GRAPHIC OUTPUT, TRACE, CROSS-REFERENCE, DUMP
FORAN 4.2.9	ARC	OPERATIONAL	FORTRAN CDC 6400																					CROSS REF
FORTRAN CODE AUDITOR 4.2.8	RADC/TRW	OPERATIONAL	FORTRAN HIS 600/6000											U				U*		A				
FORTRAN STRUCTURING ENGINE 4.2.22	CAINE, FARBER GORDON, INC.	OPERATIONAL	FORTRAN IBM 360/370											U										
ISMS 4.2.35	FAIRLEY	EXPERIMENTAL IMPLEMENTATION	ALGOL 60 FORTRAN																					
JAVS 4.2.5	RADC/GRC	OPERATIONAL (MULTIPLE SITES)	JOVIAL (J3) HIS 6180 CDC 6400																	A				BREAKPOINT, TRACE, GRAPHIC OUTPUT, CROSS-REFERENCE, DUMP
JOYCE 4.2.25	MDAC	OPERATIONAL	FORTRAN CDC																					CROSS REF

U(USED) - THE TOOL OR TECHNIQUE USES THIS TECHNOLOGY

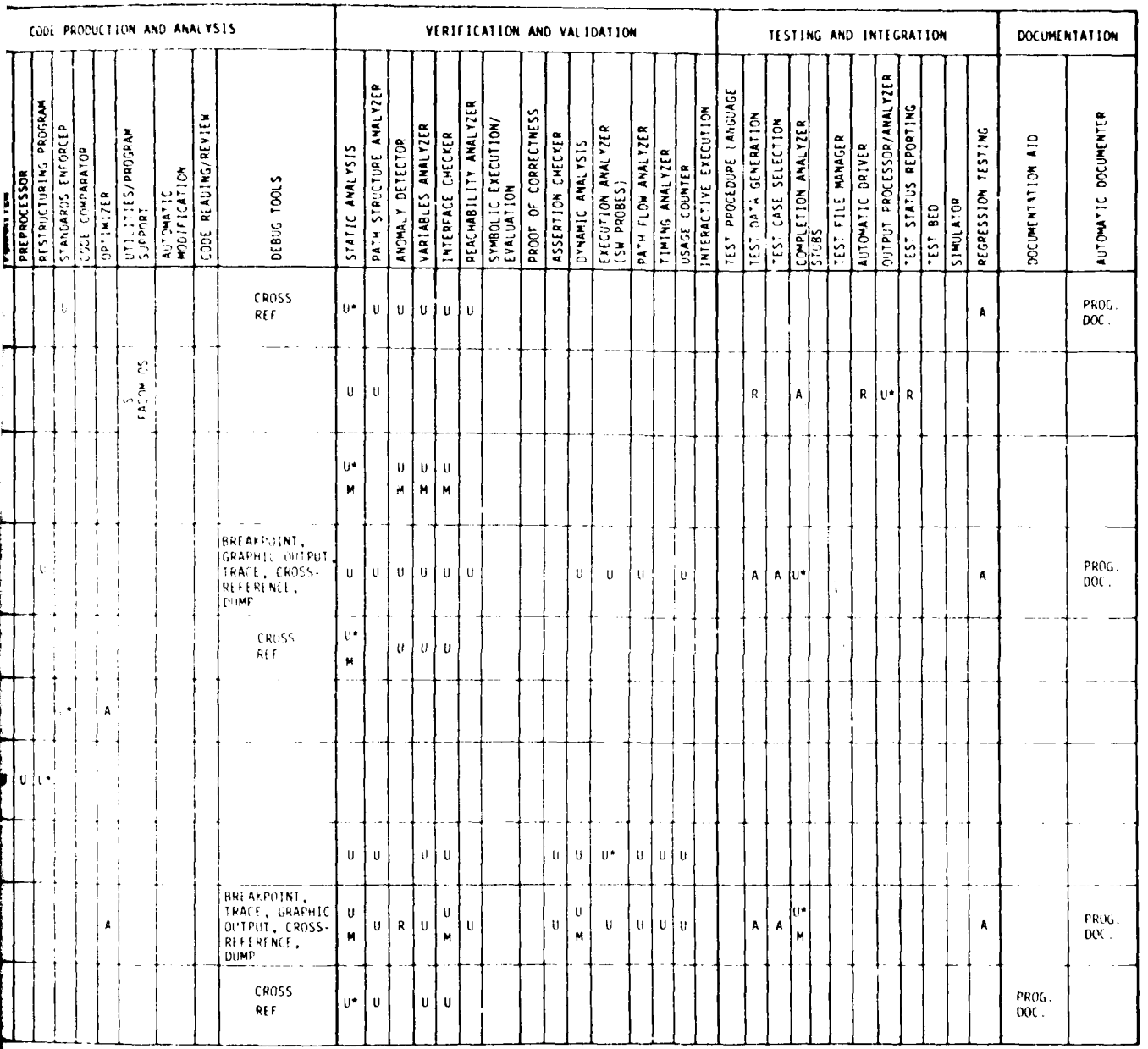
M (MAINTENANCE) - AN APPLICATION TO THE MAINTENANCE PHASE IS MENTIONED IN THE LITERATURE

R(RECOMMENDED) - THE USER OR DEVELOPER RECOMMENDS OR PLANS THE EXPANSION OF THE TOOL OR TECHNIQUE TO INCLUDE THIS TECHNOLOGY

T(TRIED AND DISCARDED) - THE USER OR DEVELOPER TRIED THIS APPLICATION OF THE TOOL OR TECHNIQUE BUT WAS NOT SATISFIED WITH THE RESULTS

E(EXPERIMENTED) -

*INDICATES PRIMARY FUNCTION OF TECHNIQUE/TOOL



AND DISAGREES) - THE USER OR DEVELOPER TRIED THIS APPLICATION OF THE TOOL OR TECHNIQUE BUT WAS NOT SATISFIED WITH THE RESULTS

E(EXPERIMENTAL) - THIS APPLICATION OF THE TOOL OR TECHNIQUE IS PARTIALLY OR EXPERIMENTALLY IMPLEMENTED

S(STANDARD) - WHEN REFERRING TO THIS TECHNOLOGY (EITHER "UTILITIES" OR "DEBUG TOOLS"), SUCH TERMS AS "THE STANDARD SET" OR "THE USUAL ARRAY" WERE USED IN THE SOURCE ARTICLE. (SEE PARA. 3.3.2 FOR FURTHER DEFINITION)

A(AID) - THIS APPLICATION IS NOT AUTOMATED IN ITSELF, BUT THE TOOL OR TECHNIQUE PROVIDES INFORMATION TO ASSIST THE USER IN PERFORMING THE APPLICATION MANUALLY

Figure 4-1. Maintenance Tool & Techniques Applications Matrix (Continued)

APPLICATIONS / MAINTENANCE TOOLS AND TECHNIQUES	SOURCE	STATUS	LANGUAGE/ SYSTEM	CONFIGURATION CONTROL			OPERATIONS MONITORING/EVALUATION				REDESIGN			CODE PRODUCTION AND ANALYSIS										
				SUPPORT LIBRARY	AUTOMATIC RECONFIGURATION	STATUS REPORTING	PERFORMANCE ANALYSIS	FAILURE DATA ANALYSIS	PERFORMANCE MONITORING	AUTOMATIC RECOVERY	REQUIREMENTS DETERMINATION	REDESIGN ANALYSIS	PSEUDO-CODE	STRUCTURED PROGRAMMING AID	FORMATTER	PREPROCESSOR	RESTRUCTURING PROGRAM	STANDARDS ENFORCER	CODE COMPARATOR	OPTIMIZER	UTILITIES/PROGRAM SUPPORT	AUTOMATIC MODIFICATION	CODE READING/REVIEW	DEBUG TOOLS
LIBRARIAN 4.2.40	ADR	OPERATIONAL	IBM 360/70	U* M		U M																	EDITOR
MPP 4.2.10	VARIOUS	OPERATIONAL	VARIOUS	U M									U	U* M									U M	
NUMERICAL SOFTWARE TESTBED 4.2.37	HENNELL ET AL	OPERATIONAL	FORTRAN ALGOL												U									
OPTIMIZER II 4.2.32	CAPEX	OPERATIONAL	COBOL IBM 360/370																U*					
PACE 4.2.23	TRW	OPERATIONAL	FORTRAN CDC 7600 CDC 6500 UNIVAC 1108																A					
PERFORMANCE/ MAINTENANCE ALGORITHMS 4.2.12	YAU & COLLOFELLO	BEING DEVELOPED	LARGE SCALE SYSTEMS				U M*				U M	U M												
PET 4.2.18	MDAC	OPERATIONAL	FORTRAN IBM, CDC, HONEYWELL, UNIVAC															U	A					
PFORT VERIFIER 4.2.21	BELL LABS	OPERATIONAL	FORTRAN IBM, CDC, HONEYWELL, BURROUGHS, OTHERS															U*						CROSS REF
TPE 4.2.17	BOOLE & BABBAGE, INC.	OPERATIONAL	LANGUAGE INDEPENDENT IBM						U* M										A M					
RELIABILITY MEASUREMENT MODEL 4.2.30	MUSA (BELL LABS)	OPERATIONAL	MOST LARGE COMPUTER SYSTEMS					U* M																

U(USED) - THE TOOL OR TECHNIQUE USES THIS TECHNOLOGY

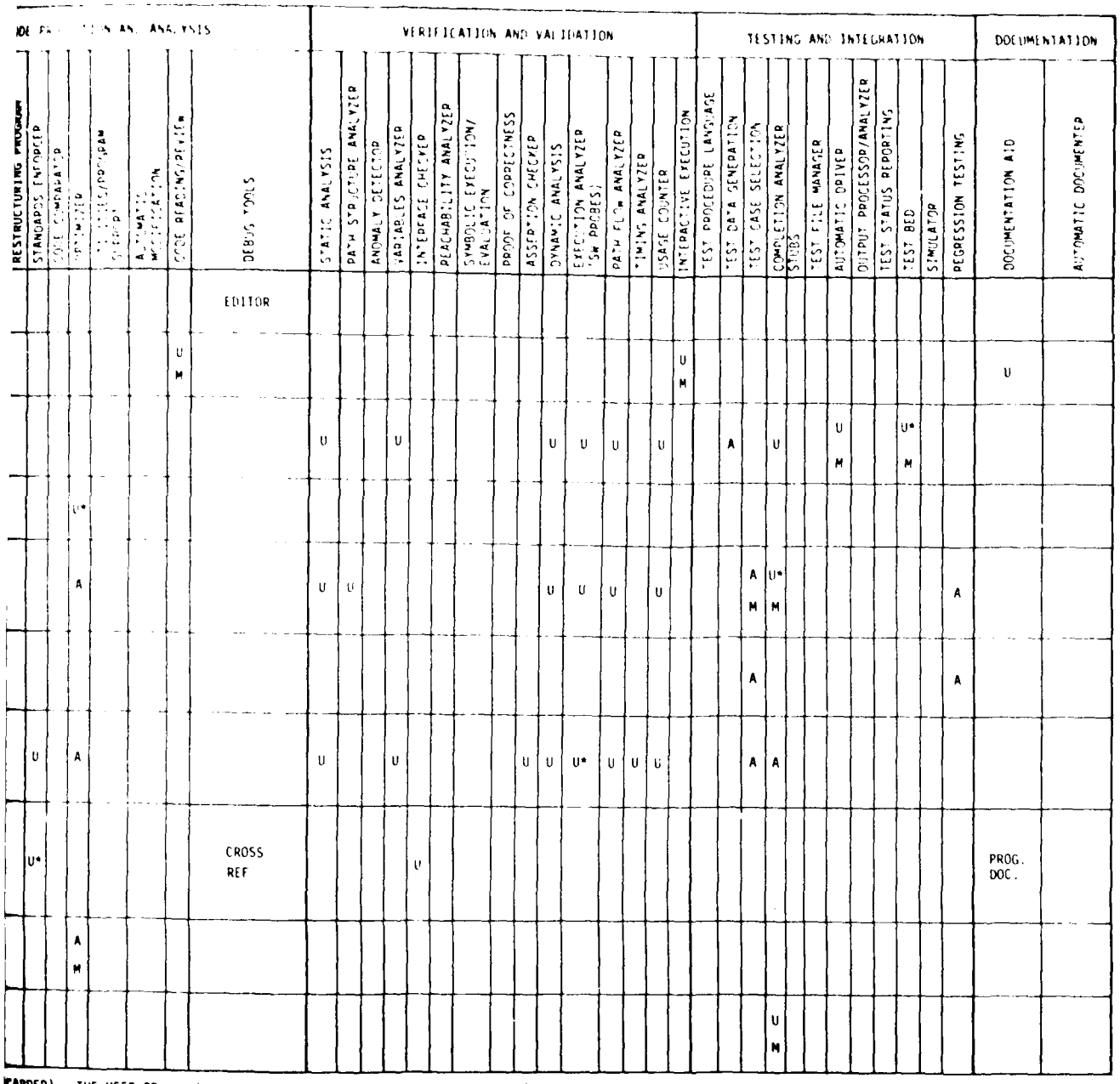
M(MAINTENANCE) - AN APPLICATION TO THE MAINTENANCE PHASE IS MENTIONED IN THE LITERATURE

R(RECOMMENDED) - THE USER OR DEVELOPER RECOMMENDS OR PLANS THE EXPANSION OF THE TOOL OR TECHNIQUE TO INCLUDE THIS TECHNOLOGY

T(TRIED AND DISCARDED) - THE USER OR DEVELOPER TRIED THIS APPLICATION OF THE TOOL OR TECHNIQUE BUT WAS NOT SATISFIED WITH THE RESULTS

E(EXPERIMENT TO

*INDICATES PRIMARY FUNCTION OF TECHNIQUE/TOOL



ICARDED) - THE USER OR OPER TRIED THIS APPLI- N OF THE TOOL OR TECH- BUT WAS NOT SATISFIED THE RESULTS

E(EXPERIMENTAL) - THIS APPLICATION OF THE TOOL OR TECHNIQUE IS PARTIALLY OR EXPERIMENTALLY IMPLEMENTED

S(STANDARD) - WHEN REFERRING TO THIS TECHNOLOGY (EITHER "UTILITIES" OR "DEBUG TOOLS"), SUCH TERMS AS "THE STANDARD SET" OR "THE USUAL ARRAY" WERE USED IN THE SOURCE ARTICLE. (SEE PARA. 3.3.2 FOR FURTHER DEFINITION)

A(AID) - THIS APPLICATION IS NOT AUTOMATED IN ITSELF, BUT THE TOOL OR TECHNIQUE PROVIDES INFORMATION TO ASSIST THE USER IN PERFORMING THE APPLICATION MANUALLY

Figure 4-1. Maintenance Tool & Techniques Applications Matrix (Continued)

APPLICATIONS / MAINTENANCE TOOLS AND TECHNIQUES	SOURCE	STATUS	LANGUAGE/ SYSTEM	CONFIGURATION CONTROL			OPERATIONS MONITORING/EVALUATION				REDESIGN			CODE PRODUCTION AND ANALYSIS											
				SUPPORT LIBRARY	AUTOMATIC RECONFIGURATION	STATUS REPORTING	PERFORMANCE ANALYSIS	FAILURE DATA ANALYSIS	PERFORMANCE MONITORING	AUTOMATIC RECOVERY	REQUIREMENTS DETERMINATION	REDESIGN ANALYSIS	PSEUDO-CODE	STRUCTURED PROGRAMMING AID	FORMATTER	PREPROCESSOR	RESTRUCTURING PROGRAM	STANDARDS ENFORCER	CODE COMPARATOR	OPTIMIZER	UTILITIES/PROGRAM SUPPORT	AUTOMATIC MODIFICATION	CODE READING/REVIEW	DEBUG TOOLS
RXVP 4.2.24	GRC	OPERATIONAL	FORTRAN IFTRAN CDC, IBM, UNIVAC, OTHERS														U								CROSS REF
SEF (MODTST) 4.2.16	IRVINE & BRACKETT	PARTIALLY OPERATIONAL NAVAL AIR DEVEL. CENTER (BEING EXTENDED)	NOT SPECIFIED			U*						U									S R				S R
SELECT 4.2.2	BOYER ET AL (SRI)	EXPERIMENTAL RESEARCH TOOL	LISP SUBSET																						
SEMANTIC UPDATE SYSTEM 4.2.26	HIRSCHBERG ET AL	PROTOTYPE BEING DEVELOPED	FORTRAN LARGE-SCALE	U M		U M						U M											A* M		
SPTRAN 4.2.15	ELLIOTT (HONEYWELL)	OPERATIONAL	FORTRAN PORTABLE											U*	U	U									
STRUCTRAN I&II 4.2.6	RADC	OPERATIONAL	FORTRAN UNIVAC 1108 HIS 600/6000											U	(STRUCTRAN I) ⊆ (STRUCTRAN I) ⊆		U*								DUMP (STRUCTRAN I
SYSTEM MONITOR 4.2.29	YAU, CHEUNG, AND COCHRANE	EXPERIMENTAL	NOT SPECIFIED		U M					U* M		U M							U M				U M		INTERACTIVE
TESTING SYSTEM 4.2.34	HONEYWELL	PROTOTYPE IMPLEMENTED	HONEYWELL OS																						DUMPS
TPL/F SYSTEM 4.2.38	PANZL	OPERATIONAL	FORTRAN																						
TRANSFORMATIONS 4.2.36	VARIOUS	EXPERIMENTAL IMPLEMENTATION	VARIOUS									U							U	A			U*		

U(USED) - THE TOOL OR TECHNIQUE USES THIS TECHNOLOGY

M(MAINTENANCE) - AN APPLICATION TO THE MAINTENANCE PHASE IS MENTIONED IN THE LITERATURE

R(RECOMMENDED) - THE USER OR DEVELOPER RECOMMENDS OR PLANS THE EXPANSION OF THE TOOL OR TECHNIQUE TO INCLUDE THIS TECHNOLOGY

T(TRIED AND DISCARDED) - THE USER OR DEVELOPER TRIED THIS APPLICATION OF THE TOOL OR TECHNIQUE BUT WAS NOT SATISFIED WITH THE RESULTS

E(EXPERIMEN

*INDICATES PRIMARY FUNCTION OF TECHNIQUE/TOOL

CODE PRODUCTION AND ANALYSIS								VERIFICATION AND VALIDATION														TESTING AND INTEGRATION										DOCUMENTATION	
RESTRUCTURING PROGRAM	STANDARDS ENFORCER	CODE COMPARATOR	OPTIMIZER	UTILITIES/PROGRAM SUPPORT	AUTOMATIC MODIFICATION	CODE READING/REVIEW	DEBUG TOOLS	STATIC ANALYSIS	PATH STRUCTURE ANALYZER	ANOMALY DETECTOR	VARIABLES ANALYZER	INTERFACE CHECKER	REACHABILITY ANALYZER	SYMBOLIC EXECUTION/ EVALUATION	PROOF OF CORRECTNESS	ASSERTION CHECKER	DYNAMIC ANALYSIS	EXECUTION ANALYZER (SW PROBES)	PATH FLOW ANALYZER	TIMING ANALYZER	USAGE COUNTER	INTERACTIVE EXECUTION	TEST PROCEDURE LANGUAGE	TEST DATA GENERATION	TEST CASE SELECTION	COMPLETION ANALYZER	STUBS	TEST FILE MANAGER	AUTOMATIC DRIVER	OUTPUT PROCESSOR/ANALYZER	TEST STATUS REPORTING	TEST BED	SIMULATOR	REGRESSION TESTING	DOCUMENTATION AID	AUTOMATIC DOCUMENTER
U							CROSS REF	U	U	U	U						U	U	U					A	U	U*		U						
				S R			S R	U				U												R		R			U M	U M			U M	U
								U	U	R				U*		U							E											
					A* M			U M			U M	U M																					U M	
																																		PROG. DOC.
U*							DUMP (STRUCTRAN I)																												
		U M			U M		INTERACTIVE																												
							DUMPS															U		U	U	A		R	U* M		U M			R (TEST DOC.)
																U	U				U		U			U	U	U*	U	U		U		
		U	A		U*										E																			

ISCARDED) - THE USER OR LOPER TRIED THIS APPLI- ON OF THE TOOL OR TECH- E BUT WAS NOT SATISFIED THE RESULTS

E(EXPERIMENTAL) - THIS APPLICATION OF THE TOOL OR TECHNIQUE IS PARTIALLY OR EXPERIMENTALLY IMPLEMENTED

S(STANDARD) - WHEN REFERRING TO THIS TECHNOLOGY (EITHER "UTILITIES" OR "DEBUG TOOLS"), SUCH TERMS AS "THE STANDARD SET" OR "THE USUAL ARRAY" WERE USED IN THE SOURCE ARTICLE. (SEE PARA. 3.3.2 FOR FURTHER DEFINITION)

A(AID) - THIS APPLICATION IS NOT AUTOMATED IN ITSELF, BUT THE TOOL OR TECH NIQUE PROVIDES INFORMATION TO ASSIST THE USER IN PERFORMING THE APPLICATION MANUALLY

Figure 4-1. Maintenance Tool & Techniques Application Matrix (Continued)

be shown in the Applications Matrix with a "U" in the "static analysis" column but no entry in the "redesign analysis" column, because the literature did not address the redesign application of the tool. The Technology/Activity Matrix shows that static analyses tools are also applicable to the redesign activity.

4.2 TOOL/TECHNIQUES DESCRIPTIONS

4.2.1 EFFIGY

a. Category - Verification and Validation Tool

- Interactive Symbolic Execution - with normal execution as a special case
- Standard Interactive Debug Tools (including trace, breakpoints, and state saving)
- Path Structure Analyzer
- Static Analysis
- Assertion Checker
- Proof of Correctness
- Interactive Execution

b. Sources

(1) King, J. C., "A New Approach to Program Testing", Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp 228-233.

(2) King, J. C., "Symbolic Execution and Program Testing", Communications of the ACM, Volume 19, Number 7, July 1976, pp 385-394.

c. Status

EFFIGY is an experimental system which has been under development at IBM since 1973. The system is limited in practical use but, as a research tool, it has given considerable insight into the concepts of symbolic execution and its varied applications, including proving the correctness of programs.

EFFIGY applies to programs written in a simple PL/I style programming language that is restricted to integer valued variables

and vectors (one dimensional arrays). The system runs on CMS under VM 370 on an IBM 370, Model 168. The CMS filing system and context editor are used as an integral part of EFFIGY for creating, changing, and storing procedures and command files.

d. Description

The EFFIGY system is an interactive symbolic execution tool incorporating standard debug tools and expanded to include assertion checking, a simple program testing manager and a program verifier. Normal program execution is provided as a special case.

The interactive debugging tools include:

- Tracing

  During execution, the user can request to see the statement number, the source statement, the computational results, or any combination of these for any or all program statements.

- Breakpoints

  The user can insert breakpoints before or after any statement. When the breakpoint is reached, execution is interrupted and control passes to the user's terminal for further interrogation or input.

- State Saving

  The user can save the state of execution before selecting a specific branch and later return to begin the exploration of alternative paths.

EFFIGY accepts one statement at a time (from the terminal or from a stored program), building a symbolic execution tree that defines the paths through the program. Either symbolic or integer values can be assigned to the input variables. Because the execution is symbolic, some of the conditional branches may be unresolvable; that is, both alternatives are possible, and parallel execution must proceed on the alternate paths in order to generate a complete execution tree. EFFIGY allows the user to interactively choose the single alternative path to be taken at any one time. The "state saving" capability can be

used to save the state of the execution and return later to follow the alternate paths.

A test manager is available for systematically exploring the alternatives presented in the symbolic execution tree. The exhaustive search, which could become an infinite process because of unresolvable branches, can be limited by the user to those paths traversing less than a specified number of statements. If the user supplies output assertions, the system automatically checks test case results against the assertions.

The program verifier generates verification conditions from user supplied assertions in conjunction with the symbolic execution.

An "input predicate" and an "output predicate" must be supplied with the program, defining the "correct" behavior of the program. The program is determined to be correct if for all inputs which satisfy the input predicate, the results produced by the program satisfy the output predicate. Inductive predicates must be inserted at appropriate points in the program to reduce the proof of correctness to a finite set of finite length paths. The ASSERT statement is used to place the predicates in the program. A managerial controller enumerates the paths and forces path choices at unresolved IF statement executions. This capability is being used for conducting research into correctness proof techniques.

e. Research and Findings (King)

King believes that a powerful interactive debugging/testing system combined with a symbolic execution capability offers a natural evolutionary growth from today's systems to achieving the systems of tomorrow. The system provides a continuing basis for further research in other forms of program analysis, such as program proving, test case generation, and program optimization.

4.2.2 SELECT (Symbolic Execution Language to Enable Comprehensive Testing)

a. Category - Verification and Validation Tool

- Symbolic Execution (also normal execution)
- Static Analysis
- Path Structure Analyzer
- Assertion Checker
- Test Data Generation

b. Source

Boyer, R. S., Elspas, B., and Levitt, K. N., "SELECT-A Formal System for Testing and Debugging Programs by Symbolic Execution," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp 234-245.

c. Status

SELECT is an experimental system which has been under development at Stanford Research Institute since 1974. The system is similar to EFFIGY and its purpose is to assist in the formal systematic debugging of programs.

SELECT applies to programs written in a LISP subset that includes arrays.

d. Description

SELECT is a symbolic execution tool which is intended to be a compromise between an automated program proving system and ad hoc debugging practices. Experimentally, SELECT includes:

- Semantic analysis of programs
- Construction of input data constraints to cover selected program paths
- Identification of (some) unfeasible program paths
- Automatic determination of actual (real number) input data to drive the test program through selected paths
- Execution (actual or symbolic) of the test program with optimal intermediate assertions and output assertions

- Generation of simplified expressions for the values of all program variables, in terms of symbolic input values
- Path analysis for each potentially executable path or for a user-selected subset of paths. Multiple executions of a loop within a path are defined as separate paths, producing a potentially infinite number of distinct paths. The number of loop traversals may be constrained by the user.

For conditions which form a system of linear equalities and inequalities, SELECT generates a specific input data example that will cause the associated program path to be executed in a test run. During the path analysis, each line of code is exercised for each path that includes that line of code. When a branch point is reached, SELECT determines which of the two branch units is satisfied by the data and execution continues along that branch. If the alternative path is feasible, a backtrack point is established for further analysis of the alternative branch. The execution and the backtracking continue until the program exit is reached and each loop has been traversed the number of times specified by the user.

SELECT also offers the capability of accepting user-supplied assertions. Assertions can be used to determine the numerical value of the symbolic data during execution, to constrain the input space bounds for the generation of test data and to provide specification of the intent of the program for verification purposes.

e. Research and Findings (Boyer, Elspas, and Levitt)

(1) SELECT appears to be a useful tool for rapidly revealing program errors, but there is a need to expand its manipulative powers beyond those of inequalities and algebraic simplification.

(2) In four demonstration examples, SELECT was not always successful in automatically generating useful test data. For the unsuccessful cases, user interaction was required in the form of output assertions.

(3) SELECT requires a method for handling program abstractions; as a minimum, a mechanism for hierarchically handling subroutine calls.

(4) SELECT should be expanded to detect invalid operations and conditions such as potential overflow and underflow, division by zero, and referencing an unitialized variable.

(5) SELECT is better suited to the analysis of moderate-sized data processing programs with numerous paths, each path corresponding to a different case, than to the analysis of complex, "clever" algorithms.

(6) The authors conclude that "automatic generation of test data can be useful but certainly should not be viewed as a total solution to the problem. A more promising method is to incorporate user interaction in the test generation process". However, the symbolic processing system can be used to assure that all cases of interest are covered during the debugging activity.

4.2.3 ATTEST

a. Category - Verification and Validation Tool

- Automatic Test Data Generation
- Static Analysis
- Symbolic Execution
- Path Structure Analyzer
- Stubs

b. Sources

(1) Clarke, L. A., "Testing: Achievements and Frustrations", Proceedings COMPSAC 78, Chicago, IL, November 1978, pp 310-320.

(2) Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, Volume SE-2, September 1976, pp 215-222.

(3) Chandrasekaran, B., "Test Tools: Usefulness Must Extend to Everyday Programming Environment", Computer, Volume 12, Number 3, March 1979, pp 102-103.

c. Status

ATTEST is an experimental system which is being developed at the University of Massachusetts. ATTEST applies to programs written in ANSI FORTRAN and can be used in the context of top down testing.

d. Description

ATTEST is an automated aid to structural testing which can either augment previously selected data sets or select all the test data for a program. The major purpose of the system is to generate test data for unexercised sections of the code and to create data sets (assertions) to exercise some of the implementation dependent special cases such as an array out of bounds and division by zero.

ATTEST automatically selects a subset of a program's paths according to the following testing criteria:

- Executing all statements
- Executing all possible conditional branches
- Executing all loops a minimum number of times
- Executing all program paths.

Each selected path is analyzed by symbolic execution to determine the path's computations, and the constraints for the range of possible input values (that is, a set of equality or inequality expressions with variables representing the input data). Nonexecutable paths are identified.

The test generation algorithm attempts to find a solution to the set of constraints. If solution fails, the user is shown the set of constraints for manual resolution. If the solution succeeds, test data is generated to drive execution down the selected path. In support of top-down testing and stepwise refinement, the ATTEST interface description language AID enables the user to describe both predicted and presumed relationships among program variables. Specifications of unwritten

modules (stubs) can be stated by means of AID commands, and symbolic execution can proceed as if the module were present. AID has conditional execution constructs for the easy description of conditional procedure computations in early versions of a program. ATTEST also supports symbolic I/O.

e. Research and Findings (Clarke)

(1) At least 80 percent of the requested path coverage is being achieved. Other testing criteria are being explored as well as methods for designating critical areas of code that should receive additional consideration.

(2) Methods of incorporating further analysis of incompatible predicates are being investigated.

(3) Test data generation is restricted to systems of linear predicates. Other methods for solving systems of inequalities are being examined.

(4) Some difficulties in handling FORTRAN arrays and file implementations are being researched.

4.2.4 DISSECT

a. Category - Verification and Validation Tool

- Symbolic Evaluation
- Static Analysis
- Assertion Checker
- Path Structure Analyzer (the paths are selected by the user)
- Documenter (a complete record of the evaluation is provided)
- Test Data Generation (when automatic test data generation has not been successful, the DISSECT output can be used as a guideline for the manual preparation of test data)

b. Sources

(1) Howden, W. E., "DISSECT - A Symbolic Evaluation and Program Testing System", IEEE Transactions on Software Engineering, Volume SE-4, Number 1, January 1978, pp 70-73.

(2) Stucki, L. G. et al., Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two volumes.

c. Status

The DISSECT system is designed for use as a research tool and is available from the National Bureau of Standards. Plans for a production version depend on the results of continuing research into the effectiveness of various validation and testing techniques.

DISSECT applies to ANSI FORTRAN programs having sequence or line numbers. It is implemented in LISP and runs under the PDP-10 LISP system.

The current version of DISSECT consists of 4100 lines of LISP source code and requires user space of at least 70K 36-bit words to run effectively.

d. Description

The DISSECT system is a symbolic evaluation tool which allows the user to "dissect" a FORTRAN program in order to examine the composition and computational effects of each program path. The DISSECT Command Language is used to specify each case to be studied. A case consists of:

- The path(s) or partial path(s) to be evaluated (the branches to be followed)
- The symbolic or actual values to be assigned to the input variables
- The output or sets of output to be generated.

The symbolic evaluation technique allows the validation of a program over a large subset of the input domain in a single execution "run". Complex programs having many paths can be divided into segments and analyzed using separate cases.

DISSECT analyzes the ANSI FORTRAN program to determine:

- The computations that are carried out along the selected paths

- The system of predicates; that is, the set of symbolic values which cause each path to be executed
- The symbolic values of the output variables.

Options in the Command Language allow the user to specify the exact branches to be followed within a path or to follow all branches which do not cause the system of predicates associated with the path to become inconsistent. Conditional commands provide the capability to execute the command only if some specified condition is true and path traversal can be automatically halted for unfeasible paths. Since the symbolically evaluated system of predicates for a path describes the set of all input values which cause the path to be executed, the output can be used as a guideline for the manual preparation of test data. Similarly, the identification of the unfeasible program paths can be used to eliminate these paths from consideration during test data preparation.

e. Research and Findings

(1) Howden - Source (1)

(a) The interactive use of DISSECT was attempted, requiring the user to interact with the system during execution whenever it was necessary to specify which branch to follow. The results were not satisfactory, in Howden's view, because the choices must be made more carefully and systematically than is usually possible in interactive real time.

(b) A study was conducted to determine the effectiveness of symbolic evaluation (DISSECT) in discovering program errors. Six programs containing 28 natural (unseeded) errors were analyzed using various error detection techniques, including symbolic evaluation, branch testing, and anomaly analysis. The findings were:

- DISSECT found 68 percent of the errors
- The combined use of all techniques including DISSECT resulted in the discovery of 3-4 percent more errors than the combined use of all techniques without DISSECT.

- DISSECT alone was 10-20 percent more effective then structured testing alone; that is, symbolic evaluation was the "natural" error discovery technique for 10-20 percent of the errors in the six programs.
- Automatic test data generation was not useful in any of the six programs
- Elimination of unfeasible paths was useful in preparing structured tests for four of the six programs.

(c) A major result of Howden's research is a clear indication that no single program analysis technique should be used to the exclusion of all others.

(d) Howden states that the distinctive feature of DISSECT is the DISSECT Command Language, which allows the user to describe a set of program analyses in a convenient and efficient manner.

(2) Stucki et al - Source (2)

DISSECT was among the automated verification tools which were evaluated under contract to the NASA Goddard Space Flight Center (GSFC). The evaluations for DISSECT were as follows:

(a) The operating cost was evaluated at 4 in a scale of 1 = low to 5 = high.

(b) The ease of use was evaluated at 3 in a scale of 1 = easy to 5 = difficult.

(c) The output from the analysis of a large number of paths was easy to read and a single examination could document all important cases for a program.

(d) Automated generation of test data from systems of predicates and automated proof of verification conditions are not planned for inclusion in DISSECT because attempts to automate these two features have not been completely successful in specially designed test data generation and proof of correctness systems.

(e) DISSECT was not recommended for use at GSFC at the time of study due to programmer time required to invoke the tool and to interpret the output in terms of program correctness. However, support for continued refinement and development was recommended.

4.2.5 JAVS (Jovial Automated Verification System)

a. Category - Testing Aid, Verification and Validation Tool

- Test Completion Analyzer
- Test Case Selection (guideline for manual test case selection)
- Test Data Generation (guideline for manual data preparation)
- Dynamic Analysis
- Execution Analyzer - software probes
- Path Flow Analyzer
- Static Analysis
- Path Structure Analyzer
- Reachability Analyzer
- Interface Checker
- Optimizer (timing guidelines for manual code optimization)
- Assertion Checker
- Timing Analyzer
- Usage Counter
- Variables Analyzer
- Automatic Documenter (program documentation)
- Debug Tools - graphic output, trace, cross-reference, dump, breakpoint
- Regression Testing (guidelines to modules affected by coding changes)

b. Sources

(1) Gannon, C., "JAVS: A Jovial Automated Verification System," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp 539-544.

(2) Finfer, M., et al., Software Debugging Methodology, Final Technical Report, April 1979, RADC-TR-79-57, Three volumes.

(3) Compendium of ADS Project Management Tools and Techniques, Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

(4) TRW Systems and Space Group, NSW Feasibility Study, Final Technical Report, February 1978, RADC-TR-78-23.

c. Status

JAVS is a workable, field-tested system for validating and testing JOVIAL programs. The system was delivered to its sponsor, Rome Air Development Center (RADC), in September 1975 and has since been installed at a number of government sites.

JAVS is an overlay program which operates in batch mode. It has been installed using the GCOS, WWMCCS, and GOLETA operating systems and the HIS 6180 and CDC 6400 computers. It executes on the HIS 6180 at RADC in 53,000 words of primary storage. In general, the execution of a JAVS-instrumented program requires 1.5 times the execution time of an uninstrumented program and approximately twice the load core size.

d. Description

JAVS performs both static and dynamic analysis of modules, programs, program segments, and decision paths. Its primary use is to determine the extent to which JOVIAL programs have been tested and to provide guidelines for additional test cases.

Static analysis can be performed on JOVIAL (J3) source modules after a successful error-free compilation. Up to 250 invokable modules and an unlimited number of JOVIAL statements can be analyzed in a single process job. The analysis includes determination of program paths (control flow) inter- and intra-module relationships, unreachable modules, cross reference of symbols and usage of program variables. All symbols, including module names and loop variables, are listed in the symbol cross reference, along with an indication of where they are set, used, or

defined. The JAVS cross reference differs from the usual cross reference in that it can report on all modules in the JAVS data base library or on specified single sequences. (A module can be a program, procedure, function, CLOSE or COMPOOL.)

Instrumentation of the program with software probes permits dynamic analysis, determines testing coverage during execution, and produces comprehensive reports identifying the paths remaining to be exercised.

Execution analysis indicates which modules, decision paths, and statements have been exercised, including the number of times each statement was executed and the execution time (in CP milliseconds) spent in each module.

Tracing capabilities during execution include: invocations and returns of all modules, values of variables, and "important events", such as overlay link loading. Tracing may be extended to the D-D (decision-to-decision) path level at the user's option.

Program paths leading to untested areas are identified by means of statement numbers of all calls to the module, the module's interaction with the rest of the system, or the statements leading to a selected program segment. Test coverage reports can be obtained for a single run or using the JAVS data base, cumulatively for all runs. Reports are available by module, by test case or by test run.

For regression testing, a JAVS report showing the interaction between the selected set of modules and the rest of the system can be used to determine all modules in the system which could be affected by the code changes.

Processing is directed by user input in the JAVS command language. Computational directives are available to aid in data flow analysis and checking array sizes during execution. The directives include ASSERT to check logic expressions and EXPECT to check the boundaries of expected variables.

JAVS provides automatic program documentation in the form of enhanced module listings, module control flow pictures, module invocation reports and parameter lists, module interdependence reports and symbol cross reference lists.

e. Research and Findings

(1) Finfer et al (Source 2) report that "JAVS is a powerful tool that provides the user a good deal of control over the amount and type of debugging information produced, but it does require the user to master a rich command language."

(2) Gannon (Source 1) notes that: the JAVS software was path-tested and documented by JAVS. Since the tests were performed, 12 errors in the software have been found and corrected. Most of these were logic errors due to incomplete understanding of all possible combinations of JOVIAL constructs. All corrections were minor and required only a few statements to be changed.

Gannon recommends adding the following capabilities to JAVS:

- Static Analysis
  - -Identification of Uninitialized Variables
  - -Physical-Units Consistency Checking
- Dynamic Analysis
  - -Automatic Generation of Certain Types of Assertions
  - -Coverage Measurement of Program Functions

(3) TRW researchers (Source 4) report that JAVS is an advanced automated verification system with well-organized documentation. However, they point out the following disadvantages:

- The output of JAVS is difficult for users to analyze because of its D-D path orientation. Manual correlation is required to interpret the results at any other working level, such as statement level, which may be more familiar to users.

- The overhead caused by recording execution monitoring data on a mass storage trace file would be unacceptable for the instrumentation of an entire medium to large scale system.

f. Maintenance Experience

(1) Gannon - Source (1)

(a) At the General Research Corporation (GRC), a new syntax analyzer component for JAVS and all modified interfacing components were subjected to functional and structural tests assisted by JAVS.

(b) Maintenance of the JAVS software has been greatly facilitated by the JAVS documentation reports.

(2) AF Data Automation Agency - Source (3)

RADC reports that no major problems were experienced during the implementation of JAVS; however, no experience-related information on its performance was available at the time of the report. It is anticipated that computer resources required for testing and maintenance purposes should decrease substantially.

4.2.6 STRUCTRAN I and STRUCTRAN II

a. Category - Code Production Tool

- Restructuring Program
- Structured Programming Aid
- Preprocessor (STRUCTRAN I)
- Debug Tools (STRUCTRAN I)

  -Programmed-In Dumps
  -Recompilation
- Formatter (STRUCTRAN I)

b. Sources

(1) Compendium of ADS Project Management Tools and Techniques, Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

(2) Finfer, M., et al, Software Debugging Methodology, Final Technical Report, April 1979, RADC-TR-79-57, Three volumes.

c. Status

STRUCTRAN I and STRUCTRAN II are program translation tools which assist structured programming in FORTRAN. Both tools have been implemented on the UNIVAC 1108 computer and are exercised daily at the Defense Mapping Agency. STRUCTRAN I has been implemented on RADC's HIS 600/6000 series computer under GCOS.

d. Description

STRUCTRAN I is a preprocessor/translator which allows the use of extended structured FORTRAN source statements without changing the standard FORTRAN compilers. Listings for the input source code are automatically indented. The extended FORTRAN language, DMATRAN, contains five structured statement forms which can be mixed with FORTRAN statements in the source program. Ordinary FORTRAN control statements are no longer used. The five DMATRAN forms are:

- IF . . . THEN . . . ELSE . . . ENDIF. This construct provides block structuring of conditionally executable sequences of statements.
- DO WHILE . . . . END WHILE. This construct permits iterative execution of a sequence of statements while a specified condition remains true.
- DO UNTIL . . . END UNTIL. This construct permits iterative execution of a sequence of statements until a specified condition becomes true.
- CASE OF . . . CASE . . . CASE ELSE . . . END CASE. This construct allows multiple choices for program control flow selection.
- BLOCK - name - . . . END BLOCK. This construct (and corresponding INVOKE - name - statement) provides a facility for top-down programming and internal parameterless subroutines.

STRUCTRAN I translates the DMATRAN source statements into a FORTRAN program that is compatible with the ASA standard FORTRAN X3.9 and can be compiled by a standard ASA FORTRAN compiler.

STRUCTRAN II translates unstructured programs written in FORTRAN V into structured source programs which are logically equivalent to DMATRAN source programs.

The translator derives a graph of the program, reduces the graph to a hierarchy of single entry/exit subgraphs, adding variables as needed, and generates a structured version of the source program which implements the reduced graph.

The output of STRUCTRAN II can be examined manually in order to make improvements or modifications, and then translated back to FORTRAN by STRUCTRAN I. The DMATRAN program is easier to understand, modify, and maintain than the original FORTRAN program.

e. Research and Findings

(1) AF Data Automation Agency - Source (1)

RADC reports that STRUCTRAN I and STRUCTRAN II provide a fast and efficient solution to the promotion of structured programming in FORTRAN. STRUCTRAN I is highly transferable but there may be problems with transferring STRUCTRAN II because of its complexity and apparent system dependency.

(2) Finfer et al - Source (2)

(a) A structured DMATRAN program has highly visible form which reveals the intended function more readily than a FORTRAN program. Blocks of code and the possible sequences of blocks which can be executable are well-defined.

(b) While the use of structured programming has a positive effect on program reliability and maintainability, there is also a negative aspect to the use of DMATRAN. Although the user is familiar with the DMATRAN version of the program rather than the FORTRAN version, most of the available debugging and program analysis tools operate on the FORTRAN or object code version of the program.

4.2.7 FAVS (FORTRAN Automated Verification System)

a. Category - Testing Aid, Verification and Validation Tool

- Test Completion Analyzer
- Test Case Selection (guideline for manual test case selection)
- Test Data Generation (guideline for manual data preparation)
- Dynamic Analysis
- Execution Analyzer - software probes
- Path Flow Analyzer
- Static Analysis
- Path Structure Analyzer
- Anomaly Detector
- Reachability Analyzer
- Debug Tools - dumps, breakpoint, trace, cross-reference, graphic output
- Interface Checker
- Usage Counter
- Variables Analyzer
- Automatic Documenter (program documentation)
- Regression Testing (guidelines to modules affected by coding changes)
- Restructuring Program

b. Sources

(1) Compendium of ADS Project Management Tools and Techniques, Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

(2) Finfer, M., et al, Software Debugging Methodology, Final Technical Report, April 1979, RADC-TR-79-57, Three volumes.

c. Status

FAVS is a system for validating and testing FORTRAN programs which is based on a modified and enhanced version of General Research Corporation's proprietary RXVP Automated Verification System. The system has been implemented at Rome Air Development Center (RADC) on the Honeywell 6180 under GCOS.

d. Description

FAVS performs both static and dynamic analysis of FORTRAN and DMATRAN programs and provides automatic program documentation. Its primary use is to determine the extent to which the programs have been tested and to provide guidelines for the preparation of additional test cases.

The static analysis includes determination of program paths (control flow), inter- and intra-module relationships, cross reference of all symbols, common block and symbol usage, and syntax analysis. Inconsistencies not normally checked by compilers can be identified, including mixed mode expressions, improper subroutine calls, variables that may be used before being set, graphically unreachable code, and potentially infinite loops.

Instrumentation of the program with software probes permits dynamic analysis, determines testing coverage during execution, and produces comprehensive reports identifying the paths remaining to be exercised.

Execution analysis indicates which modules, D-D (decision-to-decision) paths and program statements have been exercised, including the number of times each statement was executed and each D-D path was traversed. The entry and exit values of the variables are traced dynamically.

Reports are generated for the current test run and cumulatively for all past test cases, for a single module or a group of modules. Details of individual D-D path coverage on a module are optional. D-D paths not traversed for the current test case and for all test cases are also identified.

Processing is directed by user input in the FAVS command language. The commands include REACHING, by which the user can determine which D-D path must be executed for a particular statement to be reached, and RESTRUCTURE, which translates existing FORTRAN programs into the structured language DMATRAN.

FAVS provides automatic programming documentation as follows:

- Source listings showing the number of each statement, the levels of indentation and the D-D paths.
- Statement analysis by type (classified or declaration, executable, decision, documentation).
- Common blocks and modules (subroutines) dependencies.
- Common block symbol matrix for modules.
- Cross reference listing of all symbols set or used by all modules on the library.
- List of read statements within a module.

For regression testing, the FAVS reports showing module, path, and variable dependencies can be used to determine the effect of proposed coding changes on the rest of the system.

e. Research and Findings (Source 1)

A performance evaluation of FAVS was not available at the time of the report. However, RADC anticipates that computer resources required for testing and maintenance purposes should decrease substantially with the use of FAVS.

4.2.8 FORTRAN Code Auditor

a. Category - Code Production and Analysis Tool

- Standards Enforcer
- Structured Programming Aid
- Optimizer (Checks predefined optimization standards)

b. Sources:

(1) Compendium of ADS Project Management Tools and Techniques, Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

(2) Finfer, M., et al, Software Debugging Methodology, Final Technical Report, April 1979, RADC-TR-79-57, Three volumes.

(3) TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

c. Status

The FORTRAN Code Auditor described here is a representative example of automated tools which analyze FORTRAN programs for conformance to predefined programming standards and conventions. The auditor is being exercised approximately 100 times a week on the U. S. Army's Site Defense Software Development Program.

d. Description

The FORTRAN Code Auditor analyzes a FORTRAN source program and audits each statement for conformance to standards pertaining to readability, structured programming techniques, and optimization of object code. The auditor does not modify the users' source code but lists the code indicating the source statements which deviate from the predefined standards. Summary reports of deviations are provided for each subroutine and for the entire program.

The FORTRAN source statements must be free of syntax errors, since syntax errors are not detected by the auditor. The standards which are monitored are:

- Rules for quantity and placement of comments
- Rules for physical placement and grouping of code elements on the source code listing
- Limitations to module size
- Restrictions on the use of certain instructions (for the purpose of optimization of the object code execution time)
- Rules for top-down design and hierarchical structure.

e. Research and Findings

(1) AF Data Automation Agency - Source (1)

RADC anticipates that proper application of the tool will help reduce the cost of software development and maintenance. Transferability to other computer environments is estimated to be: 750 hours to IBM, 600 hours to UNIVAC, and 460 hours to CDC.

(2) Finfer, M., et al - Source (2)

The FORTRAN Code Auditor is an effective mechanism for enforcing standards and improving both verification and maintenance activities.

4.2.9 FORAN (FORTRAN Analyzer Program)

a. Category - Verification and Validation Tool

- Static Analysis
- Variables Analyzer
- Debug Tools - cross reference
- Interface Checker
- Anomaly Analyzer

b. Source

Finfer, M., et al, Software Debugging Methodology, Final Technical Report, April 1979, RADC-TR-79-57, Three volumes.

c. Status

FORAN is a FORTRAN source code analyzer used at the U. S. Army Advanced Research Center (ARC) in Huntsville, Alabama, in support of the Ballistic Missile Defense Advanced Technology Center (BMDATC).

The FORAN analysis is limited to 4095 data items and a total of 24,000 unique references for all named items.

d. Description

FORAN performs static analysis on source code written in any dialect of FORTRAN. Usage of program labels, tags, data variables, constants, subroutines, and other program elements are analyzed for a main program and its related subroutine components. Each item name is listed, showing the statement numbers where the item is referenced and how it is referenced (assigned, used, input, output, subroutine CALL, etc.). Individual program units can be separately analyzed. FORAN also identifies symbols defined but not used, discrepancies in variable type and dimension, and number and type of parameters in functions and subroutines. Syntax errors are flagged during the analysis.

FORAN's primary use is to determine possible computation or logic errors from the static analysis of data usage. It is also valuable in analyzing the effect of a program modification on data usage.

e. Research and Findings

Finfer, et al report that "FORAN is easy to use and its output contains more information and is easier to read than a compiler's symbolic reference map."

4.2.10 Modern Programming Practices (MPP)

a. Category - Code Production and Analysis Technique, Configuration Control, Redesign

- Structured Programming Aid
- Pseudo-Code
- Support Library
- Code Reading/Review
- Interactive Debugging/Testing
- Documentation Aid

b. Sources

(1) Baker, F. T., "Structured Programming in a Production Programming Environment," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pp 241-252.

(2) McNurlin, B. C., "Using Some New Programming Techniques," EDP Analyzer, Volume 15, Number 11, November 1977, pp 1-13.

(3) Holton, J. B., "Are the New Programming Techniques Being Used?" Datamation, Volume 23, Number 7, July 1977, pp 97-103.

(4) Compendium of ADS Project Tools and Techniques, Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

c. Status

IBM introduced the concept of Improved Programming Technologies as design and programming aids in the early 1970's. The ideas of structured programming have gained widespread recognition and it is generally agreed that the use of these "Modern Programming Practices" during system development will improve maintainability. Literature references concerning the application of the techniques to the operations and maintenance phase are extremely limited. However, the following comments and recommendations apply to their application in any life cycle phase.

- The major problem in implementing the technique is that people do not like to change.
- The introduction of the technique requires careful planning, substantial training, and dedicated management support.

d. Description

(1) Structured Programming

A structured program is a program in which the logic flow proceeds from beginning to end without arbitrary branching. The purpose of the structured programming technique is to make the program easier to read, comprehend, debug, and maintain. First,

the user must develop or adopt a set of standards which explicitly define the required structuring techniques, along with coding, formatting, and naming conventions. Some typical requirements for a structured module (Source 1) are:

- It has an intelligible name
- It is about one page of source code in length
- It contains functions that logically fit together
- It has one entry and one exit
- Coding is restricted to three basic control structures - sequence, alternation (IF THEN ELSE) and repetition (DO WHILE), and possibly two optional structures - DO UNTIL and CASE
- Indentation is used to show the structure of the program on the source listing
- Exceptions to standards (for example, use of GOTO) are approved and documented
- Team members must review each other's code

The four approaches to providing the control flow required for structured programming (Source 1) are:

- Structures are directly available as statements in the programming language. (For example, PL/I provides the three basic control structures.)
- Structures may be simulated using a few standard statements. (For example, CASE can be simulated in PL/I.)
- A standard preprocessor may be used to augment the basic language. (For example, a macro assembler for assembly languages.)
- A special precompiler may be written to compile augmented language statements into standard statements. (For example, a special precompiler for FORTRAN.)

Tools may be specially written by the user or vendor-supplied; for example, IBM offers SCOBOL which supports structured programming in COBOL.

(2) Top-Down Development/Programming

In the development phase, structured programming is closely associated with top-down development. In production, IBM Federal Systems Division (Source 1) has separated the concept of top-down development from that of structured programming in order to provide for the use of structured programming techniques on maintenance projects as well as development projects. According to this definition, top-down development refers to the process of concurrent design and development of program systems containing more than a single compilable unit. Development proceeds in a way which minimizes interface problems by integrating and testing modules as soon as they are developed. In everyday production programming, top-down development means eliminating (or at least minimizing) writing any code whose testing depends on code not yet written or on data not yet available.

Top-down programming applies the concept to a single program, typically consisting of one or a few load modules and a number of independently compilable units.

(3) Development Support Library (DSL)

The DSL keeps all machine readable data on a project in a series of data sets which comprise the internal library. Corresponding to each type of data in the internal library, there is a set of current status binders which comprise the external library. Sufficient archives are maintained to provide complete recovery capability. The DSL is maintained by the project librarian using an automated library package and includes such data as:

- Current and backup versions of programs
- Operating systems instructions
- Test data
- Test results
- Documentation
- Project performance data
- Library procedures.

The use of a DSL is usually associated with the use of structured programming; however, Baker (Source 1) states "because ongoing projects may not be able to install a DSL, an implementation of only structured coding is acceptable in these cases."

(4) Chief Programmer Teams (CPT)

A CPT is a functional programming organization built around a nucleus of three experienced persons doing well-defined parts of the programming development process using top-down development, structured programming techniques, and support libraries. The nucleus organization consists of a chief programmer, a backup programmer and a project librarian. Other programmers are added to the team only to code specific, well-defined functions within the framework established by the chief programmer and the backup programmer.

(5) Structured Walk-Through

Structured Walk-Throughs are a series of technical reviews at various stages of program development during which the developer "walks through" the design, code, test plan, etc. for review by a peer group. The group is usually limited to no more than six people and the emphasis of the walk-through is on error detection rather than correction.

(6) Pseudo Code (or metacode or program design language)

Pseudo code is a structured, natural language notation used during software design. Its purpose is to provide precision and readability for a smoother and more accurate transition between design and coding.

(7) Interactive Debugging and Testing

For interactive (or on-line) debugging and testing, the programmer uses a terminal to debug and test a program after the code has been entered into the system by a data clerk.

(8) Project Workbook (or Unit Development Folder)

The workbook contains the history and current status of each program module. As a minimum for program maintenance, the workbook should include:

- The design structure showing the module's place in the hierarchy
- The data flow analysis showing the I/O requirements
- The program listing (the structured code)
- The test plan and test case results
- The problem reports
- The log of changes.

e. Research and Findings

(1) Structured Programming

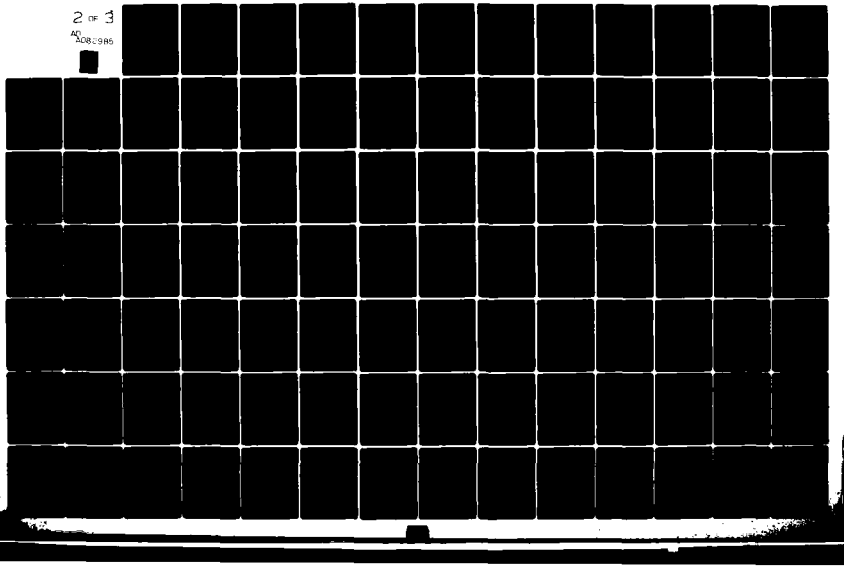
Opinions concerning the role of structured programming techniques in program maintenance range from the view that their use for maintenance means starting all over to the approach that they can be used to modify a system without restructuring it (Source 2). Tools are available (Source 4) to structure existing programs for ease of maintenance modification and then translate the modified program back to the original language for system processing. Another possibility is the "phasing in" of structured programs during the maintenance phase. For example, at IBM Federal Systems Division (Source 1), programs were structured when modules had to be rewritten or replaced.

(2) Structured Walk-Throughs

Holton's survey (Source 3) indicated that structured walk-throughs are more often used in the planning and design phase than for detailed design and coding. McNurlin (Source 2) reports that many people think walk-throughs are more effective as a design review technique, with a corollary opinion that code walk-throughs are not required because the design walk-throughs are so effective.

AD-A082 985 IIT RESEARCH INST CHICAGO IL F/G 9/2
A REVIEW OF SOFTWARE MAINTENANCE TECHNOLOGY.(U)
FEB 80 J D DONAHOO, D R SWEARINGEN F30602-78-C-0255
UNCLASSIFIED RADC -TR-80-13 NL

2 OF 3
AD A082985



f. Maintenance Experience

(1) Multiple Implementation

Charles Holmes (Source 2) described two attempts at McDonnell-Douglas Automation Company to modify four programs by implementing structured programming, chief programmer teams, an on-line production library, top-down programming, and structured walk-throughs. The first attempt (implementing all the new techniques at once) failed "because we were too ambitious and because we lacked coaching and the visibility of the experience of other companies." The second attempt (implementing the new techniques in phases beginning with a two week course on structured programming, top-down programming, pseudo code, and structured walk-throughs) was considered to be successful, with projects being completed more quickly and with improved quality.

(2) Support Libraries

Surveys (Sources 2 and 3) have indicated that although the DSL is seldom implemented as elaborately or completely as originally defined, many organizations use procedure libraries and automated library functions with favorable results.

Columbus Mutual Life Insurance Company reported (Source 2): "We wouldn't do our program development and maintenance any other way than by using the support library function." Columbus Mutual uses the LIBRARIAN package from Applied Data Research and its ROSCOE remote job entry system. For program maintenance, production source code is copied to create a test version. Modifications to the test version are "temporary" until the program is compiled, tested and approved, and then LIBRARIAN is used to make the changes permanent.

HQ AFSC/ACDPL reported (Source 4) on the use of WWMCCS Program Support Library (PSL) to support the development and maintenance of computer programs in a top-down structured

programming environment. The PSL supports a large organization with many persons working on different and often unrelated programming development and maintenance projects.

(3) Interactive Processing

The Mellon Bank (Source 2) found that the use of interactive debugging techniques resulted in "somewhere between a 3 to 1 and 5 to 1 improvement in the manhours needed to do a maintenance job." The interactive debugging was accomplished by inserting testing aids into the programs to monitor progress through the program execution and to detect the cause of abnormal termination.

TRW Defense and Space Systems Group (Source 2) reported that the use of terminals and CRT displays of the program has eased the problem of fitting modifications into the indented format of structured programs.

AFDAA, HQ AFDSC/GLD (Source 4) reported that "on-line terminals greatly assist the programmer to develop structured code, but they likewise reduce his inclination to desk check his program," and that the use of "top-down design and structured programming does not guarantee that the programmer will write a good program."

4.2.11 CSPP (COBOL Structural Programming Precompiler)

a. Category - Code Production Tool

- Structured Programming Aid
- Preprocessor

b. Source

Compendium of ADS Project Management Tools and Techniques, Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

c. Status

CSPP was developed under RADC's Structured Programming Contract with IBM to demonstrate the feasibility of the precompiler specification task. Its purpose is to provide programmers with a structured programming capability for ANS COBOL. The tool is highly transferable and is being used at several installations including:

- HQ Air Force Systems Command
- Defense Mapping Agency
- Warner-Robbins Air Logistics Center
- Air Force Institute of Technology at WPAFB
- Air Training Command
- Defense Communications Agency
- Electronics Systems Division

d. Description

The CSPP (COBOL Structured Programming Precompiler) is a tool to facilitate structured programming in ANS COBOL, X.3.23 - 1968. The additions to COBOL are in the form of structuring verbs, as defined in the RADC Structured Programming Series. The CSPP accepts the augmented ANS COBOL as input and produces a source program in ANS COBOL.

CSPP is itself written in ANS COBOL.

4.2.12 Performance/Maintenance Algorithms

a. Category - Performance Evaluation Technique

- Performance Analysis
- Requirements Determination
- Redesign Analysis

- Test Case Selection (aid)
- Regression Testing (aid to manual test case selection)

b. Source

Yau, S. S. and Collofello, J. S., Performance Considerations in the Maintenance Phase of Large-Scale Software Systems, Interim Report, June 1979, RADC-TR-79-129.

c. Status

Algorithms are being developed to apply a maintenance technique for predicting the system performance requirements which will be affected by a proposed modification. An interim report has been published and preparation of a second report containing descriptions of the algorithms is in progress.

d. Description

The maintenance technique described in the interim report can trace repercussions introduced by maintenance changes in large-scale systems and predict which performance requirements may be affected by the program modifications. The techniques will enable maintenance personnel to incorporate performance considerations in their criteria for selecting the type and locations of required software modifications and also to identify which performance requirements must be verified following the modifications.

The second report will cover the following topics:

- Formal description of the algorithms for identifying the eight mechanisms for the propagation of performance changes in a large-scale program. Also included will be proofs of the correctness of these algorithms as well as illustrative examples.
- Formal description of the algorithms for identifying the critical software sections of a large-scale program.
- Formal description of the algorithms for identifying performance dependency relationships in a large-scale program.

- Formal description of the algorithms composing the technique for predicting performance requirements affected by maintenance activity. Also included will be proofs of the correctness of these algorithms.
- Demonstration of the maintenance technique during the maintenance phase of a typical program.

4.2.13 CCS (Change Control System)

a. Category - Configuration Control and Code Production Technique

- Support Library (programming support and central administrative data base)
- Configuration Status Reporting (status of changes)
- Code Comparator
- Utilities (standard utilities such as compilers, editors, and loaders)
- Code Reviews and Walk-Throughs
- Requirements Determination (determines dependencies among changes and enforces concurrent processing of interdependent changes)

b. Source

Bauer, H. A. and Burchall, R. H., "Managing Large-Scale Software Development with an Automated Change Control System," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp 13-17.

c. Status

CCS is a programming support system providing an interface between production and maintenance programmers and the No. 1 Electronic Switching System (ESS) software. No. 1 ESS is an extremely large (100,000 - 1,000,000 lines of code), complex system operating in a very dynamic environment. Programming support of this system is applied under conditions which are characterized by:

- High rate of software changes
- Multiple software development projects
- Large and growing programming staff

- Large, tightly coupled program units
- Overlapping responsibilities for program unit modification among programmers.

CCS provides a capability for minimizing the adverse effects of the above factors on the programming staff. It has been in operation since January 1977 at Bell Laboratories. In its first year of operation over 4000 program modifications were processed.

d. Description

The major functions of CSS are:

- Controls application and approval status for program source changes
- Generates and maintains object code changes
- Permits early source code changes
- Determines change dependencies and enforces concurrent processing
- Supports continuous integration of releases in sequence
- Permits identification of alternative system versions
- Provides language/compiler independent processes
- Maintains central, administrative data base
- Provides standard administrative syntax.

Through the use of CCS programmers have access to a standard set of support programs including compilers, editors, and loaders for implementing modifications and updates to No. 1 ESS. In addition, CCS manages the program change process by recording both source and object code versions of program units, by providing selective comparison of program versions for analysis of changes, and by storing and retrieving information about change status for review. Thus, CCS provides a common access to a variety of support software and facilitates management control of the change process.

CCS evolved as a result of combining a data management system with three extensions to the Source Code Control System

(SCCS) concept at Bell Laboratories. SCCS enables reconstruction of any prior version of a source module by applying recorded changes (deltas) to a baseline, in time sequence until the desired version is reached. The first extension relaxed the time sequence constraint for reproducing module versions. Through a program called MANAGER, source versions containing any specified deltas can be created. The second extension provided a capability for incremental installation of object code changes in any order desired. Use of this COMPARE program eliminates the need for recompiling and reloading for each change to a module. The final extension added a formal, finite state change model to the system. It is used to track and control change status for No. 1 ESS.

e. Maintenance Experience

At the time of the reference source article over 200 programmers were engaged in repairing and updating No. 1 ESS software. All programmers were using a common master copy of ESS source code. From a baseline such as this, CCS can produce past versions of the system for operations support or generate new versions for testing. A single module may undergo modification by more than one programmer at a time under control of CCS. As an example, in 1977, 546 modules were changed with an average of 13 deltas per module by four different programmers. A substantial improvement is productivity is attributed to the use of CCS, although the use of other techniques such as reviews and walkthroughs had a "large" effect.

For system corrections or enhancements CCS is used in the following way. An approved software change (Failure/Feature Report) is tagged with a unique identifier in the CCS data base and as a programmer submits a source delta for that change it is logged against that identifier. The set of deltas for that change is available for testing when all have been submitted and are successfully compiled. The programmer may test against any system version desired. Once his testing is completed he

informs CCS and the change is processed through a sequence of quality assurance steps until it is released for incorporation in the next reload of the system.

4.2.14 FADEBUG-I (FACOM Automatic Debug)

a. Category - Verification and Validation, and Testing and Integration Tool

- Output Analyzer (compares output results to desired output specified by user)
- Static Analysis
- Path Structure Analyzer
- Completion Analyzer (aid to user analysis)
- Utilities (standard FACOM OS)

b. Source

Itoh, Daiju and Izutani, Takao, "FADEBUG-I, A New Tool for Program Debugging," Proceedings IEEE Symposium Computer Software Reliability, 1977, pp 38-43.

c. Status

The FADEBUG-I program is a new debugging aid to be used at the early debugging stage of programs written in assembler language. It is an option of the FACOM 230-60 OS debug utility.

d. Description

FADEBUG-I has two primary functions as a debug aid. Comparing the set of output data produced by a program with user-specified output data is identified as its most important function. The other function involves automatic isolation and definition of all possible execution paths from entry to exit in a program module. These capabilities aid in detecting and removing program bugs.

In the module test stage of program development the following areas of difficulty are identified:

- Examination and verification of output data from module test execution
- Examination of module processing paths for logical errors
- Evaluation of module logic paths for omissions.

FADEBUG-I is designed to reduce or eliminate these difficulties through its test function or route definition function. For the test function the user must provide a set of desired output data and a set of input data. FADEBUG-I initiates execution of the module using the input data and after execution checks the actual output data against the desired output data. Discrepancies are identified and listed. In the route definition mode FADEBUG-I analyzes the source assembler language version of the module and lists all physical routes through the module code from entry to exit.

The source article identifies areas for improvement of FADEBUG-I and recommends some additional functions. The improvements consists of simplifying certain control statements and developing a better algorithm for identifying logical paths. Additional functions that might be useful are program production management data collection and automatic system test and data generation.

e. Research and Findings

The authors of the source article made a survey of FADEBUG-I users and nonusers within a programming group. Use of FADEBUG-I in the module test stage resulted in manpower reductions of 11 percent and improvement in debugging speed of 17 percent. After the module test stage, the users reported that preparation of test data and control statements was in general more difficult than similar preparation reported by nonusers. However, they also indicated that analysis of output data from the tests was easier with FADEBUG-I results.

4.2.15 SPTRAN

a. Category - Code Production Tool

- Structured Programming Aid
- Formatter (from free-form to FORTRAN)
- Automatic Documenter (program listing)
- Preprocessor

b. Source

Elliott, I. B., "SPTRAN: A FORTRAN-Compatible Structured Programming Language Coverter," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp 331-336.

c. Status

As of the date of the source article, SPTRAN was resident on a 16-bit Honeywell H316 computer. The converter is written in ANSI FORTRAN, and occupies 16,000 words of core, and can be implemented on any computing system having a word length of at least two bytes. Successful use of the converter in reducing program development time for three applications is reported in the source article.

The SPTRAN is listed as being available with the following items:

- Source code listing
- Source deck (029 cards)
- User manual
- Installation manual.

d. Description

As described in the source article, "The SPTRAN converter allows a user to apply structured programming concepts by coding in a language that has the structured and free-format features of PL/I, but is otherwise like FORTRAN. This language, called SPTRAN, was designed such that it combines those features of

FORTRAN and PL/I most often used for scientific programming applications."

Free-form SPTRAN input is translated into ANSI FORTRAN and the original SPTRAN source code is retained as FORTRAN remarks.

SPTRAN advantages are described as follows:

- Provides actual experience in encoding and executing structured programs
- Enables timely creation of FORTRAN programs
- Results in annotated program listings that are easy to read and can supplement documentation
- Provides early introduction to structured constructs and principles.

e. Research and Findings (Elliott)

There was no direct reference to the use of the SPTRAN converter for support of software maintenance. However, references to use of SPTRAN in program development cited reductions in debugging time attributed to use of this converter.

4.2.16 Software Engineering Facility (SEF)

a. Category - Configuration Control, Verification and Validation, Testing and Integration Techniques

- Configuration Control - Status Reporting
- Redesign Analysis
- Static Analysis
- Interface Checker
- Automatic Test Driver
- Test Output Analyzer
- Regression Testing
- Documentation Aid
- Simulator

b. Source

Irvine, C. A. and Brackett, J. W., "Automated Software Engineering Through Structured Data Management," IEEE Transactions

on Software Engineering, Volume SE-3, Number 1, January 1977, pp 34-40.

c. Status

The Software Engineering Facility concept has been implemented at the Naval Air Development Center, Warminster, Pennsylvania. As reported in the source article, this SEF is undergoing continuing upgrading to expand its capabilities to support development of well-engineered software. The authors state that a fully implemented SEF will provide "central support for an integrated collection of subsystems, and the subsystems (will) provide appropriate facilities for all phases of the software development process, from requirements definitions through maintenance and enhancement."

d. Description

An SEF consists of a software engineering data base, Structured Data Management System (SDMS), Structured Data Language Processor (SDLP) and a set of subsystems which provide specialized support to software development functions. A primary objective of the SEF is to provide a capability for automated capture of as much software development data as possible for the software engineering data base. In addition, the SEF, through the Structured Data Language, provides a common protocol for involving the subsystem processors. These processors may be linked directly to the SEF as an "SDMS integrated processor" or may be free-standing with no SDMS dependencies, an "SEF compatible processor."

The SEF components are described as follows:

- Software Engineering Data Base. This data base contains information stored by the various SEF processors (subsystems) and also used by them. This information relates to software modules, properties and structure of the system under development. The data base structure is compatible with the structure of the software being developed. Typical of information stored in the data base is hierarchic structure, intermodule dependencies, quantified system properties and dynamic sequencing constraints.

- Structured Data Management System. The SDMS functions as the manager of SEF resources. It is the facility for storage, inspection and manipulation of software engineering data and it provides control to the interaction among SEF processors and the software engineering data base.

- Structured Data Language Processor. The SDLP provides a capability for involving a standard set of SDMS commands through the Standard Data Language.

- Subsystem Processors. The types of subsystems which are selected for inclusion in a given SEF processor set are dependent upon the particular requirements of the software system to be supported by the SEF. For example the following subsystems could be included in a SEF processor set:

  Analysis, Design and Specification Tools. All information for data base entry will be captured by these processors as they perform their respective functions.

  System Analyzer. Data captured by the analysis, design and specification processors will be used by the system analyzer to develop a model of the system software. This model can be used to simulate system behavior for the analysis process.

  Interface Auditor. The interface auditor will perform various interface consistency analyses using the system software structure as recorded in the data base. This capability can be used to evaluate software change impact on interfaces and modules.

  Report Generator. Both management and system performance data are produced in report format by the report generator. These reports can be used in status reporting and in documenting the system for maintenance.

  Software Testing and Validation Tools. An array of testing and validation processors may be included in the SEF to provide automated testing of software revisions and validation of performance against established performance criteria.

e. Maintenance Experience

The source report authors cite success with the SEF at NADC even though only limited capability is implemented.

NADC uses an SEF process called MODTST. This process accepts a set of source code modifications as input, produces new executable modules, locates the associated test data, runs the tests, compares the test results with historical test results, and produces a discrepancy report. The discrepancy report indicates whether or not unintended changes occurred and is the basis of regression testing.

No performance data is given in the article. NADC is planning to integrate other tools into the SEF in the future.

4.2.17 PPE (Problem Program Evaluator)

a. Category - Performance Evaluation Tool

- Performance Monitoring
- Optimizer (aid)

b. Sources

(1) Stucki, L. G. et al., Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two volumes.

(2) Compendium of ADS Project Tools and Techniques, Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

c. Status

PPE is a proprietary software monitor package distributed by Boole and Babbage, Inc., Sunnyvale, California. It has been implemented on IBM computers and is language independent.

d. Description

PPE is a performance measurement tool which samples a program's execution at regular intervals and reports timing data for user analysis. The measurements include:

- Percent of time the program is actively executing.
- Percent of time the program is waiting for I/O to complete.
- Percent of time the program is waiting for non-I/O events to complete.
- The distribution of the I/O among the program's data sets.
- Percent of time the problem program is executing SVC's (Supervisor Calls).
- The instruction location in core where the various activities and waits occurred.

Since PPE resides in the same region as the program being measured, it can readily be applied to programs during production runs to analyze program performance in the operational environment. The analysis is performed externally on load modules and does not require any modification to the executing program.

The Study Report output from PPE provides information concerning the distribution of CPU time throughout the program. The ten most frequently executed intervals are listed separately. These and other timing measurements enable the programmer to concentrate optimization activity on the most time consuming or least efficient areas of code and on the data sets that show the highest percentage of activity.

e. Research and Findings - Source (1)

PPE was among the automated tools which were evaluated under contract to the NASA Goddard Space Flight Center (GSFC) and was in use at GSFC at the time of the study. The evaluations for PPE were as follows:

(1) PPE is a well established proven tool of considerable use in performance measurement applications.

(2) PPE is recommended as a good tool for giving a total picture of a program's efficiency but does not provide performance statistics at the source statement level.

(3) The operating cost was evaluated at 1 in a scale of 1 = low to 5 = high.

(4) The ease of use was evaluated at 2 in a scale of 1 = easy to 5 = difficult.

f. Maintenance Experience - Source (2)

HQ AFAFC/ADRR, Lowry AFB, reports that PPE has been used for over five years and is the primary tool for evaluating production programs. Inefficiencies disclosed during the earlier use of PPE were used to derive programming "tips" and standards which assist in preventing the reoccurrence of the inefficiencies in new programs. A current savings of 10 hours IBM 360/65 time per year was estimated at the time of the report.

4.2.18 PET (Program Evaluator and Tester)

a. Category - Verification and Validation Tool

- Execution Analyzer (software probes)
- Dynamic Analysis
- Path Flow Analyzer
- Timing Analyzer
- Usage Counter
- Variables Analyzer
- Test Case Selection (aid to manual preparation)
- Test Completion Analyzer (aid to user analysis)
- Optimizer (aid to manual optimization)
- Standards Enforcer
- Static Analysis (syntactic profile)
- Assertion Checker

b. Sources

(1) Stucki, L. G. et al, Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two volumes.

(2) Stucki, L. G. and Foshee, G L., "New Assertion Concepts for Self-Metric Software Validation," Proceedings IEEE Conference on Reliable Software, Los Angeles, CA, April 1975, pp 59-65.

(3) TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

(4) Gilb, T., Software Metrics, Winthrop Publishers, Inc., Cambridge, MA, 1977, 282 pp.

c. Status

PET, produced by the McDonnell Douglas Corporation, is a validation/testing tool which performs both static and dynamic analysis of FORTRAN programs. Since 1972, it has been implemented on IBM, CDC, Honeywell, and Univac computers and was recently installed at NASA Langley, Hampton, Virginia on the CDC 6000.

d. Description

The PET preprocessor inserts software probes into the program code and the postprocessor analyzes the data collected by the probes during program execution. Four one-page reports are produced for each program:

- A syntactic and operational profile
- A subprogram operational summary
- A subprogram execution summary
- A subprogram timing summary.

An annotated source listing is used for displaying the execution statistics associated with each source statement.

The data collection includes:

- The number of times each executable statement was executed.
- The number of times each branch was executed. This includes branch counts for logical and arithmetic IF conditions, plus computed and assigned GOTO's branching histories.
- The number and percentage of the total of executable statements, non-executable statements, and comments.
- The number of coding standards violations.
- The number and percentage of all potential executable statements that were executed one or more times.
- The number and percentage of program branches tested.

- The number and percentage of subroutine calls that were executed.
- The number of times each subroutine was called, and the names of those subroutines that were never entered.
- Relative timing for subroutine executions.
- The minimum and maximum values attained by an assignment statement variable or DO loop parameter.
- The first and last values attained by an assignment statement variable or DO loop parameter.

All data items except the counts of statement executions and branch executions are optional at the user's request.

The PET reports and annotated listings provide insight into dead code, impossible branches, highly utilized code, and the degree of program documentation by COMMENT statements. The timing summary and execution frequency counts provide information for optimizing program performance. Since the reports reveal the untested code for each test case, they can be used to assist the user in developing test cases which exercise the entire program.

The assertion checking capability of PET allows the user to specify error checking criteria in the form of mathematical statements and to direct their placement over any range or position in the program. The system then generates and places probes to monitor the progress of program execution for adherence to these assertions.

e. Research and Findings

(1) Stucki et al - Source (1)

PET was evaluated under contract to the NASA Goddard Space Flight Center (GSFC) as follows:

(a) The ease of use was evaluated at 1 in a scale of 1 = easy to 5 = difficult.

(b) The operating cost was evaluated at 2 or 3, depending on the options used, in a scale of 1 = low to 5 = high.

(c) PET was recommended for use in situations where operating cost is not a major factor in the selection.

(2) Gilb - Source (4)

(a) Gilb reports that PET was used on an operational system of 40,869 program statements along with the test data that had been used to test the system prior to release. PET showed that the test data covered only 44.5% of the executable source statements and only 35.1% of the branches. PET also detected that 519 source statements (1.3%) contained coding standards violations.

(b) The increase in execution time resulting from the PET instrumentation varies from 25 percent to 150 percent depending on the options used.

4.2.19 ATDG (Automated Test Data Generator)

a. Category - Verification and Validation Tool

- Test Case Selection (unit testing)
- Static Analysis
- Path Structure Analyzer
- Anomaly Detector
- Variables Analyzer
- Test Data Generation (aid to unit testing)

b. Sources

(1) Stucki, L. G. et al, Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two volumes.

(2) TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

c. Status

ATDG was developed by TRW for NASA in Houston. It is an experimental interactive tool for static error analysis and test data generation to be used on FORTRAN programs for the UNIVAC 1110. There is no batch mode version of the tool.

d. Description

ATDG examines a FORTRAN source program at the unit level to identify effective test paths, data constraints for path execution, and path dependent errors. (Unit level refers to a single subroutine, function or main program.) The values of program variables which will produce optimal test cases are generated.

ATDG constructs a network by identifying the transfers and connective properties between program segments. (A segment is a set of contiguous FORTRAN statements with no branch points.) The resulting directed graph is represented by a Boolean connective matrix. The structural and logical characteristics of the unit are used to define executable paths based on data constraints. The goal is to exercise each transfer at least once, using the fewest number of cases. Loops are considered to require one iteration and unexecutable paths are eliminated.

The static analysis function supplements the error detection function of conventional FORTRAN compilers by identifying path-dependent errors such as uninitialized variables, infinite loops and unreachable code.

e. Research and Findings - Source (1)

ATDG was among the automated verification/testing tools which were evaluated under contract to the NASA Goddard Space Flight Center (GSFC). At the time of the evaluation, the tool was considered to be representative of current research in program testing and proving. The advantage in the ATDG concept of connectivity matrices is the ability to handle complex units of code. Support for continued development of the tool was recommended.

4.2.20 DAVE (Documentation, Analysis, Validation and Error Detection)

a. Category - Verification and Validation Tool

- Anomaly Detector
- Static Analysis

- Variables Analyzer
- Standards Enforcer (detects violation of ANSI standards)
- Path Structure Analyzer
- Interface Checker
- Documentation Aid

b. Sources

(1) Osterweil, L. J. and Fosdick, L. D., "DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs," Software Practice and Experience, Volume 6, September 1976, pp 473-486.

(2) Stucki, L. G. et al, Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two volumes.

(3) Browne, J. C. and Johnson, D. B., "FAST: A Second Generation Program Analysis System," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp 142-148.

c. Status

DAVE is a FORTRAN validation tool and documentation aid developed at the University of Colorado. It applies to programs written in ANSI FORTRAN and can be modified to apply to other languages.

DAVE is written in FORTRAN and was designed for ease of portability. It contains approximately 20,000 source statements and executes in four overlaid phases, the largest of which requires approximately 50,000 decimal words of memory on the CDC 6400.

d. Description

The DAVE system for static analysis of FORTRAN programs detects the symptoms of a wide variety of errors, assures the absence of these types of errors, and exposes subtle data relations

and flows within programs. As an aid to program documentation, DAVE produces information about the use of all local and global variables in the program.

The primary methodology of the system is the use of data flow analysis to reveal suspicious or erroneous data usage within and between subprograms. A data base of information about each program unit is created. Each data base contains a symbol table, a COMMON table, a label table, and a statement flow table that represents the control flow graph. A program call graph is constructed to represent relationships between programs being called.

The system analyzes the sequential pattern of definitions, references, and "undefinitions" of values for variables on the principle that many common programming errors cause either of the following rules to be violated:

- That a variable must not be referenced unless previously defined (without an intervening undefinition).
- That once defined, it must subsequently be referenced before being redefined or undefined.

Among the errors that can be detected from violations of these rules are uninitialized variables, misspelling of variable names and labels, unequal lengths of corresponding argument and parameter lists, transposed variables, use of exhausted DO indices, and mismatched types and dimensions of arguments and parameters.

In order to detect the rule violations, it is necessary to know the usage of every variable in every statement; that is, whether it is used as input or output in each case. Therefore, a search of the subprograms is conducted along the paths defined by the structural analysis, to look for the rule violations. Unexecutable paths are not detected and are included in the analysis. DAVE addresses the problem of data passing through calling parameters and through COMMON, and provides information about these variables in terms of their input/output classification.

DAVE does not attempt to positively identify the exact nature of every error in a program. Instead, the program is probed for suspicious and elusive constructs. The programmer must then use the messages and warnings produced by DAVE to improve the program.

e. Research and Findings

(1) Osterweil and Fosdick - Source (1)

DAVE will be revised to reduce unwanted output, detect additional anomalies, use faster algorithms developed for global variable optimization, and provide for interactive user participation. It is expected that a future version of DAVE, redesigned for efficiency rather than flexibility, will be capable of analyzing larger program units in a smaller data base area and will execute faster.

(2) Browne and Johnson - Source (3)

Browne and Johnson state that DAVE represents one of the best FORTRAN validation tools available. However, the system does not provide the full range of analyses that its data collection facilities could support.

(3) Stucki et al - Source (2)

DAVE was among the automated verification/validation tools which were evaluated under contract to the NASA Goddard Space Flight Center (GSFC). The researchers state that DAVE has proven to be a valuable documentation and checkout aid. The operating cost was evaluated at 5 in a scale of 1 = low to 5 = high and the ease of use was evaluated at 2 in a scale of 1 = easy to 5 = difficult.

4.2.21 PFORT Verifier

a. Category - Code Production Tool

- Standards Enforcer
- Debug Tools - Cross Reference
- Documentation Aid
- Interface Checker

b. Source

(1) Stucki, L. G. et al, Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two volumes.

c. Status

The PFORT Verifier checks the portability of FORTRAN programs and is representative of a class of automated verification tools known as standards auditors or enforcers. PFORT has been widely implemented and is available from Bell Laboratories, Murray Hill, New Jersey.

d. Description

The PFORT Verifier checks a FORTRAN source program for adherence to PFORT, a portable subset of ANS FORTRAN. Subprogram communication is checked through common and argument lists. Debugging and documentation aids include a subprogram cross reference giving type, usage, and attributes of each identifier with a list of the statements in which the identifier occurs. A subprogram summary is also provided, listing argument attributes, COMMON blocks used, subprograms called, and the calling programs.

e. Research and Findings (Stucki et al)

The PFORT Verifier was among the automated verification tools which were evaluated under contract to the NASA Goddard Space Flight Center (GSFC). It was recommended for use at GSFC with the following observation: "can be especially valuable at GSFC when verifying that vendor programs coming into GSFC adhere to a prescribed set of standards." The operating cost was evaluated at

1 in a scale of 1 = low to 5 = high and the ease of use was evaluated at 2 in a scale of 1 = easy to 5 = difficult.

4.2.22 FORTRAN Structuring Engine

a. Category - Code Production and Analysis Tool

- Restructuring Program
- Preprocessor
- Formatter
- Structured Programming Aid

b. Sources

(1) Stucki, L. G. et al, Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two volumes.

(2) TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

c. Status

The FORTRAN Structuring Engine, which transforms FORTRAN programs into a structured superset of FORTRAN, has been implemented on the IBM 360/370 and is available from Caine, Farber & Gordon, Inc. (CFG), Pasadena, California. Feasibility studies have been performed for the development of a COBOL Structuring Engine.

d. Description

The FORTRAN Structuring Engine is a set of software tools which produce a SFORTRAN source program and listing from a FORTRAN source program. SFORTRAN is a highly structured superset of FORTRAN which was developed by CFG to extend and improve FORTRAN for the support of structured coding. A SFORTRAN to FORTRAN preprocessor is available to allow compilation and execution of SFORTRAN programs. The SFORTRAN program listing is designed to easily identify blocks of code and thus obtain a picture of the overall logical structure of the program.

The Structuring Engine accepts programs written in ANSI FORTRAN, IBM FORTRAN, UNIVAC FORTRAN V, CDC FORTRAN extended or unextended, and Honeywell FORTRAN.

e. Research and Findings (Source 1)

The FORTRAN Structuring Engine was among the automated tools evaluated under contract to the NASA Goddard Space Flight Center (GSFC). It was not recommended for GSFC due to the extreme resource requirements (one million bits of memory and high execution time).

The following comments are quoted from the evaluation report:

"Probably the best product of this type on the market today, in terms of structuring a given FORTRAN program. Note that even if the preprocessor is never used to execute the SFORTRAN representation of a particular FORTRAN program, the SFORTRAN listing can be an invaluable aid in determining what is going on in a large and complicated program."

"The Structuring Engine has serious practical limitations to go along with its sophisticated output, however, since a 1000K load module size makes it rather machine dependent. Also, the number of SFORTRAN lines starts to go up extremely fast as the number of branch points in the FORTRAN program increases."

4.2.23 PACE (Product Assurance Confidence Evaluator)

a. Category - Testing Tool

- Test Completion Analyzer
- Dynamic Analysis
- Path Flow Analyzer
- Execution Analyzer (software probes)
- Optimizer (aid to manual action)
- Usage Counter
- Test Case Selection (aid)
- Static Analysis

- Path Structure Analyzer (directed graph)
- Regression Testing (aid)

b. Sources

(1) TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

(2) Ramamoorthy, C. V. and Ho, S. F., "Testing Large Systems with Automated Software Evaluation Systems," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp 382-394.

c. Status

The PACE quality assurance tool is a product of TRW (SEID Software Product Assurance) and has been implemented on the CDC 7600, CDC 6500, and UNIVAC 1108. Various versions of PACE have been developed to meet the special requirements of individual users.

d. Description

PACE is a collection of tracing and managerial tools which assist in assessing test coverage for FORTRAN programs. Dynamic analysis includes the generation of software probes to record the number of times each statement is executed. Static analysis includes a segment transfer table, a segment description table, and a program structure summary which identifies and counts the various kinds of FORTRAN statements.

The major purpose of PACE is to quantitatively assess how thoroughly and rigorously a program has been tested with the objective of testing every logical and arithmetic instruction in every branch.

PACE also acts as a driver for the TRW FORTRAN Standards Auditor (STRUCT).

e. Research and Findings (Source 1)

TRW reports that very large subroutines may overflow internal PACE tables and very large programs may overflow core due to the added instrumentation.

f. Maintenance Experience (Source 2)

The following report concerning experience with PACE is quoted from Ramamoorthy and Ho:

"In testing and maintenance of the Houston Operations Predictor/Estimator (HOPE) program, cost savings achieved by the use of the PACE system was $8,000 per year. The PACE system disclosed that the existing test file consisting of 33 test cases covered only 85% of the programs and that one-half of this number were exercised by almost every test case. It required 4.5 hours of computer time and 35-50 manhours of test results evaluation. Consideration of these statistics initiated the subsequent analysis to produce a more effective test file. A file of six cases was generated. With this set of test cases, 93% of the subprograms were exercised and required less than 24 manhours of test results examination."

4.2.24 RXVP (formerly RSVP - Research Software Validation Package)

a. Category - Verification and Validation Tool, Testing Tool

- Test Completion Analyzer
- Static Analysis
- Path Structure Analyzer
- Anomaly Detector
- Debug Tools - Cross Reference
- Dynamic Analysis
- Execution Analyzer (software probes)
- Path Flow Analyzer
- Variables Analyzer
- Usage Counter
- Test Data Generation (guidelines for manual data preparation)
- Test Case Selection

- Restructuring Program (FORTRAN to IFTRAN)
- Test File Manager
- Test Status Reporting

b. Sources

(1) Miller, E. F. and Melton, R. A., "Automated Generation of Test Case Data Sets," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp 51-58.

(2) TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

(3) Ramamoorthy, C. V. and Ho, S. F., "Testing Large Systems with Automated Software Evaluation Systems," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp 382-394.

(4) Gilb, T., Software Metrics, Winthrop Publishers, Inc., Cambridge, MA, 1977, 282 pp.

c. Status

RXVP is an automated verification system and test case generator developed by the General Research Corporation (GRC). RXVP applies to programs written in FORTRAN and IFTRAN, a structured programming extension of FORTRAN. It has been implemented on CDC, IBM, and UNIVAC computers and is commercially available from GRC.

d. Description

RXVP offers an organized approach to program testing and guides the programmer in the systematic development of test cases that ensure thorough test coverage. A detailed static analysis of individual program modules and groups of modules is also provided. The various options of RXVP are selected by means of a comprehensive command language.

RXVP automatically performs a complete structural analysis of FORTRAN programs and stores the results in a data base. Additional or changed source code causes the existing data base to be updated. The detailed static analysis of individual program modules and groups of modules includes:

- Statement classification by type of statement
- Number of statements in each module and percentage of total
- Module cross reference table of variables, the statements that reference the variables, and their type of usage
- Check of array subscripts for indexing appropriate to the definition.

The control structure of the source code is automatically instrumented for dynamic analysis and statistics on program variables are recorded at the statement level during execution.

The internal program structure is used to identify a minimal set of test case patterns which will comprehensively exercise the program(s). The requirement is that every executable statement and every possible outcome of each branch statement be exercised at least once. A partitioning process is used to identify a hierarchy of subschema within the program's digraph; each subschema having a single entry and single exit. Appropriate aggregations of modules are then made and the iteration structure for each is identified. Backtracking techniques are used to recognize impossible paths and to generate specifications for the input variables which will cause execution of the required program flow for the test case. After the generated test case is executed with manually prepared test data and verified using the instrumentation and data collection facilities of RXVP, the process is repeated with another untested segment as the testing objective. RXVP keeps track of the untested segments and reports on the test coverage achieved. The effectiveness of the test case data is verified by manual comparison with the system functional specifications.

e. Research and Findings

(1) Ramamoorthy and Ho (Source 3) report that extensive man-machine interactions are required for the testing of programs and the test data preparation.

(2) Gilb (Source 4) reports that RXVP was run on a group of 26 programs that GRC programmers had approved as having met the criteria that 100 percent of the test paths had been exercised at least once and every possible outcome of each program decision statement had occurred at least once. RXVP showed that 23 of the 26 programs had less than 80 percent of these paths tested. Gilb states "This example illustrates the strength of automated tools compared with well-intentioned humans" and recommends that manual test monitoring and evaluation "should probably be augmented by such aids."

4.2.25 JOYCE

a. Category - Verification and Validation Tool

- Static Analysis
- Path Structure Analyzer
- Debug Tools - symbol cross reference
- Variables Analyzer
- Interface Checker
- Documentation Aid

b. Source

Stucki, L. G., et al, Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two volumes.

c. Status

JOYCE is a static analyzer and documentation aid for FORTRAN programs. It was developed by McDonnell Douglas, Huntington Beach, California and has been implemented on CDC computers.

d. Description

JOYCE is an automatic static analysis tool for FORTRAN programs. It accepts as primary input FORTRAN source decks in

the form of card decks or CDC compile files. The source decks are edited and the edited information is combined to produce several combinations of descriptive reports.

JOYCE compiles tables of symbols and cross references of symbol usage within each routine of a program. These symbols include FORTRAN variable names, the names of any referenced function or module, any entry points, and all I/O file references. Flowlists are provided in the form of microfilm FORTRAN listings with all transfers indicated by arrows to the right of the statement text and all DO loops indicated by brackets to the left.

Symbolic descriptions may be input on data cards to produce a completely cross-referenced program glossary. The data cards may describe or designate a variable definition, a mathematical symbol, flags for grouping related subjects, or subroutine usage information. The glossary may be listed on microfilm in a variety of formats at a program and/or subroutine level, and may be sorted on FORTRAN and/or mathematical symbol or any of several special modes (e.g. by storage location).

e. Research and Findings (Stucki et al)

JOYCE was among the automated verification tools evaluated under contract to NASA Goddard Space Flight Center (GSFC). The operating cost was evaluated at 2 in a scale of 1 = low to 5 = high and the ease of use was evaluated at 2 in a scale of 1 = easy to 5 = difficult. JOYCE was not recommended for use at GSFC because an IBM version was not available.

4.2.26 Semantic Update System

a. Category - Code Production and Analysis Tool, Configuration Control

- Automatic Modification (aid)
- Configuration Status Reporting
- Support Library (System Master Library)
- Redesign Analysis

- Static Analysis (analyzes side effects of proposed changes)
- Variables Analyzer
- Interface Checker
- Regression Testing

b. Source

Hirschberg, M. A., Frickel, W. G., and Miller, E. F., Jr., "A Semantic Update System for Software Maintenance," Proceedings COMPCON Spring 1979, San Francisco, CA, March 1979, pp 307-309.

c. Status

The Semantic Update System is an automated software maintenance tool for large-scale FORTRAN programs. The system is in prototype development; however, some user exposure has been obtained. Semantic Update is to be applied initially to a hydro-dynamics simulation system.

d. Description

Semantic Update is described as "a tool that assists in the incremental modification of software systems treated as systems". It can be applied to a program part, an entire module, a software subsystem, or an entire software system. The system is language dependent and is not capable of handling data files. However, it can determine the extent of side effects to proposed or directed changes and provide trial updates to determine the extent of side effects.

Ten classes of system commands have been identified in the preliminary design for the Semantic Update system.

(1) Semantic Update system control commands for choosing processing options, setting limits on side-effect scanning, etc.

(2) Single module modification commands having no multiple module side effects, such as requests to change local FORTRAN variable DIMENSIONality within a module.

(3) Single module modification commands having multiple module side effects, such as SUBROUTINE name changes or formal parameter type changes.

(4) Commands controlling the content of global program declarations, such as addition, modification, or deletion of items from a particular COMMON block.

(5) Software system module redefinition commands that may be used for changing module names or formal parameter lists.

(6) Global macro redefinition commands for making modifications to source-level macro definitions.

(7) Software system structure commands that permit selection of an alternative module version from among a set of nominally equivalent versions resident in the system master library.

(8) Semantic Update operations commands that are used to control the system logic and values in conditional operations.

(9) Information storage commands for adding information to the system development archive, such as annotations by system maintenance personnel.

(10) System status report commands that initiate report processing for presentation of status information, such as modules listings and side-effect interaction levels.

In addition the system will process commands in different modes based on "(1) the extent to which command-to-command interaction is to be considered during a particular Semantic Update session; and (2) the current limits placed on side-effect analysis".

e. Research and Findings (Hirschberg, Frickel, and Miller)

The authors cite estimation of an overall cost-benefit ratio based on use of Semantic Update compared with conventional update methods as an important requirement. This should be accomplished after significant user experience with Semantic Update has been accrued. It is anticipated that Semantic Update will provide measurable improvement in update capability and perhaps will apply to some types of applications that could not be treated by any other means.

4.2.27 FAST (FORTRAN Analysis System)

a. Category - Verification and Validation Technique

- Static Analysis
- Variables Analyzer
- Interface Checker
- Anomaly Detector
- Redesign Analysis

b. Source

Browne, J. C. and Johnson, D. B., "FAST: A Second Generation Program Analysis System," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp 142-148.

c. Status

The FAST FORTRAN analyzer is a specific purpose program product which is built upon the use of existing general purpose products. It uses the commercially available data management system, System 2000 (MRI Systems Corporation) as its data handler and data correlator along with the FACES source program parser and the BOBSW parser generator. The developers of FAST propose it as a model for the development of similar specific purpose program products in the future, particularly since FAST was implemented with only 3 1/2 man months of effort.

FAST is specifically designed for interactive usage. There is no batch interface, but terminal output can be directed to a line printer.

d. Description

The FAST system creates a data base of the attributes of modules, statements and names in a FORTRAN program and interactively processes a wide range of queries concerning these attributes.

The FAST data base is generated from the FORTRAN source program by using

- the FACES parser
- a program to map the output of the parser onto System 2000 load strings
- the System 2000 data management system.

The FAST command/query language, which is used to query the data base, defines approximately 100 attributes of FORTRAN names and statements. These attributes can be combined in logical expressions to qualify or isolate very broad or very narrow program contexts. The command language interpreter was implemented through the use of the BOBSW parser generator (University of Texas at Austin).

The FAST command/query language is used to request displays of statements or variables which satisfy specified attributes or a logical expression of attributes. The range of the query may be program wide, intra-module, or within specified program lines. Displays include

- The attributes of specified variables (such as type, class, scope, and module environment)
- Variables satisfying a specified range of attributes (such as DO-control variables or actual parameters)
- Statements satisfying a specified combination of attributes (such as referencing selected statement labels containing specified variables, or containing variables used in a certain way)
- Variables which may be affected by a change to a specified variable
- Trace of variables which may affect the value of a specified variable
- Improperly aligned parameters
- Improperly aligned COMMON blocks
- Uninitialized variables.

e. Research and Findings

Browne and Johnson report that much of FAST's power and flexibility as well as its low cost of implementation derives from the use of a general purpose data management system as an integral component. FAST's capabilities are designed to be especially well suited to the program maintenance environment as well as for program development and debugging. The authors state: "The capability for qualifying and isolating segments of program text by association with key attributes should be particularly valuable in the program maintenance environment where the programmer/analyst will not usually be familiar with the local program context and associations".

4.2.28 ASSET (Automated Systems and Software Engineering Technology)

a. Category - Integrated Verification and Validation Tools (Requirements, design, and code)

- Redesign Analysis
- Configuration Status Reporting
- Support Library
- Requirements Determination
- Pseudo-code (requirements and design languages)
- Utilities
- Static Analysis
- Symbolic Execution
- Dynamic Analysis

b. Source

Osterweil, L. J., Brown, J. R., and Stucki, L. G., "ASSET: A Lifecycle Verification and Visibility System," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp 30-35.

c. Status

Development of a system implementing the ASSET architecture is currently underway at Boeing Computer Services Company. Early efforts are focusing on implementation of key analytic capabilities and front ends to process requirements, design and specific coding languages. The manipulative functions for the integrated data base are also being developed.

d. Description

ASSET is an integrated system of tools and techniques which is designed to facilitate the transition from one development phase to the next and to determine that the transitions have been made correctly. Upgrading the system in the maintenance phase is considered to be a reiteration of the development phase (requirements analysis, preliminary design, detail design, and code). Testing and verification are considered to be a continuing activity throughout the development (or maintenance) process rather than a separate phase.

The principal component of ASSET is a central data base containing all of the information needed for making and implementing management decisions about a program, including

- Source code
- Object code
- Documentation
- Support libraries
- Project utilities
- Requirements specifications
- Design specifications (all available levels)

Incoming source representations (code, design specifications or requirement specifications) are first scanned by a static analyzer using graph analysis techniques. (DAVE is mentioned as an example.)

Symbolic execution is applied to the design and requirements specifications. Although the technique is applicable to source code, the cost is considered to be too high for source code verification.

Dynamic analysis is considered to be the more successful verification technique for source code. (PET is mentioned as an example.) This approach can also be applied to simulated processes which model early requirements and analyze their interactions.

Formal verification is offered as an option which is expected to be most effective at the higher levels of requirements and design specification. In formal verification the complete definitive functional effect of an algorithmic specification is determined and compared to the complete definitive statement of the program's intent. The determination of effect is made by symbolically executing every algorithmic path.

4.2.29 System Monitor

a. Category - Redesign, Code Production and Analysis Tool

- Automatic Recovery
- Automatic Modification
- Comparator
- Redesign Analysis
- Debugging Tools - interactive
- Automatic Reconfiguration

b. Source

Yau, S. S., Cheung, R. C., and Cochrane, D., C., "An Approach to Error-Resistant Software Design," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp 429-436.

c. Status

Exact status of this maintenance tool is not identified in the source article. However, it might be inferred from the information presented that System Monitor is operational at least in a prototype state and that it has been applied to operational application software systems.

d. Description

System Monitor is a software system which has been developed to provide error detection, error containment, and functional recovery support to applications software at the program, module, and system levels. It is comprised of five components: Internal Process Supervisor, External Process Supervisor, Interaction Supervisor, System Monitor Kernel, and Maintenance Program. These components and their functions are described as follows:

(1) Internal Process Supervisor. One Internal Process Supervisor (IPS) will be created for each internal process of the application software to be monitored. The IPS contains two components, the Program Supervisor and the Module Supervisor, that will check the process for reliable execution. The Program Supervisor monitors control and data flow between modules. The Module Supervisor checks module functional reliability. Upon detection of an error or software failure, global data values are saved and alternate versions of modules are called in during recovery attempts.

(2) External Process Supervisor. The programmer creates one External Process Supervisor (EPS) per global data structure. The EPS determines if program modules have access to reliable data. The EPS is composed of error detection routines, recovery data, and abstract data types support facilities. Detection of an error causes initiation of attempts at alternate module access, error correction or data repair by EPS.

(3) Interaction Supervisor. The Interaction Supervisor (IS) monitors and controls external process interactions to ensure the reliability of these interactions. The IS first validates interactions, then provides facilities for validated interactions and finally supervises the interactions. For failure of an interacting process, the IS coordinates recovery among the interacting processes.

(4) System Monitor Kernel. The System Monitor Kernel is responsbile for checking the integrity of the Process Monitor and thus ensuring the integrity of System Monitor itself.

(5) Maintenance Program. The Maintenance Program is a part of the System Monitor operating system and is executed to attempt repair of faulty system processes. It compares failed software and data to backup copies of code and data, and uses some interactive debug capabilities. The corrected versions are automatically put into service by the System Monitor.

e. Research and Findings (Yau, Cheung, and Cochrane)

The authors point out that System Monitor does impose certain overhead costs through its internal and external interfaces, state saving processes for recovery, and error detection procedures.

Further research is planned in order to improve System Monitor performance and develop more effective tests and recovery techniques.

4.2.30 Reliability Measurement Model

a. Category - Operations Evaluation Tool

- Failure Data Analysis
- Test Completion Analyzer

b. Source

Musa, John D., "Software Reliability Measures Applied to System Engineering," Proceedings AFIPS Conference, Volume 48, AFIPS Press, Montvale, NJ, pp 941-947.

c. Status

A portable FORTRAN program for measuring software reliability is available on magnetic tape from John D. Musa, Bell Laboratories, Whippany, New Jersey. A feedback of the data collection (with appropriate safeguards) is requested for use in refining and improving the reliability theory.

The program can be run interactively or in batch mode on most large computer systems.

d. Description

Mean-time-to-failure (MTTF) is a useful metric for characterizing system operation and for controlling change during the maintenance phase. An automated model has been developed using a number of fundamental equations which relate failures experienced, present MTTF, MTTF objective, and time required to meet the MTTF objective. The model requires the input of the execution time intervals between experienced failures, the MTTF objective and a parameter describing the environment.

The model can provide a quantitatively-based mechanism for change control in the Operations and Maintenance phase. Generally, the MTTF will drop after the installation of software changes and improve during the following period of error removal. If the MTTF can be tracked and if MTTF service objectives can be

set for the system, the model can be used as a tool for the management of system modifications. When the MTTF falls below the service objective, the system can be frozen until improvement occurs. The manager may use the amount of margin above the service objective as a guide to the size of the changes permitted at any given time.

e. Research and Findings (Musa)

The reliability theory has been applied to several software development projects and operational systems. It can be used in system engineering, test monitoring, and change control of operational software. Experience in application of the theory should lead to its further refinement and broadening, resulting in greater accuracy and wider utility.

4.2.31 FACES (FORTRAN Automated Code Evaluation System)

a. Category - Verification and Validation Tool

- Static Analysis
- Path Structure Analyzer
- Anomaly Detector
- Variables Analyzer (intramodule)
- Interface Checker
- Standards Enforcer
- Debug Tools - cross reference
- Reachability Analyzer
- Regression Testing (aid)
- Automatic Documenter

b. Sources

(1) Ramamoorthy, C. V. and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp 382-394.

(2) Browne, J. C. and Johnson, D. B., "FAST: A Second Generation Program Analysis System," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp 142-148.

(3) TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

c. Status

FACES is an ANSI FORTRAN analysis program developed for the Marshall Space Flight Center, and is commercially available from COSMIC, University of Georgia. The system is composed of a front end consisting of 6,000 source statements and a collection of diagnostic and interrogation routines consisting of 2000 source statements.

FACES is designed for transferability among machines and has been implemented on the UNIVAC 1108, the CDC 6400, and the IBM 360/65.

d. Description

FACES is a FORTRAN static analysis system consisting of three parts: the FORTRAN front end, Automatic Interrogation Routine (AIR), and a report generator.

The FORTRAN front end constructs a data base from the program source code. The data base consists of three main tables: a SYMBOL TABLE which contains an entry for each symbol in the module along with its type, a USAGE TABLE which contains an entry for each occurrence of a symbol along with its associated usage and a NODE TABLE which identifies the type of each statement and its logical predecessors and successors. Each of these tables contains explicit pointers to the others for efficient movement among the entries.

The Automatic Interrogation Routine (AIR) interprets queries and automatically searches the data base. The user may query the entire system or an individual routine, by variable names or by lists of attributes. AIR can be used to check for:

- Error prone constructs, such as calls to subroutines which pass explicit constants as parameters.
- Interface
  - --To verify that all COMMON blocks with the same name have the same number of elements and that corresponding elements agree in type and dimensionality. Optionally, corresponding elements may be checked for agreement in name.
  - --To verify agreement between the formal parameter lists of routines defined within the program and the actual parameter lists used to reference those routines. Actual parameter lists must agree in argument number, type and, optionally, dimensionality.
- Redundant and unreachable code
- Loop construction and termination
- Coding standards
- Uninitialized variables (checks are limited to local variables within a module).

A variable trace routine displays for a given variable, at a particular line in a program module, either those variables which have affected its value up to that line or those variables it will affect after that line. The trace is limited to local variables within a module.

The FACES data base can also be used for documentation generation, including cross reference tables (variable versus statement and COMMON block versus subroutine), a subroutine calling sequence table, and a program graph.

e. Research and Findings - Source (1)

(1) Design

The FACES system is designed to provide automated maintenance support as well as program development support. Maintenance support is based on documentation generation and interrogation of the data tables of program characteristics. The maintenance applications include:

- Aid in predicting the effect of proposed changes
- Validation of the modifications
- Aid in the selection of test cases for retesting.

(2) Validation Experience

FACES has been used to evaluate other software systems, and has also been used to analyze itself on subsequently developed versions.

While analyzing AIR by FACES, the following errors were discovered:

- Three instances of misspelling of variables and one instance of transposition of variables in COMMON block declaration were detected by the Common Block Alignment Check.

- Two subtle keypunch errors that changed the names of two variables were detected as uninitialized variables.

4.2.32 Optimizer II

a. Category - Code Production Tool

- Optimizer

b. Source

TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

c. Status

Optimizer II is an automatic optimizer for ANS COBOL programs on the IBM 360/370. It is commercially available from the Capex Corporation, Phoenix, Arizona.

d. Description

Optimizer II is included in this report as an example of tools which automatically optimize the performance of computer programs.

Optimizer II automatically improves the efficiency of the object code generated by the IBM ANS COBOL compilers. Its object level analysis provides savings in main processor time and main storage requirements which cannot be obtained at the source-code level.

e. Research and Findings

TRW reports that savings of up to 25 percent in execution time and 20 to 30 percent in memory requirements may be achieved.

4.2.33 CASEGEN

a. Category - Testing Tool

- Test Data Generation
- Test Case Selection
- Symbolic Execution

b. Sources

(1) Ramamoorthy, C. V., Chen, W. T., Han, Y. W., and Ho, S. F., "Techniques for Automated Test Data Generation," Proceedings Ninth Asilomar Conference on Circuits, Systems and Computers, Pacific Grove, CA, November 1975, pp 324-329.

(2) Ramamoorthy, C. V., Ho, S. F., and Chen, W. T., "On the Automated Generation of Program Test Data," IEEE Transactions on Software Engineering, Volume SE-2, November 4, December 1976, pp 293-300.

c. Status

CASEGEN is a prototype test data generation system which has been designed as a part of the FORTRAN Automated Code Evaluation System (FACES) described in Paragraph 4.2.31.

d. Description

CASEGEN is designed to generate test data automatically for testing FORTRAN programs. It consists of four major subsystems:

- FORTRAN source code processor (FSCP)
- Path generator
- Path constraint generator (PCG)
- Test data generator.

FSCP generates a data base consisting of the program graph, the symbol table and the internal representation of the source code. The path generator, by partitioning the program graph, generates a minimal set of paths to cover all edges. (Some of the generated paths are infeasible.) The path constraint generator uses symbolic execution to produce a set of equality and inequality constraints on the input variables. The test data generator creates a set of inputs which satisfy the constraints and can be used to execute the program path. The constraints are solved by means of random number generation and systematic trial and error procedures, with values being assigned to program variables until all constraints are satisfied.

A user-oriented language has been designed to allow the user to specify additional information about the program such as upper and lower bounds of input variables, number of loop iterations and relations among program variables.

e. Research and Findings - Source (2)

CASEGEN is undergoing further testing and tuning. Determining the number of loop iterations is still a major obstacle and user assistance is sometimes necessary.

4.2.34 Testing System

a. Category - Testing and Integration Technique

- Automatic Test Driver
- Test Case Selection
- Test Completion Analyzer (aid to user analysis)
- Test Status Reporting
- Test Data Generation (through programs designed specifically for the system being tested)
- Interactive Execution
- Debug Tools - dumps, etc.

b. Source

Cicu, A., Maiocchi, M., Polillo, R., and Sardoni, A., "Organizing Tests During Software Evolution," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp 43-50.

c. Status

Testing System (TS) is operational in prototype form with a limited set of functions being exercised. It has been used to monitor testing of the Honeywell GCOS Level 62 operating system. TS is specifically designed as an aid to organizing tests for operating systems which are undergoing modification for every new release. However, the concepts for automated testing are generally applicable to any software which is undergoing extensive maintenance.

d. Description

The source authors describe Testing System requirements in terms of system adaptability and expandability. That is, a testing system should be able to "locate areas where testing activities cannot be pursued any further", (for example, where a fault has been detected but not yet repaired), and adapt system testing to alternative areas. Also, as the operating system which is being tested experiences modifications a testing

system must be capable of adding new tests and altering or suppressing existing tests through easily implemented increments or changes. With these requirements in mind the TS is being developed into a highly structured and, to the extent possible, automated system.

The automatic testing is performed primarily through the execution of testing programs which are activated from a hierarchical "catalog" of testing programs according to the specific catalog structure or to operator requests. Within each testing program, dependent modules are activated according to the outcome (success of failure) of the previously executed module or to operator requests. When actual test results do not match the expected results, messages are issued to help locate the error and debug tools are activated.

The TS contains the following components:

Testing Modules and Testing Programs.

A testing module is a code segment structured with one entry and one exit point. It is designed to test one well-defined feature of the system. Testing modules make as few assumptions as possible about the operating system internal structure, are self-documenting, and are portable.

Modules which test related features of the system are grouped to form a testing program. The testing program is a job step which is activated from within job control language (JCL) procedures.

Mutual dependencies among modules within a testing program are described by means of a module dependency graph. Two dependency situations are typical:

- Two testing modules are independent and can be executed in any order
- Two testing modules are sequenced and one can be executed only after successful completion of the other.

The TESTER (Testing Modules Handler)

The TESTER routine scans the dependency graph and controls execution of the testing modules according to the outcome (success or failure) of the previously executed module or to operator requests. TESTER can stop execution when unsuccessful runs occur at key points, bypass blocked modules, execute a single module, execute a particular module and all previous modules required for its execution, issue messages defined by the programmer to describe the purpose of the test and issue messages on the overall results (success or failure) of the test.

Testing Units

Testing units, the JCL statements which activate and control the testing programs and test data generator programs, are the smallest meaningful entity in the management of testing activities. Testing units perform disc preparation, files preallocations, etc., prior to execution of a testing program and provide messages and dumps if the test fails. Test results which cannot be checked automatically are interactively communicated to the operator. Testing units, like the test modules, are designed to be self-documenting and portable.

Testing Structure

The testing structure is a hierarchical catalog of all testing units for the system, organized according to the area of testing activity. Testing structures are defined through graph notation, similar to the module dependency graphs, to establish testing unit dependencies.

HLTESTER (Testing Unit Handler)

The testing unit handler (HLTESTER) scans the testing structure, monitors the execution of testing units, and selects the next testing unit according to the results of the previous execution and/or to operator requests. That is, it performs

essentially the same services as TESTER, but at a higher linquistic level.

Data Base

The data base was not implemented at the time of the source article. The primary objective of a data base is to provide automatic test documentation and a capability for review of testing activity status at any desired level of detail. The criteria stated for organizing a data base are that it reflect the hierarchical structure of the TS and that it be capable of being updated directly at any level with test results.

e. Research and Findings (Cicu, Maiocchi, Polillo, and Sardoni)

The authors report that completing a comprehensive set of operating system tests under control of a TS was difficult because: testing coordination problems existed among different development groups, tests to check job-management functions cannot be monitored from within a job, and all operating system features required to support TS operation were not available with early system releases. As a result of these difficulties the TS was implemented in a bottom-up approach typically beginning with test module and test data generator design. Also, the requirement for a test unit handler and data base is created only after a "relevant number of tests is available" and those tests have been effectively used.

Use of TS is said to improve system maintainability and simplify test activity management.

Future research will be devoted to increasing maintainability factors and reducing the clerical overhead required for release management by expanding the data base content.

4.2.35 ISMS (Interactive Semantic Modeling System)

a. Category - Verification and Validation Tool

- Execution Analyzer (software probes)
- Dynamic Analysis
- Static Analysis
- Path Structure Analyzer
- Variables Analyzer
- Interface Checker
- Assertion Checker
- Path Flow Analyzer
- Usage Counter
- Timing Analyzer

b. Source

Fairley, R. E., "An Experimental Program Testing Facility," IEEE Transactions on Software Engineering, Volume SE-1, November 4, December 197[illegible], pp 350-357.

c. Status

ISMS is an experimental research tool for static and dynamic analysis of computer programs. It is designed to permit rapid implementation of a variety of tools for collecting, analyzing, and displaying testing information with the purpose of determining the most useful types of information and the most meaningful way to display that information.

At the time of the source article, ISMS was partially implemented. Programs written in ALGOL 60 were being used for the experimentation, with a FORTRAN version in progress.

d. Description

ISMS establishes a data base from the static analysis of the program structure and the dynamic analysis history of one or more program executions. The data base is accessed by means of "semantic models" of program execution which display the program behavior in either the control flow domain or the

data flow domain. The data base may be accessed in either batch or interactive mode.

The ISMS preprocessor performs syntactical analysis of the program text to record the structural attributes of the program in the data base and instruments the source code with subroutine calls to collect the execution history.

Execution of the instrumented program collects all or a specialized part of the program's execution history. Symbol names are interfaced with data values and program text is interfaced with flow of control. The history of program events (any change in the computational state) may be collected by type of event, such as the sequence of subroutine calls through the entire execution, or region of event, such as data flow and control flow in a given segment. If collection criteria are not specified, all events for the entire execution are collected.

Displays from the data base allow each step in the execution history to be reconstructed either forward or backward. Backward interpretation permits analysis of how a given computation was influenced by previous computations. Also, the entire execution history can be scanned to collect global information and summary statistics. The types of information available for display include:

- Program graph structure
- Overview of source code structure
- Ranges of variables
- Statement execution counts by statement type or statement number
- Branch execution counts
- Control flow traces and tracebacks
- Data flow traces
- Data sensitivity analysis (the effects of input data inaccuracies and finite word length on the computation being performed)

- Sequence checks
- Parameter passing environments
- Timing estimates
- Assertion checks
- Data values of variables (complete or selective)
- Dependence of data values on other data values

e. Research and Findings (Fairley)

Fairley states that "the important design features of the ISMS are: 1) the syntax driven nature of the preprocessor; 2) the isolation of data collection from data analysis and display; and 3) the independence of the collection, analysis, and display routines from the internal details of data base implementation".

The major disadvantage of ISMS is the potentially large size of the data base. Determining the extent of this problem is an important aspect of the continuing research.

4.2.36 Computer Program Transformation

a. Category - Redesign, and Code Production and Analysis Technique

- Automatic Modification
- Redesign Analysis
- Code Comparator
- Optimizer

b. Sources

(1) Boyle, J., and Matz, M., "Automating Multiple Program Realizations," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp 421-449.

(2) Dershowitz, N., and Manna, Z., "The Evolution of Programs: Automatic Program Modification," IEEE Transactions on Software Engineering, Volume SE-3, Number 6, November 1977, pp 377-385.

(3) Arsac, J. J., "Syntactic Source to Source Transforms and Program Manipulation", Communications of the ACM, Volume 22, Number 1, January 1979, pp 43-53.

c. Status

Program transformation as described in the source articles has been implemented through both automated and manual procedures. These articles present program transformation as a methodology with significant potential for improving program redesign and production capabilities both for development and maintenance functions. Automated procedures have been realized through the Transformation-Assisted Multiple Program Realization (TAMPR) system developed at Argonne National Laboratory. Both manual and semiautomated (using interactive software support) procedures have been implemented to achieve program transformation.

The systems described in the source articles are presented as prototype or experimental systems. In general they are discussed in the context of their application to a limited range of program transformation environments.

d. Description

The techniques of program transformation as presented by the source articles evolve from the following concepts:

- Achieve economics in the application of valid software routines within multiple hardware environments by automated transformation of the source statements.
- Extend program capabilities by automatically modifying the program process to produce new or expanded output.
- Reduce program complexity through syntactic and semantic source to source transformation of program code.

The automated transformation system TAMPR is described as "a program manipulation system which permits one to describe (and to automate) the construction of different realizations

of the same prototype program by means of source-to-source transformations" (Source 1). The authors propose that certain well-defined, automated processes (such as mathematical routines) may be economically extended to alternate program realizations through automatic transformation. Thus, a validated routine implemented on one processor may be transformed to an alternative, valid routine to run on a different processor. Source 1 describes in detail the application of TAMPR to transformation of a set of linear algebra modules. As discussed in this article, TAMPR operates on routines written in FORTRAN.

Automatic program modification is based upon finding "an analogy between the specifications of the given and desired programs, and then transforming the given program accordingly" (Source 2). Global transformations are emphasized; that is, transformations in which all occurrences of a particular symbol throughout a program are affected. This process may be automated and has been implemented in the high level language QLISP. As described in the source article the modification process consists of eight steps in three phases:

- Premodification Phase
  - (1) Annotation of the given program. Establish invariant assertions such as relation between input and output variables.
  - (2) Specifications rephrasing. Express the given and desired program specifications in equivalent form.
- Modification Phase
  - (3) Analogy definition. Develop transformations that yield desired program specifications from given program specifications.
  - (4) Modification validity determination. Check verification conditions and develop any additional transformations needed to preserve program correctness.

(5) Transformations application. Apply transformations to given program.

(6) Unexecutable statements rewrite. Incorrect variable assignments and non-primitive expressions must be replaced.

- Postmodification Phase

(7) New segment synthesization. Introduce new code where necessary to complete the new program expression.

(8) Transformed program optimization. Optimize all new properties of the transformed program.

Automatic program modification may be applied as a program debugging tool to develop a new program realization when errors prevent the old program from operating according to specifications.

Source code transform using syntactic and semantic manipulation is presented as an effective technique for creating a "clear, simple and reasonably efficient" program. Syntactic transform is implemented using catalogs of known syntactic transformation properties. By applying certain of these along with selected local semantic transforms a new realization of a given program can be achieved. The goal for this transformation process is to create less complex program realizations.

The Source 3 author reports that this technique has been implemented through an interactive system written in the high order language SNOBOL. A detailed presentation of the syntactic and semantic transform methodology is given in Source 3.

e. Research and Findings

(1) Boyle and Matz (Source 1)

The authors state that TAMPR does provide automated program manipulation and transform notation for aiding construction of related program realizations. By extending the validity of a

particular program to a set of related (transformed) programs the testing requirements for these programs can be significantly reduced.

(2) Arsac (Source 3)

Program manipulation is presented as a "worthwhile tool" where program efficiency is important and for creation of uncomplicated programs. In the view of the author more experimentation with the tool remains to be done in order to develop a better understanding of the methodology and its interactive implementation.

4.2.37 Numerical Software Testbed

a. Category - Testing Technique

- Testbed
- Static Analysis
- Formatter
- Variables Analyzer (symbol table)
- Dynamic Analysis
- Execution Analysis (software probes)
- Path Flow Analyzer
- Usage Counter
- Test Data Generation (guidelines)
- Completion Analyzer
- Automatic Driver (special test program for each routine to be tested)

b. Source

Hennell, M. A., Hedley, D., and Woodward, M. R., "Experience with an ALGOL 68 Numerical Algorithms Testbed," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp 457-463.

c. Status

The Numerical Algorithms Group (NAG) library consists of approximately 300 numerical algorithm routines in both FORTRAN IV and ALGOL 60, with a lesser number in ALGOL 68. The library is available to all British universities and a number of other universities, covering the equipment of five major manufacturers. Each routine has associated with it a stringent test program and a results file so that if a fault or an incorrect implementation is suspected, the results produced by the stringent program can be compared with the expected results. All new routines are subjected to testbed analysis as an integral part of the normal code auditing and validation process. The analysis is not intended to test the numerical algorithm itself (NAG has a separate validation process for that purpose), but rather to rigorously exercise the program code.

d. Description

The term testbed is used in analogy with aeronautical procedures, where the engine is tested exhaustively on a testbed to determine its operating characteristics. The testbed was orginally designed to study run-time performance and consists of three major components: a static component, a dynamic component, and an analysis component displaying various aspects of the execution history.

The static component performs static analysis of FORTRAN or ALGOL source code and collects statistics, including the identification of all possible program jumps. The FORTRAN analysis also gives statement count by type, annotates the source code and constructs a symbol table. The ALGOL 68 analysis reformats the source code so that each part of choice clauses (such as IF ... THEN ... ELSE ...) is on a separate line.

The program is then run with a modified compiler which stores the run-time execution history in a data base.

The events monitored are jumps, subroutine entry and exit, predicate values, assignment values, and loop entry and exit. The events monitored can be freely switched on and off at any point within the testing program.

The analysis of the data base can be carried out in interactive or batch mode. Typical information produced by the analysis includes statement execution frequency counts, jump execution frequency counts, a trace of control flow, and a breakdown of the executed paths. In interactive mode, the execution history can be interrogated in either a forward or backward direction so that the control flow sequences which reach a particular point may be investigated.

Each NAG routine is tested with its associated stringent test program which attempts to exercise each statement and each program jump at least once. A jump is defined as occurring when the line number of the next (reformatted) statement differs from that of the current executable statement by other than +1 or 0.

e. Research and Findings (Hennell, Hedley and Woodward)

The authors have found that:

(1) The testbed analysis of the stringent tests is of great value. Certain goals, such as the execution of every statement in a large body of code, become realizable only through the automated testbed tools.

(2) The testbed does not directly detect program bugs but does focus attention on areas of code, such as unexecuted code, where undetected bugs may be present.

(3) The results of the dynamic analysis provide the most effective guidelines available for obtaining better tests.

f. Maintenance Experience

(1) FORTRAN and ALGOL 60 routines were translated into ALGOL 68. The testbed was then used to exhaustively test the ALGOL 68 programs. Analysis of the unexecuted statements and attempts to derive test data to exercise these statements, revealed a number of previously undetected program bugs in the ALGOL 68 versions and provided new insights into the requirements for test data generation.

(2) The stringent test program associated with each rcutine in the NAG library is a part of the standard maintenance process. The goal of the stringent test is to exercise every statement and every possible program jump at least once. The stringent test program is used in corrective maintenance when a fault is suspected and in adaptive maintenance to verify that implementation on a particular machine is correct.

4.2.38 TPL/F (FORTRAN Test Procedure Language System)

a. Category - Testing and Integration Technique

- Automatic Test Driver
- Test Procedure Language
- Completion Analyzer
- Stubs
- Assertion Checker
- Output Processor/Analyzer
- Test Status Reporting
- Dynamic Analysis
- Usage Counter
- Regression Testing

b. Sources

(1) Panzl, D. J., "Test Procedures - A New Approach to Software Verification," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp 477-485.

(2) Panzl, D. J., "A Language for Specifying Software Tests," Proceedings National Computer Conference, Anaheim, CA, June 1978, pp 609-619.

c. Status

The FORTRAN Test Procedure Language (TPL/F) was developed by General Electric at their Schenectady, New York facility. It is used by GE to specify test procedures for FORTRAN software. The TPL/F system is designed to provide software verification for FORTRAN programs, however, the concepts upon which this approach was developed are not language specific, and could be used for languages other than FORTRAN.

d. Description

The TPL/F system automatically executes software test cases and verifies the test results. A formal test procedure coded in a special test procedure language (TPL) is used to describe the test cases and control an automatic test driver. The test procedure takes the place of the test data and test setup instructions of conventional testing and provides a standard format for software test specification.

A test procedure specifies one or more test cases which cause actual execution of a target subprogram, module, or group of modules. Stub versions of missing subprograms are coded in FORTRAN-like statements embedded in the test procedure. Typically, a test procedure for a FORTRAN module of 50 to 100 statements may contain 20 to 50 test cases. To reduce the notation required to represent a test case, the TPL/F system uses a built-in macro processor and each specific test case uses a single macro call. The test procedure approach permits a great deal of freedom in executing either module, integration, or regression testing.

Test cases consist of execution instructions, input values, and model output values for the program to be tested. Execution instructions specify where to begin and where to

terminate a test case execution and how many times to execute it. A VERIFY statement is used to specify an assertion about the target program which can be checked at a given point in the execution.

The automatic software test driver applies a test procedure to one or more program modules, executes the specified test cases, verifies that the results of each test are correct, and reports the degree of testing coverage achieved.

Execution of TPL/F test cases is accomplished in three steps. First, the target program is initialized using initialization code within the test procedure to assign initial data values. Second, target program execution occurs. The range of execution is governed by the execution directive that appeared in the test definition. Third, upon termination of execution the execution states are verified in accordance with the specified assertions. Results of the verification process are produced in the form of a test execution report which contains execution success/failure information and data on the percentages of target program statements and branches executed.

e. Research and Findings (Panzl)

The author expresses the opinion that implementation of the TPL and test procedure concept may improve the quality of software test design. This is likely because now software tests may be formalized in executable and readable form. Through use of the automatic test driver, software testers can receive feedback on the degree of testing thoroughness. The importance of automatic test drivers and formal test procedures to the maintenance of production software is also cited. Now, test cases may be retained over the entire life-cycle of the software. This means that post-release test results of the software can be automatically compared with pre-release test results and the impact of post-release program modifications can be assessed.

4.2.39 DATFLOW (Data Flow Analysis Procedure)

a. Category - Testing and Integration Tool

- Variables Analyzer
- Static Analysis
- Path Structure Analyzer
- Reachability Analyzer

b. Sources

(1) Allen and Cocke, J., "A Program Data Flow Analysis Procedure," Communications of the ACM, Volume 19, Number 3, March 1976, pp 137-147.

(2) Osterweil, L. J., "A Methodology for Testing Computer Programs," Proceedings Computers in Aerospace Conference, Los Angeles, CA, November 1977, pp 52-62.

c. Status

The DATFLOW data flow analysis tool has been implemented and used in a PL/I oriented Experimental Compiling System. A PL/I listing of the procedure is available from the source article.

d. Description

The DATFLOW procedure determines data flow relationships within a program by a static, global analysis. Definition-use ("def-use") relationships are expressed in terms of the control flow graph of the program.

A data definition is an expression or part of an expression which modifies a data item. A data use is an expression or part of an expression which references a data item without modifying it. For a given data definition, DATFLOW identifies the uses which will be affected. For a given use, DATFLOW identifies the data definitions which supply the value. The procedure also identifies "live" data definitions at a given instruction in the program; that is, the data definitions prior to the instruction that are used following the instruction.

The basic data flow analysis algorithms and a listing of a PL/I procedure are given in the source article (Source 1).

e. Research and Findings

Osterweil (Source 2) makes the following comments concerning data flow analysis techniques: "With the realization that so much valuable static analysis can be carried out by adaptations of a few basic data flow analysis algorithms, comes an appreciation of the pivotal importance of these algorithms. Hence, the further study of these algorithms must be recognized as a research area of importance" (to the development of an integrated testing, analysis and verification system).

4.2.40 LIBRARIAN

a. Category - Configuration Control Tool

- Support Library
- Configuration Status Reporting
- Debug Tools - Editor

b. Sources

(1) TRW (Catalog), Software Tools Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

(2) McNurlin, B. C., "Using Some New Programming Techniques," EDP Analyzer, Volume 15, Number 11, November 1977, pp 1-13.

c. Status

LIBRARIAN is mentioned in this report as an example of commercially available library maintenance tools. LIBRARIAN is distributed by Applied Data Research, Inc., Princeton, New Jersey, for use on the IBM 360/370. Many similar library maintenance tools are available from various vendors.

d. Description

The LIBRARIAN is a source program management system. Source programs can be stored and subsequently retrieved and

updated using system commands. System facilities are included to protect against unauthorized access to master files. Programming facilities include commands for inserting, deleting and replacing source statements, syntax checking of COBOL programs, editing and scanning, provisions for copying, renaming and applying temporary changes to source programs, user exits for specialized local code interfaces, and the ability to rearrange and expand statements within a source program. Management facilities include the ability to produce reports showing the status and attributes of all source programs within a master file, including a historically accurate, date-stamped audit trail of all changes made to a program.

A TSO (Time Sharing Option) interface option permits TSO users direct access to program modules. The Space Saver option produces reports which allow monitoring and optimization of all disk-oriented direct access resources. The LIBRARIAN on-line option features Customer Information and Control System (CICS) and Virtual Memory (VM) on-line interfaces. There is also an entry version designated LIBRARIAN/E.

e. Research and Findings (Source 2)

Columbus Mutual Life Insurance Company used the LIBRARIAN in converting the IBM's ALIS (Advanced Life Information System). LIBRARIAN was very helpful for controlling the creation of the ALIS support programs plus changes and extensions to the ALIS package. Prior to installing the ADR remote job entry system, a great deal of time was spent carrying card decks to computer operations and printouts back to programmers. After obtaining the remote job entry system, turnaround time was considerably reduced.

Columbus Mutual reports the following benefits from using a source code control tool:

- The source code is secure
- Computer operations cannot erroneously use an outdated version of a program

- There is no longer a worry about the only copy of the source code being destroyed
- Code is accessible for review
- There is an audit trail of all changes made to each program.

SECTION V
TECHNOLOGY REVIEW ASSESSMENT

5.1 MAINTENANCE STATE-OF-THE-ART

Many state-of-the-art tools and techniques, such as debug tools and utility programs, which are well-known aids in software development are equally applicable to the operations and maintenance phase. However, the emphasis in this report is directed towards tools and techniques which are being used or proposed for use in solving the problems specific to maintaining operational software systems.

5.1.1 Problems Addressed by State-of-the-Art Technologies

Maintenance technology is being directed toward the following problem areas.

a. The Law of Increasing Entropy

Belady and Lehman (Reference 9) state that the entropy (unstructuredness) of a system increases with time, unless specific work is executed to maintain or reduce it. Incremental design typically becomes increasingly difficult to implement as the structure deteriorates. These authors emphasize that integrity of structure is the most important consideration in software maintenance and that local optimization of large systems "is in many cases a delusion".

b. Ripple Effect

Correcting an error may introduce additional errors into the system. Ramamoorthy and Ho (Reference 28) report a study by J. D. McGonagle which indicates that ripple effect constitutes 19 percent of software errors detected. Brooks (Reference 29) states that a software modification carries with it a 20 percent to 50 percent chance to induce a secondary error.

c. Lack of Documentation

Maintenance activities are usually performed by personnel who are not familiar with the software to be changed. The documentation required for understanding the software is often inadequate or, if originally adequate, has not been updated to reflect changes made following delivery.

d. Regression Testing

Jensen and Tonies (Reference 30) state that "the two fundamental questions in maintaining software" are:

- If one module is changed, what other modules have to be retested?
- If a series of changes are made throughout the system, which modules have to be retested to restore the system testedness level?

Maintenance technology is attempting to answer these questions, in addition to assisting in the testing activity after the test cases have been selected.

5.1.2 Applications of Maintenance Technologies

The following applications of maintenance technologies have been identified from the literature survey.

a. Data Base

A software data base is an essential requirement for configuration management and for using automated tools to maintain software. Ramamoorthy and Ho (Reference 28) state, "The success of maintenance tools depends heavily on the data base which provides a convenient means of storing test cases, error history and statistics, and cataloging detailed program characteristics."

b. Documentation

Documentation is an important aspect of software maintenance and should be given a high priority in the maintenance phase. Documentation tools are needed to:

- Assist in understanding the software to be maintained.
- Record all changes and test history during the maintenance phase for the purposes of future maintenance.

c. Static and Dynamic Analysis

Static and dynamic analysis tools are capable of performing many functions of value during the maintenance phase. Related to their primary function as error isolation and program revalidation tools, one study (Reference 31) showed that dynamic analysis detected more errors but static analysis detected errors earlier. It is generally agreed that the two techniques are complementary and should be used in combination.

In addition to their primary function, static analysis tools usually provide:

- Automatic program documentation
- Program structure data for redesign analysis
- Ripple effect analysis data

  -The set of modules that invoke a changed module
  -Sets of modules that the changed module invokes
  -Program variables affected by a proposed change.

In general, dynamic analysis tools also provide:

- Data for regression testing, such as paths or single modules that remain untested following a program modification
- Performance data for perfective maintenance and redesign analysis

d. Code Production

Code production tools used in the development phase are equally applicable to the maintenance phase, with the same limitations. As Gries (quoted by Wegner in Reference 23) states:

"The importance is emphasized of distinguishing between goals such as understandibility, flexibility and efficiency, and means of achieving the goal such as not using GOTOs or relying on elaborate debugging and test tools rather than on writing error-free well-structured programs."

e. Performance Evaluation

Tools in this area have been developed primarily in connection with operating systems but are now being directed toward evaluating the performance of large application systems. The importance of performance considerations during the maintenance phase has been recognized (Reference 12).

f. Reliability Measurement

Reliability measurement tools are available for determining software reliability changes during the maintenance phase and managing the maintenance schedule.

g. Integrated Systems

The current trend is toward the development of integrated tool systems with standard procedures for their use throughout the life cycle. The System Development Corporation's Software Factory (Reference 32) is an example of a system using a data base and a set of automated tools for development and maintenance. Tools include static and dynamic analysis, test analysis, automatic documentation, and test case generation. Maintenance functions include configuration control, evaluating the impact of changes, tracking the completeness and accuracy of changes, and reporting the status of changes and corrections.

Bell Laboratories' Programmer's Workbench (Reference 33) provides a general purpose tool kit which resides entirely on a small dedicated machine (PDP-11 based UNIX) and is application and machine independent. Standard procedures are used for program maintenance functions. One advantage of this approach is that the tools are still applicable when the user changes computers or operating systems.

The National Software Works (Reference 34), through the use of computer netting, allows the use of tools which are resident on the host machine best suited to support the tool. The large selection of software tools residing on different host computer systems are integrated into a unified tool kit under a single monitor with a single file system. Expertise concerning the individual computers and formats is not required in order to use the tools.

h. Research Areas

Attention is being directed toward the following areas with potential application to the maintenance phase:

- Upgrading current systems for fault tolerance and automatic fault recovery
- Providing automated tools for the maintenance of embedded software and mini- and micro-computer software
- Concurrent process programming systems maintenance
- Automatic programming
- Advanced verification techniques using symbolic execution, proof of correctness, mutation analysis and software sneak analysis
- Automatic testing, automatic selection of optimal test cases, and automatic generation of test data
- Requirements statement languages, design statement languages and test procedure languages which can be easily automated and subjected to analysis by automated tools.

5.1.3 Problems with the Application of Automated Tools

The use of automated tools is not widespread for the following reasons, summarized from the literature reviewed:

- Management is reluctant to forsake traditional methods.
- Management sometimes imposes tools when the intended user does not perceive the problem to be solved.
- There is a lack of confidence in the capability of the available tools to solve the problem.
- Some tools are available only to the companies that have been able to finance their development.

- The use of tools is too expensive

  -In time and resources for development
  -In purchase price
  -In computer operating time and equipment
  -In manpower resources for analysis of the results.

- Using automated tools may increase verification or testing time beyond schedule limitations.
- Information concerning the availability and capabilities of tools is limited.
- Some tools are difficult to use because

  -Documentation is poor.
  -Input is complicated.
  -Output is too extensive for analysis.

- The equipment being used may be too small to support the tool.
- The tool may be available only for a certain machine or a certain language, and modification of the tool is too difficult or too expensive
- The tools themselves have not been adequately maintained and updated.
- Tools, especially simulation tools, which have been developed for designing, verifying and testing the software prior to delivery have been considered "throwaway items" and are not delivered as part of the software package.
- A well-defined methodology for the use of tools is lacking.

5.1.4 Criteria for Selection of Automated Tools

General guidelines for the evaluation and selection of automated tools by the user are summarized below. Basically, an automated tool should:

- Provide rapid identification of problems and rapid implementation of solutions
- Be thoroughly documented and tested
- Fulfill a need without creating new problems
- Be inexpensive to use
- Be easy to learn, or at least worth the effort of learning in terms of the benefits obtained
- Be easy to use after the initial learning period

- Provide usable information and not simply more data
- Be available for the user's equipment and program language

5.1.5 Support of Automated Tools

Some guidelines and general comments on the use of automated tools are summarized as follows:

- Tools must be supported with sound management, organizational concepts and procedures
- Tools must be reviewed periodically for enhancement, utilization of new technologies, or retirement.
- A single technique or tool is insufficient; a combination of consistent and complementary tools should be selected. Ideally, the techniques and discipline should be the same for using all tools.
- Existing tools should be used if possible, rather than redeveloping similar tools
- Tools require the interaction of human experience and judgment and can only assist the user, not replace him.

5.1.6 Evaluation of Tools

It is generally agreed that the tools are insufficiently evaluated in terms of their full range of capabilities, their limitations and their proper application. Goodenough (quoted by Wegner in Reference 23) states: "There has been an overemphasis on tool development and insufficient emphasis on analysis and evaluation of effectivenss of tools." Concerning testing tools, he reports: "Although a great deal of effort has been expended on the development of testing tools, little is known about the relative or absolute effectiveness of different testing strategies in finding errors in production programs."

Evaluation of the tools should also consider their applicability and use in an integrated tool system. Osterweil (Reference 35) states that the focus of the evaluation "should be not on what can be done by each individual technique, but rather

on what should be done by each in the context of the overall system" and should "explore the significant problems involved with interfacing the techniques to each other."

5.2 MAINTENANCE RESEARCH DIRECTIONS

It is apparent from the literature reviewed for this report that present knowledge and understanding of software maintenance technology is inadequate. Delineation of the technology application environment or measurement of technology performance effectiveness are illustrative of the kind of data that has only begun to appear in the literature. As this report demonstrates, there is an abundance of information available concerning maintenance needs and technology development. Much of this information has been developed from research and analysis into specialized maintenance problems and environments. These limited scope efforts have led to piecemeal definition of potential maintenance technology applications that are narrowly defined and generally of limited usefulness. This area of software engineering research demands a program of basic research into the fundamental properties of software evolution. An understanding of these properties is essential to the development of a unified and universally applicable software maintenance technology. Development of such a technology would establish maintenance engineering as a legitimate engineering subdiscipline under the aegis of software engineering.

For the present, several areas for software maintenance research have been suggested as a result of the review of maintenance technology literature. Basic research into the properties of software evolution must encompass these areas; however, individually their study would materially enhance understanding of software maintenance requirements and application.

- Quantitative definition of universally applicable software characteristics.
- Quantitative definition of maintenance technology performance metrics.

- Comprehensive analysis of maintenance technology performance in operational environments.
- Definition of a maintenance engineering discipline and the principles governing application of maintenance techniques and tools.

SECTION VI

REFERENCES

1. Munson, J. B., "Software Maintainability: A Practical Concern for Life-Cycle Costs," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp 54-59.

2. Mills, H. D., "Software Development," IEEE Transactions on Software Engineering, Volume SE-2, Number 4, December 1976, pp 265-273.

3. Canning, R. G., "That Maintenance 'Iceberg,'" EDP Analyzer, Volume 10, Number 10, October 1972, pp 1-13.

4. Boehm, B. W., "Software Engineering," IEEE Transactions on Computers, Volume C-25, Number 12, December 1976, pp 1226-1241.

5. Zelkowitz, M. V., "Perspectives on Software Engineering," Computing Surveys, Volume 10, Number 2, June 1978, pp 197-216.

6. Canning, R. G., "Progress In Software Engineering: Part 2," EDP Analyzer, Volume 16, Number 2, March 1978, pp 1-13.

7. Walters, G. F. and McCall, J. A., "The Development of Metrics for Software R&M," Proceedings Annual Reliability and Maintainability Symposium, Los Angeles, January 1978, pp 79-85.

8. Swanson, E. B., "The Dimensions of Maintenance," Proceedings Second International Conference on Software Reliability, San Francisco, CA, October 1976, pp 492-497.

9. Belady, L. A. and Lehman, M. M., A Model of Large Program Development, IBM System Journal, Volume 15, Number 3, 1976.

10. Lehman, M. M., "Evolution Dynamics - A Phenomenology of Software Maintenance," Proceedings Software Life Cycle Management Workshop, August 1977, pp 313-323.

11. Uhrig, J. L., "Life Cycle Evaluation of System Partitioning," Proceedings COMPSAC 1977, Chicago, IL, November 1977, pp 2-8.

12. Yau, S. S. and Collofello, J. S., Performance Considerations in the Maintenance of Large-Scale Software Systems, Interim Report, June 1979, RADC-TR-79-129, A072380.

13. Sharpley, W. K., Jr., "Software Maintenance Planning for Embedded Computer Systems, Proceedings COMPSAC 1977, Chicago, IL, November 1977, pp 520-526.

14. Cooper, J. D., "Corporate Level Software Management," IEEE Transactions on Software Engineering, Volume SE-4, Number 4, July 1978, pp 319-326.

15. Lientz, B. P., et al, "Characteristics of Application Software Maintenance," Communications of the ACM, Volume 21, Number 6, June 1978, pp 466-471.

16. Gunderman, R. E., "A Glimpse into Program Maintenance," Datamation, Volume 19, Number 6, June 1973, pp 99-101.

17. Daly, E. B., "Management of Software Development," IEEE Transactions on Software Engineering, Volume SE-3, May 1977, pp 229-242.

18. Curtis, B., et al, "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," IEEE Transactions on Software Engineering, Volume SE-5, Number 2, March 1979, pp 96-104.

19. Lindhorst, W. M., "Scheduled Maintenance of Applications Software," Datamation, Volume 19, Number 5, May 1973, pp 64-67.

20. Gilb, T., Software Metrics, Winthrop Publishers, Inc., Cambridge, MA, 1977, 282 pp.

21. Gilb, T., "Controlling Maintainability: A Quantitative Approach for Software," Unpublished paper.

22. Reifer, D. J. and Trattner, S., "A Glossary of Software Tools and Techniques," Computer, Volume 10, Number 7, July 1977, pp 52-60.

23. Wegner, P., "Research Directions in Software Technology," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp 243-259.

24. Stucki, L. G., et al., Methodology for Producing Reliable Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769.

25. Finfer, M., Fellows, J. and Casey, D., Software Debugging Methodology, Final Technical Report, April 1979, RADC-TR-79-57, Three Volumes, A069539, A069540, A069541.

26. Taylor, R. N., and Osterweil, L. J., "A Facility for Verification Testing and Documentation of Concurrent Software," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 36-41.

27. Adkins, G. and Pooch, V. M., "Computer Simulation: A Tutorial," Computer, Volume 10, Number 4, April 1977, pp. 12-17.

28. Ramamoorthy, C. V., and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp 382-394.

29. Brooks, F. P., Jr., The Mythical Man-Month, Addison-Wesley Publishing Company, Philippines, July 1978, 195 pp.

30. Jensen, R. W. and Tonies, C. C., Software Engineering, Prentice-Hall, Incorporated, New Jersey, 1979, 580 pp.

31. Rubey, R. J., et al., "Quantitative Aspects of Software Validation," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pp 150-155.

32. Bratman, H. and Court, T., "The Software Factory," Computer, Volume 8, Number 5, May 1975, pp 28-37.

33. Ivie, E. L., "The Programmer's Workbench-A Machine for Software Development," Communications of the ACM, Volume 20, Number 10, October 1977, pp 746-753.

34. Robinson, R. A., "National Software Works: Overview & Status," Proceedings COMPCON Fall 1977, Washington, D.C., September 1977, pp 270-273.

35. Osterweil, L. J., "A Methodology for Testing Computer Programs," Proceedings Computers in Aerospace Conference, Los Angeles, CA, November 1977, pp 52-62.

SECTION VII

BIBLIOGRAPHY

Adams, J. M., "Experiments on the Utility of Assertions for Debug ging," Proceedings Eleventh Hawaii International Conference on System Sciences, Honolulu, HI., January 1978, pp. 31-39.

Adkins, G. and Pooch, V. W., "Computer Simulation: A Tutorial," Computer, Volume 10, Number 4, April 1977, pp. 12-17.

Alford, M. W., "Software Requirements Engineering Methodology (SREM) at the Age of Two," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 332-339.

Alford, M. W. and Burns, I. F., "R-Nets: A Graph Model for Real-Time Software Requirements," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 97-108.

Allen, F. E. and Cocke, J., "A Program Data Flow Analysis Procedure, "Communications of the ACM, Volume 19, Number 3, March 1976, pp. 137-147.

Anderson, P. G., "Redundancy Techniques for Software Quality," Proceedings Annual Reliability and Maintainability Symposium, Los Angeles, CA, January 1978, pp. 86-93.

Arsac, J. J., "Syntactic Source to Source Transforms and Program Manipulation," Communications of the ACM, Volume 22, Number 1, January 1979, pp. 43-53.

Baker, F. T., "Structured Programming in a Production Programming Environment," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pp. 241-252.

Basili, V. R. and Turner, A. J., "Iterative Enhancement: A Practical Technique for Software Development," IEEE Transactions on Software Engineering, Volume SE-1, Number 4, December 1975, pp. 390-396.

Bate, R. R. and Ligler, G. T., "An Approach to Software Testing: Methodology and Tools," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 476-480.

Bauer, H. A. and Birchall, R. H., "Managing Large-Scale Software Development with an Automated Change Control System" Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 13-17.

Beaver, E. W. and John, F. C., "Operational Software Management for the Peace Rhine Integrated Weapon Control System," Proceedings National Aerospace Electronics Conference, Dayton, OH, May 1978, pp. 1322-1326.

Becker, R. A. and Chambers, J. M., "Design and Implementation of the 'S' System for Interactive Data Analysis," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 626-629.

Belady, L. A. and Leavenworth, B., "Program Modifiability," IBM Technical Memo Number 15.

Belady, L. A. and Lehman, M. M., A Model of Large Program Development, IBM System Journal, Volume 15, Number 3, 1976.

Belford, P. C., "Experience Utilizing Components of the Software Development System," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 340-344.

Belford, P. C. and Taylor, D. C., "Specification Verification-A Key to Improving Software Reliability," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 83-96.

Bell, T. E. and Bixler, D. C., "A Flow-Oriented Requirements Statement Language," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 109-122.

Benson, J. P. and Melton, R. A., "A Laboratory for the Development and Evaluation of BMD Software Quality Enhancement Techniques," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 106-109.

Berri, R. E., "Specifying Milestones for Software Acquisitions," Proceedings Computers in Aerospace Conference, Los Angeles, CA, November 1977, pp. 23-33.

Bersoff, E. H., Henderson, V. D. and Siegel, S. G., "Software Configuration Management: A Tutorial," Computer, Volume 12, Number 1, January 1979, pp. 6-14.

Bianchi, M. H. and Wood, J. L., "A Users' Viewpoint of the Programmer's Workbench," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 193-199.

Bielski, J. P. and Blankertz, W. H., "The General Acceptance Test System (GATS)," Proceedings COMPCON Spring 1977, San Francisco, CA, March 1977, pp. 207-210.

Blum, B. I. and Richeson, K. E., "Inexpensive Computer Assisted Software Engineering for Moderate Sized Programs," Proceedings COMPCON Spring 1977, San Francisco, CA, March 1977, pp. 202 206.

Boehm, B. W., Brown, J.R. and Lipow, M., "Quantitative Evaluation of Software Quality," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 592 605.

Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," Datamation, Volume 19, Number 5, May 1973, pp. 48 59.

Boehm, B. W., et al., "Some Experience with Automated Aids to the Design of Large Scale, Reliable Software," IEEE Transactions on Software Engineering, Volume SE-1, Number 1, March 1975, pp. 125 133.

Boehm, B. W., "Software Engineering," IEEE Transactions on Computers, Volume C 25, Number 12, December 1976, pp. 1226 1241.

Boeing Computer Services, BCS Software Production Data, Final Technical Report, March 1977, RADC TR 77 116, A039852.

Bogden, W. R., "Life Cycle Support of Navy Airborne Antisubmarine Warfare Tactical Software," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 499 503.

Boyer, R. S., Elspas, B. and Levitt, K. N., "SELECT A Formal System for Testing and Debugging Programs by Symbolic Execution," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 234 245.

Boyle, J. and Matz, M., "Automating Multiple Program Realizations," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 421 456.

Branscomb, L. M., "The Everest of Software," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1975, pp. XVII XX.

Brantley, C. L. and Osajima, Y. R., "Continuing Development of Centrally Developed and Maintained Software Systems," Proceedings COMPCON Spring 1975, San Francisco, CA, February 1975, pp. 285 288.

Bratman, H. and Court, T., "The Software Factory," Computer, Volume 8, Number 5, May 1975, pp. 28 37.

Braun, C. L. and Wohman, B. L., "Tools and Techniques for Implementing a Large Compiler on a Small Computer," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 408-414.

Braverman, P. H., "Managing Change," Datamation, Volume 22, Number 10, October 1976, pp. 111-113.

Brooks, F. P., Jr., The Mythical Man-Month, Addison-Wesley Publishing Company, Philippines, July 1978, 195 pp.

Brown, J. R. and Lipow, M., "Testing for Software Reliability," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 518-527.

Brown, J. R. and Fischer, K. F., "A Graph Theoretic Approach to the Verification of Program Structures," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 136-141.

Browne, J. C. and Johnson, D. B., "FAST: A Second Generation Program Analysis System," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 142-148.

Bucher, D. E. W., "Maintenance of the Computer Sciences Teleprocessing System," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 260-266.

Budd, T., Majoras, M. and Snood, H., "Experiences with a Software Test Factory," Proceedings COMPCON Spring 1979, San Francisco, CA, February 1979, pp. 319-329.

Burger, R. T., "AUTASIM: A System for Computerized Assembly of Simulation Models," Proceedings Winter Simulation Conference, Washington, DC, January 1974, pp. 15-22.

Caine, S. H. and Gordon, E. K., "PDL - A Tool for Software Design," Proceedings National Computer Conference, Anaheim, CA, May 1975, pp. 271-276.

Campbell, J. S., "Harnessing the Software Revolution to Meet Navy Needs," Defense Management Journal, Volume 14, Number 3, May 1978, pp. 17-23.

Campos, I. M. and Estrin, G., "Concurrent Software System Design Supported by SARA at the Age of One," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 230-242.

Canning, R. G., "Progress in Software Engineering, Part 1," EDP Analyzer, Volume 16, Number 2, February 1978, pp. 1-13.

Canning, R. G., "Progress in Software Engineering, Part 2," EDP Analyzer, Volume 16, Number 3, March 1978, pp. 1-13.

Canning, R. G., "That Maintenance 'Iceberg'," EDP Analyzer, Volume 10, Number 10, October 1972, pp. 1-13.

Carpenter, L. C. and Tripp, L. L., "Software Design Validation Tools," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 395-400.

Cave, W. C. and Salisbury, A. B., "Controlling the Software Life Cycle - The Project Management Task," IEEE Transactions on Software Engineering, Volume SE-4, Number 4, July 1978, pp. 326-334.

Chandrasekaran, B., "Test Tools: Usefulness Must Extend to Everyday Programming Environment," Computer, Volume 12, Number 3, March 1979. pp. 102-103.

Chandy, K. M., "A Survey of Analytic Models of Rollback and Recovery Strategies," Computer, Volume 8, Number 5, May 1975, pp. 40-47.

Chen, W-T, HO, J-P and Wen, C-H., "Dynamic Validation of Programs Using Assertion Checking Facilities," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 533-538.

Chrysler, E., "Some Basic Determinants of Computer Programming Productivity," Communications of the ACM, Volume 21, Number 6, June 1978, pp. 472-483.

Cicu, A., Maiocchi, M., Polillo, R. and Sardoni, A., "Organizing Tests During Software Evolution," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 43-50.

Clarke, L. A., "Testing: Achievements and Frustrations," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 310-320.

Clarke, L. A., "A System to Generate Test Data and Symbolically Execute Programs," IEEE Transactions on Software Engineering, Volume SE-2, September 1976, pp. 215-222.

Compendium of ADS Project Management Tools and Techniques, Air Force Data Automation Agency, Gunter AFS, AL, May 1977.

Computer Sciences Corporation, Software Production Data, Final Technical Report, July 1977, RADC-TR-77-177, A042686.

Cooper, J. D., "Corporate Level Software Management," IEEE Transactions on Software Engineering, Volume SE-4, Number 4, July 1978, pp. 319-326.

Curry, R. W., "A Measure to Support Calibration and Balancing of the Effectiveness of Software Engineering Tools and Techniques," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 199-214.

Curtis, B., et al., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," IEEE Transactions on Software Engineering, Volume SE-5, Number 2, March 1979, pp. 96-104.

Daly, E. B., "Management of Software Development," IEEE Transactions on Software Engineering, Volume SE-3, May 1977, pp. 229-242.

Davidson, D. and Jones, C., "A Comprehensive Software Design Technique," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 513-529.

Davidson, S. and Shriver, B. D., "An Overview of Firmware Engineering," Computer, Volume 11, Number 5, May 1978, pp. 21-33.

Davis, C. G. and Vick, C. R., "The Software Development System: Status and Evolution," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 326-331.

DeRoze, B. C., Defense System Software Management Plan, Office of the Secretary of Defense, March 1976, AD-A022 558.

DeRoze, B. C. and Nyman, T. H., "The Software Life Cycle - A Management and Technological Challenge in the Department of Defense," IEEE Transactions on Software Engineering, Volume SE-4, Number 4, July 1978, pp. 309-318.

Dershowitz, N. and Manna, Z., "Inference Rules for Program Annotation," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 158-167.

Dershowitz, N. and Manna, Z., "The Evolution of Programs: Automatic Program Modification," IEEE Transactions on Software Engineering, Volume SE-3, Number 6, November 1977, pp. 377-385.

Dolotta, T. A. and Mashey, J. R., "An Introduction to the Programmer's Workbench," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 164-168.

Elliott, I. B., "SPTRAN: A Fortran - Compatible Structured Programming Language Converter," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 331-351.

Fabry, R. S., "How to Design a System in Which Modules can be Changed on the Fly," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 470-476.

Fairley, R. E., "Modern Software Design Techniques," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 11-30.

Fairley, R. E., "An Experimental Program Testing Facility," IEEE Transactions on Software Engineering, Volume SE-1, Number 4, December 1975, pp. 350-357.

Finfer, M., Fellows, J. and Casey, D., Software Debugging Methodology, Final Technical Report, April 1979, RADC-TR-79-57, Three Volumes.

Fink, R. C., "Major Issues Involving the Development of an Effective Management Control System for Software Maintenance," Proceedings COMPSAC 1977, Chicago, IL, November 1977, pp. 533-538.

Fischer, K. F., "A Test Case Selection Method for the Validation of Software Maintenance Modifications," Proceedings COMPSAC 1977, Chicago, IL, November 1977, pp. 421-426.

Fleischer, R. J. and Spitler, R. W., "SIMON: A Project Management System for Software Development," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 547-560.

Fleischman, S. L., "Software Configuration Management for Minicomputers Using National Software Works Tools," Proceedings COMPCON Fall 1977, Washington, DC, September 1977, pp. 279-283.

Friedman, F. L., "Decompilation and the Transfer of Assembly-Coded Minicomputer System Programs," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 301-330.

Gannon, C., "JAVS: A Jovial Automated Verification System, Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 539-544.

Gansler, J. S., "Keynote: Software Management," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 1-9.

Gaulding, S. N., "A Software Design Methodology and Tools," Proceedings COMPCON Spring 1977, San Francisco, CA, February 1977, pp. 198-201.

Gaulding, S. N., "A Software Engineering Discipline and Tools for Real-Time Software," Proceedings Tenth Hawaii International Conference on System Sciences, Honolulu, HI, January 1977, pp. 220-223.

German, S. M. and Wigbreit, B., "A Synthesizer of Inductive Assertions," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pp. 68-75.

Gilb, T., Software Metrics, Winthrop Publishers, Cambridge, MA, 1977, 282 pp.

Gilb, T., "Controlling Maintainability: A Quantitative Approach for Software," Unpublished paper.

Godoy, S. G. and Engels, G. J., "Software Sneak Analysis," Proceedings Computers in Aerospace Conference, Los Angeles, CA, November 1977, pp. 63-67.

Good, D. I., London, R. L. and Bledsoe, W. W., "An Interactive Program Verification System," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 482-492.

Goodenough, J. B. and Gerhart, S. L., "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pp. 156-173.

Gordon, S. C., "The Development of a Computer Software Management Discipline,"Proceedings National Aeronautics and Electronics Conference, Dayton, OH, May 1978, pp. 1345-1354.

Goullon, H., Isle, R. and Lohr, K., "Dynamic Restructuring in an Experimental Operating System," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 295-304.

Green, J. S., Jr., "Dynamic Software Engineering: An Evolutionary Approach to Automated Software Development and Management," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 373-377.

Green, T. F., Schneidewind, N. F., Howard, G. T. and Pariseau, R. J., "Program Structures, Complexity and Error Characteristics," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 139-154.

Greenspan, S. J. and McGowan, C. L., "Structuring Software Development for Reliability," Microelectronics and Reliability, Volume 17, 1978, pp. 75-84.

Gunderman, R. E., "A Glimpse into Program Maintenance," Datamation, Volume 19, Number 6, June 1973, pp. 99-101.

Gunther, R. C., Management Methodology for Software Product Engineering, John Wiley and Sons, New York, NY, 1978, 379 pp.

Hallin, T. G. and Hansen, R. C., "Toward a Better Method of Software Testing," Proceedings COMSAC 1978, Chicago, IL, November 1978, pp. 153-157.

Hamilton, M. and Zeldin, S., "Higher Order Software - A Methodology for Defining Software," IEEE Transactions on Software Engineering, Volume SE-2, Number 1, March 1976, pp. 10-32.

Hamlet, R. G., "Testing Programs with the Aid of a Compiler," IEEE Transactions on Software Engineering, Volume SE-3, Number 4, July 1977, pp. 279-290.

Hamlet, R., "Test Reliability and Software Maintenance," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 315-320.

Hammond, L. S., Murphy, D. L. and Smith, M. K., "A System for Analysis and Verification of Software Design," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 42-47.

Hennell, M. A., Hedley, D. and Woodward, M. R., "Experience with an ALGOL 68 Numerical Algorithms Testbed," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 457-463.

Hetzel, W. C. and Hetzel, N. L., "The Future of Quality Software," Proceedings COMPCON Spring 1977, San Francisco, CA, March 1977, pp. 211-212.

Hirschberg, M. A., Frickel, W. G. and Miller, E. F., Jr., "A Semantic Update System for Software Maintenance," Proceedings COMPCON Spring 1979, San Francisco, CA, March 1979, pp. 307-309.

Hodges, B. C. and Ryan, J. P., "A System for Automatic Software Evaluation," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 617-623.

Holton, J. B., "Are the New Programming Techniques Being Used?" Datamation, Volume 23, Number 7, July 1977, pp. 97-103.

Howden, W. E., "DISSECT - A Symbolic Evaluation and Program Testing System," IEEE Transactions on Software Engineering, Volume SE-4, Number 1, January 1978, pp. 70-73.

Howden, W. E., "Reliability of the Path Analysis Testing Strategy," IEEE Transactions on Software Engineering, Volume SE-2, Number 3, September 1976, pp. 208-214.

Howden, W. E., "Functional Program Testing," Proceedings COMPSAC 1978, Chicago, IL, November, 1978, pp. 321-325.

Howley, P. P., Jr., "Software Quality Assurance for Reliable Software," Proceedings Annual Reliability and Maintainability Symposium, Los Angeles, CA, January 1978, pp. 73-78.

Howley, P. P., Jr. and Scholten, R. W., "Test Tool Implementation," Proceedings Computers in Aerospace Conference, Los Angeles, CA, November 1977, pp. 372-377.

Ingrassia, F. S., "The Unit Development Folder (UDF) an Effective Management Tool for Software Development," Datamation, Volume 24, Number 1, January 1978, pp. 171-176.

Irvine, C. A. and Brackett, J. W., "Automated Software Engineering Through Structured Data Management," IEEE Transactions on Software Engineering, Volume SE-3, Number 1, January 1977, pp. 34-40.

Itoh, D. and Izutani, T., "FADEBUG-1, A New Tool for Program Debugging," Proceedings IEEE Symposium on Computer Software Reliability, 1977, pp. 38-43.

Ivie, E. L., "The Programmer's Workbench - A Machine for Software Development," Communications of the ACM, Volume 20, Number 10, October 1977, pp. 746-753.

Jensen, R. W. and Tonies, C. C., Software Engineering, Prentice-Hall, Incorporated, Englewood Cliffs, NJ, 1979, 580 pp.

Jessop, W. H., et al., "ATLAS - An Automated Software Testing System," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 629-634.

Johnson, D., Kolberg, C. and Sinnamon, J., "A Programmable System for Software Configuration Management," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 402-407.

Johnson, J. N. and Shaw, J. L., "Fault-Tolerant Software for a Dual Processor with Monitor," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 395-407.

Kane, J. R. and Yau, S. S., "Concurrent Software Fault Detection," IEEE Transactions on Software Engineering, Volume SE-1, Number 1, March 1975, pp. 87-99.

Kaplan, R. S., "ISSUE: An Information System and Software Update Environment," Proceedings COMPSAC 1977, Chicago, IL, November 1977, pp. 527-532.

King, J. C., "Symbolic Execution and Program Testing," Communications of the ACM, Volume 19, Number 7, July 1976, pp. 385-394.

King, J. C., "A New Approach to Program Testing," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 228-233.

Knudsen, D. B., Borofsky, A. and Satz, L. R., "A Modification Request Control System," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 187-192.

Koppang, R. G., "Process Design System - An Integrated Set of Software Development Tools," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 86-90.

Krause, K. W. and Diamant, L. W., "A Management Methodology for Testing Software Requirements," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 749-760.

Krause, K. W., Smith, R. W. and Goodwin, M. A., "Optimal Software Test Planning Through Automated Network Analysis," Proceedings IEEE Computer Software Reliability Symposium, New York, NY, April 1973, pp. 18-22.

Laffan, A. W., "The Software Maintenance Problem," Proceedings Eleventh Hawaii International Conference on System Sciences, Honolulu, HI, January 1978, pp. 119-123.

Lehman, M. M., "Evolution Dynamics - A Phenomenology of Software Maintenance," Proceedings Software, Life Cycle Management Workshop, August 1977, pp. 313-323.

Lehman, M. M. and Parr, F. N., "Program Evolution and Its Impact on Software Engineering," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 350-357.

Lientz, B. P., Swanson, E. B. and Tompkins, G. E., "Characteristics of Application Software Maintenance," Communications of the ACM, Volume 21, Number 6, June 1978, pp. 466-471.

Lientz, B. P. and Swanson, E. B., "Discovering Issues in Software Maintenance," Data Management, Volume 16, Number 10, October 1978, pp. 15-18.

Lientz, B. P. and Swanson, E. B., On The Use of Productivity Aids in System Development and Maintenance, Technical Report 79-1, January 1979, AD A067 947.

Lindhorst, W. M., "Scheduled Maintenance of Applications Software," Datamation, Volume 19, Number 5, May 1973, pp. 64-67.

Liskov, B. and Zilles, S., "Specification Techniques for Data Abstractions," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 72-87.

Liu, C. C., "A Look at Software Maintenance," Datamation, Volume 22, Number 11, November 1976, pp. 51-55.

Lloyd, D. K. and Lipow, M., Reliability: Management, Methods, and Mathematics, (Second Edition), Published by the Authors, Redondo Beach, CA., 589 pp.

London, R. L., "A View of Program Verification," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 534-545.

McDonald, W. C. and Williams, J. M., "The Advanced Data Processing Testbed," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 346-351.

McGregor, B., "Program Maintenance," Data Processing, Volume 15, Number 3, May-June 1973, pp. 172-174.

McHenry, R. C. and Walston, C. E., "Software Life Cycle Management: Weapons Process Developer," IEEE Transactions on Software Engineering, Volume SE-4, Number 4, July 1978, pp. 334-344.

McKissick, J., Jr. and Price, R. A., "The Software Development Notebook - A Proven Technique," Proceedings Annual Reliability and Maintainability Symposium, Washington, DC, January 1979, pp. 346-351.

McNurlin, B. C., "Using Some New Programming Techniques," EDP Analyzer, Volume 15, Number 11, November 1977, pp. 1-13.

Malmberg, A. F., Maintenance for the NET-2 Network Analysis Program, BDM Corporation Report, April 1977, AD A041 074.

Manley, J. H., "Embedded Computer System Software Reliability," Defense Management Journal, Volume 11, Number 4, October 1975, pp. 13-18.

Martin, G. N., "Managing Systems Maintenance," Journal of Systems Management, Volume 29, Number 7, July 1978, pp. 30-33.

Martin Marietta Corporation, Viking Software Data, Final Technical Report, May 1977, RADC-TR-77-168, A040770.

Mashey, J. R. and Smith, D. W., "Documentation Tools and Techniques," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 177-181.

Merwin, R. E., "Software Management: We Must Find a Way," IEEE Transactions on Software Engineering, Volume SE-4, Number 4, July 1978, pp. 307-308.

Miller, C. R., "Software Maintenance and Life Cycle Management," Proceedings Software Life Cycle Management Workshop, August 1977, pp. 53-61.

Miller, E., et al., "Workshop Report: Software Testing and Test Documentation," Computer, Volume 12, Number 3, March 1979, pp. 98-107.

Miller, E. F. and Melton, R. A., "Automated Generation of Test Case Data Sets," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 51-58.

Mills, H. D., "Software Development," IEEE Transactions on Software Engineering, Volume SE-2, Number 4, December 1976, pp. 265-273.

Miyamoto, I., "Toward an Effective Software Reliability Evaluation," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 46-55.

Mohanty, S. N. and Adamowicz, M., "Proposed Measures for the Evaluation of Software," Proceedings MRI Symposium on Computer Software Engineering, New York, NY April 1976, pp. 485-497.

Montgomery, H. A. and Turk, R. L., "An Approach for Identifying Avionics Flight Software Operational Support Requirements-PAVE TACK an Example," Proceedings National Aerospace Electronics Conference, Dayton, OH, May 1978, pp. 418-425.

Mooney, J. W., "Organized Program Maintenance," Datamation, Volume 21, Number 2, February 1975, pp. 63-64.

Munson, J. B., "Software Maintainability: A Practical Concern for Life-Cycle Costs," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 54-59.

Musa, J. D., "Software Reliability Measures Applied to System Engineering," Proceedings AFIPS Conference, Volume 48, AFIPS Press, Montvale, NJ, pp. 941-946.

Nakamura, Y., et al., "Complementary Approach to the Effective Software Development Environment," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 235-240.

Okumoto, K. and Goel, A. L., "Availability and Other Performance Measures of Software Systems Under Imperfect Maintenance," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 66-70.

Osterweil, L. J., "A Methodology for Testing Computer Programs," Proceedings Computers in Aerospace Conference, Los Angeles, CA, November 1977, pp. 52-62.

Osterweil, L. J., et al., "ASSET: A Life-Cycle Verification and Visibility System," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 30-35.

Osterweil, L. J. and Fosdick, L. D., "DAVE - A Validation, Error Detection and Documentation System for FORTRAN Programs," Software Practice and Experience, Volume 6, September 1976, pp. 473-486.

Overton, R. K., et al., Developments in Computer Aided Software Maintenance, Technical Report, September 1974, AD A005 827.

Paige, M. R., "Software Design for Testability," Proceedings Eleventh Hawaii International Conference on System Sciences, Honolulu, HI, January 1978, pp. 113-118.

Paige, M. R., "The Technology Base for Automated Aids to Program Tests," Proceedings Tenth Annual Hawaii International Conference on System Sciences, Honolulu, HI, January 1977, pp. 240-243.

Paige, M. R., "An Analytical Approach to Software Testing," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 527-532.

Panzl, D. J., "A Language for Specifying Software Tests," Proceedings National Computer Conference, Anaheim, CA, June 1978, pp. 609-619.

Panzl, D. J. "Test Procedures - A New Approach to Software Verification," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 477-485.

Panzl, D. J., "Automatic Revision of Formal Test Procedures," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 320-326.

Para, P. S., "CLIO - A Relational Data Base System," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 289-294.

Parnas, D. L., "Designing Software for Ease of Extension and Contraction," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 264-277.

Perry, W. E. and Fitzgerald, J., "Designing for Auditability," Datamation, Volume 23, Number 8, August 1977, pp. 46-50.

Peters, L. J. and Tripp, L. L., "Software Design Representation Schemes," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 31-56.

Peterson, R. J., "TESTER/1: An Abstract Model for the Automatic Synthesis of Program Test Case Specification," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 465-484.

"Program Maintenance: User's View," Data Processing, Volume 15, Number 5, September-October 1973, pp. 1-4.

Punter, M., "Programming for Maintenance," Data Processing, Volume 17, Number 4, September-October 1975, pp. 292-294.

Ramamoorthy, C. V., et al., "The Status and Structure of Software Testing Procedures," Proceedings COMPCON Spring 1977, San Francisco, CA, March 1977, pp. 367-369.

Ramamoorthy, C. V. and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 382-394.

Ramamoorthy, C. V., et al., "On the Automated Generation of Program Test Data," IEEE Transactions on Software Engineering, Volume SE-2, Number 4, December 1976, pp. 293-300.

Ramamoorthy, C. V., et al., "Techniques for Automated Test Data Generation," Proceedings Ninth Asilomar Conference on Circuits, Systems and Computers, Pacific Grove, CA, November 1975, pp. 324-329.

Ramamoorthy, C. V. and Jahanian, P., "Formalizing the Specification of Target Machines for Compiler Adaptability Enhancement," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 353-366.

Ramamoorthy, C. V. and Kim, K. H., "Software Monitors Aiding Systematic Testing and Their Optional Placement," Proceedings First National Conference on Software Engineering, Washington, DC, September 1975, pp. 21-26.

Ramamoorthy, C. V. and Kim, K. H., "Optimal Placement of Software Monitors Aiding Systematic Testing," IEEE Transactions on Software Engineering, Volume SE-1, Number 4, December 1975, pp. 403-411.

Randell, B., "System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pp. 220-232.

Reifer, D. J. and Trattner, S., "A Glossary of Software Tools and Techniques," Computer, Volume 10, Number 7, July 1977, pp. 52-60.

Reifer, D. J., "Automated Aids for Reliable Software," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 131-142.

Riggs, R., "Computer Systems Maintenance," Datamation, Volume 15, Number 11, November 1969, pp. 227-232.

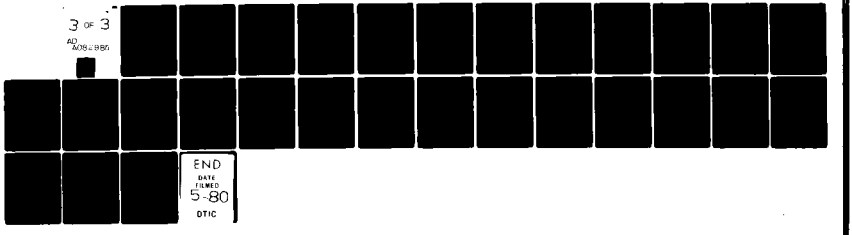
Ripley, G. D., "Program Perspectives: A Relational Representation of Measurement Data," IEEE Transactions on Software Engineering, Volume SE-3, Number 4, July 1977, pp. 296-300.

Robinson, R. A., "National Software Works: Overview & Status," Proceedings COMPCON Fall 1977, Washington, DC, September 197., pp 270-273.

Rochkind, M. J., "The Source Code Control System," IEEE Transactions on Software Engineering, Volume SE-1, Number 4, December 1975, pp. 364-370.

Roman, G., "An Argument in Favor of Mechanized Software Production," IEEE Transactions on Software Engineering, Volume SE-3, Number 6, November 1977, pp. 406-415.

AD-A082 985 IIT RESEARCH INST CHICAGO IL F/G 9/2
A REVIEW OF SOFTWARE MAINTENANCE TECHNOLOGY.(U)
FEB 80 J D DONAHOO, D R SWEARINGEN F30602-78-C-0255
UNCLASSIFIED RADC -TR-80-13 NL



Ross, D. T., et al., "Software Engineering: Process, Principles and Goals," Computer, Volume 8, Number 5, May 1975, pp. 17-27.

Rubey, R. J., et al., "Quantitative Aspects of Software Validation," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pp. 150-155.

Saib, S. H., "Executable Assertions - An Aid to Reliable Software," Proceedings Eleventh Annual Asilomar Conference on Circuits, Systems and Computers, Pacific Grove, CA, November 1977, pp. 277-281.

Saib, S. H., et al., Advanced Software Quality Assurance Final Report, General Research Corporation, May 1978.

Schurre, V., "A Program Verifier with Assertions in Terms of Abstract Data," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, April 1976, pp. 267-280.

Sharpley, W. K., Jr., "Software Maintenance Planning for Embedded Computer Systems," Proceedings COMPSAC 1977, Chicago, IL, November 1977, pp. 520-526.

Sholl, H. A. and Booth, T. L., "Software Performance Modeling Using Computation Structures," IEEE Transactions on Software Engineering, Volume SE-1, Number 4, December 1975, pp. 414-420.

Sperry Univac, Modern Programming Practices Study Report, Final Technical Report, April 1977, RADC-TR-77-106, A040049.

Stearns, S. K., "Experience with Centralized Maintenance of a Large Application System," Proceedings COMPCON Spring 1975, San Francisco, CA, February 1975, pp. 281-284.

Stephens, S. A. and Tripp, L. L., "A Requirements Expression and Validation Tool," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 101-108.

Stockenberg, J. E. and Van Dam, A., "STRUCT Programming Analysis System," Proceedings First National Conference on Software Engineering, Washington, DC, September 1975, pp. 27-36.

Straeter, T. A., et al., "MUST - An Integrated System of Support Tools for Research Flight Software Engineering," Proceedings Computers in Aerospace Conference, Los Angeles, CA, November 1977, pp. 442-446.

Strong, E. J., III, "Software Reliability and Maintainability in Large-Scale Systems," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 755-760.

Stucki, L. G. and Foshee, G. L., "New Assertion Concepts for Self-Metric Software Validation," Proceedings IEEE Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 59-65.

Stucki, L. G., et al., Methodology for Producing Reliabile Software, McDonnell Douglas Astronautics Company, March 1976, NASA CR 144769, Two Volumes.

Swanson, E. B., "The Dimensions of Maintenance," Proceedings Second International Conference on Software Reliability, San Francisco, CA, October 1976, pp. 492-497.

System Development Corporation, An Investigation of Programming Practices in Selected Air Force Projects, Final Technical Report, June 1977, RADC-TR-77-182.

Taylor, R. N. and Osterweil, L. J., "A Facility for Verification Testing and Documentation of Concurrent Software," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 36-41.

Teichroew, D., "ISDOS and Recent Extensions," Proceedings MRI Symposium on Computer Software Engineering, New York, NY, Aril 1976, pp. 75-81.

Teichroew, D. and Hershey, E. A., III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions on Software Engineering, Volume SE-3, Number 1, January 1977, pp. 41-48.

Todoriki, M. and Handa, M., "A Hierarchical Model of Computer Systems by SIMULA 67," Proceedings Summer Computer Simulation Conference, Chicago, IL, July 1977, pp. 795-797.

TRW (Catalog), Software Tools, Catalogue and Recommendations, TRW, Defense and Space Systems Group, January 1979.

TRW Systems and Space Group, NSW Feasibility Study, Final Technical Report, February 1978, RADC-TR-78-23, A052996.

Uhrig, J. L., "Life Cycle Evaluation of System Partitioning," Proceedings COMPSAC 1977, Chicago, IL, November 1977, pp. 2-8.

Van Horn, E. C., "Software Evolution Using the SEER Data Base," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 147-152.

Van Vleck, T. H. and Clingen, C. T., "The Multics System Programming Processes," Proceedings Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp. 278-280.

Victor, K. E., "A Software Engineering Environment," Proceedings Computers in Aerospace Conference, Los Angeles, CA, November 1977, pp. 399-403.

Von Henke, F. W. and Luckham, D. C., "A Methodology for Verifying Programs," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 156-164.

Walters, G. F. and McCall, J. A., "The Development of Metrics for Software R&M," Proceedings Annual Reliability and Maintainability Symposium, Los Angeles, CA, January 1978, pp. 79-85.

Wasserman, A. I., et al., "Software Engineering: The Turning Point," Computer, Volume 11, Number 9, September 1978, pp. 30-41.

Wegner, P., "Research Directions in Software Technology," *Proceedings* Third International Conference on Software Engineering, Atlanta, GA, May 1978, pp 243-259.

White, B. B., "Planning for Software Quality," Proceedings National Aerospace Electronics Conference, Dayton, OH, May 1978, pp. 230-235.

White, B. B., "Program Standards Help Software Maintainability," Proceedings Annual Reliability and Maintainability Symposium, Los Angeles, CA, January 1978, pp. 94-98.

Williams, R. D., "Managing the Development of Reliable Software," Proceedings International Conference on Reliable Software, Los Angeles, CA, April 1975, pp. 3-8.

Witt, J., "The COLOMBUS Approach," IEEE Transactions on Software Engineering, Volume SE-1, Number 4, December 1975, pp. 358-363.

Wulfe, W. A., "Reliable Hardware/Software Architecture," IEEE Transactions on Software Engineering, Volume SE-1, Number 2, June 1975, pp. 233-240.

Yau, S. S., et al., "An Approach to Error-Resistant Software Design," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 429-436.

Yau, S. S. and Cheung, R. C., "Design of Self-Checking Software," Proceedings International Conference on Reliable Software, Los Angeles, CA, pp. 450-457.

Yau, S. S. and Collofello, J. S., Performance Considerations in the Maintenance Phase of Large Scale Software Systems, Interim Report, June 1979, RADC-TR-79-129, A072380.

Yau, S. S., et al., "Ripple Effect Analysis of Software Maintenance," Proceedings COMPSAC 1978, Chicago, IL, November 1978, pp. 60-65.

Zelkowitz, M. V., "Automatic Program Analysis and Evaluation," Proceedings Second International Conference on Software Engineering, San Francisco, CA, October 1976, pp. 158-163.

Zelkowitz, M. V., "Perspectives on Software Engineering," Computing Surveys, Volume 10, Number 2, June 1978, pp. 197-216.

Zempolich, B. A., "Effective Software Management Requires Consideration of Many Factors," Defense Management Journal, Volume 11, Number 4, October 1975, pp. 8-12.

Zinkle, A. L., An Automatic Software Maintenance Tool for Large-Scale Operating Systems, Final Report, December 1978, AD A067 799.

APPENDIX
GLOSSARY

The objective in including this glossary of terms is to present definitions for the more frequently used terms that are consistent with common usage. These definitions have been taken from the comprehensive, software engineering glossary compiled by the Data and Analysis Center for Software. The terms included here represent a subset of those found in the DACS glossary.

As stated in the forward to the DACS glossary, the terms and definitions originated with a variety of sources; software engineering literature, individual usage, and data processing dictionaries. Specific credits for selected term definitions and the list of definition sources have been ommitted here. Reference to the DACS glossary is made for this information and for definition of terms not included in this glossary.

Algorithm

A collection of operations organized to be performed in a certain order when applied to data objects. The arrangement of the operations may lead to some of the operations being performed multiple times and others not being performed at all. The selection and ordering of the performance of the operations may depend in part on the data objects to which the algorithm is applied. If an algorithm is applied twice to the same data object, the operations will be performed in the same order (yielding the same results). The arrangement of the operations of an algorithm which determines their selection and order of performance is indicated by the control structures (and control statements) used to define the algorithm. An algorithm may be used to define an operation (on one level of abstraction) in terms of other operations (on a lower level of abstraction).

A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps. In principle, the steps are sufficiently basic and definite that a human can compute according to the prescribed steps exactly and in a finite length of time, using pencil and paper.

Analytical Modeling

The technique used to express mathematically (usually by a set of equations) a representation of some real problem. Such models are valuable for abstracting the essence of the subject of inquiry, because equations describing complex systems tend to become complicated and often impossible to formulate, it is usually necessary to make simplifying assumptions which may distort accuracy. Specific language and simulation systems may serve as aids to implementation.

Analyzer

An analyzer is a computer program which is applied to another program to provide analytical information. An analyzer breaks the program into identifiable small parts called segments, and uses the resulting segments to produce statistical information. This information can include execution frequency statistics, program path analysis, and/or source code syntax analysis. An analyzer may be used to determine (1) the degree to which test cases exercise the structure of the program; (2) which program segments are not executed; (3) which segments are heavily executed (and thus are candidates for optimization); (4) which test cases need to be rerun if a program segment is changed.

A computer program used to provide source language or execution frequency statistics at the program or source-statement level to assist in performance evaluation and determination of test case coverage.

Assertion

An assertion is a logical expression that specifies an instantaneous condition or relation among the variables of a program. Assertions are used in various methods of program verification as well as for program testing, synthesis, and abstraction.

A statement defining properties or behavior at a specific point in a computer program.

Assignment Statement

An instruction used to express a sequence of operations, or used to assign operands to specified variables or symbols, or both.

All statements that change the value of a variable as their main purpose (e.g. assignment or read statements, but the assignment of the DO loop variable in a DO statement should not be included).

Augmentability

Code possesses the characteristic augmentability to the extent that it can easily accommodate expansion in component computational functions or data storage requirements, this is a necessary characteristic for modifiability.

Batch Processing

The processing of data or the accomplishment of jobs accumulated in advance in such a manner that each accumulation thus formed is processed or accomplished in the same run.

Pertaining to the technique of executing a set of computer programs such that each is completed before the next program of the set is started.

Usage of a computer where the entire job is read into the machine before the processing begins. (Interactive usage always is via a terminal, batch usage may be via a terminal or a card deck.)

Case

A case statement is a statement that transfers control to one of several locations depending on the value of the control expression. ...The "case" construct provides a n-way transfer of control and is considered a "GOTO" replacement. One type of case statement is the "arithmetic if" in FORTRAN.

Change

A modification to design, code, or documentation. A change might be made to correct an error, to improve system performance, to add a capability, to improve appearance, to implement a requirements change, etc.

Any alteration (addition, deletion, correction) of the program code whether it be a single character or thousands of lines of code. Changes made to improve documentation or satisfy new specifications are important to record and study, but are not counted as bugs. --Compare with maintenance or with modification.

Code Analysis

Code analysis is the process of verifying that the computer program, as coded, is a correct implementation of the specified design.

Command Language

A source language consisting primarily of procedural operators, each capable of invoking a function to be executed.

The language through which a user directs a system.

Compiler

A computer program used to compile. Synonomous with compiling program.

A tool, used in the production of software systems, that allows programs to be written in higher-order languages, examples include the PL/I compiler, FORTRAN compiler, and COBOL compiler.

A program which translates a higher-order language source program into either assembly or machine language.

Computer Program

A computer program is a series of instructions or statements in a form acceptable to computer equipment designed to cause the equipment to execute an operation or operations.

An identifiable series of instructions, or statements in a form suitable for execution by a computer, prepared to achieve a certain result.

Computer Software

A combination of associated computer programs and data required to command the computer equipment to perform computational or control functions.

The terms software and computer software are used interchangeably.

Computer System

A computer system is an interacting collection of computer equipment, computer programs, and computer data.

Concurrent Processes

Processes may execute in parallel on multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other during execution. Individual processes within a collection of concurrent processes may suspend their execution pending receipt of information from another of the processes.

Configuration

The collection of interconnected objects which make up a system or subsystem.

The total software modules in a software system or hardware devices in a hardware system and their interrelationships.

Configuration Control

A methodology concerned with procedures for controlling the contents of a software system. A way of monitoring the status of system components, preserving the integrity of released and developing versions of a software system, and controlling the effects of changes throughout the system.

A process by which a configuration item is baselined, and thereafter, only changeable by approval by a controlling agency.

Configuration Management

Configuration management involves the systematic and disciplined application of the principles of good technical and administrative practices to ensure that all requirements are identified, evaluated, transformed into and maintained as hardware configuration items and software configuration items. It is the function of configuration management to provide the framework for technical control and status accounting during configuration item acquisition or modification to best direct maintenance effort and to minimize impact of maintenance and testing on operational service.

All activities related to controlling the contents of a software system. It monitors the status of system components, preserves the integrity of released and developing versions of a system, and controls the effects of changes throughout the system. It is a process dealing as much with procedures as with tools.

A discipline applying technical and administrative direction and surveillance to identify and document a configuration item, to control changes to it, and to report status of change processing and implementation.

Control Structures

Control structures are the logical expressions that determine the flow of control through a computer program. Structured programming restricts flow of control constructs to simple structures and avoids transfers of control that create flow complexities (i.e., excessive GOTO statements).

An organization used to build a control segment. A control structure relates two or more operations or control segments within an algorithm. A control structure provides the framework to determine: 1) whether its component operations and control segments will be performed; and 2) the order in which they will be performed during execution of an algorithm.

Corrective Maintenance

Maintenance specifically intended to eliminate an existing fault... Contrast with preventive maintenance.

Correctness

Agreement between a program's total response and the stated response in the functional specification (functional correctness), and/or between the program as coded and the programming specification (algorithmic correctness).

Correctness Proofs

Proof that a program produces correct results for all possible inputs. Validation of a program in the same way a mathematical theorem is proved correct, i.e., by mathematical analysis of its properties.

An alternative to executing tests of software to demonstrate its correctness is the method of analytic proofs. The verification process consists of making assertions describing the state of a program initially, at intermediate points in the program flow, and at termination, and then proving that each assertion is implied by the initial or prior assertion and also by the transformations performed by the program between each two consecutive assertions. An assertion consists of a definition of the relationships among the variables at the point in the program where the assertion is made. The proofs employ standard techniques for proving theorems in the first order predicate calculus. Proof of the correctness of a program using this approach obviates the need for executing test cases, since all possibilities are covered by the proofs.

The technique of proving mathematically that a given program is consistent with a given set of specifications. This process can be accomplished by manual methods or by program verifiers requiring manual intervention.

Automated verification systems exist which allow the analyst to prove small programs are correct by means similar to those used in proving mathematical theorems. Axioms and theorems derived are used to establish validity of program assertions and to provide a fundamental understanding of how the program operates.

Data Base

(1) A set of data, part or the whole of another set of data, and consisting of at least one file, that is sufficient for a given purpose or for a given data processing system. (2) A collection of data fundamental to a system, (3) A collection of data fundamental to an enterprise.

Data Repository

A facility for gathering, storing and disseminating data related to a particular topic or group of topics.

Debugging

Testing is the process of determining whether or not errors/faults exist in a program. *Debugging is an attempt* to isolate the source of the problem and to find a solution...debugging is required only in the event that one or more tests fail. It is the process of locating the error/fault which caused a test to fail.

The identification and correction of software discrepancies.

Debugging Tools

Those programs designed to locate and eliminate programming errors and to test a program for proper execution.

Software tools available to the system operator and used to locate errors in software. (The tools may include dump, snap, inspect and change, and time capabilities.)

Design Analysis

Design analysis ensures that the computer program design is correct and that it satisfies the defined software requirements with respect to design completeness and the various design elements: mathematical equations, algorithms, and control logic.

Design Hierarchy

In software, a program design in which the identified programmable elements are arranged in order of dependency from the most dependent elements to the least dependent elements.

Desk Checking

Desk checking (DC) is a term covering the totality of verification efforts performed manually during program checkout without benefit of a computer or simulator...most commonly, desk checking refers to (1) doing arithmetic calculations to verify output value correctness, and (2) "playing computer" (i.e., manually simulating program execution) in order to understand and verify program logic and data flow. Desk checking is an essential part of any verification process. It usually concentrates on areas of special problems, especially suspected errors or code inefficiencies.

Documentation

Software documentation is technical data, including computer listings and printouts, in human-readable form which (1) documents the design or details of the software, (2) explains the capabilities of the software, or (3) provides operating instructions for using the software to obtain desired results from computer equipment.

Written material, other than source code statements, that describes a system or any of its components.

The production of all the paper work necessary to describe the final product. Examples include: Cross-reference listings, dictionary listings, and flow charts.

The comprehensive description of a computer program in various formats and levels of detail to clearly define its content and composition.

Efficiency

Code possesses the characteristic efficiency to the extent that it fulfills its purpose without waste of resources, this implies that choices of source code constructions are made in order to produce the minimum number of words of object code, or that where alternate algorithms are available, those taking the least time are chosen; or that information-packing density in core is high, etc., of course, many of the ways of coding efficiently are not necessarily efficient in the sense of being cost-effective, since portability, maintainability, etc., may be degraded as a result. The process whose end is to increase efficiency is optimization. Efficiency is the ratio of useful work performed to the total energy expended. It can also be expressed as the effectiveness/cost ratio.

Entry

Entry is the instruction at which the execution of a routine begins,... A "proper program" is one having only one entry and one exit. Additional entries imply increased complexity both in coupling and in internal functional composition. Multiple entries are prohibited in structured programming guidelines.

Error

An error is a discrepancy which results in software containing a fault.

An error is an action which results in software containing a fault. The act of making an error includes omission or misinterpretation of user requirements in the software subsystem specification, incorrect translation or omission of a requirement in the design specification and programming errors. Also, programming errors include: algorithmic (fails proof of correctness), algorithmic approximation (accurate for some inputs, inaccurate for others), typographical (e.g., I for 1,* for **, etc.), data structure (e.g., dimensions, linkages incorrect), semantic (compiler works differently than programmer believes), SYNTAX (e.g., parentheses omitted), logic (e.g., or for XOR), interface (I/O mismatch), timing (e.g., execution time of instruction sequence greater than required).

A discrepancy between a specification and its implementation, the specification might be requirements, design specifications, coding specifications, etc.

A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition.

Extensibility

The extent to which software allows new capabilities to be added and existing capabilities to be easily tailored to user needs.

Fault Tolerance

Use of protective redundancy. A system can be designed to be fault-tolerant by incorporating additional components and abnormal algorithms which attempt to insure that occurrences of erroneous states do not result in later system failures-a quantitative prediction of system reliability.

Fault-Tolerant Software

A software structure employing functionally redundant routines with concurrent error detection, and provisions to switch from one routine to a functional alternate in the event of a detected fault.

Flow of Control

Flow of control is the ordered sequence of operations performed in the execution of a series of algorithms...the control structures of a high-level programming language (FORTRAN, COBOL, PL/1, etc.) allow sequential processing and branching. Examples of flow-of-control statements in a high-level programming language are: GOTO, case, while, if-then-else, etc.

Foreign Debug

Foreign debugging (FD) is an in-depth program review conducted by someone other than the implementor to find program errors and improve program reliability...A non-implementor learns the internal characteristics of the program to be debugged, constructs appropriate test cases, and debugs just as the implementor would.

Function

(1) A mathematical notation used to specify the set of inputs, the set of outputs, and the relationship between the inputs and outputs.

(2) A function is a subprogram which returns a particular value that is dependent upon the independent value(s) given with the calling instruction. ...Normally the value returned by a function is directly associated with the name of the function such as sin(k).

(3) A grouping of routines which performs a prescribed function.

(4) A sub-division of processes.

(5) In computer programming, synonym for procedure.

(6) A purposeful role or action based on a specified relationship between circumstances and responses.

(7) The natural, required, or expected activity of a program element in carrying out a program requirement.

Functional Testing

(1) The execution of independent tests designed to demonstrate a specific functional capability of a program or a software system.

(2) Validation of program "functional correctness" by execution under controlled input stimuli. This testing also gauges the sensitivity of the program to variations of the input parameters.

GOTO

In a high-level programming language (FORTRAN, COBOL, PL/1, etc.) GOTO is a statement which tells the computer where the sequence of execution should continue...A GOTO statement normally transfèrs control of the sequence of instructions to some other point in the program. The GOTO statement became a debating point when Dijkstra said in 1965 that the quality of a programmer was inversely proportional to the number of GOTO statements in his programs. Others argued for the retention of the GOTO statement because of its usefulness in a limited number of situations...also see - flow of control.

Inductive Assertion

An invariant predicate appearing within a procedure iteration. Usually placed just following the loop-collecting node. These predicates are used as an aid toward proving correctness.

Input Assertion

An input assertion is an assertion (usually denoted by the Greek letter phi) that imposes conditions on the input to a program. It is used to specify the domain of input values over which a program is intended to operate. A program is said to be totally correct with respect to an input assertion phi, if it yields the desired output for all sets of input values satisfying phi.

Integration

The combination of subunits into an overall unit or system by means of interfacing in order to provide an envisioned data processing capability.

Integration Test

Integration test - test of several modules in order to check that the interfaces are defined correctly.

Full integration test - testing of the entire system (i.e., top level component).

Partial integration test - test of any set of modules but not the entire system.

Interactive

Usage of a computer via a terminal where each line of input is immediately processed by the computer.

Interactive Debug

Interactive debugging (ID) is the process of seeking and correcting errors in a computer program while communicating with the computer executing the program...typically, the communication takes the form of monitoring program progress, inspecting intermediate values, inserting data corrections as needed, and, in general, controlling program execution. ID can dramatically reduce the time needed to debug a program since the programmer can accomplish in a short session with the "computer" (often, a remote terminal attached to the computer) what would normally take several batch turnarounds (e.g., in many installations, several days).

Interface

(1) A shared boundary. An interface might be a hardware component to link two devices or it might be a portion of storage or registers accessed by two or more computer programs.

(2) Interface - The set of data passed between two or more programs or segments of programs, and the assumptions made by each program about how the other(s) operate.

(3) The common boundary between software modules between hardware devices, or between hardware and software.

(4) When applied to a module, that set of assumptions made concerning the module by the remaining program or system in which it appears. Modules have control, data, and services interfaces.

Invariant

An invariant is an assertion associated with a point in a program that is satisfied whenever execution reaches that point... An invariant that cuts a loop in the program is sometimes called a "loop invariant." Such an assertion is said to "carry" itself around the loop...also see - assertion.

Link Editor

A utility routine that creates a loadable computer program by combining independently translated computer program modules and by resolving cross references among the modules.

Loader

A routine, commonly a computer program, that reads data into main storage.

A computer program that enables external references of symbols among different assemblies as well as the assignment of absolute addresses to relocatable strings of code. This program provides diagnostics on assembly overlap, unsatisfied external references, and multiple defined external symbols.

A program which produces absolute machine code from a relocatable code object program.

Loop

(ISO) A set of instructions that may be executed repeatedly while a certain condition prevails. In some implementations, no test is made to discover whether the condition prevails until the loop has been executed once.

Macro

A macro is a single instruction in a source language that is replaced by a defined sequence of source instructions in the

same language. The macro may also specify values for parameters in the instructions that are to replace it. Default values may exist for the parameters.

A test replacement mechanism whereby a predefined sequence of assembly language statements are inserted wherever prescribed during the translation process.

Maintainability

Code possesses the characteristic maintainability to the extent that it facilitates updating to satisfy new requirements or to correct deficiencies. This implies that the code is understandable, testable and modifiable; e.g., comments are used to locate subroutine calls and entry points visual search for locations of branching statements and their targets is facilitated by special formats, or the program is designed to fit into available resources with plenty of margins to avoid major redesign, etc.

Maintainability is the probability that, when maintenance action is initiated under stated conditions, a failed system will be restored to operable condition within a specified time.

Maintainability Measurement

The probability that when maintenance action is initiated under stated conditions, a failed system will be restored to operable condition within a specified time.

Maintainable

A software product is maintainable to the extent that it can be changed to satisfy new requirements or to correct deficiencies... Some of the characteristics which indicate the extent to which a software product is maintainable are: (A) Ease of modifying its documentation; e.g., insertions and deletions can be made without renumbering other pages, and revision records are available. (B) Code modifications are traceable to any previous state (e.g., source code lines sequentially numbered, and comment marks used to convert previously executable source code statements to "comments" which remain in the listing as a change record). (C) Documentation includes cross-references of variable names with subroutines in which they are used, and subroutines calling sequences. (D) Comments are used to locate subroutine calls and entry points. (E) Source code format facilitates visual search for locations of branching statement and their targets. Alternatively, up-to-date flowcharts are available.

Maintenance

(1) Any activity, such as tests, measurements, replacements, adjustments, and repairs, intended to eliminate faults or to keep a functional unit in a specified state.

(2) Activity which includes the detection and correction of errors and the incorporation of modifications to add capabilities

and/or improve performance. See also preventive maintenance, corrective maintenance.

(3) Software maintenance - the process of modifying existing operational software while leaving its primary function intact.

(4) Alterations to software during the post-delivery period in the form of sustaining engineering or modifications not requiring a reinitiation of the software development cycle.

Modifiability

Code possesses the characteristic modifiability to the extent that it facilitates the incorporation of changes, once the nature of the desired change has been determined. Note the higher level of abstractness of this characteristic as compared with augmentability.

Modifiability implies controlled change, in which some parts or aspects remain the same while others are altered, all in such a way that a desired new result is obtained.

Modifiable

Modifiability is the characteristic of being easy to modify... modifiability or to be modifiable implies controlled change in which some parts or aspects remain the same, while others are altered; all in such a way that a desired new result is obtained. Modifiability is one aspect of maintainable. Also see - maintainable.

Modification

The process of altering a program and its specifications so as to perform either a new task or a different but similar task. In all cases, the functional scope of a program under modification changes.

Module

A program unit that is discrete and identifiable with respect to compiling, combining with other units and loading.

A program; (A) Characterizable externally as performing a single operation; and (B) Characterizable internally as limited in complexity. The complexity of a module may be measured in terms of: I) The depth of nesting of its control structures; II) The total number of its control segments (i.e. control structures); and III) The total number of its operations.

A portion of a computer program which performs identifiable functions in a somewhat autonomous manner, and which is usually constrained to some maximum size.

Modules are characterized by lexical binding, identifiable proper boundaries, named access, and named reference. The word "module" may apply to a subprogram, subroutine, routine, program, macro,

or function. A "compile module" is a module or set of modules that are discrete and identifiable with respect to compiling, combining with other units, and loading.

Module Testing

The intent of the module or unit test is to find discrepancies between the module's logic and interfaces, and its module external specifications. (The description of the module's function, inputs, outputs, and external effects). The step of compiling the module should also be considered as part of the module test since the compiler detects most syntax errors and a few semantic or logic errors.

Mutation

A technique for creating high quality test data. The approach is based on the competent programmer assumption; that after the programmer has completed his job, the program is either correct or "almost" correct in that it differs from a correct program in only simple ways, and is thus a mutant of a correct program. The central idea of program mutation is the construction of a set of mutants of the target program. A mutant is a copy of the target program which differs only by a single "mutation". A mutation is a transformation of a program statement in a way which stimulates typical program errors. Some mutants may turn out to be equivalent, functionally, to the target program. The remainder should be distinguished from the target program by sufficiently powerful test data. Test data which is able to distinguish all non-equivalent mutants of a target program must thoroughly exercise the program and, hence, provide strong evidence of the program's correctness.

Operational

The status given a software package once it has completed contractor testing and it is turned over to the eventual user for use in the applications environment.

Output Assertion

An output assertion, usually denoted by the greek letter psi, is a statement that expresses a relation between the input and output values of a program. An output assertion is used in conjunction with an input assertion to specify formally the intended function of a program. A program is said to be totally correct with respect to an input assertion phi and output assertion psi if it halts satisfying psi on all inputs.

Path Analysis

A software technique which scans source code in order to design an optimal set of test cases to exercise the primary paths in a software module.

A technique which defines a practical measurable means of determining an optimal number of test cases by examining source code and determining the minimum set of paths which exercise all logical branches of a program.

Path Condition

The compound condition which must be satisfied by the input data point in order that the control path be executed. It is the conjunction of the individual predicate conditions which are generated at each branch point along the control path. Not all the control paths that exist syntactically within the program are executable. If input data exist which satisfy the path condition, the control path is also an execution path and can be used in testing the program. If the path condition is not satisfied by any input value, the path is said to be infeasible, and is of no interest in testing the program.

Performance

The evaluation of non logical properties (i.e. computer run time, resource utilization) of a software system. Performance is measured in terms of the amount of resources required by a software system to produce a result.

A measure of the capacity of an individual or team to build software capabilities in specialized or generalized contexts. Performance distinguishes between work and effort, as it includes productivity as one component of its measure. However, performance also measures quality of work as measured by other criteria as well, as set forth in a prioritized list of "competing characteristics" early in development.

Performance Evaluation

The degree to which a system meets stipulated or generally accepted goals.

Portability

Portability is the property of a system which permits it to be mapped from one environment to a different environment.

"Portability" designates the fact that for many different machines and operating systems, copies of the product can be delivered with uniform operating characteristics. From the user's point of view, any input which is valid on one supported system is valid on any other supported system, and will produce identical output.

Code possesses the characteristic portability to the extent that it can be operated easily and well on computer configurations other than its current one...This implies that special language features, not easily available at other facilities are not used; or that standard library functions and subroutines are selected for universal applicability, etc.

Portability is the property of a system which allows it to be moved to the new environment with relative ease.

Predicate

A logical proposition or assertion concerning the state of a program at a given point, having either a true or false value. Concerning program correctness, all such assertions must be axioms or be proved true.

Preventive Maintenance

Maintenance specifically intended to prevent faults from occurring. Corrective maintenance and preventive maintenance are both performed during maintenance time. Contrast with corrective maintenance.

Production

That portion of a software implementation that has to do with the generation of code and documentation and the checkout for correctness by production personnel. Production programming is characterized by the application of tradeoffs, known algorithms, and state-of-the-art solution methods toward software generation, as opposed to programming performed to extend the current state of the art.

Production Libraries

A technique used to provide constantly up-to-date representations of the computer programs and test data in both computer and human readable forms. The current status and past history of all code generated is also maintained. Specific library programs are available to serve as aids to implementation.

Production Run

The operation of a software system under real operating conditions and the production of useful products for the customer. This is contrasted with a test run, which is the operation of a software system to test its performance.

Program Module

A program module is a discrete, identifiable set of instructions usually handled as a unit by an assemblier, a compiler, a linkage editor, a loading routine, or other type of routine or "subroutine."

Program Segment

The smallest coded unit of a program which can be loaded as one logical entity.

A combination of program steps and calls to lower-level program segments.

Program Transformations

To replace one segment of a program description by another, equivalent description.

Programmer

A programmer is a person who produces computer programs. A senior level programmer is normally capable of performing all software development activities including design, code, test, and documentation. The activities of a more junior level programmer may be limited to coding, test case preparation, and/or assisting in the modification of existing programs and documentation.

Proof of Correctness

A proof of correctness is a statement of assertions about a program that is verified by analytic methods... An alternative to executing tests on software to demonstrate its correctness is the method of analytic proofs. The verification process consists of making assertions describing the state of a program, initially, at intermediate points in the program flow and at termination; and then proving that each assertion is implied by the initial or prior one and by the transformations performed by the program between each two consecutive assertions. An assertion consists of a definition of the relationships among the variables at that point in the program where the assertion is made. The proofs employ standard techniques for proving theorems in the first-order predicate calculus. Proof of the correctness of a program using this approach lessens the need for executing test cases, since all possibilities are covered by the proofs.

Quality

The degree to which software conforms to quality criteria. Quality criteria include, but are not limited to, correctness, reliability, validity, resilience, useability, clarity, maintainability, modifiability, generality, portability, testability, efficiency, economy, integrity, documentation, understandability, flexibility, interoperability, modularity, reusability.

Quality Assurance

A planned and systematic pattern of all action necessary to provide adequate confidence that the item or product conforms to established technical requirements.

The process of activity during which the system design is audited to determine whether or not it represents a verifiable and certifiable specification, and during which test plans and test procedures are formulated and implemented. This activity ensures the technical compliance of the software system--a product--to its requirements and design specifications. Quality assurance

is an independent audit review of all products to ensure their compliance to a management-directed standard of quality.

Guarantee made by the developer to the customer that the software meets minimum levels of acceptability. The criteria for acceptability should be mutually agreed upon, measurable, and put into writing. Primarily, although not necessarily, quality is assured through some form of testing.

Regression Testing

Regression testing (RT) is a method for detecting errors spawned by changes or corrections made during software development and maintenance. A set of tests which the program has executed correctly is rerun after each set of changes is completed, if no errors occur, confidence is increased that spawned errors were not created in that change... RT is an invaluable aid during program maintenance to prevent the "X step forward, Y steps backward" syndrome. Spawned errors are particularly onerous from a program user point of view, since they contribute to user distrust ("it used to work; why doesn't it now). RT is primarily used in a maintenance-intensive environment. However, it has applicability to any program in maintenance, regardless of the quantity of frequency of change... A set of tests is maintained and utilized prior to release of each new software version. If errors or deviations are detected, they are corrected and the regression test is repeated prior to release. If acceptance tests are used, they should form the basis for the regression tests. Tests should be added as new soft spots are identified during maintenance. Because of the frequency of rerunning, tests should be self checking whenever possible. Also see - testing.

Repairable

A software product is repairable to the extent that a change to correct a deficiency can be localized, so as to have minimal influence on other program modules, logic paths, or documentation. Repairability is a subcategory of maintainability, but the implication is that a software product becomes non-repairable when the effects of a proposed code fix are not understood with sufficient confidence, owing to previous poor maintenance practices, including lack of traceability. In other words, a state of non-repairability is reached when it can be concluded that it is cost effective to redesign a significant portion of the program. Also see maintainable.

Scenario

An automated test control package consisting of test execution control words, test data, and even stimuli used to activate and test a target program.

Software

Software is computer program code and its associated data, documentation, and operational procedures.

Software Engineering

Software engineering combines the use of mathematics to analyze and certify algorithms, engineering to estimate costs and define tradeoffs, and management science to define requirements, assess risks, oversee personnel, and monitor progress in the design, development and use of software. Software engineering techniques are directed to reducing high software cost and complexity while increasing reliability and modifiability.

Software engineering is that branch of science and technology which deals with the design, development, and use of software. Software engineering is a discipline directed at the production of computer programs that are correct, efficient, flexible, maintainable, and understandable in reasonable time spans at acceptable costs.

The practical and methodical application of science and technology in the design, development, evaluation, and maintenance of computer software over its life cycle.

Software Life Cycle

The software life cycle is that period of time in which the software is conceived, developed, and used.

The life cycle is normally divided into the six phases of conception, requirements definition, design, implementation, test, and operational phases. The conceptual phase encompasses problem statement definition, preliminary systems analysis, and the identification of alternative solution categories. The requirements definition phase consists of producing a statement of project objectives, system functional specifications, and design constraints. During the design phase the software component definitions, interface, and data definitions are generated and verified against the requirements. The implementation phase consists of the actual program code generation, unit testing of the programs, and documenting the system. During the test phase, system integration of the software components and system acceptance tests are performed against the requirements. The operational phase involves the use and maintenance of the system.

This includes the detection and correction of errors and the incorporation of modifications to add capabilities and/or improve performance.

Software Sneak Analysis

A formal technique involving the use of mathematical graph theory, electrical sneak theory, and computerized search algorithms which are applied to a software package to identify software

sneaks. A software sneak is defined as a logic control path which causes an unwanted operation to occur or which bypasses a desired operation without regard to failure of the hardware system to respond as programmed.

Software Testing

The process of exercising software in an attempt to detect errors which exist in the code. Software testing does not prove that a program is correct.

Standards

Any specifications that refer to the method of development of the source program itself, and not to the problem to be implemented (e.g., using structured code, at most 100 line subroutines, all names prefixed with subsystem name, etc.).

Procedures, rules, and conventions used for prescribing disciplined program design (program structuring, and data structuring) and implementation. Architecture and partitioning rules, documentation conventions, configuration and data management procedures, etc. are among those standards to be disseminated.

A design criterion. An entity conforms to a standard if the attribute(s) defined by the standard apply to the entity.

Conventions, ground rules, guidelines, procedures, and software tools employed during the software development process to benefit software design quality, coding quality, software reliability, viability and maintainability.

Stepwise Refinement

Step-wise refinement is the process whereby steps are taken in the following order: (1) the total concept is formulated, (2) the functional specification is designed, (3) the functional specification is refined at each intermediate step where the intermediate steps include code or processes required by the previous step, and (4) final refinements are made to completely define the problem.

The process of defining data in more and more detail as the need arises during the programming process.

The defining of more general operations in terms of more specific, lower level operations. The design of a programming system through stepwise refinement is called top down design.

Submodule

A module appearing within a module or invoked by a module on a flowchart, the procedure appearing within or referred to (e.g., invoked by) any charted symbol.

Support Software

All programs used in the development and maintenance of the delivered operational programs and test/maintenance programs. Support programs include, but are not limited to: A) Compilers, assembliers, emulations, builders, and loaders required to generate machine code and to combine subprograms or components into a complete computer program. B) Debugging programs. C) Stimulation and simulation programs used in operator training sites. D) Data abstraction and reduction programs applicable to operational programs. E) Test programs used in development of operational programs. F) Programs used for management control, configuration management of document generation and control during development.

A computer program which facilitates the design, development, testing, analysis, evaluation, or operation of other computer programs.

Software tools used by project personnel for software design, debugging, testing, verification, and management.

SYNTAX

The part of a grammar dealing with the way in which items in a language are arranged.

The set of rules that defines the valid input strings (sentential forms) of a computer language as accepted by its compiler (or assembler). Therefore, the structure of expressions in a language, or the rules governing the structure of a language.

Test

Any program or procedure that is designed to obtain, verify, or provide data for the evaluation, research and development (other than laboratory experiments), progress in accomplishing development objectives; or performance and operational capability of systems, subsystems, components, and equipments items.

Test Procedure

A formal document developed from a test plan that presents detailed instructions for the set up, operation, and evaluation results for each defined test.

Testing

Testing is the part of the software development process where the computer program is subject to specific conditions to show that the program meets its indended design. It is the process of feeding sample input data into a program, executing it, and inspecting the output and/or behavior for correctness. The cornerstone of reliability methodology is testing. Traditionally, testing is the development phase where the largest quantity of errors is detected and corrected. But, given this expenditure, the software developer has no real assurance of developing

error-free software, for the testing cycle only demonstrates the presence of error. The following techniques or tools are considered part of the testing cycle: analyzers, assertions, source language debug, intentional failure, test drivers, regression testing, environment simulators, standardized testing, symbolic execution, interactive debug, foreign debug, sneak circuit analysis.

Exercising different modes of computer program operation through different combinations of input data (test cases) to find errors.

Tree

An acyclic connected graph. If the tree has N > OR = Nodes, then it also has N - 1 edges. Every pair of nodes is connected by exactly one path. The tree often represents a heirarchy, in which edges are directed to denote a subordinating relationship between the two joined nodes.

Understandable

A software product is understandable to the extent that its purpose is clear to the inspector...Many techniques have been proposed to increase understandability. Prominent among these are code structuredness which simplifies logical flow. Local commentary, to explain complex coded instructions, and consistently used mnemonics. In addition, references to readily available and up-to-date documents need to be included in source commentary so that the inspector may comprehend more esoteric contents. Input, outputs, and assumptions should be stated in the form of glossaries or prose commentary. In general, a coding standard encompassing format of headers and indentation should be followed for all modules so that information can be found where expected...

Unit

(1) A set of computer program statements treated logically as a whole. The word "unit" is restricted in the context of a computer program structure. This usage does not refer to a device unit, or logical unit.

(2) A named subdivision of a program which is capable of being stored in a program support library and manipulated as a single entity. See also: Program segment.

User

(1) The individual at the man/machine interface who is applying the software to the solution of a problem, e.g. test or operations.

(2) Any entity using the facilities of an operating system. In addition to "normal" users, this includes at least programs, networks, and operators.

Validation

The process of determining whether executing the system (i.e., software, hardware, user procedures, personnel) in a user environment causes any operational difficulties. The process includes ensuring that specific program functions meet their requirements and specifications. Validation also includes the prevention, detection, diagnosis, recovery, and correction of errors.

Validation is more difficult than the verification process since it involves questions of the completeness of the specification and environment information. There are both manual and computer based validation techniques.

The process of ensuring that specific program functions meet their detailed design requirement specifications.

Verification

Computer program verification is the iterative process of determining whether or not the product of each step of the computer program acquisition process fulfills all requirements levied by the previous step. These steps are system specification verification, requirements verification, specification verification, and code verification.

The process of determining whether the results of executing the software product in a test environment agree with the specifications. Verification is usually only concerned with the software's logical correctness (i.e., satisfying the functional requirements) and may be a manual or a computer based process (i.e., testing software by executing it on a computer).

The process of ensuring that the system and its structure meet the functional requirements of the baseline specification document.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of technical competence is provided to ESD Program Offices (POs) and other ESD elements. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.