IDA PAPER P-2132

SDS SOFTWARE TESTING AND EVALUATION: A REVIEW OF THE STATE-OF-THE-ART IN SOFTWARE TESTING AND EVALUATION WITH RECOMMENDED R&D TASKS

Christine Youngblut
Bill R. Brykczynski
John Salasin
Karen D. Gordon
Reginald N. Meeson

February 1989

Prepared for
Strategic Defense Initiative Organization

IDA

INSTITUTE FOR DEFENSE ANALYSES
1801 N. Beauregard Street, Alexandria, Virginia 22311-1772

DEFINITIONS

IDA publishes the following documents to report the results of its work.

Reports

Reports are the most authoritative and most carefully considered products IDA publishes. They normally embody results of major projects which (a) have a direct bearing on decisions affecting major programs, or (b) address issues of significant concern to the Executive Branch, the Congress and/or the public, or (c) address issues that have significant economic implications. IDA Reports are reviewed by outside panels of experts to ensure their high quality and relevance to the problems studied, and they are released by the President of IDA.

Papers

Papers normally address relatively restricted technical or policy issues. They communicate the results of special analyses, interim reports or phases of a task, ad hoc or quick reaction work. Papers are reviewed to ensure that they meet standards similar to those expected of refereed papers in professional journals.

Documents

IDA Documents are used for the convenience of the sponsors or the analysts to record substantive work done in quick reaction studies and major interactive technical support activities; to make available preliminary and tentative results of analyses or of working group and panel activities; to forward information that is essentially unanalyzed and unevaluated; or to make a record of conferences, meetings, or briefings, or of data developed in the course of an investigation. Review of Documents is suited to their content and intended use.

The results of IDA work are also conveyed by briefings and informal memoranda to sponsors and others designated by the sponsors, when appropriate.

The work reported in this document was conducted under contract MDA 903 84 C 0031 for the Department of Defense. The publication of this IDA Paper does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that Agency.

This Paper has been reviewed by IDA to assure that it meets the high standards of thoroughness, objectivity, and appropriate analytical methodology and that the results, conclusions and recommendations are properly supported by the material presented.

REPORT DOCUMENTATION PAGE

1a REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
2a SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION/AVAILABILITY OF REPORT Public release/unlimited distribution.
2b DECLASSIFICATION/DOWNGRADING SCHEDULE	
4 PERFORMING ORGANIZATION REPORT NUMBER(S) IDA Paper P-2132	5 MONITORING ORGANIZATION REPORT NUMBER(S)
6a NAME OF PERFORMING ORGANIZATION Institute for Defense Analyses 6b OFFICE SYMBOL IDA	7a NAME OF MONITORING ORGANIZATION OUSDA, DIMO
6c ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311	7b ADDRESS (City, State, and Zip Code) 1801 N. Beauregard St. Alexandria, VA 22311
8a NAME OF FUNDING/SPONSORING ORGANIZATION Strategic Defense Initiative Organization 8b OFFICE SYMBOL (If applicable) SDIO	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER MDA 903 84 C 0031
8c ADDRESS (City, State, and Zip Code) 1E149, The Pentagon Washington, DC 20301-7100	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. T-R2-597.21 / WORK UNIT ACCESSION NO.
11 TITLE (Include Security Classification) SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation with Recommended R&D Tasks	
12 PERSONAL AUTHOR(S) Christine Youngblut, Bill R. Brykczynski, John Salasin, Karen D. Gordon, Reginald N. Meeson	
13a TYPE OF REPORT Final 13b TIME COVERED FROM _____ TO _____ 14 DATE OF REPORT (Year, Month, Day) 1989 February 15 PAGE COUNT 182	
16 SUPPLEMENTARY NOTATION	
17 COSATI CODES FIELD / GROUP / SUB-GROUP	18 SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Testing and evaluation; SDS; system architectures; software engineering; reliable software.

19 ABSTRACT (Continue on reverse if necessary and identify by block number)

This document identifies the technology required for effective and efficient testing and evaluation of Strategic Defense System (SDS) software. This document provides an overview of current testing and evaluation technology, a mapping of available technology against SDS needs, and recommendations to close critical gaps in technology.

20 DISTRIBUTION/AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21 ABSTRACT SECURITY CLASSIFICATION Unclassified
22a NAME OF RESPONSIBLE INDIVIDUAL Mr. Bill Brykczynski	22b TELEPHONE (Include area code) (703) 824-5515 22c OFFICE SYMBOL IDA/CSED

IDA PAPER P-2132

SDS SOFTWARE TESTING AND EVALUATION: A REVIEW OF THE STATE-OF-THE-ART IN SOFTWARE TESTING AND EVALUATION WITH RECOMMENDED R&D TASKS

Christine Youngblut
Bill R. Brykczynski
John Salasin
Karen D. Gordon
Reginald N. Meeson

February 1989

INSTITUTE FOR DEFENSE ANALYSES

Contract MDA 903 84 C 0031
Task T-R2-597.21

CONTENTS

LIST OF FIGURES

LIST OF TABLES

PREFACE

The purpose of IDA Paper P-2132, *SDS Software Testing and Evaluation: A Review of the State-of-the-Art in Software Testing and Evaluation With Recommended R&D Tasks*, is to identify the technology required for effective and efficient testing and evaluation of Strategic Defense System (SDS) software. This document was prepared for the Strategic Defense Initiative Organization (SDIO), and provides an overview of current testing and evaluation technology, a mapping of available technology against SDS needs, and recommendations to close critical gaps in technology.

IDA Memorandum M-496 is a related document which provides a comprehensive, annotated bibliography of the reference material acquired in the course of this work. IDA Memorandum M-513, another related document, is a collection of the papers provided by leading experts in testing and evaluation technology preparatory to an IDA workshop held in support of this work.

The authors gratefully acknowledge the support given by all those who participated in the workshop and reviewed this paper. The insights and experience these people so willingly shared has been invaluable. In particular, special thanks go to the following people:

Dr. Frank Ackerman
Dr. Victor Basili
Mr. David Card
Mr. Joseph Cavano
Dr. Lori Clarke
Dr. Rich DeMillo
Dr. Laura Dillon
Dr. Michael Evangelist
Dr. Amrit Goel
Dr. William Howden
Dr. Warren Hunt Jr.
Dr. Richard Kemmerer
Dr. John Knight
Dr. Timothy Lindquist
Dr. David Luckham
Mr. Karl Nyberg
Dr. Leon Osterweil
Dr. Richard Platek
Dr. Debra Richardson
Dr. Dieter Rombach
Dr. Richard Selby
Dr. Vincent Shen
Dr. Richard Taylor
Dr. Lee White
Dr. Steven Zeil

EXECUTIVE SUMMARY

Introduction

Testing and evaluation are recognized as pivotal problems in the development of the Strategic Defense System (SDS). Consequently, a development approach combining design, prototyping, and simulation is being used to allow early evaluation of system requirements and designs. Loosely-coupled, decentralized system architectures are being examined for their facility to reduce the testing problem to a manageable level. But what testing and evaluation technology shall be used to ensure the reliability of, and provide the necessary level of confidence in, SDS software? The purpose of this report is to address this question.

For the past decade, software testing and evaluation have been relatively unpopular subjects. Why this happened is less important than its consequences. It has been a contributing factor to the lag between the state-of-the-art and state-of-the-practice, which is the largest of any in the diverse areas of software engineering, and has resulted in a shortage of research and development (R&D) resources and a small, weak research community. Since SDS prototype software is already being developed, software testing and evaluation are now critical concerns. The Strategic Defense Initiative Organization (SDIO) *cannot* wait for testing and evaluation issues to pop up later in the development process (a traditional approach which has proven costly in the past), but must meet the challenge up front.

Once the importance of this challenge is recognized, appropriate actions can be taken. There is a substantial body of testing and evaluation technology whose use in an industrial environment is ready to be investigated, preparatory to transitioning the technology into practice. The main body of this report describes the current status of different areas of this technology. It also outlines a number of tasks for bringing promising elements of the technology into practice. While the use of advanced technology will certainly help, it is by no means sufficient to resolve all testing and evaluation problems. In particular, technology for the testing and evaluation of large, distributed and real-time software systems is still in its infancy. The final part of this report, therefore, recommends a number of tasks to extend the boundaries of technology to meet SDS needs.

Testing and evaluation technology is not, by itself, enough to ensure improved practices. Policy must evolve in step with the technology to ensure its proper application. The process of setting and revising policy is invariably lengthy and, therefore, policy inevitably lags behind the state-of-the-art. Consequently, it is important that SDIO testing and evaluation policies be designed to encourage carefully considered innovation, rather than dictate the use of particular techniques. The disciplined development approaches needed to produce reliable software and facilitate testing and evaluation are another policy issue, and one already under consideration in the evolving SDIO Software Policy (see Section 2.4.3). Indeed, it is important to emphasize that reliability *cannot* be tested or evaluated into software. Reliable SDS software can only be achieved through improvements in the "upstream" stages of the development life cycle. Testing and evaluation during these activities will provide the necessary feedback to ensure that these activities are performed properly.

Although this report specifically addresses testing and evaluation for SDS software, advances in testing and evaluation practices achieved for SDS could be exploited to benefit all DOD software efforts.

SDS Software Testing and Evaluation Needs

The SDS will possess many characteristics which stress current testing and evaluation technology. This fact, combined with the inability to conduct full-scale operational testing in the usual manner,

require that software testing and evaluation be assigned a central role of the development of the system as a whole.

At a high level of detail, critical needs for SDS software testing and evaluation revolve around the following issues:

- Planning for software testing and evaluation must start with the user's definition of operational system requirements, and system requirements must be reviewed against the ability to conduct needed testing and evaluation.
- Testing and evaluation must be thoroughly integrated into a development life cycle which includes prototyping, simulation, and the use of formal specifications.
- The introduction and use of testing and evaluation technology must be carefully planned to reflect programmatic concerns, including the need for well-defined organizational support, roles, and policies.

The key mechanisms proposed for meeting these needs are:

- An overall test plan concept which institutionalizes an SDS testing and evaluation process model. The model, itself, should provide an evolutionary framework showing (1) how testing and evaluation fit into development activities, and (2) what specific technology elements should be exploited. Moreover, the model allows the flexibility necessary to allow the continuing adoption of improved practices and tools.
- Explicit software testing requirements. These requirements should be initially derived from system requirements and refined during the progression to code to guide the application of testing and evaluation technology.

Dynamic Analysis Technology

There is an evolving body of technology for the dynamic analysis of sequential programs. There is no doubt that disciplined application of available state-of-the-art techniques offers substantial improvements over current testing practices. Although these techniques may be expensive to apply, their cost largely accrues from execution costs; with appropriate automation, they do not place excessive burdens on the skill or labor required from software developers. Major short-term deficiencies in dynamic analysis of sequential programs arise from an absence of quantitative information on the error and fault detection capabilities and costs of existing techniques, and the slow growth of understanding on how to integrate the application of several techniques. One promising area deserving of greater attention is the use of formal specifications to facilitate functional analysis, allow greater automation of the testing process, and resolve problems due to the lack of effective oracles.

Technology for dynamic analysis of concurrent and real-time software is less mature. While testing of sequential programs evolved from graph-theory based modeling of control flow properties, there is no comparably stable basis for analysis, debugging, or run-time monitoring of concurrent and real-time software. Even so, some emerging techniques seem very promising. The resources required to develop necessary automated support and conduct trials of these techniques on realistic software efforts should be provided. The slow progress in this area, however, requires consideration of alternative approaches. For example, the use of self-testing software for critical SDS components must be investigated.

Dynamic techniques are rarely applied to precode products. The applicability of existing techniques to

executable, pre-implementation representations such as the Strategic Defense Initiative (SDI) Architecture Dataflow Modeling Technique (SADMT) (see Section 2.2.1) should be examined.

Static Analysis Technology

Again, there are many static techniques for the analysis of sequential code. Since these are largely automated, requiring minimal human effort to apply, a base set of static analyses should be routinely required for all SDS software. There are relatively few techniques for the analysis of concurrent and real-time software. As with dynamic approaches, development of the most promising techniques should be fostered by providing the resources needed to develop necessary tools and to allow evaluation of these techniques in realistic software development environments.

Unlike dynamic approaches, there are a few techniques for static analysis of pre-implementation products. The use of a common set of formal representation forms for early SDS development products which facilitate the use of existing static analysis techniques and provide a common framework for additional techniques must be examined. In particular, approaches such as fault tree analysis (which identifies combinations of conditions which may lead to critical system or software failures) will be extremely important in the SDS software testing and evaluation planning. For example, they will help to identify critical SDS components where additional testing dollars, or special fault-tolerance approaches, are required.

Automated Support for Dynamic and Static Analysis

From the SDS perspective there are two promising trends here. First, recent tool developments are focusing on supporting testing and evaluation of Ada programs. In particular, the Ada language is clearly becoming the target of choice for tools applying advanced techniques. Second, a few organizations are undertaking the development of comprehensive testing and evaluation environments which will provide a broad range of capabilities. Although these efforts are tackling difficult problems, researchers expect sophisticated prototype environments to become available within two years (see Section 4.6).

It must be clearly understood, however, that existing tools are more or less exclusively prototypes. There are many techniques which are sufficiently mature to justify production quality automated support, but the lack of research resources has prevented development of tools which are suitable for widespread use. Available tools generally lack robustness, complete documentation, speed, and the ability to handle very large software systems. The need for productization of existing prototype tools is urgent and itself requires research to develop increased understanding of the issues involved. One of the pertinent issues is flexibility. Large scale tool integration efforts will only remain viable and useful if they can continue to integrate the increasing numbers and varieties of tools that will emerge in the coming years.

Formal Verification Technology

The foundations of formal verification were laid in the 1960's and followed by prototype verification system development in the 1970's. In the early 1980's, the complexity of formal proofs and practical limitations on the size of systems that could be verified at the program code-level became apparent. Incremental improvements in verification tools and environments are being made but are not likely to make code-level proofs of large systems feasible in the near term. On the other hand, the use of formalism in software requirements, programming languages, and test specifications (prompted by the verification community) has increased assurance of correct operation of large systems significantly. Proofs of high-level designs are feasible, even though code-level proofs remain attainable only for smaller, critical components and subsystems.

Measurement Technology

The field of software measurement has traditionally been concerned with the application of stand-alone metrics to software products and processes. Unfortunately, this approach has failed to produce empirical results which are of major use to a software developer or manager. Problems with the current technology include an inability to validate metrics or compare metric results across multiple projects and organizations. Metrics are often abused by inappropriately viewing them as the *goal* of the software measurement effort themselves, rather than low-level *indicators* of product and process qualities which only make sense in the context of some measurement goal. However, when used with discretion, metrics can provide insights into desirable and undesirable software characteristics.

There are several aspects of software measurement which must be substantially improved before major benefits can be achieved. A well-established measurement methodology which helps in selecting appropriate project metrics, collecting and validating the data obtained, and analyzing and interpreting both the data and the metrics must be developed. Methods for deriving metrics for specific application domains are also needed. Measurement processes must be integrated into software development activities in order to ensure early data collection, feedback to the development processes, and reuse of collection methods. Tools must be developed to automate the activities of the measurement process to the fullest extent possible.

Reliability Assessment Technology

The thrust of current software reliability assessment technology is prediction of a software product's future failure behavior from its past failure behavior. This prediction effectively supports management activities such as estimating project schedules, optimizing the allocation of project resources, and optimizing the timing of new software releases. However, it does not adequately support the feedback of reliability information into the *construction* of highly reliable software and systems, which is of paramount importance in the case of the SDS.

Software reliability assessment technology needs to evolve in a number of directions. First, to support the construction of reliable software, emphasis must shift toward the software development process. That is, the targets of software reliability assessment should be software development methodologies, practices, tools, techniques, and other elements of the software development process, rather than individual software products. The motivation here is that the best way to construct reliable software is to utilize software development methodologies that have been shown to afford the highest degree of reliability.

Second, the technology must enable software reliability to be assessed in a system context. In distributed real-time systems, the software is responsible for dealing with timing constraints, hardware failures, and software faults. Accordingly, software correctness and reliability depend on whether the software meets its requirements with respect to real-time and fault tolerance.

Third, since software reliability must be taken into account when assessing system reliability, the technology must support *system* reliability assessment. In regard to this issue, the traditional practice of casting software reliability in hardware reliability terms and then using combinatorial analysis to derive system reliability needs to be rethought. In particular, the distinction between design faults and age-related faults needs to be given further consideration.

Recommended Tasks to Exploit and Extend Technology

Tasks to begin the process of making a sophisticated body of testing and evaluation technology

available for SDS software efforts fall into two groups. The first set of tasks provide the foundation for bringing state-of-the-art technology into play, while learning more about its effectiveness. The technology that can be immediately transitioned into practice lies largely in the areas of testing and evaluation of sequential code products. The problem with less developed technologies lies in identifying those deficiencies where intensive research has a strong probability of producing practically useful techniques and tools in time for full scale development of SDS software. A number of tasks are needed to investigate emerging technology and to identify those areas where fundamental research should be sponsored. These two groups of recommended tasks are outlined in Figures E-1 and E-2.

One of the important findings of this report is that the research community has not grown sufficiently in the past decade or so and is not large enough or strong enough to meet the challenges raised by SDS software testing and evaluation. Steps to strengthen and expand the software testing and evaluation research community must be promptly taken.

In addition, it must be recognized that technology transfer is, at least, as big a problem as technology development. A well-supported effort devoted to technology transition for SDS purposes should be put in place in the immediate future.

- Implement the test plan and testing requirements concepts to provide the mechanisms for:
  - — Better integration of testing and evaluation into development activities.
  - — Providing increased visibility, control, and repeatability of testing and evaluation activities.
- Establish a SDS Software Data Collection System which:
  - — Supports analysis of both SDS software and the technology used to develop, test, and support that software.
  - — Provides a historical database capability and acts as a focal point for research.
- Develop a comprehensive testing and evaluation environment by:
  - — Exploiting promising ongoing environment development efforts.
  - — Meanwhile, assembling an interim "environment" from available tools.
- Embark on a program of process modeling to explore effective, flexible ways of integrating testing and evaluation into software development activities in such a way as to enable SDIO to keep up with emerging technology.

Figure E-1. Tasks to Exploit Technology

- Conduct a series of technology demonstrations applying evolving technology to specific SDS problems. Example problems are:
  - — Identify critical SDS properties to be formalized and verified.
  - — Develop methods for reasoning about "degraded" systems.
  - — Develop testing and evaluation process specifications in software contexts.
- Sponsor a number of R&D tasks directed at specific gaps in technology (see Section 9). For example:
  - — Promote the use of increased formalism for early life cycle products.
  - — Develop a methodology, tools, and policy to support planning and execution of regression testing.
  - — Identify a minimum set of preconditions and postconditions which can be required in the specification of all FSD SDS software.
  - — Develop a comprehensive measurement methodology.
- Monitor ongoing research efforts so that:
  - — Promising developments are promptly considered for SDS practice.
  - — The SDIO supports efforts which indicate solutions to specific SDS problems.

Figure E-2. Tasks to Extend Technology

1. INTRODUCTION

Software testing and evaluation is widely recognized as one of the primary challenges in the development of the Strategic Defense System (SDS). This challenge arises from several factors. First and foremost, no system of comparable size has ever before been developed. Second, the system must be reconfigurable to adapt to rapidly changing requirements, some of which will arise dynamically due to countermeasures by adversaries. Third, it must perform massively parallel computations distributed on a network of complex system components. Fourth, there will be hard real-time deadlines that must be met by the SDS to achieve its mission and to ensure the safety of the system and environment. Fifth, the system must be fault tolerant not only to continue operation in the face of inherent software and hardware failures, but to survive in a hostile environment. The practical and political limitations on full-scale testing in an operational environment further exacerbate the testing problem.

It must be accepted that SDS software testing and evaluation needs cross the boundaries of current technology. If the traditional approach of postponing investigation of testing problems until testing activities are due to commence is followed, then the predictions of failure made by Parnas [Parn85] and others [Lin85,Bump87] may be fulfilled. Moreover, testing and evaluation cannot be addressed independently of software development. Software must be developed with error prevention in mind and designed to facilitate testing and evaluation. Since SDS prototypes are already being developed, the software testing and evaluation challenge must be squarely faced and immediate actions taken to investigate solutions.

This report is the first step in an effort directed at identifying the technology that is needed for SDS software testing and evaluation. The SDS development approach and special software testing and evaluation concerns are discussed in Section 2. These are used to develop a conceptual model of the SDS software testing and evaluation process which provides a framework for inducing the necessary synergism between development and testing activities. Sections 3 through 7 review the state-of-the-art in software testing and evaluation technology. Section 8 then maps current technology against needs to determine what must be done to exploit the best of available technology. Although transitioning the state-of-the-art testing and evaluation technology into practice can be expected to yield substantial improvements in software reliability, it by no means ensures a sufficient technology for SDS purposes. Section 9 recommends a number of R&D tasks to begin the process of resolving these deficiencies.

The remainder of this section sets the scene for the following discussions by outlining the role of testing and evaluation activities in the software development life cycle. The current state-of-the-practice is reviewed to provide a baseline against which possible improvements can be assessed. Finally, the role of the Institute for Defense Analyses (IDA) Testing and Evaluation Workshop held in support of this work is briefly described. First, some definitions of primary terms are appropriate.

1.1 Definitions of Terms

In this report, the term *testing and evaluation* is used in the general sense to refer to the planning, conducting, and reporting of all activities involved in software validation and verification. In this context, *validation and verification* are not distinguished as two separate activities but used jointly to refer to the process of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services, or documents conform to specified requirements. For the purposes of this report, the terms dynamic analysis, static analysis, formal verification, and measurement will be used to refer to the different types of validation and verification activities.

Dynamic analysis approaches rely on executing a piece of software with selected test data to detect, or

in some cases demonstrate the absence of, software faults. *Static analysis* approaches have the same goals, but do not base the recognition of faults on expected software outputs. *Formal verification* techniques, the most rigorous analysis approaches, apply formal, mathematical principles to prove the correctness of software designs and program code with respect to a formal specification of the behavior in question. *Measurement* techniques are concerned with the quantitative evaluation of critical properties of both software products and the processes used to develop and support software. Techniques for assessing software reliability can be isolated as a subset of measurement techniques which base evaluation of the property in question on the occurrence of software failures experienced during testing.

An *error* is a mental mistake made by a software developer. Its manifestation may be a textual problem within the software called a *fault*. A *failure* occurs when an encountered fault prevents the software from performing a required function within specified limits.

Debugging is an activity related to testing and evaluation. While testing and evaluation activities are designed to detect faults, *debugging* is concerned with textually isolating these faults and eliminating the underlying error.

Definitions of additional terms are given in the accompanying glossary.

1.2 Overview of the DOD Testing and Evaluation Process

As software development proceeds, successive products are reviewed against the requirements specified by their predecessors. Typically, these reviews focus on the consistency and completeness of the new products, and little rigorous testing and evaluation is performed on precode software products. Testing and evaluation of code products occurs in three stages: unit, integration, and system testing. As code is developed, each unit is tested individually. In a programming language such as Ada [MIL83], these units may be subprograms, packages, tasks, or generic units. Units are then incrementally combined to test the interfaces between them. In later stages, this integration testing combines hardware and software elements, proceeding until the entire system is integrated. Integration testing may follow a top-down or bottom-up strategy. In top-down testing, the most abstract software units are combined first and stubs are used to represent the more detailed units they invoke. Bottom-up testing starts with examining the interfaces among the most detailed units, which are executed by a test driver that invokes the units in the proper manner and provides the necessary input data to each. There are variations on these basic strategies; for example, outside-in testing allows the units most directly concerned with handling inputs and outputs to be tested first. The final stage of testing is system testing. Here the system as a whole is examined to determine whether it meets its specified requirements.

The approved DOD model for software reviews and audits is shown in Figure 1-1, reproduced from DOD Military Standard 2167A [DOD88], entitled Defense System Software Development. Here computer configuration items (CSCIs) are partitioned into computer software components (CSCs), which may themselves be decomposed into further CSCs and computer software units (CSUs). DOD-STD-2167A specifies the products required from each development, review, audit, and testing activity, together with evaluation criteria for major products. Although these evaluation criteria do address such issues as traceability, understandability, and test coverage of requirements, they are largely subjective criteria questioning the use of *appropriate* techniques or *adequate* test coverage and do not require measurement of quantifiable properties.

Major defense acquisition programs require a Test and Evaluation Master Plan (TEMP). This document is used by the Office of the Secretary of Defense (OSD) and all DOD components for oversight of test and evaluation (T&E) activities. It is the basic planning document for all T&E related to a particular

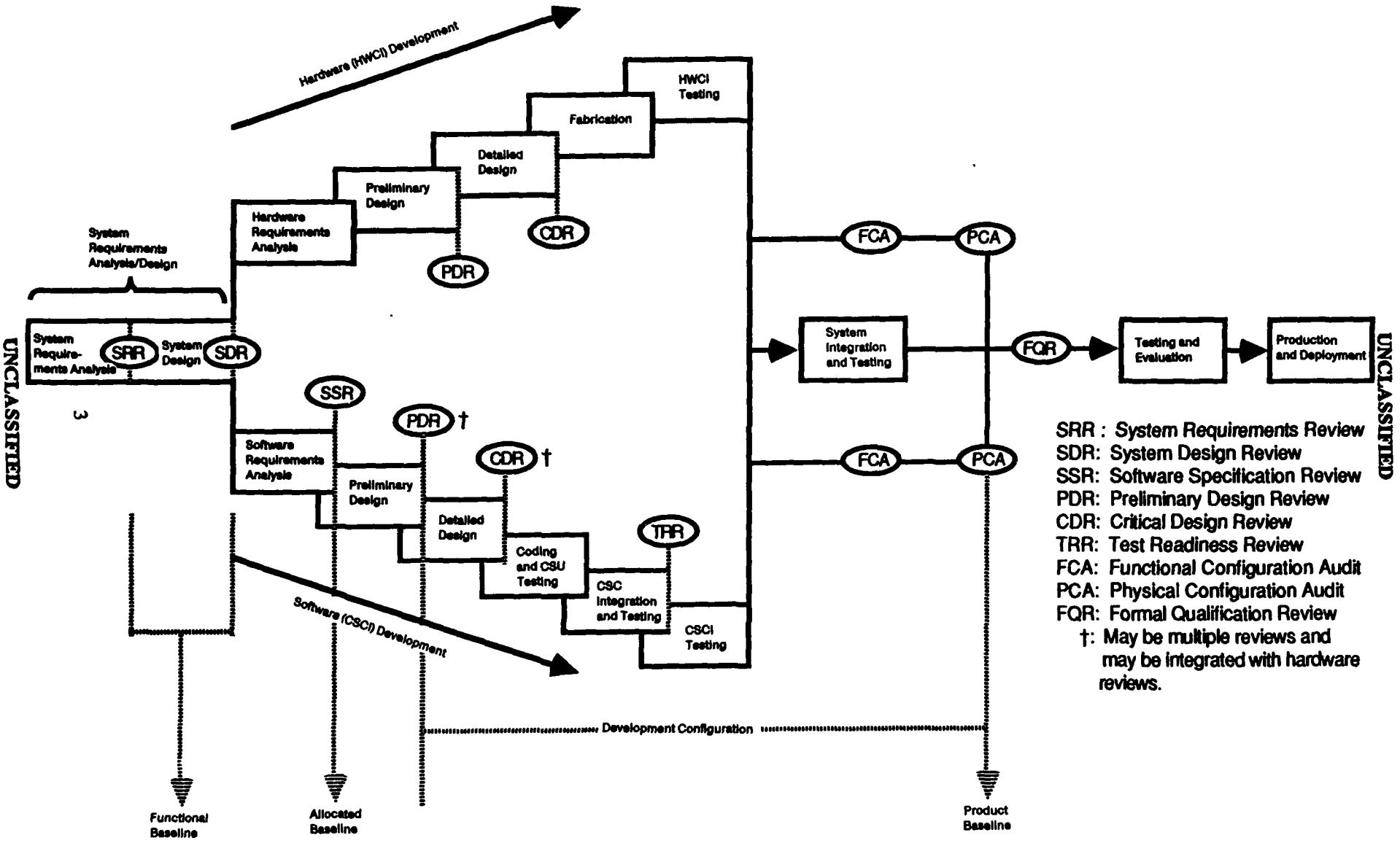

Figure 1-1. DOD 2167A - An Example of System Development Reviews and Audits

system acquisition, addressing T&E of both hardware and software elements. It specifies required technical and operational characteristics for the system, the T&E responsibilities of all participating organizations, a timing sequence for T&E activities, and necessary T&E resources. As a development effort proceeds, the conduct and results of T&E activities are included.

Traditionally, a software TEMP discusses T&E activities in the context of a waterfall software development life cycle. These activities fall into two categories. Developmental Test and Evaluation (DT&E) addresses attainment of critical *technical* characteristics such as correctness and certifiability. Operational Test and Evaluation (OT&E), on the other hand, is concerned with the *operational* effectiveness and suitability that are critical to the system's mission. OT&E may be preceded by Qualification Testing (QT) and Initial Operational Test and Evaluation (IOT&E). Once the system is deployed, it may be continued with Follow-on Operational Test and Evaluation (FOT&E).

It is generally good practice to have test teams which are independent of the software developers. This prevents the developers from propagating any misconceptions of the software through to testing and evaluation activities. It also allows for distinguishing between the different thought processes and skills required in testing; a good software developer is not necessarily a good tester, and vice versa. Many DOD projects require software to be tested and evaluated by an independent verification and validation (IV&V) organization prior to acceptance. Ideally, the IV&V organization is provided by the eventual software support agency.

1.3 State-of-the-Practice in Software Testing and Evaluation

Testing and evaluation have long been regarded as one of the weakest areas in the development of software systems. The structured programming movement started by Dijkstra in the 1960's [Dijk76a] was motivated by a desire to improve software quality, specifically by espousing development of demonstrably correct code from rigorous specifications. This movement has since grown into a much wider activity and the focus has broadened to address the introduction of increased rigor and structure into early life cycle activities. These efforts have led to considerable advances in testing and evaluation technology over the past decade. Unfortunately, these advances have not been matched by a corresponding improvement in the state-of-the-practice. It is not uncommon for intrinsic flaws to become apparent late in the development of a system, necessitating the discard of efforts that have already consumed substantial resources. Even worse, relatively trivial failures, often arising from unforeseen combinations of circumstances, have caused several medical, automotive, aeronautical, and defense systems to result in the loss of life [Neum87].

The typical practice in software testing and evaluation revolves around a software developer's intuition. As part of the DOD's Software Test and Evaluation Project (STEP), a review of current practices in twelve development and IV&V organizations undertaking DOD software efforts was performed [DeMi87a]. By and large, the only formalized techniques used in unit testing were structural testing (to ensure that a given percentage of statements or control paths were executed during testing), exercising the code with extremal and special values, and manual reviews. Only one development organization used functional testing (to examine the conformance of a software implementation to its specification), and only one IV&V organization used metrics to evaluate critical software properties. Integration and system testing primarily consisted of functional testing, with only one development organization performing reliability assessment. There was minimal use of automated tools to reduce the traditionally labor-intensive nature of testing. The only tools used by three or more organizations were file comparators, analyzers or code auditors, and dynamic execution verifiers. Although the STEP report was prepared in 1981 (it was slightly revised in 1987) these findings still reflect current practices.

The SDIO must ensure that *all* contractors developing SDS software use the best available technology. This cannot be achieved by simply imposing contractual requirements on software developers. Software developers must be provided incentives to adopt improved practices. To this end, the technology must first be applied to real software efforts to prove its utility and demonstrate the benefits that accrue, and the results of these efforts widely disseminated. Furthermore, a rigorous testing and evaluation approach can only be introduced into widespread practice when the automated tools that support its application are available.

1.4 The IDA Testing and Evaluation Workshop

Although several weak areas in technology are clearly obvious, it is difficult to determine which constitute *critical* shortcomings where intensive R&D over the next few years can be expected to result in practically applicable technology that can be exploited for Full Scale Development (FSD) SDS software. The leading researchers in the various areas of testing and evaluation technology were invited to join in a workshop to address this issue. Prior to the workshop, these researchers were requested to provide position papers outlining their views. Since several of the researchers are conducting promising research efforts discussed later in this report, they were also asked to provide explicit data on the current status of these efforts and describe how useful they expect this evolving technology to be for SDS purposes. The resulting collection of papers is presented in IDA Memorandum M-513 [Bryk89]. These researchers were asked to review an earlier version of this document. This version reflects the inputs and comments received from the workshop attendees.

In addition to researchers from academia, private companies, and DOD R&D centers, representatives from DOD organizations involved in Strategic Defense Initiative Organization (SDIO) software efforts were invited to attend the workshop. The complete list of participants is provided in Appendix C.

2. SDS SOFTWARE TESTING AND EVALUATION NEEDS

The SDS possesses characteristics which stress the application of current testing and evaluation technology. While these characteristics are individually not unique, never before have they been combined in such a large and safety-critical application. The concerns arising from these characteristics have long been recognized and the overall SDS development approach has been designed to facilitate software testing and evaluation.

This section reviews these special characteristics and outlines the SDS development approach. Against this background, specific objectives for SDS software testing and evaluation technology are identified and a practical mechanism for integrating testing and evaluation more fully into software development activities presented. Finally, related ongoing SDIO testing and evaluation activities are identified.

2.1 Impact of SDS Characteristics on Testing and Evaluation

The SDS is conceived as a multi-layered defense that destroys attackers in their boost, mid-course, and terminal phases. Some battle management functions, such as detection, acquisition and tracking, classification, and resource allocation are common to each layer of the defense. On a more global level, there are functions which occur in all phases of battle or span multiple phases of battle management. Examples in this case include surveillance, engagement, and situation assessment functions. Concurrency will be used widely, both to coordinate subsystems that are distributed geographically and spatially, and to provide the computational power needed on a single platform. The Battle Management/Command, Control, and Communication (BM/C3) system will have to integrate a software system significantly larger than any previous system. The characteristics of weapons and sensors are yet unknown and may remain fluid for several years. Consequently, system components will be subject to independent modification with possibly changing interfaces. While the system is expected to idle during most of its operational life, in an engagement it must operate under extreme and inflexible real-time constraints. Battle characteristics, and the SDS components available at that time, cannot be predicted with any certainty.

The system will be dynamically, as opposed to statically, linked. This is necessary (1) because of the constant physical movement of system components, (2) to allow for rapid reconfiguration of the system in response to enemy countermeasures, and (3) to compensate for the loss of components that can be expected in a hostile operating environment. The system itself is expected to be developed and deployed in an evolutionary manner, with each version of the system meeting different requirements. These requirements will evolve to provide increasing functionality, respond to major advances in technology employed in (or countered by) the SDS, and reflect changes in the political arena.

2.1.1 Concurrent, Distributed Software

Concurrent software, whether with apparent parallelism on a uniprocessor or actual parallelism on multiprocessor or distributed architectures, is subject to types of failure which do not occur with sequential software. These failures arise from problems with the synchronization and communication between processes. Examples include deadlock and the simultaneous update of shared variables.

In the initial testing process individual concurrent processes can be treated, in some senses, as independent, sequential programs. When the time comes to test the combined run-time behavior of the processes, however, the non-determinism inherent in their concurrent behavior results in additional

testing problems. When several processes execute with pseudo parallelism on a single processor, for example, the scheduler may make different choices about the order of execution in response to conditions external to a program. Consequently, a program supplied with the same set of test data on two different executions can exhibit markedly different behavior. Moreover, if a program is ported across environments, the new environment may employ an entirely different scheduling algorithm. It may even include different processor construction which invalidates previous testing, for example, a different real-time clock.

The non-determinism of concurrent software considerably increases the difficulty of detecting and correcting errors and faults dependent on the relative scheduling of events and resources. Special techniques and tools are required to mitigate these problems. Even relatively straightforward tasks, such as monitoring testing coverage, become more complex. In this example, new types of coverage measures are required to reflect the coverage achieved for both individual processes and synchronization activities. The instrumentation needed to collect coverage information can distort the execution of the program, thus introducing a complicating factor which exacerbates the non-determinism problem.

2.1.2 Real-Time Software

The principal distinction of a real-time system is that the physics of the application imposes time constraints on some of the computations. These time constraints are not merely performance metrics but a correctness property for the computations. In other words, performance is a critical factor. The software functionality cannot be tested independently of performance and, of course, this is dependent upon the execution environment. In systems where the requested processing in a given time period is likely, at times, to exceed available computation power, the issues of timeliness and importance are tightly coupled. Testing must demonstrate that performance has been optimized for the most important cases. In command, control, and communication systems, such as the SDS BM/C3, these important cases are typically the exceptional ones, not the most frequent ones.

Process control systems are good examples of real-time systems. These systems are required to respond to inputs within restricted time constraints to control ongoing external processes. There is inherent non-determinism in the relative scheduling of events that arises from the varying order in which inputs may occur. As with concurrent software, this non-determinism poses problems for testing and evaluation. The rigorous timing constraints inherent to all real-time systems give rise to a further difficulty; namely, any instrumentation of code has a significant impact on the software performance and considerably complicates the analysis and repeatability of test results.

Since the execution environment is typically highly specialized, real-time systems are usually developed on a different machine (the host) than that on which they will operate (the target). This allows the host environment to provide development tools which may be unavailable in the target environment. Testing and evaluation are performed on both machines. While testing practices vary from one project to another, it is common for testing on the host to emphasis unit testing using the standard techniques employed for non real-time applications. Integration testing may also be performed, but later stages of integration testing require an environment simulator. Unit and integration testing are repeated on the target machine, along with system testing. Here again, an environment simulator may be required. If the target machine has no debugging facilities, failures occurring during target testing may require reconstructing the test in question on the host. This is exceptionally difficult and not always possible.

Although the limitations of the target environment may necessitate testing on the host, differences between these environments may cast doubt on the validity of the testing.

2.1.3 Dynamic Environments and Fault Tolerance

Dynamic linking of subsystems and components introduces another level of uncertainty. If all possible reconfigurations, and the operational requirements which apply to each, could be explicitly identified, it would still be infeasible to fully test each instantiation of the system. Nevertheless, techniques for rigorously analyzing the factors which may necessitate reconfiguration and determining how the system should respond to each factor individually or in combination must be developed. Innovative testing and evaluation approaches are needed to provide the greatest possible coverage of the diverse environments and operating modes expected to be encountered. In particular, all safety critical conditions leading to, or arising from, reconfiguration must be identified.

One of the uses of dynamic environments is to increase the fault tolerance of a system. The infeasibility of routine maintenance and repair of space-based components, and lack of time for any repair activities during an engagement, mandate that SDS make extensive use of fault tolerance. Testing and evaluation of fault tolerant systems is not a well-understood process. Although some aspects, such as the nee to drive the system to failure, are recognized, they present additional testing challenges for the SDS.

At a more software specific level, fault tolerance poses additional questions. Consider for a moment N-version programming, which is one of the best known techniques for increasing software fault tolerance. The basic premise of N-version programming is that different software developers are likely to introduce different faults into the software. Therefore, a large number of versions of a piece of software are developed independently and executed together. The results from all the versions are compared and the consensus assumed to be correct. Some researchers hold such high expectations of the inherent tolerance of faults in the different versions as a group, that they claim individual versions need less testing than usual [Aviz85]. Unfortunately, recent experiments [Knig86a] indicate that the necessary statistical independence of faults rarely occurs in practice, undermining the fundamental assumption of N-version programming. Until there are approaches for analyzing the fault independence achieved in any particular application, and determining the resultant impact on reliability, this technique must be used with care. Similar cautions apply to other software fault tolerance techniques.

2.1.4 In-Line Testing

The inability to perform *full-scale* operational testing in its deployed environment is one of the most frequently stated arguments against the feasibility of the SDS. It is a legitimate concern. Even ignoring the various political constraints, there are very real technical and economic prohibitions against such full-scale testing. It would be very visible, providing potential adversaries with information which could be used to negate the usefulness of the system. Moreover, the validity of the testing would be short-lived, outdated by even minor changes in battle characteristics that are not under U.S. control.

Nevertheless, the system must be designed to facilitate in-line testing once deployed. Although the bulk of the testing will be performed prior to deployment, some testing will subsequently be necessary if only to validate assumptions made in the earlier testing, or diagnose failures reported by monitoring processes. The prime difficulty here is that, once deployed, the system must operate continually. Although most system components will be inactive for an indefinite period, the SDS must be ready to respond instantly to any threat. The capability for in-line software testing has to be designed into the system.

2.1.5 Evolving Requirements

The impact of continually changing requirements on software testing and evaluation is more pragmatic

than the concerns so far raised. The problem is largely an economic one; the need to perform expensive retesting after a change is made. Planning for such regression testing is often forgotten during system and software engineering activities when developers are already faced with considerable technical challenges. If, however, the retest concern is not addressed in a timely manner, the result will be software which, when modified, requires retesting out of all proportion to the scale of the change. It would rapidly become *economically infeasible* to keep such a system responsive to changing threats.

The amount of regression testing required after a software change must be (1) predictable, and (2) proportional to the *size* of the change. Traceability from system requirements to later development products is not sufficient to ensure this. Good engineering practices, such as information hiding, must be employed with regression testing specifically in mind, at the earliest stages of system development. Regression testing will also be facilitated by maintaining good records of testing activities. Since it is unrealistic to expect that full details about all testing events can be stored, a mechanism for identifying and structuring pertinent details must be developed.

Revisions to system requirements are not necessarily confined to new versions of the system. For example, a change in the attack capabilities of adversaries may require immediate modifications to the system. The speed with which these modifications can be implemented and validated will be critical since the system may be unable to fulfill its mission in the interim and would itself be increasingly vulnerable to attack. These support activities cannot be left to less experienced software developers, as is often the case. Skilled software developers who have an intimate knowledge of the system must be available to devote their immediate attention to implementing necessary changes.

2.2 The SDS Development and Testing Paradigm

As is apparent from the preceding discussion, consideration of software testing and evaluation cannot be postponed until the software is designed. It must be an integral part of the whole development process. The SDIO has begun to act upon this insight, and some initial decisions have already been made.

2.2.1 The SDS Software Development Approach

The SDS software development approach will integrate design, prototyping, and simulation to allow formulation of SDS requirements to be tied to early analyses of the effectiveness and suitability of alternative architectures. A key ingredient of this approach is the use of formal specifications for representation of SDS architectures. Formal specifications offer several benefits. They help to ensure that system interfaces and functions are cleanly and modularly separated, thus improving the testability of the system as a whole. They are also a prerequisite for all formal testing and evaluation approaches, including testing formal specifications and proving formal properties about the specifications. One of the most significant benefits, however, accrues from their potential for simulation. The SDIO has developed a formal notation, called the Strategic Defense Initiative Architecture Dataflow Modeling Technique (SADMT) [Linn88], to exploit this potential. This notation is designed to meet the specific needs of the SDS and BM/C3 architectures. It represents architecture designs in such a way that they can be directly input to simulation.

Expanded capabilities of this nature are being developed for the National Test Bed (NTB) where requirements range from high-level system simulation through full-fidelity simulation of individual components. This approach supports the detection of requirement and design errors early in the development process when their correction is easy and does not require undoing much completed work.

Early operational testing is a natural consequence of the prototyping approach and timely analysis and comparison of architectural models is possible. Prototypes can be exercised to determine, for example, the impact of various damage scenarios on the ability of an architecture to continue functioning. Results can be refined and expanded through an eventual full-scale engineering decision and deployment. This allows the system and its components to be tested and evaluated over an extended period of time and under a wide variety of operating modes, conditions, and environments.

The feasibility of building a testable SDS is tied to the choice of an SDS architecture. The SDS has been described as a system that:

> "... would need to respond to an offensive strike as a single organism, coordinating perhaps millions of separate actions in a schedule timed in milliseconds." [Adam85]

System architectures that require such coordination between elements demand excessively sophisticated software and cannot be adequately tested. The complex interaction of the units or components in such an architecture means that the test of an individual unit will provide little assurance about the adequacy of functioning for the system as a whole. Consequently, distributed, decentralized architectures which reduce the complexity of the software by eliminating needless coordination have been chosen. Independence allows clear design and separation of functions and explicit specification of the interfaces among the functions. This in turn allows use of implicit coordination schemes where elements can act independently based on their limited knowledge of the status of other elements and how these are anticipated to behave. Of course, achieving a good, distributed, decentralized architecture is not easy. It requires an effective modular decomposition of the system although there is little experience in how to accomplish this and how to evaluate the results. Nevertheless, the use of decentralized architectures makes testing SDS analogous to testing current offensive weapons. In addition to increasing the testability of the overall system, concepts such as these, that reduce the complexity of required software, also increase system security, evolvability, and robustness.

2.2.2 Use of the Ada Programming Language

In accordance with DOD Directives 3405.1 and 3405.2 [DODD87c,DODD87b], the SDIO has adopted a policy requiring all software to be developed in Ada. This required use of a single programming language provides an enormous advantage for testing and evaluation; namely, it allows a common set of code-level testing and evaluation techniques and tools to be used across all SDS software efforts. This makes the software developers' work easier, facilitates the introduction of state-of-the-art technology, and increases the cost-effectiveness of necessary tool development.

Ada was specifically developed to support "programming-in-the-large." Productivity and maintenance issues were key drivers in its definition. Savings both through increased productivity and decreases in maintenance cost are expected from the use of Ada. The techniques for interface specification and the constructs for modularity and information hiding of Ada have been specially designed to support state-of-the-art software development methods and technology, including the development of portable and reusable code.

The choice of Ada as the required programming language offers further advantages. Ada combines advantages of many previous separate languages, such as strong typing and data abstraction. Of course, techniques for evaluating conformance to proper Ada coding practices are necessary to ensure that strong typing and other desired practices are exploited to take maximum advantage of the language. Through the use of predefined and user-defined exceptions, Ada also provides a basic capability for run-

time detection and recovery from certain types of faults. Additionally, the Ada Compiler validation effort has resulted in an impressive improvement in code portability across widely differing machines (in comparison with other languages). Further advantages in code reuse from the combination of portability, abstract interfaces, and generics, are becoming more evident as Ada experience increases, especially in the area of constructing large systems.

One of the major challenges facing the SDS is that of producing *trusted* software. While Ada alone does not ensure the development of trusted software, it does provide language features important to the implementation and verification of security attributes.

2.3 Integrating Testing and Evaluation into Development Activities

All too frequently, software testing and evaluation are regarded as tack-on activities to be performed after code has been written. A software product is developed and then testing and evaluation performed as a discrete step to get the "bugs" out. Herein lies a fundamental misconception that must be eliminated before significant increases in software reliability can be achieved. Reliability *cannot* be tested into software. Instead, development must be seen as an error prevention activity, with testing and evaluation providing continual feedback on the validity of the current activity in its own right, and its implications for subsequent activities. Only then will the increased visibility into the development process necessary for timely identification of factors which impact testability and early diagnosis of problem areas be possible.

Consideration of testing and evaluation concerns must be brought forward to the earliest development activities; this is true at both the system and software levels. For example, the time to develop the system test plan is when system requirements are defined. If system requirements are accompanied by details on *how* the conformance of the final system with those requirements will be determined, untestable requirements or those whose testing incurs unacceptable costs can be immediately identified; preventing millions of dollars being invested in an undeployable system. In the same way, the integration test plan should be developed during architectural design activities. Usually, the unit test plan would be developed during detailed design and coding of the software, leading to an overall relationship between test plan elements as shown in Figure 2-1[1].

This figure contains some simplifications. For a system the size and complexity of SDS, for example, at least one additional level of test planning is necessary, between integration and unit testing. Additionally, only the major flows between activities and products are shown, the additional flows necessary to establish the traceability of testing requirements have been omitted.

Figure 2-1 provides the starting point for an SDS software testing and evaluation process model. It demonstrates a practical mechanism for the integration of testing and evaluation concerns into development activities. The model shows the necessary planning for testing final code products. It also indicates the need to test intermediate products. Experience has repeatedly shown that it is most effective to detect and correct an error early in the lifecycle; the cost of detecting and correcting software faults increases by a factor of 100 or more as the system is integrated [Boeh81]. At each stage in the

1. This model of software processes for developing and testing SDS software was first proposed by Prof. L.J. Osterweil from the University of California at Irvine. It was subsequently elaborated by Prof. Osterweil and Prof. L.A. Clarke (from the University of Massachusetts).

development cycle, testing requirements for the next stage must be specified. Of course, during the following stage, the software developers will accumulate additional testing requirements for the products of that stage. The initial testing requirements passed on from the previous stage will, however, ensure that previously identified concerns are addressed in a timely manner.

Testing requirements must be traceable through the entire system development process. Consequently, they must be maintained as permanent attributes of the software and expressed in a formal notation which permits the necessary analysis of consistency and completeness. Additionally, these requirements must be stated in explicit and measurable terms against which testing and evaluation activities can be monitored and analyzed to determine the effectiveness of testing-to-date and identify any outstanding testing needs. Thus, testing requirements are envisioned as driving the testing at each stage and providing a mechanism for integrating testing and evaluation activities into an overall testing strategy.

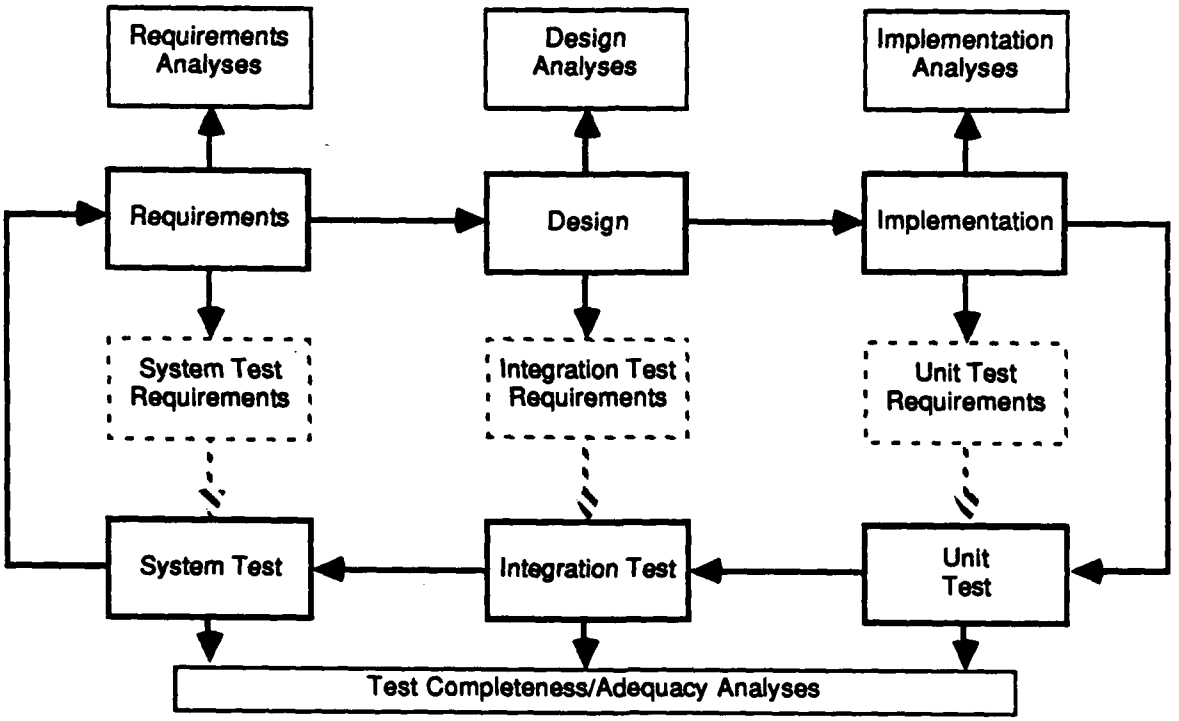

Figure 2-1. Preliminary SDS Software Testing and Evaluation Process Model

What is most important about this proposed process model is not the specific process itself. Rather that this process is an example of the flexibility in performing testing and evaluation that arises from the principle of software process modeling and the adoption of software process programming as the basis for software development specification and management. The particular development and testing model discussed in this report is intended largely to indicate the innovative ways in which testing and evaluation could be proposed, implemented, and experimentally evaluated if process modeling in general, and process programming in particular, are understood and exploited. SDS software development efforts should adopt the principle of software process programming and use it as the basis for exploring effective software life cycle modeling and as the basis for realizing the kinds of flexibility that will be needed.

In view of the existing lack of research results, experimental technologies, and finished products, the SDIO must expect that it will have to adopt and assimilate techniques and tools of unexpected types and

varieties. If a specific, fixed and rigid software development process is assumed, it is certain that the assimilation of these new techniques and tools will be awkward, if not impossible. Thus development life cycle flexibility is essential.

It is important to note that the ability to perform rigorous testing and evaluation on intermediate products is dependent on the use of formal notations at all stages of development. Furthermore, the need to provide timely feedback on development activities, particularly for large systems, implies the need for incremental testing and evaluation performed on possibly incomplete products at each development stage. As noted by Clarke [Clar88a], although systems start out incomplete and are developed incrementally, most of the tools implementing current techniques work on complete software representations. This handicap must be resolved. Finally, since no single technique is sufficient at any one stage, testing and evaluation techniques must be applied cooperatively. Simply applying one technique sequentially after another yields neither increased nor efficient fault detection. Instead, different testing techniques must be tightly integrated. Quantitative information on the capabilities of particular techniques when applied to different types of software is also needed.

Many issues remain to be resolved. What information should be captured in the various test plan elements and in testing requirements? How can the necessary level of confidence in results dictated by the possible impact on mission performance and other consequences of failure be specified? What is the minimum synopsis of testing activities sufficient to facilitate retesting? Tasks to address these, and other, questions are discussed in Sections 8 and 9.

2.4 Related On-Going Activities

Before proceeding, it is useful to note some of the other SDIO testing and evaluation activities that are underway. While by no means an exhaustive list, the following identifies those activities relevant to the purpose of this report.

2.4.1 Organizational Activities

The SDIO T&E Directorate has established an SDS T&E Working Group. One of the tasks being initiated by this group is an activity designed to address outstanding policy, organizational, and logistics issues for SDS software testing and evaluation. For example, the NTB will be the focal point for element and system level testing and evaluation and provide a communications network linking geographically distributed facilities within the National Test Facility (NTF). One of the goals of this group is to determine whether the NTB should be the IV&V organization for SDS software.

The Simulation Engineering Group has been established as an advisory group to the SDIO on the subject of simulation and system performance evaluation. It has brought together the leading researchers in this area to review ongoing research. A working group of this panel is currently developing requirements for the Advanced Simulation Framework of the NTB.

2.4.2 Software Center

The SDIO is establishing a Software Center which will provide a focal point for software technology awareness across the development activities in the SDS by an extensive training program for senior software engineers. This training will also address the SDS development approach, for example, software engineering environments, and software methodology and standards. Evaluated, working examples of approved environments will be available for inspection, and will also be used at the Center in the

evaluation and certification of software submitted to the NTF by the elements. The Center will execute research projects at the direction of the SDIO, and will track technology advances made in other programs of interest for the SDS. The Center will also provide technical support for the development of a secure repository of trusted reusable components (mainly in Ada) for the SDS.

2.4.3 Development of Technical Policy

Proper use of software engineering approaches is an essential prerequisite for the development of reliable and testable software. Accordingly, the SDIO is developing a Software Policy [SDIO88a,SDIO88b] which specifies those practices and techniques that must be employed in the development of mission-critical SDS software. In a related effort, the SDIO C3 System Operational and Integration Function (SOIF) is sponsoring the development of guidelines for tailoring DOD Quality Assurance standard DOD-STD-2168 [DOD86a] for use on SDS software efforts.

The National Computer Security Center (NCSC) is developing an SDS Security Policy which will provide guidance on the use of formal verification technology to develop secure and trusted software. The first draft of this policy is expected before the end of 1988.

2.4.4 Planning Activities

Development of the SDS TEMP began in early 1987. The most recent version [SDIO87] was completed in June 1988. While the evolving software annex of the TEMP essentially conforms to DOD TEMP guidelines [DODD87a], the non-waterfall SDS development life cycle has necessitated some divergence. For example, to take maximum advantage of the prototyping approach, the software annex requires early OT&E of an evolving series of prototypes and experimental versions. Early OT&E will revolve around examination of architecture designs. Subsequently, DT&E and OT&E will be performed on all major prototypes and experimental versions, through to operational SDS software. T&E of prototypes and experimental versions will not only examine the technical and operational properties of the products under test, but will serve as the basis for projecting properties of the deployed system. Emphasis will be placed as well on an early operational assessment of the process of T&E in conjunction with the objects of T&E.

3. DYNAMIC ANALYSIS TECHNOLOGY

This section provides an overview of the state-of-the-art in dynamic analysis technology. It discusses the major techniques and ongoing research efforts relating to the testing of both sequential and real-time/concurrent software. Since these techniques require executing software products with test data, they are largely used for testing program code. In those cases where software requirements and designs are executable, however, many of the techniques can be applied to pre-implementation products. Although the burgeoning development of large-scale testing and evaluation environments is discussed in the Section 4, special concerns relating to oracles, coverage analyzers, and debuggers are covered in this section. Finally, the critical gaps in technology are elucidated.

By necessity, the following material has been kept brief. Sources for further information are given in a supporting bibliography [Youn88a]. This bibliography includes an extensive index to guide those seeking sources for information on a particular topic.

3.1 Techniques for Dynamic Analysis of Sequential Programs

Of the areas covered by this report, technology for the dynamic analysis of sequential programs is the most mature. This is not to say that there is a complete technology sufficient for all practical purposes. Indeed, advances in fundamental theory can be expected for many more years. Currently, while there is a small body of techniques that can be applied to SDS software considerable work is needed to turn the few existing research prototypes into industrial strength, practical tools.

Testing techniques can be categorized as either white-box or black-box. White-box approaches derive test data from consideration of the program structure. *Path selection* approaches are based on graph-theoretic notions of control flow or data flow. They divide the input space of a program into domains which cause particular control or data paths to be followed, and the program is executed on test cases that are constructed by selecting test data from these domains. Remaining white-box approaches can be further distinguished as either error-based or fault-based[2]. *Error-based* approaches apply the whole realm of programming knowledge, such as information about error-prone language constructs, to the task of selecting test data which can find faults in the execution of a particular program path. *Fault-based* approaches, as a group, are the most recent techniques to be developed. Here test data is designed to demonstrate the absence of a predetermined set of faults in program statements. Black-box approaches, also called *functional testing* approaches, derive test data from the functional requirements of a program, without regard to the internal structure of the program.

The following subsections identify the notable dynamic analysis techniques in each category. As variations on several of the described techniques exist, these techniques should be regarded as a representative subset of those available, not an exhaustive listing.

The key features of several of the techniques discussed are summarized in Table 3-1. This table identifies the types of faults that can be detected and whether this detection is guaranteed. It identifies if

2. Within the field there is a lot of confusion between error-based and fault-based testing. This is partly due to the fact that before the adoption of the IEEE terminology which gives different meanings to the words "error" and "fault," these two words were used interchangeably. Consequently, the classifications used in this report are sometimes historical artifacts and the distinctions between the identified methods is not always so clear.

automated tools are available to support application of the techniques to Ada programs, or programs written in some other language. Under the heading of Inputs to the Testing Process, it indicates whether common inputs such as program specifications, program text, and test data are required. Path analysis of a program (usually provided by symbolic evaluation, see Section 4.1) is included in this category as another relatively common input. Additionally required inputs are identified separately. Similarly, under Outputs of the Testing Process, the table shows which techniques provide program outputs, which require oracles to predict the correct outputs against which program outputs can be compared, and which support locating the position of a fault in the program text. Again, other less common outputs are described separately. Finally, the table indicates those techniques that are suitable for practical application and those that remain, to varying degrees, impractical for widespread application at the present time.

CLASSIFICATION	NAME	ERRORS DETECTED: STATUS	ERRORS DETECTED: TYPES OF ERRORS	ERRORS DETECTED: DETECTION GUARANTEED	AUTO SUPP: Ada	AUTO SUPP: OTHER	INPUTS TO TESTING PROCESS: SPECIFICATION	INPUTS TO TESTING PROCESS: PROGRAM TEXT	INPUTS TO TESTING PROCESS: PATH ANALYSIS	INPUTS TO TESTING PROCESS: TEST DATA	INPUTS TO TESTING PROCESS: OTHER	OUTPUTS OF TESTING PROCESS: PROGRAM OUTPUTS	OUTPUTS OF TESTING PROCESS: ORACLE	OUTPUTS OF TESTING PROCESS: ERRORS LOCATED	OUTPUTS OF TESTING PROCESS: OTHER
STRUCTURAL	Data Flow Testing	1	General	-	x	x	-	x	l	x	Path Selection Criteria	x	x	-	Coverage Measure
	Structural Testing	1	General	-	x	x	-	x	l	x	Path Selection Criteria	x	x	-	Coverage Measure
ERROR-BASED	Algebraic Testing	3	Computation	x	-	-	-	x	x	x	Algebraic Equivalence Theorems	x	x	-	Symbolic Traces
	Computation Testing	2	Computation	-	-	p	-	x	x	x	Path Computations	x	x	p	
	Domain Testing	2	Path selection	x	x	x	-	x	x	x	Test Data Selection Criteria	x	x	p	Error Bound
	Perturbation Testing	3	Predicate and computation	x	x	x	-	x	x	x	Path Selection Criteria, Perturbing Functions	x	x	x	Adequacy Measure
	Partition Analysis	3	Missing path, other domain, computation	-	-	p	F	x	x	x	Test Data Selection Criteria, Symbolic Path Computations	x	x	-	Compatibility, Equiv., Consistency wrt Spec.
	Revealing Subdomains	3	Chosen potential	x	-	-	x	x	x	x	Potential errors	x	x	p	Related Errors
FAULT-BASED	ESTCA Testing	2	Extremal/Special	-	-	-	x	x	l	x	Heuristics	x	x	p	
	Mutation Testing-strong	2	Set of frequently occuring	x	x	x	-	x	x	x	Error Operators	x	x	x	Adequacy Measure
	Mutation Testing-weak	3	Set of frequently occuring	-	x	x	-	x	x	x	Mutation Transformations	x	x	x	Adequacy Measure
	RELAY	2	Chosen potential	x	-	-	-	x	x	x	Potential errors, Symbolic Evaluation	x	x	x	Adequacy Measure
FUNCTIONAL	Boundary Value Analysis	1	Errors/Ambiguities wrt spec.	-	-	-	x	x	-	x	Boundary Conds., I/O Equiv. Classes	x	x	-	
	Cause-Effect Graphing	1	Errors/Ambiguities wrt spec..	-	-	-	x	x	-	x	Cause-Effect Graph, I/P Combinations	x	-	-	
	Equivalence Partitioning	1	Errors/Ambiguities wrt spec.	-	-	-	x	x	-	x	ESTCA Heuristics, I/P Equiv. Classes	x	x	-	
	Grammar-Based Testing	3	Errors/Ambiguities wrt spec.	-	x	x	F	x	-	x	Heuristics	x	-	-	
	Stat./Random Testing	1	General	-	-	x	x	x	-	x	I/P Distributions	x	x	-	Reliability Estimates

KEY: x feature required/exhibited; p partial ability; l may be used; F formal specification required
1 ready for productization; 2 possibly ready for use; 3 experimental

Table 3-1. Key Features of Dynamic Analysis Techniques for Sequential Programs

3.1.1 Path Selection Techniques

Path selection techniques focusing on the control flow through a program were the first systematic testing strategies to be developed. Here the control structure of a program is represented as a finite, directed graph with single entry and single exit points which correspond to the program beginning and end. The nodes in the graph represent program statements, connected by edges which correspond to possible control flows between statements.

The set of techniques based on control flow graphs are collectively known as structural testing techniques. The initial three categories of structural testing techniques were: statement testing, branch testing, and path testing. Of these, statement testing requires executing a program with test data that cause each program statement to be executed at least once and is the weakest approach. At the other extreme, testing of all program paths is considered the ideal. However, since the total number of possible program paths in a typical large computer program is in the range of 10^5 to 10^7 (without counting the number of loop traversals within each path), exhaustive path testing is generally infeasible. Branch testing requires executing all program branches at least once and is commonly agreed to be the minimum acceptable path selection criterion.

More recently, different approaches for developing intermediate strategies between testing all branches and all paths have been developed. Woodward, Hedley, and Hennel [Wood80a], for example, consider Linear Code Sequence and Jump (LCSAJ) program units. These are sections of the code through which the flow of control proceeds sequentially until terminated by a jump. Program execution paths can be described in terms of concatenated LCSAJs. A series of progressively more demanding coverage measures is then based on examining the proportion of distinct subpaths of length n LCSAJs exercised by the test data.

Another set of techniques intended to bridge the gap between branch and path testing are based on data flow analysis. These techniques reflect the intuition that the path from a variable assignment to its use must be executed to provide confidence that the correct value was assigned to that variable. Consequently, data flow testing techniques select test data forcing the execution of different interactions between a variable definition and references to that variable. As an example of the complexity and maturity of current testing techniques, the underlying theory of data flow testing is summarized in Figure 3-1 (taken from [Fran86]).

Several different path selection criteria based on data flow relationships have been developed. Clarke and Richardson have formulated a uniform model for three families of data flow testing criteria (Rapps-Weyuker, Ntafos' required k-tuples, and Laski-Korel) and defined the criteria in each of these families in terms of that model. They analyzed the path coverage of each criterion and developed a subsumption hierarchy that demonstrates how these criteria relate to each other [Clar85a,Clar86a]. This analysis showed that the most comprehensive of the criteria in each family are incomparable with each other as a consequence of the different foci of the families. More recently, a new family of criteria [Fran86] has been developed to circumvent the problem, common to all path-directed testing, of identifying non-executable paths.

Weyuker has recently completed an empirical evaluation of the complexity of data flow testing techniques [Weyu88]. This study focused on the number of test cases needed to satisfy different criteria, one of the cost elements in applying these techniques. While theoretical upper bounds on the number of test cases needed for most criteria are quadratic or exponential, Weyuker found that, in practice, far fewer test cases are necessary. Figure 3-2 reproduces the information presented in [Weyu84a] and [Weyu88].

In a series of papers [Howd78a,Howd78b,Howd82a], Howden has addressed the theoretical

The subprogram under test is represented by a flow graph. Variable occurrences are classified as *definitions, undefinitions,* or *uses*; where uses either directly affect the computation or reveal the result of some earlier definition (*c-use*), or directly affect the flow of control (*p-use*).

A c-use of a variable x in node i is defined to be a *global c-use* if the value of x has been assigned in some block other than block i. A path $(i,n_1,\ldots,n_m,j)$, $m \geq 0$, containing no definitions or undefinitions of x in nodes $n_1,\ldots,n_m$ is called a *def-clear path* wrt x from node i to node j and from node i to edge(n_m,j). A node i has a global definition of x if it has a definition of x and there is a def-clear path wrt x from node i to some node containing a global c-use or edge containing a p-use of x. The subprogram's *def-use* graph is obtained by associating with each node i, the sets c-use(i) and def(i), and with each edge(i,j) the set p-use(i,j). In addition, assumptions include that the entry node has a definition of each parameter and each global variable which occurs in the subprogram, and the exit node has an undefinition of each local variable and a c-use of each variable parameter.

V	=	the set of variables
N	=	the set of nodes
E	=	the set of edges
def(i)	=	{x ∈ V | x has a global definition in block i}
c-use(i)	=	{x ∈ V | x has a global c-use in block i}
p-use(i,j)	=	{x ∈ V | x has a p-use in edge (i,j)}
dcu(x,i)	=	{j ∈ N | x ∈ c-use(j) and there is a def-clear path from i to j}
dpu(x,i)	=	{(j,k) ∈ E | x ∈ p-use(j,k) and there is a def-clear path from i to (j,k)}

A *def-c-use* association is a triple (i,j,x) where i is a node containing a global definition of x and j ∈ dcu(x,i). A *def-p-use* association is a triple (i,(j,k),x) where i is a node containing a global definition of x and (j,k) ∈ dpu(x,i). A *simple path* is one in which all nodes, except possibly the first and last, are distinct. A *loop-free path* is one in which all nodes are distinct. A path $(n_1,\ldots,n_j,n_k)$ is a *du-path* wrt x if n_1 has a global definition of x and either: i) n_k has a c-use of x and $(n_1,\ldots,n_j,n_k)$ is a def-clear simple path wrt x, or ii) (n_j,n_k) has a p-use of x and $(n_1,\ldots,n_j)$ is a def-clear loop-free path wrt x.

Informally, the testing criteria require that test data execute def-clear paths from each node containing a global definition of a variable to specified nodes containing global c-uses and edges containing p-uses of that variable. For each variable definition, all def-clear paths wrt that variable from the node containing the definition to some of the uses reachable by some such path must be executed. More precisely:

THE DATA FLOW TESTING CRITERIA

Test T satisfies criterion C for subprogram P if for each node i and x ∈ def(i) the set ρ of paths executed by T covers the following associations:

CRITERION	ASSOCIATIONS REQUIRED
All-defs	Some (i,j,x) s.t. j ∈ dcu(x,i) or some (i,(j,k),x) s.t. (j,k) ∈ dpu(x,i).
All-p-uses	All(i,(j,k),x) s.t. (j,k) ∈ dpu(x,i).
All-p-uses/some-c-uses	All (i,(j,k),x)s.t. (j,k) ∈ dpu(x,i). *In addition*, if dpu(x,i) = ϕ then some (i,j,x) s.t. j ∈ dcu(x,i). Note that since i has a *global* definition of x, dpu(x,i) = ϕ => dcu(x,i) ≠ϕ.
All-c-uses/some-p-uses	All (i,j,x) s.t. j ∈ dcu(x,i). In addition, if dcu(x,i) = ϕ then some (i,(j,k),x) s.t. (j,k) ∈ dcu(x,i). Note that since i has a *global* definition of x, dcu(x,i) = ϕ => dpu(x,i) ≠ϕ.
All-uses	All (i,j,x) s.t. j ∈ dcu(x,i) and all (i,(j,k),x) s.t. (j,k) ∈ dpu(x,i).
All-du-paths	All du-paths from i to j wrt x for each j ∈ dcu(x,i) and all du-paths from i to (j,k) wrt x for each (j,k) ∈ dpu(x,i).

For comparison we also define the criteria *all-nodes* (respectively all-edges, all-paths) which require that ρ cover every node (respectively every edge, every path) in the flow graph.

The following relationship holds:

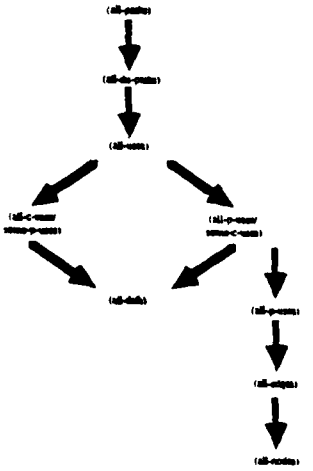

Figure 3-1. Data Flow Testing Theory

Theoretical Complexity

Let P be a program with n variables, m assignments, i input statements, and t conditional statements. Let a test case consist of a single vector of input variables.

Then, the all-nodes and all-edges criteria require at most t+1 test cases. All-defs requires at most $m+i*n$ test cases. The all-p-uses, all-c-uses/some-p-uses, all-p-uses/some-c-uses and all-uses criteria require at most 1/4 (t^2 +4t+3) test cases. All-du-paths requires at most 2^t test cases.

Empirical Complexity

The automated data flow testing tool ASSET was applied to a suite of programs "Software Tools in Pascal" by Kernighan and Plauger [Kern81]. Testers were instructed to select atomic test cases using the strategy of their choice. The data flow criteria were used as adequacy measures. For each program the following was computed:

	all-c-uses	all-p-uses	all-uses	all-du-paths
1. Least squares line: $t = \alpha+\beta$, where t is the number of test cases sufficient to satisfy each criteria and d is the number of decision statements	.52d+1.87	.76d+1.01	.81d+1.42	.93d+1.40
2. Weighted average of the ratios of d to t	.43di	.70di	.72di	80di
3. Maximum value of the ratio of t to d	3.5	2.33	3.67	3.67
4. Weighted average of the ratios of the theoretical upper bound on the number of test cases needed to satisfy t	7.61	4.30	4.15	243.93
5. Weighted average of the ratios of the number of test cases sufficient to satisfy all-defs to t			1.79	1.96
6. Weighted average of the ratios of the number of test cases sufficient to satisfy all-uses to the number sufficient to satisfy all-du-paths				.93

Although the theoretical upper bounds on the number of test cases needed to satisfy most of the criteria are quadratic or exponential, in practice only small numbers of test cases (as compared to the program size) were needed.

Qualitative observations from empirical study:

— All-p-uses was generally harder to satisfy than all-c-uses, and a test which satisfied all-p-uses usually satisfied all-c-uses too (despite the fact that these are independent criteria).

— Although all-uses is more demanding than all-p-uses, a test set which was adequate when assessed by all-p-uses was generally also adequate using all-uses.

— Even though the all-du-paths criterion has an exponential upper bound whereas the all-uses criterion has a quadratic upper bound, in practice test sets sufficient to (almost) satisfy all-uses were frequently also sufficient to (almost) satisfy all-du-paths.

Figure 3-2. Complexity of Data Flow Testing

effectiveness of several path selection and test data selection techniques. Of particular interest is Howden's evaluation of path analysis since the reliability of path testing places an upper bound on the reliability of all techniques that are based on testing of a subset of a programs' paths. The conclusion given in [Howd76c] is: "If it were possible to test every path in a program, then path testing is found to be reliable or almost reliable for about 65% of the [faults] found in the small survey of 11 [small, sequential] programs in Kernighan and Plauger [Kern74a]."

In addition to being used as test data generation strategies, these path selection techniques can be used as coverage measures. In this role, the code is instrumented to monitor the different control or data elements that are executed in the course of testing. The adequacy of the completed testing is then measured as a function of the percentage of the structural units executed.

3.1.2 Error-Based Techniques

Much recent research has been directed at the notions of reliable and adequate test data first introduced by Goodenough and Gerhart in 1975. Their theorem, called the "fundamental theorem of testing," characterized the properties of a completely effective test data selection strategy based on definitions of reliability, validity, and completeness. Essentially, a test data selection strategy is said to be *reliable* if it guarantees to generate test data capable of detecting every fault in a program [Good75a]. This work was the first formal, systematic approach to a technology which had previously been characterized by its dependence on the software developer's intuition. It has been one of the major influences on direction and scope of later work.

Although Goodenough and Gerhart provided insight into how to develop effective program tests, they did not develop an actual testing approach. Weyuker and Ostrand modified and refined Goodenough and Gerhart's properties for ideal tests to investigate a testing strategy that combines consideration of likely potential faults with more traditional path selection and functional testing approaches [Weyu80c]. The program's input domain is partitioned based on program-independent and structural properties of the program, as well as potential faults that have been identified as likely for the problem being solved. This results in the identification of *revealing subdomains*. The key property of a revealing subdomain is that the existence of one element of the subdomain which leads to incorrect processing when used as an input implies that all of the domain's elements are processed incorrectly. Equivalently, if any input is processed correctly, then all inputs are processed correctly. Selection of test data, therefore, is reduced to choosing an arbitrary element from each subdomain. This is sufficient to show the absence, or presence, of the particular types of faults being considered. Although this strategy is largely a research vehicle and has not been developed to the state of a practical testing technique, it remains interesting since it demonstrates the goals to which many researchers have been working.

There is still no general theory that states whether a piece of test data protects all of the program executions along a particular path from all kinds of faults. Even so, several testing techniques have been developed that achieve test data adequacy for certain limited, well-defined types of faults. Indeed, some of these techniques reliably demonstrate the absence of prespecified types of faults.

A commonly used classification is to distinguish program faults as either computation, domain, or missing path faults. (This classification was first introduced and analyzed by Howden in [Howd76c].) *Computation faults* result from incorrect operations performed along a correct execution path, such as missing or inappropriate assignment statements. *Domain faults* result from incorrect path traversals that occur due to path selection faults. Finally, *missing path faults* occur when some special case requires a unique sequence of actions, but the program does not contain a path whose execution will cause that sequence of actions. Following this distinction, Clarke and Richardson have developed a strategy for

selecting test data sensitive to particular computation faults based on analyzing a symbolic representation of the path computation [Clar83b].

Howden and Zeil have proposed additional computation testing approaches based on the use of algebraic techniques for defining a neighborhood of functions. Howden's algebraic testing [Howd78b] establishes rules for choosing data to differentiate among all members of a functional class, and then applies those rules to any program whose output is expected to fall within that class. Zeil's perturbation testing [Zeil83a], developed for testing numeric code, involves the derivation of those members of a chosen functional class which are indistinguishable from the program function using all the test data so far applied. The key idea is to add a perturbing function to expressions occurring in the software and derive the conditions under which that fault could go undetected by a given test path. In this way, perturbation testing is really a path selection adequacy method, though it can also be used to generate test data to reveal faults in arithmetic expressions.

In a system called EQUATE, Zeil merges perturbation testing and mutation testing (see Section 3.1.3) to provide another technique capable of finding faults in the execution of a program path. Primary goals are to remove the limitation of perturbation testing to numeric domains and the limitation of mutation testing to detection of simple faults. An additional goal is to overcome the decrease in effectiveness suffered by both methods for programs employing high-levels of data and functional abstraction. EQUATE selects a number of test locations throughout the program and chooses a set of expressions derived from the abstract syntax tree of the module being tested. Test data is required that distinguishes each pair of these expressions from one another at every test location [Zeil86]. Zeil describes a set of designated expressions and constants, called *terms*, that are formed from the union of the following three subsets:

1. The set of all expressions and subexpressions from the abstract syntax tree of the module under test, called the *expression set* of the module.
2. The set of values first taken on by each expression set term at each test location during testing, called the *initial value set*.
3. The set of expressions that can be formed by substituting any member of the expression set for any subexpression of another expression set member, called *operand substitution terms*.

Test locations occur at the beginning of each basic block and immediately following each statement in the block.

Cohen and White have developed a technique called domain testing that guarantees, within a given error bound, to detect path selection faults. This technique guides the selection of test data by geometric analysis of path domain boundaries (where the boundary of a path domain is determined by the conditional branches that are taken along the path). It generates test points on and near each boundary that can detect whether a domain error has occurred. If so, one or more of the boundaries will have shifted, or the corresponding predicate relational operator will have changed. Otherwise, if the program yields correct results for the chosen test data, the path domain boundary is correct within the error bound. Cohen and White proposed the first domain test data selection strategies and error bound criteria [Whit78b], and later strategies which reduce the displaced domain associated with an undetectable border and offer lower complexity [Whit86]. One of the serious limitations of domain testing is the need to examine the potentially infinite number of domains that arise from iterated loops. Recent work has focused on reducing this burden [Whit88a].

Partition analysis testing [Rich85a] integrates several testing strategies, such as domain, computation,

and algebraic testing and a form of formal verification. It compares a program to its formal specification and so is one of the few testing approaches that can detect missing path faults. The overall strategy requires partitioning the input domain into procedure subdomains so that the elements of each subdomain are treated uniformly by the specification and processed uniformly by the implementation. Partition verification, a variation on symbolic testing (see Section 4.1), is performed to demonstrate the consistency between the specification and its implementation. This verification is enhanced by partition testing. Partition testing uses information related to each subdomain to guide the selection of test data which, in executing the program, helps to determine whether the program conforms to its specification.

3.1.3 Fault-Based Techniques

Fault-based techniques differ from error-based techniques in that they examine program statements to demonstrate the absence of a *predefined* set of faults.

Morell [More88] describes fault-based testing as a three stage process operating in an arena consisting of (1) a specification, (2) a program, and (3) the domain of interest which is the source of test data, together with a prescribed list of potential faults, called alternatives. The first stage requires identifying the locations in the program where the alternatives might lie. Then a test set is developed which executes these locations, yet yields correct output. Finally, information collected during the execution is used to deduce that no alternative could have been inserted into the program without being detected by the test. He goes on to provide a model of fault-based testing which can be used to investigate the theoretical limitations of this group of techniques. Two orthogonal attributes are used to categorize fault-based testing techniques. The *breadth* of a technique is given by number of potential faults considered, it may be finite or infinite. Whereas the *extent* relates to the information used to determine the absence of faults and may be local or global.

While all fault-based approaches support test data generation, they are primarily test data adequacy measurement techniques. One of the earliest and perhaps the best known is mutation testing, also called mutation analysis [Budd80a]. Mutation testing requires the definition of a set of mutation transformations, called error operators, which are applied singly to the elementary components of a program to introduce certain types of simple errors as faults. Test data that can distinguish between the original and mutated programs is then deemed adequate for detection of that particular type of error. The ability of these simple errors to cover more complex errors is derived from the "Coupling Effect." The restriction of introducing single errors is justified by an empirical principle called the "Competent Programmer Hypothesis." These two assumptions are defined as:

> Competent Programmer Hypothesis – The assumption that the program to be tested has been written by a competent programmer [DeMi88a].
>
> Coupling Effect – Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors [DeMi78].

Error operators exist to demonstrate the absence of common Fortran errors, and a set of error operators appropriate to Ada programs is under development [Appe88]. Since the set of error operators is limited and mutants are distinguished from the original program based on the program output, mutation testing has finite breadth and global extent. While an effective technique, it is expensive. The need to apply error operators individually can lead to the generation of hundreds of mutated programs, each of which must be separately compiled and executed.

There are several variations of mutation testing which attempt to overcome some of the weaknesses of the tec nique in its original form. Whereas the original strategy (*strong mutation testing*) applies to a

program as a whole, *weak mutation testing* [Howd82a,Howd87] applies to program components, usually elementary computational structures. Since it is unnecessary to perform a separate compilation and execution for each mutation, weak mutation testing is cheaper to apply, although it cannot guarantee detection of faults in the function computed by a program.

Weak mutation testing was partially derived from Error Sensitive Test Case Analysis (ESTCA) [Fost80]. This approach attempts to detect all common types of faults. As such, it was the earliest example of an infinite breadth fault-based technique. Adapted from a hardware failure analysis technique, ESTCA was developed by simulating frequently occurring code faults, identifying the most effective test patterns for detecting these faults, and then deducing rules for an algorithm to generate fault sensitive test data. The test data and expected outputs are derived from examination of a program's specification. The program is then executed on the test data and the actual results compared against the expected results to identify failures.

Trace mutation testing [Howd82a] uses program traces to compare the results of a program and its mutations rather than output values, thus allowing several mutation transformations to applied concurrently. The most recent variation, *firm mutation testing* [Wood88], claims to combine the best elements of strong and weak mutation testing. It uses components with more extensive scope than weak mutation testing, and permits partial execution of components so that many mutants can be applied in a single execution.

A form of mutation testing that allows validating a program against its specification has also been developed [Budd85]. Specifications are given in the predicate calculus, modified to clearly indicate the input-output relationships of a program. These specifications are mutated by adding logical clauses to the input and output conditions. Contrary to other forms of mutation testing, the goal here is to generate test cases which satisfy the additional constraints so that the original and mutated specifications produce the same result. First the program and the original specification are executed on the test cases. If the results from these executions indicate consistency between the specification and its implementation, the mutated specifications are executed with the same test data. A mutation is eliminated when an input clause is satisfied, but one or more output clauses are unsatisfied. Unlike the forms of mutation testing which only consider a program implementation, this method can identify missing path faults in a program.

Although most error- and fault-based approaches are designed to activate faults, few ensure that the effects of faults are propagated through the program to be revealed as failures in the output. In order to overcome this problem, much current work is investigating models for fault propagation, yielding fault-based techniques with both infinite breadth and global extent.

Morell's approach utilizes symbolic execution for a dynamic model of fault activation and propagation. This is applied to mutation testing to overcome the limitation requiring individual generation and execution of mutants. The fundamental concept underlying symbolic execution is the ability to model infinitely many *executions* of a program with test data by a single symbolic execution. Morell modifies this concept to executing infinitely many mutation *alternatives* in an execution. To use Morell's words: "For symbolic execution the key lies in encoding infinitely many inputs by a single *symbolic input*. For symbolic testing the key lies in encoding infinitely many alternatives in a single *symbolic alternative*" [More88]. The modified form of execution proceeds as usual until the expression containing the symbolic alternative is reached. Then, instead of creating a symbolic value for an input, a symbolic value simulating the result of the mutation transformations represented by the symbolic alternative is generated. This value is propagated through the program until it appears as an output embodying all the possible impacts of the alternatives. This intuitively appealing approach has some drawbacks relating to undecidability problems in program control flow. Morell is studying alternative definitions of a program path to resolve these problems.

RELAY [Rich86a,Rich88] uses static analysis of the program to analyze fault activation and propagation. It defines revealing conditions which guarantee that a possible fault is activated during execution and that the fault effect transfers through computations and data flow until it is revealed. RELAY is applied by choosing a fault classification and determining the origination conditions and transfer conditions for each class of faults. These conditions are then evaluated for the program being tested to provide the revealing conditions. Thus test data selection is treated in conjunction with path selection by selecting test data that not only originates a fault but also transfers that fault to affect the output, guaranteeing the detection of any fault of the chosen classes.

Although RELAY can be used for both test data selection and test data adequacy measurement, it was primarily developed to serve a different goal. Researchers at the University of Massachusetts are undertaking a long-term effort to evaluate test data selection techniques. One of the difficulties they have encountered is that most existing techniques use different underlying models, making comparison between techniques difficult. Therefore, RELAY was developed as a consistent model for software testing which is expressive enough that most current testing techniques can be mapped on to it, although it is particularly suited to those that are fault-based. RELAY has been used to analyze three fault-based criteria: Budd's Estimate, Weak Mutation Testing, and ESTCA. This analysis demonstrates that none of these criteria guarantees the detection of faults. Richardson and Thompson discuss two common weaknesses: "First, the criteria do not thoroughly consider the potential unsatisfiability of their rules; each criterion includes rules that are sufficient to reveal [faults] for some fault classes, yet when such rules are unsatisfiable, many [faults] may remain undetected. Second, the criteria fail to integrate their rules; although a criterion may cause an expression to take on an erroneous value, there is no effort made to guarantee that the enclosing expressions evaluate incorrectly" [Rich86a]. These weaknesses are by no means exclusive to the aforementioned techniques. As yet no effective rules for showing how to transfer faults out of loops and conditionals have been developed.

Richardson and Thompson, the developers of RELAY, are currently investigating the use of the RELAY model for integration and specification-based testing. They are also enhancing the capabilities of RELAY with regard to data flow transfer, in order to facilitate processing of loops and modeling the transfer of multiple faults. Other researchers, see [Long88], are attempting to extend the RELAY model for application in fault-based testing of concurrent, real-time software.

There are three necessary and sufficient conditions for fault-based testing to ensure that a program is correct with respect to its specification [More88]:

1. The fault-based arena must be alternate-sufficient. That is, either the original program or one of the alternate programs must be correct. Since there is no algorithm to determine alternate-sufficiency, the fault-based arena must be assumed alternate-sufficient until proven otherwise.
2. Coupling does not occur for the test set. That is, each alternate can be independently detected by a test set, along with combinations of alternatives that apply. Again, it has been shown in [More84] that no algorithm for determining coupling exists.
3. Coincidental correctness, where a fault on an executed path does not produce erroneous results, does not occur.

These conditions are undecidable. Therefore fault-based testing cannot guarantee that a program is correct. It does, however, offer valuable information on the absence of certain faults. Since software reliability is directly related to the presence or absence of faults, this information could be exploited in

software reliability measurement. Morell is investigating the use of his model of fault-based testing as the basis for a white-box model of software reliability.

3.1.4 Functional Techniques

The majority of the previously discussed white-box testing approaches consider only the program implementation. By ignoring its specification, these methods are limited to testing what the program does, rather than what it is intended to do. Functional testing techniques do not consider the internal structure of programs. Instead, generation of test data (and expected results) relies on the specification of program function, inputs, and outputs. In addition to revealing faults in programs, these black-box techniques usually have the desirable side-effect of detecting ambiguities and incompleteness in program specifications.

Howden has recently shown how different testing and analysis methods relate to an expanded interpretation of functional testing, in which a system is viewed as a structure of related functions. This approach can also be categorized as an error-based approach since it looks directly at how programmers make errors, as opposed to the possible fault effects of those errors. He provides a model of how software is constructed and of the reasoning errors that humans make during this process. From the premise that software is developed by synthesizing functions, Howden discusses two views of programs. These are: (1) hierarchical top-down functional structures, and (2) horizontal state transition diagrams. Completeness of testing is then achieved by identifying the fault effects of possible errors and constructing methods for detecting these faults. This work is reported in his book, *Functional Program Testing and Analysis* [Howd87]. Since completing the book, Howden has investigated a more general error model. This new model takes the view that abstraction and decomposition are the major tools for reasoning about complex objects, and different testing and evaluation techniques are suitable for the different kinds of errors which may be made during these processes. Although not yet completed, this later work relates the different forms of dynamic and static analysis and addresses all software products, not just code.

The earliest functional strategies were equivalence partitioning [Myer79], boundary-value analysis [Myer79], and cause-effect graphing [Elme73]. Recognizing that exhaustive testing is rarely feasible, these three strategies address the problem of trying to select the subset of all possible inputs that have the highest probability of finding the most faults.

In equivalence partitioning the input domain of a program is partitioned into a finite number of equivalence classes where, it is assumed, a test of a representative value of each class is equivalent to a test of any other value. This implies that if one test case in an equivalence class detects a fault, all other test cases would be expected to find the same fault. The minimal set of test cases covering all equivalence classes is then developed by selecting test cases that invoke as many different input conditions as possible. Boundary value analysis differs from equivalence partitioning by selecting values in both input and output equivalence classes that test the edges of each class. These values are often called extremal/special values, where *extremal* values are those that lie on the edges of variable domains and *special* values are those with special mathematical properties. Since few other testing techniques provide good coverage of extremal/special values, this approach is frequently included in other techniques. Cause-effect graphing offers some advantages over the other two techniques by exploring combinations of input conditions. Here, a program's output domain is partitioned into *effect* classes, which also results in partitioning the input domain into *causes* that correspond to particular effects. The different classes, and links between them, are expressed in a combinatorial logic network called a cause-effect graph. The dependencies thus revealed are used to derive appropriate test cases.

Another technique in this group is random testing. This technique is based on the idea of testing a program by sampling for faults, and a measure of the program's correctness is given by the proportion of

elements in the input domain for which it fails to execute correctly. It not only can uncover faults but provides a reliability measure for the program. In this latter function, the number of failures in a set of test cases is related to the correctness measure via a probability distribution function that depends on the way test data is chosen. Test cases are randomly generated from either a uniform distribution of a program's input domain or from its operational profile. Once the input domain or operational profile is determined, generation of random test data is usually much easier and cheaper than other test data generation strategies. Poor selection of the range can, however, lead to wasteful generation and execution of test data, and inconclusive results. Also accurate identification of the input domain or operational profile depends on the ability to predict the actual operating environment. This is not always possible.

Statistical testing is a variation on random testing. In this case, the emphasis is on certifying the operational effectiveness of the software, with an estimate of product reliability being given in terms of the mean time to failure (MTTF) [Dyer85a]. Fault detection plays a secondary role. Test data is selected based on the anticipated statistical distribution of the operational data to provide a realistic simulation of the product environment. Randomized sampling techniques are used to introduce efficiencies in the size and content of the samples. This approach provides a technical basis for making statistical inferences about operational effectiveness based on test results. These results can be extrapolated to provide operational reliability estimates.

While the value of random testing for detecting faults with a low probability of occurrence is uncertain, recent, small-scale empirical experiments suggest that it potentially has a valuable role to play as one element in an integrated testing strategy. For example, Duran and Ntafos have conducted several experiments which indicate that random testing gives high levels of structural and fault-based coverage adequacy measures. In the first case, these researchers report: "... of the 5 programs tested, a moderate number of random test cases on the average achieved 97% of segment testing, 93% of branch testing, 57% of structured path testing, 72% of required pairs testing [Ntaf81a], 81% of [LCSAJ measure] TER3, and 74% of [LCSAJ measure] TER4" [Dura84]. In the second case: "Seven programs were tested, using the mutation system at Georgia Tech, with from 8 to 20 random test cases. 79% of the mutants were eliminated by the test cases as compared with 84% for branch testing and 90% for required pairs testing (the number of remaining mutants includes those that are equivalent to the original program" [Ntaf81a]. Additional experimentation to substantiate these findings is needed.

The advent of formal specification languages has given rise to new types of functional testing where a program specification is actively used to support the testing process, usually through automatic generation of implementation-independent test data. Although many examples exist (see [Gogu79a,Muss79,Gorl87]), this subsection focuses on three techniques which illustrate the ongoing work in this area.

Grammar-based testing techniques rely on the use of test grammars to generate program inputs and expected outputs. The test grammar is developed from a formal specification of the program and provides a separate partial specification that can be compared against the original specification to uncover ambiguities. During implementation, the partial specification and the implementation are executed on identical test data and the results compared to identify faults. One particular strategy, using attribute context-free grammars [Dunc81], allows random generation of test cases or the application of testing heuristics (for example, boundary value analysis, or ESTCA testing) for more systematic test case generation.

The other two approaches use algebraic specifications. An algebraic specification is made up of a list of functions on a set of *sorts* (types) and a set of axioms defining properties of the defined functions. The first of these approaches uses algebraic axioms of abstract data types to aid the testing process. The second uses logic programming to generate test data sets from algebraic data type specifications.

The Data-Abstraction Implementation, Specification, and Testing System (DAISTS) [Gann81]

provides a language for formal expression of the semantics of data abstractions, independent of their implementation. This allows a program implementing abstract data types to be annotated with nonprocedural, algebraic axioms and test cases to facilitate mechanical consistency checks between the axioms and the implementation. Although the same software developer may write both axioms and implementation, the orthogonal nature of these two representation forms reduces the likelihood of the same faults occurring in each. DAISTS is a compiler-based system. The compiler prepares a "program" which utilizes the axioms as a test driver for the implementation. The software developer provides test points in the form of expressions using the abstract functions. These are fed to the program which then cycles through the test data, monitoring the execution of both axioms and implementation to determine if they agree. There are two significant benefits to this approach. First, the software developer only supplies test inputs, an oracle is not needed. Second, while the developer provides the implementation, axioms, and test points, DAISTS automates the testing by developing the necessary test drivers.

Choquet and colleagues have developed an alternative approach for generating test data from algebraic data type specifications. This approach arose from the noted similarity between an algebraic specification and a logic program (both of which describe logical properties), and exploits logic programming for test data generation. It is based on a formalized theory of testing. The basic assumption for test construction is the *Correlation Principle* which states: "There exists a narrow correlation between specification structure and implementation structure" [Boug86]. It requires hypotheses relating to the concepts of higher- and lower-level sorts occurring in hierarchical specifications. In [Choq86], these hypotheses are given as:

1. The *regularity hypothesis* for a level *n* states that if the test is successful for data of complexity less than *n* [where a level of complexity is associated with each member of an input subdomain], then the program behaves correctly for any value.
2. The *uniformity hypothesis* states that if the test is successful for one datum in a subdomain then the program behaves correctly for any data in this subdomain.

It is also required for specifications to be translated or considered as logic programs and the search strategy to be controlled by a logic interpreter.

As with DAISTS, testing consists of verifying that an implementation satisfies each axiom of the specification. In Choquet's approach, however, the specification of a function is used to produce input data for that function and predict the expected result of the function. Test data sets are generated for each axiom and collected together to form one test data set for the whole specification. Early test data generation algorithms produced extensive data sets due to problems in applying the hypotheses. These have been replaced by a more efficient procedure utilizing constraints on variables to delimit uniformity subdomains [Choq86]. This improves the implementation of the uniformity hypothesis and so simplifies the test data generated. A prototype tool implemented in Quintus Prolog is under testing at the Universite de Paris-Sud.

3.2 Techniques for Dynamic Analysis of Concurrent and Real-Time Programs

As previously stated, testing of concurrent programs invariably starts with testing each task as an independent unit using sequential program testing techniques. The tasks are then tested jointly, primarily to examine their synchronization behavior. There are, as yet, few disciplined techniques for this latter form of testing. Moreover, since synchronization patterns are partially determined by a scheduler at runtime, and are thus sensitive to timing, dynamic techniques may not be able to detect all existing faults.

Researchers at the University of California (Irvine) have developed algorithms for applying structural testing, based on control and data flow, to concurrent programs [Tayl86a]. In addition to detecting data flow anomalies, these algorithms can detect synchronization faults such as waiting for an unscheduled process, waiting for a process guaranteed to have already terminated, and scheduling a process in parallel with itself. As with sequential programs, structural testing of concurrent programs can be used as both a test data generation strategy and to measure the adequacy of completed testing. In this latter role, data on concurrency state coverage, state transition coverage, and synchronization coverage for concurrent Ada programs is collected and correlated over a set of test runs. Information on concurrency states and concurrency histories is used, in conjunction with symbolic evaluation techniques, to guide test data generation.

Recognizing that concurrency-related faults have historically proven very difficult to test for, researchers at Stanford University have developed an alternative approach. They propose a type of self-checking Ada program which provides run-time detection and recovery of faults. This approach exploits two annotation languages which are used to include assertions specifying correct behavior in the program code. ANNotated Ada (ANNA) [Luck84a] provides assertions on statements, variables, and program units. Whereas the Task Sequencing Language (TSL) [Luck87] is used to specify constraints to be satisfied by the sequences of tasking events occurring in the execution of the program. The annotations given in these languages are automatically transformed into run-time checks that monitor the consistency of the behavior of the program with its formal specification. (In the case of ANNA, optimize checks and a limited variety of proof techniques to analyze the consistency of checks prior to run-time are also available.) The run-time checking against specifications can be executed in parallel with the underlying Ada, reducing the overhead involved and allowing the checks to be a permanent part of the Ada program.

The current version of TSL, TSL-1, can be used to aid the design of Ada tasking programs, but is primarily intended to support testing and debugging. Constructs for defining abstract tasks will be added to TSL-1 to develop a new language, TSL-2, suitable for specifying distributed systems prior to their implementation.

3.3 Specific Testing-Related Automation Issues

Although the majority of dynamic analysis techniques are supported by automated tools, these tools are largely research vehicles or prototypes. The lack of widely available, production quality tools is one of the factors contributing to the lag in transitioning state-of-the-art techniques into common practice. While this deficiency must be rectified, development of individual, stand-alone tools is not the solution. It is unlikely that a single testing technique will ever be sufficient to guarantee reliable software. Current evidence indicates that a simple succession of techniques, where each is applied independently of its predecessors and successors, is neither effective nor efficient. Comprehensive testing environments which support *integrated* application of a variety of techniques are critically needed. In this context, integrated means that each technique builds on (partial) information gathered by previous techniques and may itself provide information to be used by later techniques. On-going efforts investigating the development of environments which support the cooperative application of dynamic and static techniques are discussed in Section 4.6.

There are, however, a number of automation issues which are specific to dynamic analysis. These are discussed below.

3.3.1 Coverage Analyzers

Coverage analyzers, also called coverage monitors, record information about the structural elements executed during a program run. Coverage analyzers for sequential software have been available for many years. Those for concurrent and real-time software, however, lie in the realm of research rather than practice. The significant problem that occurs in the case of concurrent and real-time software is that the coverage must reflect all possible timing relations, of which there may be an infinite number.

Research into coverage analyzers for concurrent software is following two different approaches. Program transformation techniques add logic to a concurrent program to record pertinent information about its execution. Alternatively, specialized run-time systems can be developed by modifying the run-time scheduler to directly record the requisite information. Work in the area of program transformation has been pioneered by German, Luckham, Helmbold, [Germ82a] and Rosenblum [Rose85b]. They have developed techniques for transforming a concurrent Ada program into an equivalent program that exhibits the same behavior but also records the state transitions through which the program progresses. The drawback of this type of approach is the increased overhead which arises from extra entry calls and the monitor task used to record all tasking activities. This overhead is generally unacceptable for real-time software where any instrumentation of the code can significantly alter the timing characteristics and, hence, behavior of the program. One possible solution for this problem is to use a separate processor to monitor and record information about the program execution.

3.3.2 Debuggers

Debuggers aid in localizing software faults by allowing the software developer to stop the execution of a program at interesting points and examine the value of such items as program counters and variables. An interactive debugger allows the developer to step through a program, repeatedly stopping and restarting the execution, and potentially setting and clearing monitors as necessary. As with coverage analyzers, many debuggers are available for sequential programs. There is, however, a lack of quantitative information about the capabilities and relative values of these tools that must be resolved before specific tools can be recommended for widespread practice.

The indeterminacy of events in concurrent programs poses a significant problem for debugging. The behavior of the program is potentially affected by factors out of the software developer's control. Consequently, it may be extremely difficult to reproduce the circumstances that led to a failure and diagnose its cause. Moreover, the debugger itself may interfere with the execution time and other run-time characteristics (such as paging behavior), resulting in modified patterns of interaction between the processes being executed. Collecting information which is useful in diagnosing faults is also non-trivial, since a failure may not actually occur until the program execution is severely corrupted. In consequence, debuggers for concurrent software must be more closely intertwined with testing activities than their sequential counterparts.

Researchers distinguish between two types of concurrency; namely, multiple processes executing on a single processor or concurrent processes executing on several processors. In the first case, a software developer can look at the order in which events occur to determine the causes of failures, and can stop all processes at the same instant when necessary. An example of work in this area is that being conducted by Helmbold and Luckham at Stanford University [Helm83]. These researchers differentiate between possible types of Ada tasking faults. *Task sequencing faults* occur when a program's tasks interact in a different order than anticipated. *Deadness faults* occur when a task communication failure prevents part of a concurrent computation from proceeding. Whereas most deadness faults can be detected by analysis of the computation of a program, task sequencing faults often require additional information pertaining to the

intended behavior of the program. Helmbold and Luckham have developed a prototype tool which detects and analyzes deadness faults. Program transformation techniques are used to modify each task to inform a special monitor task about tasking actions about to be performed. Based on this information, the monitor maintains a continually updated *picture* about the program's tasking state, which is checked for deadness faults whenever new information is received. To aid in debugging, the monitor can print snapshots of the tasking picture and trace task interactions. The interactive interface to the tool allows the software developer to *single-step* through task interactions and request diagnostic information as needed.

Future extensions to the TSL monitor will provide the capability to monitor user specified assertions about task events and so facilitate detection of task sequencing faults [Helm85]. This is part of an overall effort to develop a testbed for specific kinds of Ada tasking programs, such as run-time schedulers for Ada tasking on multiple processors. This effort includes the development of a monitor/debugger for TSL on multiprocessors systems (Sequent Symmetry), and the definition of debugging techniques utilizing TSL specifications and the TSL monitor.

In the second case, where processes execute on multiple processors, the order in which events occur can only be determined by looking at the communications between the processes. These communications must be disabled to suspend all processes at the same *relative* time (there is a communication delay problem which makes it impossible to suspend processes at the same *actual* time). A common approach for this class of debuggers is to use a hierarchy of tools consisting of a traditional debugger to aid in testing each process independently and a higher-level debugger to monitor communications. A review of debuggers for concurrent programs executing on different processors is given in [Gord88].

In [Gord88], Gordon and Finkel also describe their tool TAP which is implemented on the Charlotte [Fink83] *distributed operating system*. TAP is used to detect timing faults which are caused by misordering of the communication events between processes. It maintains a history of the communication events that occur during software execution as a directed, acyclic graph, called a timing graph. The nodes in a timing graph represent events, whereas arcs from one node to another indicate the temporal precedence between events. TAP is designed to be always active, thus overcoming the problem of the debugger changing the behavior of the program being monitored and potentially masking faults. Using TAP, a skeleton timing graph is continually built during the program execution. When a fault is encountered, TAP suspends all processes, constructs the full timing graph, and waits for instructions from the user. It then aids in diagnosing the cause of the fault by allowing the user to backtrack through the timing graph to examine the order in which events occurred and the contents of messages. Since, unlike most debuggers of this class, TAP can analyze communication events after the fact and does not require active control over the execution, it can be used both during program testing and to find timing faults in operational software.

The worst case debugging problem occurs when concurrent processes must operate under real-time constraints on a bare machine. Here the common practice is to develop and test the software on a host machine where extensive development support is available and then perform limited testing on the target machine. The target may provide no support for debugging activities. If a failure occurs during testing on the target, it is often necessary to return to the host to isolate the cause. Even if the same scheduler algorithm is used on both machines, different behaviors may occur due to differences in processor construction. Examples of factors which cause the target execution to deviate from the host execution include: a less (or more) precise real-time clock, real-time input simulators on the host that operate at a different rate to the actual inputs to the target, and variations in the relative speed of the processors. Taylor, at the University of California (Irvine), is investigating techniques to allow reconstruction of an erroneous target execution at the source language level (Ada) on a host machine [Tayl82b]. Although prototype tools are being developed, it will likely be several years before practical solutions to this problem are available.

3.3.3 Built-In Test

Although nearer to the hardware aspect of instrumentation, the issue of built-in-test (BIT) [Batt87] must also be considered. Beginning in the late 1960's, troubleshooting of electronic components embedded in weapon systems became too complex for traditional, manual approaches. As a result, diagnostic probes began to be built into electronic components to allow easier detection of faults. These probes, or diagnostics, must be designed into electronic components; they cannot be added as an afterthought. Most DOD embedded weapon systems are designed with BIT as an operational requirement.

In a similar vein, a separate autonomous machine can be used to monitor system execution. Although potentially expensive, this approach, theoretically, does not affect timing characteristics and can provide "playback" for reproducing specific execution sequences. It can also be used to limit the execution overhead incurred for self-checking code, or the dynamic monitoring provided by tools such as TAP. Support for permanent self-test of concurrent software by downloading the checking of TSL specifications onto spare processors is under investigation at Stanford University.

3.4 Summary of Major Gaps in Dynamic Analysis Technology

So far this section has outlined the dynamic analysis technology that has been, or is being, developed. Not all of this technology is currently available for use, or would be suitable for SDS software efforts were it available. But what technology is needed? Other sections of this report raise the question of how all the different forms of testing and evaluation should fit together, and how they should cooperate with developmental activities. Here, a number of technology gaps that are particular to dynamic analysis and independent of these larger issues can be identified. By and large, these gaps are not simple, independent problems; the state-of-the-art is not yet at this point. Instead, these gaps reflect some of the fundamental deficiencies in software testing and evaluation.

3.4.1 Need for Oracles

Most dynamic analysis techniques require some means for determining whether the program output is correct. Conceptually, at least, this implies the need for an *oracle* which can produce "correct" programs outputs against which actual outputs can be compared. In practice, humans usually play the part of the oracle, although this can result in high testing costs and is often inaccurate for all but small programs.

Some techniques resolve the lack of an oracle by creating formal, executable specifications to generate the expected output. When these derive specifications from the code itself, however, there is at least the possibility of mirroring erroneous program assumptions in the created specification. Some researchers have proposed using N-version programming to deduce correct outputs based on the consensus of the N-versions of a program. Unfortunately, recent research indicates some problems with the fundamental N-version hypothesis that requires statistical independence between failures in the N-versions of a program (see Section 2.1.3).

It is probable that no adequate solution of this problem will be achieved until increased formalization of early lifecycle activities yields sufficiently formal specifications (embodying, for example, first-order predicate calculus) to define software processes. Even then, the difficulty in defining *all* correct sequences of events poses problems for concurrent and real-time software.

3.4.2 Completeness of Analysis

What is the meaning of a successful test? A good test is one which exposes faults. The meaning of a test in which the software executes successfully is uncertain; it does not necessarily indicate the absence of faults. This is part of the larger question of *completeness* or, in other words, knowing what has been achieved in testing activities. Although completeness in general applies to the overall testing and evaluation activity, it is particularly troubling for dynamic analysis techniques. Unlike many other forms of testing and evaluation, these techniques are not simply applied or not applied; they can be applied to widely varying extents.

Much research is needed before this issue can be fully understood. Meanwhile, a *very simplistic* solution is to tie the notion of completeness to the concept of minimum levels of testing coverage. (This is an extremely limited translation of completeness and is proposed only as a stop-gap measure for practical purposes; that is, it provides a step forward but falls a long way short of the ideal.) Wide acceptance of a minimum level of testing coverage for all software systems has long been sought. This is starting to be achieved as industry increasingly accepts branch testing as the minimum coverage requirement. In the case of SDS, however, branch testing is an inadequate across-the-board minimum. A hierarchy of minimum coverage requirements must be determined such that increasingly severe coverage requirements can be mapped against increasingly critical software. These coverage measures must address different aspects of dynamic analysis, such as the coverage required for structural and data flow testing, fault-based testing adequacy measures, and functional testing coverage.

3.4.3 Assessment of Capabilities of Techniques

The last several years have seen a relative increase in experimental evaluations of dynamic analysis techniques. For illustrative purposes, the results from two of these evaluations are reproduced in Figure 3-3 and Table 3-2. Experiments such as these not only provide empirical evidence of the utility of particular techniques for detecting certain types of errors or faults, but some general guidance on how, or which, techniques can support each other for more thorough analysis. While some of these experiments were admittedly conducted on small, sample programs, others utilized realistic programs intended for practical use. The chief handicaps in comparing the results across experiments remain: (1) the continuing lack of a standardized error/fault categorization scheme, and (2) the limited volume of experiments.

This work notwithstanding, practical descriptions of the error/fault detection capabilities and costs of existing techniques are still unavailable. In a few cases, such as data flow testing and domain testing, simple cost information pertaining to the number of test cases needed to achieve various levels of testing coverage is available. This must be supported by data on the cost of generating and executing test cases which, of course, is impacted by the available automated support tools and the underlying operating environment. The cost to analyze test results can also be a pertinent factor. Even so, this information is only meaningful in terms of the increased reliability these costs deliver. Since testing is a time consuming and expensive task, it is important that "the point of diminishing returns" can be recognized in a timely manner, as this applies to the case in hand.

Information on capabilities should specify how the performance of a technique varies for different types of programs. For example, the use of certain control and data structures can severely degrade the cost-effectiveness of certain techniques. Some techniques simply do not handle array references; arrays are treated as a single variable and array elements are not differentiated. A series of weightings that distinguish between programs in terms of their testing difficulty is needed. Assessment of a technique's effectiveness must also take into account any fundamental limitations or assumptions. For example, coincidental correctness is a limiting assumption of many techniques. *Coincidental correctness* occurs when a

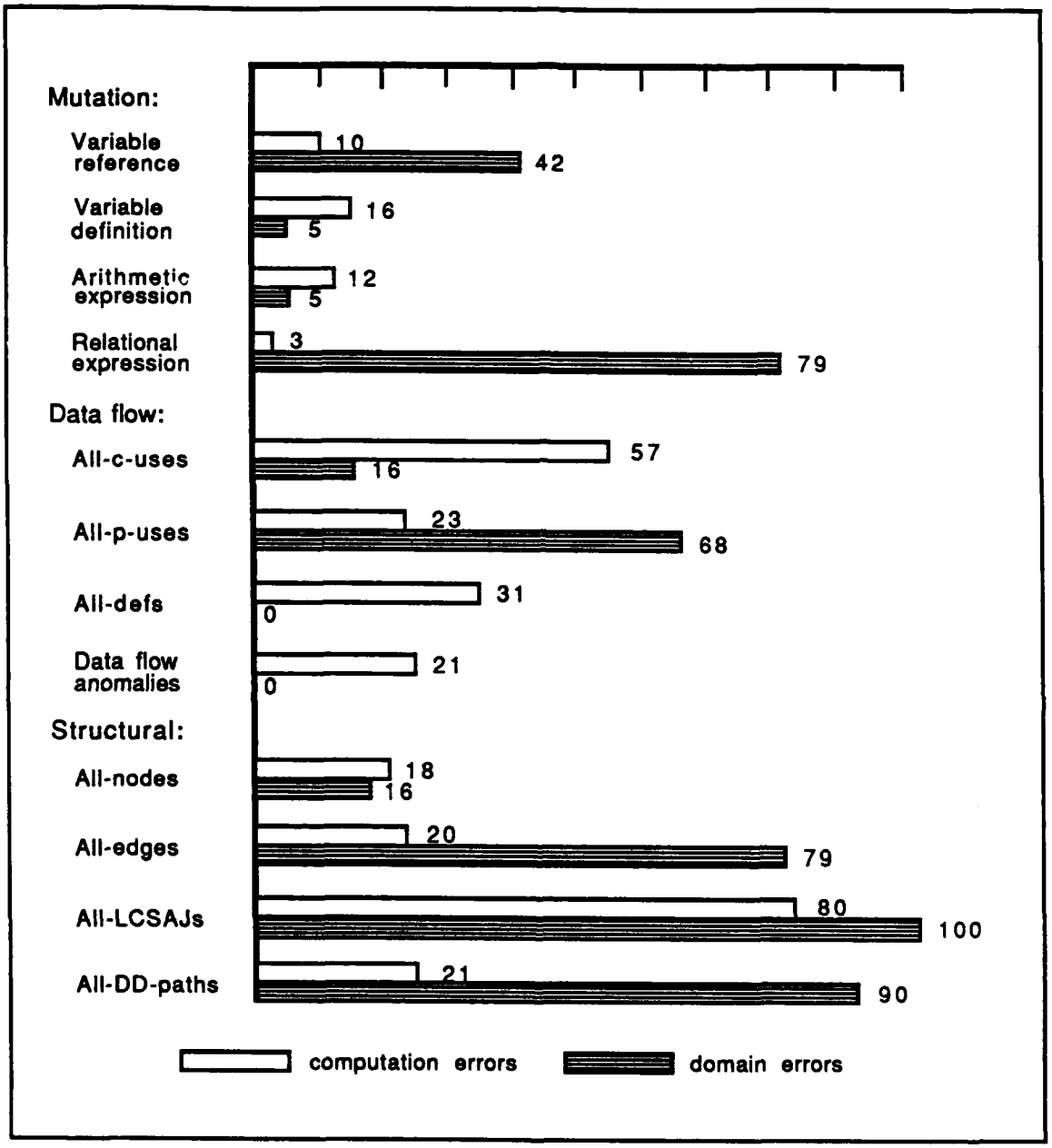

Figure 3-3. Frequency of Errors Detected

fault is executed but the program does not fail, thus an incorrect program may be falsely assumed correct. Determining the absence (or presence) of coincidental correctness is generally undecidable, but some new analysis techniques are beginning to address how to ensure that the effect of a triggered fault is indeed transferred to the output. The relative costs and benefits of using techniques as either test data generation strategies or adequacy measurement tools should also be investigated. This latter point is just one part of the more general question pertaining to the cost-effectiveness of variously combined applications of techniques.

Test Data Technique	# Test Cases	Mutants Killed	Mutation Score
Statement Analysis	5	642	.67
Specification Analysis	13	780	.81
Branch Analysis	9	749	.78
Minimized Domain Analysis	36	932	.968
Domain Analysis	75	943	.980
Mutation Testing	347	962	1.00

Table 3-2. Comparison Using the MOTHRA Mutation System

Taken as a whole, capability profiles would provide urgently needed guidance in the selection and application of appropriate techniques for each testing effort. They would also be useful in determining which techniques should be provided by a comprehensive testing environment. Both theoretical and empirical evaluations of techniques are needed to acquire the necessary data; theoretical studies to provide insight into the power and effectiveness of techniques, and empirical trials to reveal the ease of use of a technique. Experimental studies are particularly important in dynamic analysis research since worst case analysis often gives very pessimistic upper bounds that do not satisfactorily reflect practical performance.

3.4.4 Integrated Application of Techniques

No single analysis technique is sufficient to ensure highly reliable software. Researchers are discovering that a simple sequence of several different techniques applied independently does not necessarily increase error/fault detection capabilities. Moreover, many techniques rely on much of the same basic information (such as control and data flow patterns), and sequential application of techniques is highly inefficient as the same information must be generated repetitively. The relationship between techniques must be closely examined to identify those which best support one another, or even subsume others. Powerful and efficient testing environments cannot be built until such integrated techniques are available. At a higher level of concern, the relationship between the different forms of testing and evaluation needs to be studied. The distinction between, for example, dynamic analysis, static error analysis, and formal verification is narrowing as symbolic evaluation is increasingly a necessary precursor for each.

These problems are well-recognized and several researchers are examining approaches for integrating dynamic analysis techniques with some forms of static analysis. Hopefully, integrated techniques will start emerging in a couple of years. Meanwhile, the larger picture of integration must not be ignored.

3.4.5 Analysis of Concurrent and Real-Time Software

There is an acute lack of techniques for dynamic analysis of concurrent and real-time software. Although there is an emerging set of static techniques, only dynamic approaches are able to detect problems arising from the execution environment of the software.

Unlike the foundation provided by graph theory for analysis of sequential programs, there is no commonly accepted theoretical base for analysis of concurrent and real-time software. Fundamental research is needed to develop a firm foundation from which technology can evolve. There are a few instances of such work, for example that being conducted by Weiss [Weis88a]. At the current time, it would be premature to select just one of the emerging approaches and concentrate research resources in that direction. Multiple research avenues must be pursued with plenty of cross-fertilization and dissemination of information between efforts. In view of its dependence on the operating environment, this research should include efforts which take a combined view of software and hardware concerns.

4. STATIC ANALYSIS TECHNOLOGY

There are many different types of static analysis. For practical purposes, formal verification techniques and techniques oriented towards measurement of critical software properties are discussed separately, in Sections 5, 6, and 7.

Remaining static analysis techniques can be divided into four groups. The first group consists of those techniques which produce general information about a program, for example, symbol cross-referencers, rather than search for actual faults. These are relatively common and often provided by a compiler. The second group, static error analysis[3] techniques, are designed to detect specific classes of faults or anomalous constructs in a program. They focus on type and units analysis, reference analysis, expression analysis, and interface analysis. While some types of static error analysis can be automated, others are restricted to manual application. In contrast, the third group, symbolic evaluation techniques, are entirely automated. The final group consists of manual review techniques, namely code inspections and structured walkthroughs. This section concentrates on the latter three categories and discusses the state-of-the-art in these types of static analysis for sequential programs, concurrent and real-time programs, and pre-implementation products.

The final subsections summarize the major technological gaps in this area and review the current trends in automated support for both dynamic and static analysis.

4.1 Techniques for Static Analysis of Sequential Programs

Data flow analysis was one of the earliest static error analysis techniques and focuses on the detection of violations of sequencing constraints. It was derived from work in compiler code optimization where information is gathered to increase code efficiency by eliminating unnecessary computations. Initial work on data flow analysis was conducted by Fosdick and Osterweil. They defined a *data flow anomaly* as "a sequence of the events reference (r), definition (d), and undefinition (u) of variables in a program that is either erroneous in itself or often symptomatic of an error" [Fosd76a]. A software developer is required to specify all desirable and required sequences of events, the program is then analyzed to detect violations, which may arise from omitted and superfluous code errors.

Numerous notations for specifying sequencing constraints have been developed. These are based, for example, on the path expressions of Habermann [Camp79,Kieb83]; finite state machines [Howd83]; flow expressions (an extension of regular expressions) [Shaw78]; and axiomatic specifications [Gutt78a]. Various tools for applying data flow analysis have been developed [Fosd76a,Brow78,Conr85], mostly for analyzing Fortran programs. Data flow analysis has also been proposed as a way of guiding test data selection; the resulting data flow testing techniques are discussed in Section 3.1.1. Much of the early data flow analysis work focused on techniques and tools for the detection of fixed, and frequently limited, classes of data flow conditions. In the last few years, Olender and Osterweil have developed a more flexible mechanism for specifying a variety of event sequencing problems, which can be mechanically translated into algorithms capable of solving these problems [Olen86].

Another common static error analysis technique is interface analysis. In [Howd87], Howden identifies

3. As is the case with various aspects of error-based and fault-based testing, the term static *error* analysis is a historical artifact. These techniques would be more properly categorized as static *fault* analysis techniques.

three levels of interface analysis, these are: module-interface analysis, data-interface analysis, and operator-interface analysis. Module-interface analysis is the highest level of interface analysis. It is used to analyze the interfaces between system objects for consistency, completeness, and redundancy. Data-interface analysis, the next level of analysis, exploits user provided descriptions of the expected transformations to examine the transformations of one type of data into another that occur within a program module. Finally, the lowest level of analysis, operator-interface analysis, analyzes data structure operators to determine whether (1) these operators are applied to appropriately typed objects, and (2) the sequence of operators is, in fact, legal. (Of course, these types of interface analyses are automatically provided by compilers for languages such as Ada which provide encapsulation and strong-typing).

During empirical studies of large systems, Howden noted that the significant problems in these systems arise from the difficulty of keeping track of data, rather than the more specific types of faults such as computation faults. This led to the development of a new analysis technique for detecting decomposition errors, called flavor analysis [Howd87]. Here *flavors* denote the *meaning* of particular variables and can be used in much the same way as type information is used to determine proper variable usage. In order to detect incorrect assumptions (concerning variable names or missing code, for example), flavor statements are included in the code and subsequently analyzed for consistency. Howden describes two types of flavor statements, those which summarize (1) current assumptions about flavors, and (2) the flavor effects of statements or sections of code [Howd89]. Although flavor statements can be translated into both compile-time and run-time checks, Howden states that their primary use is in static detection of false assumption decomposition errors. An initial flavor analysis tool has been developed. An advanced tool which can check assumptions about scheduling and temporal assumptions is planned to support flavor analysis of concurrent and real-time software.

Symbolic evaluation is a technique whereby a program is executed over symbolic rather than actual data. Symbolic values are assigned to input variables and the user selects a path through the program to be evaluated. Each expression along this path is evaluated by substituting symbolic values for the variables. Thus, symbolic evaluation can be used to analyze the conditional branching predicates of a program, in addition to the output computations. The approach can be varied by symbolically evaluating a program over a mixture of actual and symbolic data, or examining the traces of intermediate symbolic values assigned to variables.

Symbolic evaluation is used for a variety of purposes. In symbolic testing [Howd77a], symbolic values of output variables are generated and compared (usually manually) against a program specification to identify faults. Symbolic systems of predicates for a program path can be used to analyze the subsets of the input domain that cause particular program paths to be executed; test data to execute these paths can also be generated [Rich85a]. In this capacity, symbolic evaluation techniques are an essential element of many of the dynamic analysis approaches discussed previously. They also provide the capability to detect infeasible paths, although rules for determining feasibility cannot detect all such paths. Symbolic evaluation is also used in proofs of correctness. Meanwhile, its potential continues to be investigated. Perhaps the most recent innovative application of symbolic evaluation is Morell's approach to combining symbolic evaluation with mutation testing, yielding a fault-based dynamic analysis technique with both global extent and infinite breadth (see Section 3.1.3).

4.2 Techniques for Static Analysis of Concurrent and Real-Time Programs

Taylor and Osterweil have extended data flow analysis techniques to detect faults and anomalies in concurrent programs [Tayl80a]. The same classes of variable usage faults sought in sequential programs can be found in concurrent programs, for example, referencing an uninitialized variable. A number of data flow faults unique to concurrent programs can also be detected. In this latter case, Taylor and Osterweil

have developed algorithms for detecting the following faults and anomalies: (1) waiting for an unscheduled process, (2) scheduling a process in parallel with itself, (3) waiting for a process guaranteed to have previously terminated, (4) referencing a variable which is being defined by a parallel process, and (5) referencing a variable whose value is indeterminate. Both interprocess and interprocedural data flow are analyzed, based on process augmented flowgraphs (PAFs) which provide a graph representation of a system of communicating concurrent processes.

Another area being investigated by Taylor is static concurrency analysis. This technique identifies anomalous synchronization patterns in concurrent programs through state-based program analysis techniques. Unlike dynamic analysis approaches, static concurrency analysis can potentially examine all possible synchronization patterns. It does, however, suffer some disadvantages. Static analysis cannot identify all infeasible paths and so may report spurious faults involving these paths. Moreover, static concurrency analysis has been shown to be NP-complete and is only practically useful for programs comprising a relatively small number of processes. As with all static approaches, it is weak in dealing with dynamically identified objects such as array elements and pointers. With respect to Ada, this includes task entry families and tasks that are components of dynamically allocated data objects.

Together with other researchers at the University of California (Irvine), Taylor has developed an approach for mitigating some of these problems. Here static concurrency analysis is combined with symbolic execution, by interleaving phases of the two techniques [Youn86a]. This interleaving allows symbolic execution to prune away the infeasible paths otherwise identified by concurrency analysis, and concurrency analysis to support symbolic execution by selecting paths leading to possible concurrency-related faults. Tools to apply this approach to Ada programs are under development.

Another group of researchers, at Stanford University, are developing a language for the specification of distributed Ada systems that will facilitate both static and dynamic analysis [Luck87]. TSL (see Section 3.2) is intended to provide rigorous investigation of concurrent programs without the overhead imposed by formal verification. The first version of TSL was developed to explore the underlying concepts involved in the specification of concurrent systems. Based on their success in so doing, a more general-purpose, second version of the language (TSL-2) has been developed, together with some automated analysis tools.

4.3 Techniques for Static Analysis of Pre-Code Products

Data flow, concurrency, and interface analysis are not restricted to code products. Assuming that appropriately formalized representations are used, they can also be applied to pre-implementation products. For example, researchers at the University of Massachusetts have being investigating issues revolving around the description, enforement, and analysis of relationships between system components. Their approach to providing Precise Interface Control is called PIC [Wolf86c]. PIC extends the visibility concepts of declaration, scope, and binding which underlie traditional interface analysis by distinguishing between two types of visibility. These are *requisition of access* (which occurs when an entity requests the right to make reference to, or use, some set of entities) and *provision of access* (which occurs when a entity grants the right of reference to some set of entities).

The PIC approach supports a variety of analyses. Basic interface analysis examines type and requisition/provision information to determine the interface consistency within and among modules. Stub analysis checks the consistency between the view taken by each referencing module of a particular stub, and the consistency of each of these views against some "official" specification of that module. Finally, update analysis compares two versions of the same submodule to look for changes in declarations, requisition/provision specifications, or references to non-local entities. Although update analysis does

not directly address interface analysis, it is useful for identifying parts of the software which require reanalysis following some change. In each case, different forms of these analyses are provided to correspond to different types of (sub)modules.

The AdaPIC toolset is an instantiation of the PIC approach for use with large Ada systems. This prototype toolset is being tailored to support consistent abstractions, incremental analysis, and order-independent development so that an incremental approach to interface control can be adopted.

Safety analysis is a critical concern for complex systems such as the SDS. Even if error-free software were a feasible objective, the possibility of failures in the operating environment still pose safety risks. Leveson defines a technique called Software Fault Tree Analysis (SFTA) for analyzing the safety of a software design, independently of its functionality. This technique can be performed at various levels of abstraction, on either designs or code (the use of SFTA on Ada code is discussed in [Cha88]).

SFTA is derived from fault tree analysis (FTA), a technique originally developed to measure hardware safety. This genesis has the benefit of allowing hardware and software fault trees to be linked together at their interfaces, so that an entire system can be analyzed. FTA starts with a hazard analysis of the system where potential hazards are identified and classified according to their severity. As part of this process, failures which impact system safety are distinguished from nonsafety failures; safety failures are those where the ability to provide degraded operation takes second place to the need to minimize the damage of the failure. The root of the fault tree is then given by specifying a critical failure which is assumed to have occurred, called a *loss event*. SFTA uses backwards reasoning to identify all possible conditions which may lead to this failure, building the fault tree by showing the relationships between these conditions. This analysis continues until all the leaves susceptible to analysis describe events of calculable probability. At the point where the fault tree reaches the software interface, high-level requirements for software safety have been determined based on software behavior which could compromise system safety. SFTA then (1) demonstrates that the design logic will not produce safety failures, and (2) determines the environmental conditions which could lead to a software generated safety failure.

In FTA, failure statistics can be produced and sensitivity analysis used to measure the effect of each loss event. In [Leve83a], Leveson describes how such numerical analysis is less suited for the software case and defines the following uses for SFTA:

- Identification of the most likely causes of a loss event, which can guide testing and evaluation efforts and pinpoint those areas where most testing dollars should be allocated.
- Identification of critical modules which require special fault-tolerance procedures, e.g., run-time assertions, exception handling, or redundancy.
- Identification of unsafe states and the conditions under which fail-soft and fail-safe procedures should be invoked.

Tools to aid in the production and analysis of fault trees are being developed at the University of California (Irvine). Researchers plan to use these tools to gain further understanding of SFTA.

4.4 Manual Review Techniques

Manual review techniques such as structured walkthroughs [Myer78a] and code inspections [Faga76] evolved from the simple desk checking approaches commonly used by software developers. Each type of

review is performed by a team of software developers, each of whom plays a well-defined role to focus attention on different aspects of the program. The major distinction between these two techniques lies in the goal of the review. The purpose of an inspection is to check for specific errors identified on an error checklist. Walkthroughs have a wider scope of concern. In addition to manually simulating the execution of the software, the participants review design decisions and the overall approach taken by the developer. Although these reviews are expensive in terms of the manual effort required, they are among the most cost-effective techniques for eliminating faults. They also offer a significant side benefit in team building and ensuring back-up knowledge about a piece of software.

Walkthroughs and inspections can be applied in diverse ways. For example, a review may be conducted by all participants acting in concert, or by participants reviewing the software product independently and then pooling results. Table 4-1 shows results of a small-scale experiment which investigated different applications of review techniques, as reported in [Myer78a]. Gannon has reported on the frequency with which different types of errors were identified by using inspections and branch testing independently and in conjunction [Gann79a], see Figure 4-1.

Method	Mean # of Errors Found	Variance	Median # of Errors Found	Range of Errors Found	Cumulative Errors Found	Mean Man-Minutes Per Error
A. Computer-based + specification	4.5	4.8	4.5	1-7	13	37
B. As in A, + listing	5.4	5.5	5.5	2-9	14	29
C. Manual walkthrough/inspection	5.7	3.0	6	3-9	11	75
D. Pooling independent results using A	7.3	3.6	8	4-10	13	34
E. As D, but using B	8.3	2.8	8	6-11	14	34
F. As D, but using A and B	7.2	3.4	7.5	3-10	15	37
G. Combined A and C	7.6	4.3	8	5-10	14	15

Table 4-1. Comparison of Different Applications of Review Techniques

These review techniques are not restricted to implementation products. With the appropriate roles and, in the case of inspections, lists of potential errors, they can be applied to any type of software product.

4.5 Summary of Major Gaps in Static Analysis Technology

Many of the critical gaps identified for dynamic analysis technology also apply to static analysis. For example, the issue of integrated application of techniques affects both dynamic and static analysis. Quantitative information on the capabilities and costs of static analysis techniques is also needed. While such capability profiles should be much simpler to determine for static than dynamic techniques, this information is still unavailable. There is still a lack of proven capabilities for the static analysis of concurrent software. Even more than in the case of dynamic analysis, existing techniques are at the boundaries of the state-of-the-art and have only been applied on small, example problems. Much additional research is needed. In particular, since static analysis of concurrent software is often a NP-complete problem, a better understanding of how this analysis can be made sufficiently fast is necessary. The need for transfer of promising technology into practical use is again a critical concern.

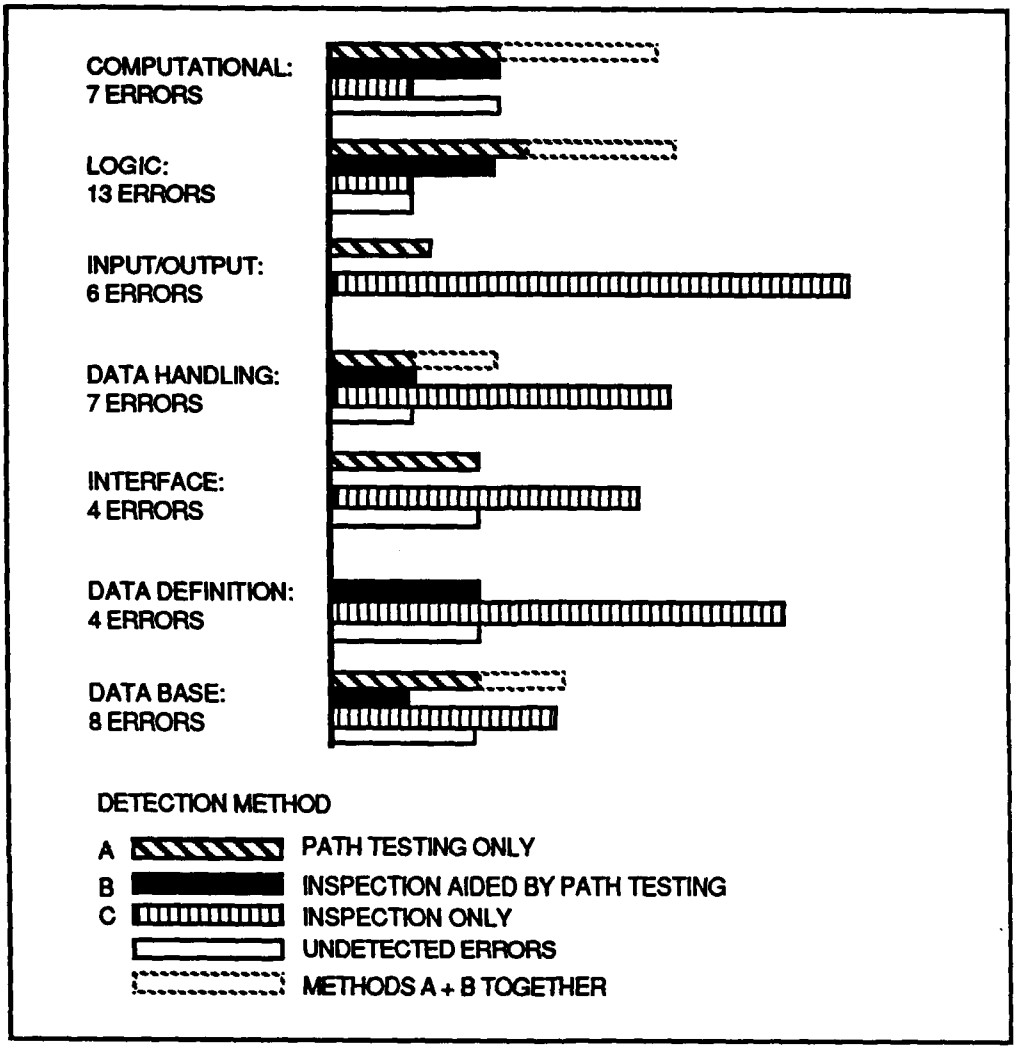

Figure 4-1. Percentage of Errors Discovered by Inspections and Branch Testing Using the Fortest System

The remainder of this subsection discusses three problems which have impeded the use and development of static analysis technology. The first relates to the programmatic issue of requiring use of available static analysis techniques. The second addresses the lack of common, formal representation forms for early life cycle products.

4.5.1 Establishing Static Analysis Policy

The earlier parts of this section focused on the more advanced types of static analysis. Looking at the entire body of software static analysis technology, some of which has been around for well over a decade, there are many available techniques. Since most of these are fully automated, requiring little effort on the part of a software developer, they are relatively cheap to apply. Although manual review techniques are not, of course, automated, sufficient data has been collected to demonstrate their usefulness.

There should be a well-established policy requiring routine use of a core set of these techniques. Instead of a collection of ad hoc tools, this core set (where appropriate) should be supported by a well-designed collection of commercially developed, cooperating tools.

4.5.2 Common Set of Formal Representations for Pre-Implementation Products

With the exception of various simulation and modeling approaches, static analysis is the main vehicle for examination of pre-implementation products. Unlike the dozen or so commonly used programming languages, there is a great number of representation forms for early life cycle products. These languages not only differ in syntax and semantics, but exploit a diverse variety of underlying conceptual models. Many embody only a limited degree of formalism which has precluded rigorous analysis of early development products. Lack of a widely-used, common set of languages has diffused the efforts of an already small research community and impeded the development of a substantial body of static analysis techniques (and supporting tools) for any particular language.

The need for increased formalism in early development activities has long been recognized. The reason why it rarely occurs in practice lies in the difficulty of using formal languages. While not all formalisms are, or need be, highly mathematical, the few existing formal languages generally employ mathematical reasoning which is beyond the educational preparation of most software developers. Although these languages could be designed for easier use, and better supported by automated tools, there is a fine line between providing easy-to-use notations and loosing the benefits of strict formalism.

4.5.3 Compatibility of Modular Interfaces

Although some work addressing the compatibility of modular interfaces is being performed, this is an area deserving much more attention. In particular, compatibility of hardware/software interfaces needs to be addressed. In as much as systems design will emphasize modularity in the future, and formal specification of interfaces, symbolic execution, testing, or formal verification methods for establishing interface consistency are needed.

4.6 Automated Support for Dynamic and Static Analysis

Increased emphasis must be placed on the development of *production quality*, automated support tools. Automated tools have long been an essential ingredient for effective and efficient testing and evaluation. As techniques become more complex, tools which automate the application of these techniques are increasingly indispensable. While extensive automation is not the solution to all testing and evaluation problems, it will significantly reduce the traditionally human-intensive nature of testing and evaluation. The provision of an integrated environment to support the use of state-of-the-art analysis activities will be a key element in their success. It is worthwhile noting, however, that as more sophisticated tools are developed to apply techniques in an integrated manner, software developers will require more education in the use of tools. Guidance will be necessary to ensure the proper and imaginative use of such tools in each type of circumstance.

At the turn of the decade, the majority of tools were those that provided analysis capabilities for Fortran, Cobol, and PL/1 programs. Several reviews of available tools are to be found in the literature. One of the most recent reviews was that conducted as part of the STEP effort [DeMi87a]. Over recent years, the trend towards supporting analysis of these older languages has been changing and the majority of new prototype tools are being developed to support the analysis of Ada programs. As research vehicles, these prototypes are not designed to be robust, easy to use, or portable. In many cases, significant effort will be

required to develop production quality counterparts that are suitable for widespread use. Yet it is *vital* that this effort be expended as part of the technology transfer activities necessary to bring new technology into everyday practice.

While stand-alone tools applying individual techniques were very much the norm in the 1970's, the recognized need for multiple techniques cooperating in an integrated strategy is promoting the development of a few powerful testing and evaluation environments. The Mothra environment [DeMi87b], developed by the Georgia Institute of Technology and Purdue University, is one of the first such environments. It currently provides mutation testing, structural testing, and a form of functional testing. Symbolic testing is expected to be added within the year. The existing system supports analysis of Fortran programs, and it has been distributed to a few sites for evaluation and beta testing. Another version supporting analysis of Ada programs is expected to become available in the near future. Since mutation testing is computationally intensive, researchers are investigating how to merge large numbers of program mutants into a small set of highly vectorizable programs which can exploit the architecture of vector processors, such as the Cray X-MP or the Alliant FX/8, for efficient execution of all the mutants [Math88a].

Researchers at the University of Massachusetts are in the early stages of developing another extensive Ada testing and evaluation environment [Clar88b]. Their intention is to support all the major, current analysis approaches, including data flow testing, EQUATE, and RELAY. This environment, called the Testing, Evaluation, and Analysis Medley (TEAM), is designed as a hierarchy of tools that is both flexible and extensible. Accordingly, there are two important aspects to the design. First, the system architecture is designed to provide layers of capabilities that support more advanced analysis techniques. Second, as many general capabilities as possible are recognized. One tool in particular, the ARIES generic interpreter, will be central to many other tools. ARIES has several distinctive features: (1) it is an generic interpretation system which is instantiated to yield a customized interpreter for use in a particular tool, (2) it is multilingual and can be used with a variety of procedural languages, and (3) it is a multi-computational model system capable of supporting both conventional models of execution and a variety of symbolic and data flow models. It is expected that this interpreter will provide the symbolic interpretation capabilities required for an Ada symbolic evaluation system.

One of the goals of the TEAM environment is to allow researchers to conduct experimental studies of existing analysis approaches. In particular, researchers plan to investigate the integration of analysis techniques, analysis support for pre-implementation products, and incremental analysis of potentially incomplete products. In addition to the emphasis on generic and language-independent capabilities mentioned above, Clarke cites several additional requirements for TEAM [Clar88b]. These include effective user interaction models providing natural interfaces which reduce the burden on the user and ensure interface uniformity across tools. A process programming approach [Oste87a] will also be adopted such that TEAM itself will prescribe the acceptable uses and interactions among tools, relieving the user of unnecessary responsibilities and facilitating the inclusion of new capabilities.

This effort, as a whole, is being carried out in the context of the Arcadia software development environment [Tayl88]. The initial prototype of the TEAM environment will contain basic data flow and symbolic evaluation capabilities and should be running by the end of 1988.

The prototype software and hardware development environment being built by researchers at Stanford University [Luck86a] has a wider focus than the efforts so far discussed. This environment is based on the use of wide-spectrum languages which provide a notation for describing the intended behavior of a system and the implementation of that behavior. These languages are ANNA, TSL, and the VHDL Annotation Language (VAL) [Luck86a]. Special emphasis is placed on distributed computing, both in providing tools for handling concurrency in the subject system, and in designing tools that utilize concurrency in the environment itself. ANNA and TSL can be used to develop specifications of Ada systems that are

susceptible to, respectively, symbolic execution and simulation. Annotations given in these languages can also be automatically transformed into run-time checks which, as mentioned in Section 3.2, provide a form of self-checking Ada software. The prototype environment contains most of the tools required to support these functions, in various stages of maturity. In addition to the potential power of the testing and evaluation functions supported, this environment is very promising in its ability to integrate testing and evaluation into system development activities.

Looking at the field as a whole, however, it is clear that the number of static analysis tools is much smaller than the number of dynamic analysis tools. In view of the complimentary roles and benefits of static and dynamic approaches, this imbalance should be rectified.

It is also important to emphasize the critical need for *flexible* environments. Any environment (whether intended to support development activities, testing and evaluation activities, or both) which cannot continue to integrate the increasing numbers and types of tools that will emerge in the coming years will have a very short-lived usefulness. The necessary flexibility must be designed into an environment. Although this requires additional upfront planning and expenditures in the environment development process, it would be "penny-wise and pound-foolish" to follow any other course.

5. FORMAL VERIFICATION TECHNOLOGY

This section reviews the state-of-the-art in formal verification technology. After a brief overview of what formal verification attempts to achieve and what benefits it offers, the current status of the technology and ongoing research and development are reviewed. Finally, some high-level technology issues are raised and recommendations are discussed.

5.1 Focus and Benefits of Formal Verification

The objective of formal verification is to prove, using mathematical logic, that systems, in theory, will behave according to their specifications. The term *formal* refers to the level of rigor required in system specifications and the construction of valid proofs. English is not a precise enough language for stating technical specifications or arguing that systems satisfy their requirements. Mathematical notations with rigorous definitions provide the necessary precision and allow system correctness arguments to be checked by automated tools. The clause *in theory* is included to recognize that proofs apply to idealized models of systems, not to the actual physical systems themselves. For example, verification of high-level language software assumes the correct and reliable operation of compilers, run-time support systems, and hardware.

Several ingredients are necessary to prove properties of systems, including:

1. Rigorous definitions or specifications of the behavior and performance required to achieve each property — typically a set of system relationships that must hold under all circumstances;
2. Rigorous definitions of the meaning (that is, effects) of all statements and expressions that can be formulated in the programming language used — also called the language's formal semantics;
3. Rigorous definitions of the effects of each machine instruction available on the target hardware; and
4. Sound rules of logical inference, which guarantee that only true assertions can be proved.

Verification and testing should be viewed as complementary technologies. Proofs address entire classes of possible circumstances, where testing exercises only a relatively small number of actual cases. Verification is useful, therefore, to assure system properties that are impossible to test adequately. Security in operating systems, for example, can be assured by proofs of security properties, but is extremely difficult to assure through testing. Verification is not a substitute for testing, however, because tests can be applied to actual physical systems as well as to idealized models. That is, testing can produce counter examples that invalidate assumptions upon which proofs are based. Proponents of formal verification are not likely to volunteer to fly in aircraft that have been verified but not tested.

5.2 Status of Current Technology

This section briefly surveys current verification techniques in three areas: software, hardware, and systems.

5.2.1 Techniques for Software

The principal techniques for verifying software are described below, starting with techniques for simple sequential software. These techniques apply to software designs expressed as high-level programs as well as to actual program code. Programming language features that complicate the verification process are then briefly discussed, followed by descriptions of additional techniques for verifying concurrent and parallel software.

5.2.1.1 Sequential Software

Techniques for proving properties of programs written in conventional procedural programming languages are the oldest and most well known to software and system developers. These techniques were introduced by Floyd [Floy67] and Hoare [Hoar69], and have been refined and improved by numerous researchers, including Dijkstra [Dijk76a] and Gries [Grie81].

The general approach assumes that programs start in an initial state with a set of initial conditions, and must complete in a state that satisfies a set of required final conditions. This may be represented symbolically in two possible forms. The first is

```
initial_state  { program }  final_state

final_state  →  required_final_conditions
```

which means: starting from the initial state, executing the program will result in the final state, and the final state satisfies (implies) the required final conditions. The second form is

```
initial_state  →  weakest_initial_conditions

weakest_initial conditions  { program }  required_final_conditions
```

The weakest initial conditions are the minimal conditions under which executing the program will always result in satisfying the required final conditions. This second form is preferred for developing a program and its proof together as one process.

Programs are made up (typically) of sequences of statements. Proof rules for sequences of statements require the result of each successive statement to satisfy the weakest preconditions for the next statement. Symbolically, this can be expressed as

```
weakest_initial_condition  { statement_1 }  intermediate_result

intermediate_result  →  weakest_intermediate_precondition

weakest_intermediate_precondition  { statement_2 }  required_result
```

Weakest preconditions are typically derived in reverse order, starting with the last statement in the program. Proving that a program satisfies its requirements amounts to deriving the effects of each statement and proving that all intermediate results imply the corresponding intermediate preconditions.

Additional rules for interpreting the effects of assignment statements, conditional statements, loops, and procedure calls can be spelled out in terms of the results they produce. For example, a conditional

statement such as

```
IF  test_condition  THEN
   then_statements
ELSE
   else_statements
END IF;
```

can produce two possible results: one produced by the *then_statements*, when the test condition is true, and one produced by the *else_statements*, when the test condition is false. Turning this around to look for a weakest precondition to achieve a required result requires solving the following expressions:

```
weakest_precondition
   AND test_condition  { then_statements }  then_result

weakest_precondition
   AND NOT test_condition  { else_statements }  else_result

then_result  →  required_result

else_result  →  required_result
```

That is, the weakest precondition for an if-then-else statement is the most general condition that allows both alternative results to satisfy the required result.

Loop statements that have conditional exits operate much like conditional statements. Loops, however, require the derivation of another condition called the loop *invariant*. Consider, for example, the loop statement:

```
WHILE  NOT exit_condition  LOOP
    loop_body
END LOOP;
```

If the exit condition is false, meaning the loop body will be executed, there are conditions that, if true before the loop body is executed, will remain true afterwards. The loop invariant is the most general of these conditions. That is,

```
loop_invariant
    AND NOT exit_condition  { loop_body }  loop_body_result

loop_body_result  →  loop_invariant
```

The weakest precondition for a loop statement as a whole is independent of its exit condition, since the exit condition may be either true or false at that point. The loop invariant is the critical precondition. In fact, the invariant is the key to a loop's overall behavior. The proof rule for a loop statement can be formulated in terms of the invariant as follows:

```
loop_invariant  { loop_statement }  loop_invariant AND exit_condition
```

Loop invariant conditions can be difficult to discover. There are no simple rules or procedures by which they can be automatically derived.

All that remains to complete the proof of a loop statement is to prove that the loop terminates. This requires demonstrating that the loop body moves a step closer to making the exit condition true on each iteration. An inductive argument can then be made to guarantee that the loop will eventually terminate.

Function and procedure subprograms can simplify large programs. They also simplify program verification. A proof of a subprogram's behavior can be derived once and used as a lemma everywhere the subprogram is called. Function calls must be embedded within statements. Any input restrictions imposed on parameters or global variables by a function, therefore, must apply to the containing statement. A symbolic expression representing the value returned by the function can then be substituted in the proof rules.

Proof rules similar to those described above for loops and conditional statements can be formulated for procedure calls. They will typically include input restrictions imposed on parameters and global variables, and will define the resulting conditions that will hold when the procedure returns control to the calling program. These proof rules generally have the form

```
preconditions  →  input_restrictions

input_restrictions  { procedure_call }  resulting_conditions

resulting_conditions  →  required_results
```

where input parameter and global variable restrictions are applied to the actual arguments and the current environment, and the resulting conditions are similarly transferred to reflect the new program state upon return.

Recursive functions and procedures require termination arguments similar to those required for loops. It must be shown that every recursive call solves a simpler version of the original problem and that, at some point, a solution can be reached directly, without further recursion.

5.2.1.2 Complications

This brief tutorial on verification techniques oversimplifies the task. There are numerous complications in proving properties of real system designs and real application software. The first complication is that proofs are more detailed, involve more steps, and are more tedious than the designs or programs themselves. While much of the tedium can be mitigated by automated tools, constructing proofs is still a demanding task.

Other sources of complication stem from language characteristics. In Ada, for example,

- Side effects from functions can change the state of a program in the middle of evaluating expressions,
- Exceptions can leave results of computations in undefined states, and
- Aliasing of procedure parameters and global data can have additional side effects.

These characteristics make rules for interpreting statements more complex and require software developers to verify many more intermediate conditions to produce complete proofs. One of the reasons for recommending that properties of software be proven as an integral part of the development process is that many of these complications can be avoided by restricting use of these language features.

5.2.1.3 Concurrent Software

The introduction of concurrency (multitasking) in programming languages adds a new dimension to the verification problem. The simple model of a single sequence of statements is no longer adequate to describe computations. Several sequences of statements may be active at one time and their execution may overlap and interleave in arbitrary ways, making it virtually impossible to isolate a program's state at any particular point.

Properties of concurrent software can be divided into two major classes [Lamp83]:

- Liveness properties — which identify program behavior that must be achieved (e.g., that tasks respond correctly and cooperate to satisfy the program's requirements); and
- Safety properties — which identify inconsistent or untenable program states that must never arise.

Liveness properties can be analyzed using process modeling techniques such as Petri nets [Pete77] or message passing schemes such as Communicating Sequential Processes (CSP) [Hoar85]. Tasking models usually assume that tasks interact only in well-defined, controlled ways; namely, when they synchronize with each other or exchange messages. This allows the techniques for verifying sequential program code to be applied to the behavior of individual tasks between synchronization points. Proofs of concurrent programs, therefore, attempt to show that:

1. Modeled tasks satisfy the program's required liveness properties;
2. Program tasks and synchronization points correspond directly with modeled tasks;
3. Synchronization points are the only places where tasks interact; and
4. Individual tasks, in isolation, satisfy their own input-output requirements.

Safety properties are *invariant* conditions similar to those described for loops. These invariants, however, are global to the program and must hold across all tasks. An example safety property is freedom from deadlock. The invariant condition to be proven is that no tasks will ever be blocked waiting for synchronization with each other in such a way that none of them can ever proceed.

Temporal logic [Pnue77] is a technique used to reason about both liveness and safety properties of concurrent programs. Temporal logic extends the predicate calculus with expressions that indicate time dependencies such as *henceforth* and *eventually*. For example, liveness properties are usually proven by showing that a program progresses from one stable, global state to another. The exact sequence of individual steps, however, cannot be determined because of their concurrent execution. A temporal logic specification of such a transition is

```
HENCEFORTH initial_stable_state
    →  EVENTUALLY  next_stable_state
```

Stable states are typically task synchronization points, including task initiation and termination.

Temporal logic obeys a full set of algebraic laws that allow formal reasoning about the behavior of concurrent programs. For example, the following equivalence relation captures the concept that a condition P is not always true if eventually it can become false.

```
NOT (HENCEFORTH condition_P)  =  EVENTUALLY (NOT condition_P)
```

5.2.1.4 Parallel Software

Parallel software is distinguished as a special case of concurrent software where concurrent operations are performed synchronously. These techniques are directed toward fine-grained machine parallelism rather than the general multi-tasking models of concurrency previously discussed.

Recently, Chandy and Misra [Chan88] introduced a new unified theory of parallel computation. Their approach addresses a wide range of granularity in concurrent operations and applies to a wide range of parallel machine architectures. In this model, program execution starts from an initial state and repeatedly selects and executes assignment statements nondeterministically. Each assignment statement may be guarded by a conditional expression and may assign values to multiple variables in parallel. The only constraint on the nondeterminism is that every assignment must be selected and executed infinitely often. The theory is simplified by not including conventional program control flow in the model and by assuming that all programs run forever. In practice, programs terminate when they reach a *fixed point*, where all open assignment statements leave the program state unchanged.

Although this model is very simple, it is fully adequate for expressing and analyzing useful and efficient parallel computations. One advantage of a simple model is that formal semantics and proofs of properties can be greatly simplified. This model also appears to support methods for efficiently mapping programs onto several types of shared-memory multiprocessor machine architectures.

5.2.2 Techniques for Hardware

Verification of high-level language software assumes the correct and reliable operation of compilers, run-time support systems, and hardware. Proofs of properties at the abstract program level are of limited value if the correctness of translation and execution cannot be equally assured. One approach to solving this problem is to define a series of abstract machines, each of which can be emulated by the next machine using a relatively simple (that is, provable) set of software macros. At the top end of the series is an abstract machine that directly executes the high-level program. At the bottom end is a machine that maps directly to the target hardware. That is, there is a one-to-one correspondence between the last abstract machine and the design of the actual physical hardware. This technique transcends the machine instruction-set level and can be used to prove program properties down to the micro-code and hardware gate level.

5.2.3 Techniques for Systems

Formal verification of complete hardware and software systems is an active research area. Capabilities that system verification will require include:

- Formal system-level specification techniques — including hardware and software performance specification and analysis techniques.
- Compatible modeling techniques for diverse system components that, for example, allow proven properties of hardware components to be incorporated in proofs of software components.
- System construction techniques that allow composition of proven components to yield provable systems.

5.2.4 Automated Support

Automated tools for verification include processors for formal specification and program annotation languages, verification condition generators, theorem provers, and proof checkers. Specification and annotation languages are typically variations of the predicate calculus. Verification condition generators attempt to describe symbolically the state transitions made by each statement in a program. Proofs are made up of arguments that program statements achieve specified conditions. Theorem provers attempt to construct proofs automatically using artificial intelligence techniques. Proof checkers are simpler systems that can verify correct proofs created manually or using other sources of automated help.

First-generation verification systems that have been developed include AFFIRM [Gerh80,Suns77], FDM [Kemm80], Gypsy [Good86a,Ambl76a], HDM [Robi79], and the Stanford Pascal Verifier [Luck77]. The first four of these systems were reviewed and evaluated in an extensive report by Kemmerer [Kemm86]. The FDM and Gypsy systems have been approved by the National Computer Security Center (NCSC) for the verification of systems targeted for *A1* security certification. NCSC has also stated that the Gypsy system will be used in all verification of SDS software. These systems are complicated, however, and it is not easy to transfer knowledge gained on one system to another.

5.3 On-Going Research and Development Efforts

This section discusses current research and development activities being conducted to advance the state-of-the-art of verification theory and practice.

5.3.1 Basic Research

Basic research is needed to develop understanding of several fundamental open verification problems. These problems are wider than simple gaps in the current technology and are likely to take some time to solve. Work is currently being done in these areas, but no results have been reported. The following sections describe current and needed efforts in the areas of real-time systems, distributed systems, and degraded system operation.

5.3.1.1 Real-Time Systems

The critical property to be verified in most real-time systems is that processes meet their deadlines. The answer is affected by the algorithms employed, efficiency of object code generated by the compiler, scheduling policies and performance of the run-time system, and target hardware performance. The problem, therefore, spans the entire system design, which is what makes it so difficult.

In the past the deadline problem has been addressed by *worst-case* analysis, which tends to produce overly pessimistic solutions. That is, systems are overbuilt for normal operations to ensure that they can handle the extreme worst-case deadline situations. Specific techniques employed include: algorithms with fixed execution times, assembly language or hand-optimized object code, fixed priorities, deterministic scheduling, and target hardware upgrades. Each of these techniques simplifies the deadline verification problem, but they often restrict capabilities that could be supported under normal (slack) operating conditions.

Adaptive algorithms and scheduling techniques that adjust to system workload have been introduced to gain processing capabilities during slack periods. For example, when system workload is light, slower but more accurate algorithms can be used and useful background operations can be performed. As the

workload picks up, background operations are dropped and required processes can switch to faster algorithms. In addition, as a process nears its deadline, it may try to increase its scheduling priority to assure its completion. Formal methods are needed for reasoning about these techniques that would allow development of proofs of adaptive program behavior.

5.3.1.2 Distributed Systems

Distributed systems are characterized by communication latencies between subsystems. This makes it extremely difficult for subsystems to synchronize their actions. Timing constraints on coordination, for example, may not allow subsystems to fully verify each other's actions or readiness. That is, they may have to proceed on the assumption that the other subsystems are performing their functions at the right time. Formal verification of such systems requires methods for reasoning about system behavior that do not require a representation of the system's global state, which cannot be known because of the uncertainty of the timing of individual local transitions.

5.3.1.3 Degraded System Operation

Fault-tolerant systems are able to recover from or adapt to certain types of component failures. Many such systems may continue to operate in a degraded mode until the failed component can be repaired or replaced. Current verification techniques assume correct operation of underlying hardware and peripheral devices such as sensors. Methods for reasoning about system behavior in the presence of potential component failures is needed to verify fault-tolerant systems.

5.3.2 Applied Research and Development

Applied research and development addresses technology gaps that do not require significant breakthroughs in fundamental understanding. In most cases these problems can be solved by creative application of known engineering techniques and careful implementation. The primary activity in this category is development of production-quality automated tools.

A second generation of verification tools is currently in development. These tools are intended, primarily, to improve the utility of earlier tools by assuring the soundness of the underlying logic system, standardizing on programming and annotation languages (for example, Ada and ANNA), improving user interfaces, and improving performance. Examples of such efforts include the Annotated Verifiable Ada (AVA) system being developed by Computational Logic, Inc. and the Penelope system being developed by Odyssey Research Associates.

5.4 Application Issues

This section discusses three important issues relating to the application of verification technology:

- Identification of critical properties and components within a system that will require verification,
- Education and training in verification techniques, and
- Insertion of verification technology into software and system development processes.

5.4.1 Identification of Critical Properties and Components

5.4.1.1 Critical Properties

Critical properties of a system as a whole must be identified as early as possible in the development process. These properties affect critical design decisions in partitioning and allocating functional responsibilities within a system. Properties that must be proven at the system level imply requirements for components with proven properties and construction techniques that preserve those properties.

5.4.1.2 Critical Components

Critical system components that will require verification also need to be identified as early as possible in the development process. Proving properties of components is much easier when the proof process is made an integral part of their design and implementation. In fact, proofs of *correct* components may be impossible to construct after the fact, because of design decisions and programming practices that increase the complexity of proofs. Early identification of critical components can significantly reduce the needed verification effort.

5.4.1.3 Levels of Criticality

As a corollary to identifying critical properties and components, identification of levels of criticality would help in assessing verification requirements and allocating assurance resources between testing and verification.

5.4.2 Education and Training

Formal verification requires high levels of skill and maturity in logic and abstract mathematics. This is not likely to change even with high-quality automated tools. Tools make programmers more productive by handling the tedious details of proofs, but generating proofs will still require abstract analytical skills, understanding of proof techniques, and considerable mathematical sophistication.

These skills are not commonly taught in computer science courses today. Instead, students must take *theoretical* mathematics courses, which may be (or seem) quite unrelated to verification applications. This arrangement does not produce enough graduates with sufficient mathematical skills to enable the industry to verify software, hardware, and systems on a regular basis. Changing computer science and related engineering curricula to include foundations for, and direct applications of, formal verification could alleviate this shortage.

5.4.3 Technology Insertion

Two factors that would facilitate the application of verification techniques within the industry are: (1) access to production-quality verification tools, and (2) publication of worked examples of formal software and system specifications and proofs. Earlier versions of verification tools were primarily research vehicles built for experimentation in academic environments. Newer versions should be much more robust, easier to use, and easier to move from one machine to another. These tools can therefore be made available to a much wider community of potential users. Anyone who might consider employing formal verification techniques should never be discouraged by the lack of available tools.

Most people learn by following examples. Small, textbook examples are useful for learning the basic concepts of formal verification. Larger examples are necessary to demonstrate how these concepts scale up to verifying full-fledged systems. Case studies of full-scale verification projects are needed to provide models of how the technology can be applied and how the efforts should be managed.

6. SOFTWARE MEASUREMENT TECHNOLOGY

This section of the report discusses the field of software measurement. An introduction to the various types of software metrics is given, along with a discussion on some of the early and existing measurement efforts. Finally, a discussion of future directions in the field of software measurement is presented.

6.1 Introduction

Software measurement deals with the understanding, characterizing, evaluating, predicting, and controlling of software products and processes. This field has traditionally been associated with the application of *metrics* to software products and processes. The majority of these software metrics, when used in a context-free fashion, have not yielded results which are demonstrably useful to either software developers or managers. Consequently, current efforts are directed at providing a framework within the software development life cycle which will facilitate better understanding of the metrics values and aid in creating higher quality software products.

6.2 Types of Metrics

Software metrics are typically classified as either process metrics or product metrics. *Process metrics* are measures which quantify attributes of the development process and of the development environment. *Product metrics* are measures of various characteristics of a software product. A general discussion on process and product metrics can be found in [Cont86].

Examples of process metrics include the education level of programmers, the degree of automated tool support, and number of design walkthroughs. These metrics have the potential to provide feedback during the software development process and help managers to predict or monitor the progress and the utilization of resources within a project. While some studies have shown significant statistical correlation between the metrics and measured quantities relating to cost or fault rates, these studies often hold true in a only a particular environment. Little research has been performed on metrics which can be effectively applied across environments. In addition, the estimation models have usually given limited consideration to the underlying software development paradigm (for example, waterfall versus prototyping).

Product metrics can be classified as external or internal product metrics. *External product metrics* are those that rely on data collected during testing and actual use of a software product. They include performance metrics, maintainability metrics (as measured through cost of maintenance), and testability metrics (as measured through the cost of testing and the number of post-release errors). They also include reliability metrics (as measured through fault rates); these reliability metrics are discussed in more detail in Section 7. These metrics are all direct measures of quantitative attributes and, as such, are excellent indicators to the extent that the test data set used to derive the metrics matches the data actually encountered in operation.

Internal product metrics are metrics which rely on examination and static analysis of a software product. The goal of these types of metrics is to provide an indirect measure of the same attributes measured by external product metrics, but in a more cost effective way. Internal product metrics can be collected much earlier in the software development process than external product metrics, thus providing improved feedback that can guide the project. Examples of high-level metrics include complexity, portability, correctness, and maintainability. These are usually assessed through examination of easily-measured low-level

metrics such as the maximum level of nesting, the use of user-defined data types, and the number of lines of code in a program. Internal product metrics are the most controversial. It is generally agreed that single low-level metrics (for example, lines of code) do not provide enough information to derive a high level indicator (for example, level of effort required, number of faults, or complexity) [Kafu85a]. Moreover, it has not been conclusively demonstrated, through statistically significant experiments operating from a sound theoretical basis, that the low-level attributes actually relate to the high-level attributes of concern, and there is a lack of empirical data for assessing the value of using sets of such metrics.

Empirical validation of these metrics have been hampered for three main reasons. First, the data used to derive a researcher's metric is commonly artificially obtained from a controlled experiment. Often, data is gathered from the results of studying software developed during a programming course(s), or collected from small, non-typical programs found in industry. While controlled metric experiments are useful for exploring new, unknown phenomena and may lead to statistically significant results, they often do not scale up in real-world case studies [Basi86a]. Second, the statistics generated from a metric are sometimes questionable. Little data is used in the development of the metrics, therefore its relevance is not valid in the general case. Finally, it is often not clear what a particular metric is measuring. Metrics are described in general terms (for example, program complexity) which leave the object of measurement unclear.

Another problem associated with the use of metrics is the feasibility of artificially manipulating software so that desired metric values are achieved while not changing the functional characteristics in any beneficial way. Any measureable criterion serves as a motivator for project personnel, thus the use of context-free metrics must be closely scrutinized. While artificial manipulation is more difficult when several sets of metrics are specified, it can lead to incidents where increasing the value of measured attributes becomes the goal of the development effort, rather than increasing the inherent quality of the software.

Internal product metrics are often used to provide input into various software cost estimation models [Boeh84a]. These models analyze various software quality factors to determine the level of effort required for software development, and provide a means for resource estimation and allocation.

Until these metrics are better understood, they must be used with caution. This is not to say they are without value; indeed, metrics can be often useful in indicating software which may require closer scrutiny. Low-level metrics, in particular, should be looked upon as *indicators* of desirable and undesirable software characteristics, and not used for ascertaining the intrinsic worth of software.

6.3 Early Metrics Research Efforts

Much of the early work in software measurement dealt with software complexity metrics. These types of metrics sought to provide quantitative estimates of program complexity by measuring a variety of software attributes. It is commonly accepted that complex programs are more difficult to understand, maintain, and modify, than simple programs. One of the earliest and most simple measures for assessing software complexity is the Lines Of Code (LOC) metric. This measure derives from when software was entered on punched cards. Each card typically represented a line of code and, as such, it was easy to compare and contrast the size of the physical card decks. While punched cards are no longer a common media for software, the term remains as the most simple measure of program size. Program size is correlated to software complexity in that, usually, when size increases, so does complexity. The LOC measure is one of the easiest complexity measures to derive and is usually used as a benchmark for comparing other complexity metrics.

In 1976, McCabe introduced a graph-theoretic complexity measure which provided initial insight into the management and control of program complexity [McCa76]. McCabe's metric, named the Cyclomatic Complexity Metric, is based on graphs representing the control flow of a program. Programs with higher numbers of basic control paths are deemed more difficult to understand, modify, and maintain and, therefore, generate a higher cyclomatic number. The cyclomatic number is used as an indicator of those modules that may contain code which will be difficult to test and maintain. While McCabe's complexity metric is being used in industry as a useful indicator of potential software problems, it has not been shown to provide a better estimate of program complexity than the LOC measure [Hame82,Evan83a].

In 1977, Halstead published a monograph entitled "Elements of Software Science" [Hals77a]. Halstead claimed that the metrics of Software Science were firmly based on the methods and principles of classical experimental science, and that the measuring process could be reduced to a few mathematical equations. Immediately after its publication, a considerable debate arose over whether or not the Software Science metrics represented software in general, or only responded to a limited class of software [Albr83,Bäke79,Curt79a,Shen83,Zweb79]. However, in recent years Halstead's work has been shown to have serious theoretical flaws which render Software Science equivalent to simple LOC metrics [Card87a].

6.4 Early Measurement Research Efforts

This section of the report discusses two major research efforts which occurred early in the software measurement field. The National Aeronautics and Space Administration (NASA) Software Engineering Laboratory (SEL) created an organization which has provided a unique ability to study the implementation and results of a large data gathering effort. The Rome Air Defense Center (RADC) has funded a variety of software measurement research efforts which have contributed to the understanding of software metrics.

6.4.1 Software Engineering Laboratory

The SEL is a joint venture between the NASA/Goddard Space Flight Center, the University of Maryland, and the Computer Sciences Corporation. One of the goals of the SEL has been to improve understanding of the impact that metric usage has on productivity and the quality of software products [Basi85a]. It has identified various metrics that are useful for evaluating and predicting the complexity, quality, and cost of Ada programs [Basi83a]. In addition, extensive data collection techniques have been developed which provide the basis for further metric research. The SEL utilizes NASA-developed, operational software to provide the basis for empirical investigation into various aspects of metrics research and validation.

6.4.2 Rome Air Development Center Software Quality Work

The RADC has been involved in a long-term program to improve and control software quality. One of the key goals of this effort has been an attempt to identify the major issues of software quality and provide a well-defined process whereby the software quality of Air Force weapon systems can be better specified and measured. RADC's initial work was to identify and define a set of software quality factors that are relevant throughout the software development lifecycle [McCa77]. Table 6-1 reproduces these quality factors, together with the primary user concern that each factor is perceived as representing. The various user concerns are grouped into three acquisition concerns, representing how well a product performs, how well it is designed, and how adaptable it is.

Acquisition Concern	User Concern	Quality Factor
Performance - How well does it function?	How well does it utilize a resource?	Efficiency
	How secure is it?	Integrity
	What confidence can be placed in what it does?	Reliability
	How well will it perform under adverse conditions?	Survivability
	How easy is it to use?	Usability
Design - How valid is the design?	How well does it conform to the requirements?	Correctness
	How easy is it to repair?	Maintainability
	How easy is it to verify its performance?	Verifiability
Adaptation - How Adaptable is it?	How easy is it to expand or upgrade its capability or performance?	Expandability
	How easy is it to change?	Flexibility
	How easy is it to interface with another system?	Interoperability
	How easy is it to transport?	Portability
	How easy is it to convert for use in another application?	Reusability

Table 6-1. RADC Quality Concerns

Figure 6-2 shows how various quality factors are decomposed into a hierarchical software quality measurement framework where each factor is broken down into several criteria. Each criterion is then further subdivided into a set of metrics. The 13 quality factors are composed of 29 criteria, while 73 metrics have been defined consisting of over 300 lower-level metrics. It is the combination of these lower-level metrics which ultimately generates a high-level software quality factor. Although there is little empirical evidence to validate these correlations, general relationships have been validated (for example, low coupling between modules seems to produce more maintainable software). Currently, these metrics and factors serve only as guides to, or simple indicators of, a program's quality.

In 1978, RADC and the US Army Computer Systems Command sponsored enhancements to the initial metrics framework [McCa80]. These enhancements provide a project manager with a description of those quality factors typically considered to be the most important. An Automated Quality Measurement Tool has been developed to automate the collection of specific metric data and to provide various quality measurement results.

In 1979, RADC sponsored research into software quality issues regarding software reusability and interoperability [RADC83a]. Metrics for assessing the quality of networked computers and distributed systems have also been developed [RADC83b].

6.5 Existing Automated Support

Although much metric research is language-independent, implementation of automated product

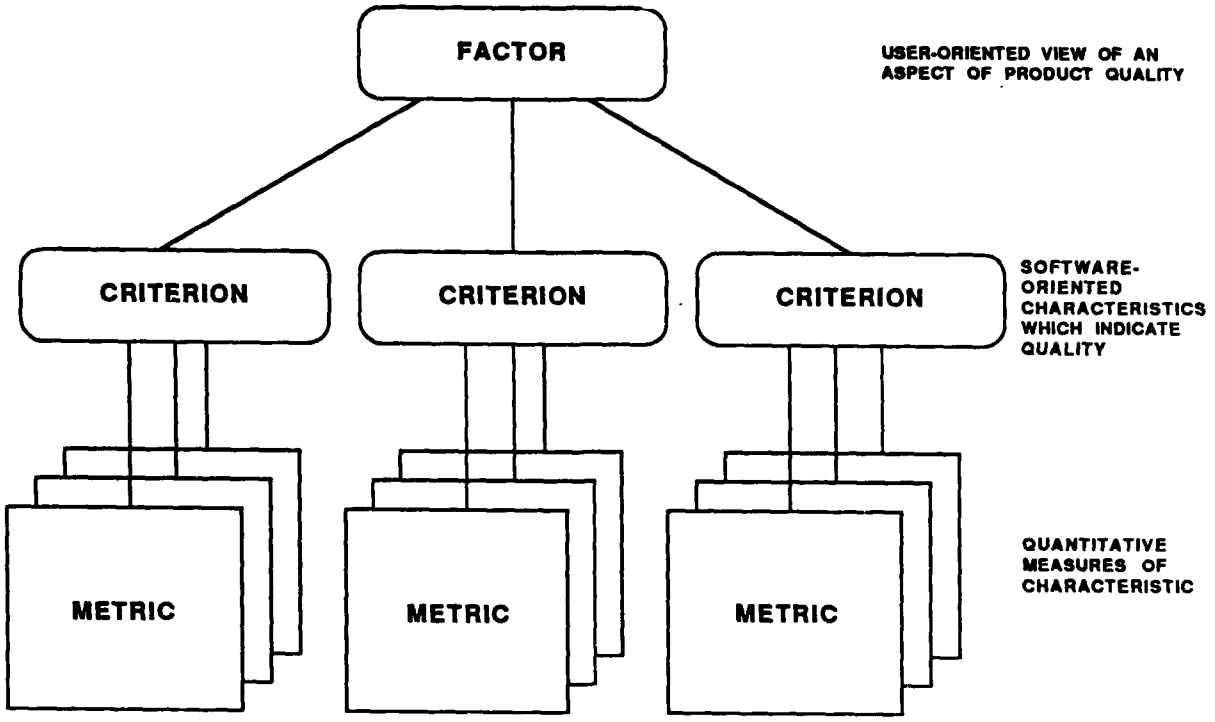

Figure 6-1. Software Quality Model

metrics usually requires tools to be geared to a specific language. While there are many metrics tools available for the analysis of software written in languages such as FORTRAN and COBOL, tools tailored for the Ada language are only beginning to emerge. However, since much of the DOD-sponsored metric research is currently focusing on automated support for the Ada language, increasing numbers of Ada-oriented tools can be expected over the next several years.

The remainder of this subsection outlines the current, major Ada-oriented metric tools/efforts .

6.5.1 AdaMAT

Dynamics Research Corporation has developed an Ada Measurement and Analysis Tool (AdaMAT) to provide automated metric analysis of Ada software [Perk86]. AdaMAT supports a metrics framework which measures six software criteria (anomaly management, independence, modularity, self-descriptiveness, simplicity, and system clarity) supported by 150 software metric elements. It consists of three separate tools: (1) a data collection tool for static analysis of the Ada software, (2) a quality analysis component providing an interactive analysis of the code and isolation of problem or unusual code, and (3) a report generator which collects the results of the completed analysis. AdaMAT metrics are arrayed in a hierarchy based upon the RADC metrics framework [Cava78] with tailorings for the Ada language. In this framework, the lowest level metrics are data items, which produce information on such concerns as maximum level of nesting and number of "out" parameters in a procedure. At the next level up, metric-elements use the data items to provide information such as local types referenced and local levels of nesting. Metric-elements are then used to create software quality sub-criteria which in turn provide information on such concerns as flow simplicity and error prevention. Finally, at the highest level, software quality criteria provide information on general aspects of software qualities, such as modularity, simplicity, and anomaly management.

6.5.2 ATVS

Although not yet completed, General Research Corporation, under contract to RADC, is developing an Ada Test and Verification System (ATVS) [RADC86]. ATVS is a test and measurement tool which provides static and dynamic analysis of Ada source code in addition to the collection of software quality measurement data. Static analysis includes call dependency, task termination dependency, and potential circular deadlock detection. Dynamic analysis is achieved by instrumenting the source code with probes, and yields information about test coverage, timing, and tasking activity analysis. In addition, ATVS will provide for the translation of manually entered assertions into executable code. Software quality measurement data is collected during testing activities. This data is then made available to both the user and other tools which are planned to be integrated into a software development environment RADC is constructing.

6.5.3 Software Metrics Data Collection (SMDC)

Developed at Purdue University, the Software Metrics Data Collection (SMDC) system [Yu88a] provides a comprehensive repository of data which can function as a testbed for the detailed analysis of information related to software development. SMDC is an APL-based system which runs on a UNIX[4] 4.3 BSD environment. It provides an extensive facility for the mathematical and statistical manipulation of data collected during software development with a view towards metric analysis. The data currently resident within SMDC has been acquired from the public domain, industry, academic, military, and other sources. Metrics such as development effort, duration, Software Science, Cyclomatic Complexity, LOC, and others are collected and stored in the SMDC.

6.5.4 The NOSC Tools

In 1983, the Naval Ocean Systems Center (NOSC), under contract to the World Wide Military Command and Control System (WWMCCS) Information System (WIS), contracted for a wide selection of software tools to be written in, and for, the Ada language. This software, collectively known as the NOSC tools, includes automated support for such tasks as database management, graphical interfacing, text processing, project management, and metric analysis. One of the metrics tools provides an implementation of the Software Science, Cyclomatic Complexity, and LOC complexity measures specifically tailored to the Ada language. The NOSC software resides in the public domain.

6.6 Future Directions in Measurement Technology

There are several areas which need to be improved if effective software measurement is to be achieved. A solid measurement methodology must be developed which will provide a framework from which appropriate metrics can be selected for a project and data collection and validation can be facilitated. Methods which provide better feedback of the results of metric analysis into software development activities are needed. In addition, there is a need for extensive automation of various tools and techniques to support the entire measurement process and its integration into software development.

---

4. UNIX is a registered trademark of AT&T

6.6.1 Measurement Methodology

Metrics have traditionally been applied to software products and processes in a somewhat bottom-up, stand-alone fashion. The measurement strategy typically revolves around a collection of independent metrics which are applied to the software undergoing analysis. Measurement begins when coding nears completion, and ends when acceptable levels of desirable attributes are attained.

It is generally recognized that the current application of stand-alone, evaluative measures of software development products or processes does not yield results that can be effectively interpreted, compared, or validated. A *methodology* is required to integrate the diverse aspects of establishing measurement requirements, guiding the selection of appropriate metrics, and supporting the collection, interpretation, and validation of results.

Basili and Rombach have proposed a software engineering process model which seeks to achieve this objective [Basi88a]. The model is based on two paradigms: the Quality Improvement paradigm, which provides a guide to improving the software development process, and the Goal/Question/Metric (G/Q/M) paradigm, which guides the selection of appropriate measures.

The Quality Improvement paradigm proposes a sequence of six steps which guide activities necessary to better understand and improve the software construction process. In the first step, the current project environment is characterized. This step attempts to identify the various factors which will influence the project development (for example, problem domain, personnel factors, product factors, and available resources). Second, goals for a successful project development are set. Example goals are improvement of the quality of the product, a reduction in production costs, and achievement of a stated software reliability threshold for a product. Third, the appropriate methods and tools for the project are chosen with the objective of maximizing project goals. Next, the software is developed, and data related to the goals, methods, and tools of the project are collected. Data can be gathered from forms, interviews, and automated tools. The Quality Improvement paradigm does not specify what data to collect or how the data is to be collected, but only provides a basic framework so that each specific step can be detailed (and perhaps automated) by the organization. The next step is a post mortem study of the gathered data in order to evaluate current practices including both the development and measurement processes and tools, determine problems, and make recommendations for the improvement of these practices in future projects. The final step simply requires that the organization does actually build upon and exploit the knowledge gained from this data collection and analysis in subsequent projects.

The G/Q/M paradigm provides (1) an operational formulation of the second step of the Quality Improvement paradigm and (2) the glue to tie together all the steps of the Quality Improvement paradigm. Here, an approach is specified for determining and specifying the goals of a software development project. These goals are then refined into a set of quantifiable questions which provide the basis for determining the appropriate software metrics and the data to be collected. Automated templates and guidelines are provided to assist in the derivation of these goals, questions, and metrics.

The G/Q/M paradigm is innovative in that the derived metrics depend heavily upon the goals and characteristics of the specific project or organization. It recognizes that the goals of most projects are different, and seeks to end the dependence of an organization upon a single set of metrics by which all software development efforts must be measured. The G/Q/M and improvement paradigms have been successfully applied to several industrial settings outside of the SEL [Romb87a,Romb87b].

Basili and Rombach are constructing an environment called TAME (Tailoring A Measurement Environment) to support the Quality Improvement and G/Q/M paradigms [Basi88a]. A series of TAME prototypes which support the measurement of Ada projects are currently being developed [Basi87a].

6.6.2 Integration of Measurement Into Software Development

The majority of current measurement programs employ a variety of stand-alone tools to perform metric analysis. However, the coherent specification, collection, and analysis of metrics requires an integrated environment for measurement; integrated in the sense of feeding back metric information into the software development process so that both current and future development can be improved. Too often, the results of metric analysis are used only for evaluating software, and not for learning how to better design, implement, and measure software. The TAME environment (mentioned above) is an encouraging effort aimed at providing an integrated environment in which software process specification languages are used to describe both the development and measurement processes as well as their interfaces.

Selby has developed a set of guidelines for incorporating metrics into a software development environment [Selb87a]. These guidelines address the varying scope of metrics an environment should possess (for example, product as well as process metrics and design as well as code metrics), and also the method for collecting and analyzing metrics.

6.6.3 Needs for Future Automated Support

Automated tools are required for effective and affordable software measurement. While there exist many tools which deal with software metric analysis, few automate the activities of evaluating, predicting, controlling, and learning. This is not surprising given the traditional focus on the use of simple, stand-alone software metrics. Efforts must be directed at developing a complete measurement environment which supports a comprehensive measurement methodology.

7. SOFTWARE RELIABILITY ASSESSMENT TECHNOLOGY

This section discusses software reliability, one of the factors that determine *software quality.* Software reliability is singled out for further discussion for two reasons:

1. Highly reliable software is essential in the SDS, and
2. Software reliability is perceived as being unique among the software quality factors.

Musa, for example, claims that software reliability is "probably the most important of the characteristics inherent in the concept 'software quality' " and "the most readily quantifiable of the attributes of software quality" [Musa87].

7.1 Scope

As its title suggests, this section focuses on how to *assess* software reliability. It is not intended to cover the issue of how to *achieve* software reliability (or, perhaps more accurately, how to achieve reliable software). Nevertheless, to place this section in context, it is worthwhile to at least identify the basic approaches that can be used to enhance software reliability.

Clearly, the most powerful approach is fault prevention. Here, the aim is to improve the software development process itself, so that faults are never introduced into the software. Ideally, the software development process would enable the construction of fault-free software. The entire field of software engineering is directed at improving the software development process.

Another approach to increasing software reliability is to facilitate the detection and correction of faults and errors. Four technologies embodying this general approach are covered in the preceding sections of this report: dynamic and static analysis, formal verification, and software measurement (that is, software quality evaluation).

A third approach to increasing software reliability is software fault tolerance, in particular, software-implemented methods for enhancing tolerance to software faults. This approach was briefly discussed in Section 2.1.3. Its aim is to minimize or eliminate the impact of software faults. The most prominent software fault tolerance methods are recovery blocks [Rand75,Ande76a] and N-version programming [Aviz85,Knig86a]. These methods have been developed in recognition of the fact that software faults cannot be totally prevented or eliminated, but that their impact (at least in the form of critical failures) must be minimized.

7.2 Current Methodology

An overview of the state of the art in software reliability assessment is presented in this subsection. The point of this overview is to clarify what is meant by software reliability, and, moreover, to indicate what type of work is being done in the name of software reliability assessment. For a comprehensive treatment of the subject of software reliability assessment, the reader is referred to [Farr83,Goel85,Musa87,Rama82].

7.2.1 Definition of Software Reliability

The term *software reliability* has taken on a narrow meaning in the software engineering literature, in particular a much narrower meaning than those not conversant with the literature might suppose. According to Musa, *software reliability* is the "probability of failure-free operation of a program for a specified time in a specified environment" [Musa87]. *Software failure*, in turn, is defined as the "departure of program operation from requirements" [Musa87]. At this point, subjectivity enters Musa's stream of definitions: *requirements* are not defined, but only discussed. Musa concludes that they can include both explicit and implicit needs. Thus, a behavior can be classified as a failure on the basis of unstated requirements, which are inherently subjective. *Environment* is equated with *operational profile*, which is defined as "the set of all possible input states (input space) with their associated probabilities of occurrence." As noted in [Rama82], "the software need be correct only for inputs for which it is designed (*specified environment*)."

Finally, although *time* may seem like a straightforward concept, it can be interpreted in a number of ways, to suit the application at hand. According to [Goel85], it "may mean a single run, a number of runs, or time expressed in calendar or execution time units."

7.2.2 General Approach to Software Reliability Assessment

The problem that has received the most attention in the field of software reliability assessment is that of estimating future failure behavior from past failure behavior. In Goel's words [Goel85],

> A commonly used approach for measuring software reliability is via an analytical model whose parameters are generally estimated from available data on software failures. Reliability and other relevant measures are then computed from the fitted model.

The analytical model is referred to as a *software reliability model*. Numerous software reliability models have been proposed [Farr83,Goel85,Musa87,Rama82].

Some work has been done on predicting reliability from properties of the software and the process by which it was developed [Musa87,McCa87a]. This work is not addressed here, because it is so closely associated with software metrics, the topic of the previous section of this report. In particular, the conclusions of the previous section apply to the specific case of software reliability prediction.

7.2.3 Classification of Software Reliability Models

In [Goel85], Goel divides software reliability models into four broad classes:

- *Times Between Failures Models*: In these models, the time between failures is treated as a random variable, drawn from a distribution whose parameters depend on the number of faults remaining in the program. Typically, the time between failures is assumed to decrease as faults are detected (and subsequently corrected). Hence, these models are sometimes referred to as software reliability growth models. Estimates of the parameters are obtained from the observed values of times between failures. Estimates of software reliability, mean time to next failure, etc., are then obtained from the fitted model.
- *Failure Count Models*: In this class of models, the failure process is represented as a stochastic process with a time dependent failure rate. Again, it is typically

assumed that the failure rate decreases over time. Parameters of the failure rate can be estimated from the observed values of failure counts or failure times.

- *Fault Seeding Models*: In these models, a known number of faults is seeded into a program with an unknown number of indigenous faults. Based on the number of seeded and indigenous faults discovered during testing, an estimate of the original (that is, prior to seeding) fault content of the program is made.
- *Input Domain Based Models*: In the basic model of this class, test cases are generated randomly from an operational profile that is assumed to be representative of the real usage of the program. Based on the number of failures observed during execution of the test cases, an estimate of program reliability is obtained.

Models of the first two classes are sometimes referred to collectively as time domain models. Time domain models are the best established and most widely used models.

7.2.4 Time Domain Models

Time domain models are strongly advocated by Musa, Iannino, and Okumoto in [Musa87]. In this book, the authors discuss the theory and application of the models. They present a model classification scheme, describe several specific models in each class, and offer a set of model comparison criteria. The criteria include predictive validity, capability, quality of assumptions, applicability, and simplicity. On the basis of these criteria, two models, both of which fall into the failure count model category, are judged superior to the others: the basic execution time model and the logarithmic Poisson execution time model.

Both models interpret "time" to be execution time, and both provide for a mapping from execution time to calendar time. Furthermore, both assume that the failure process is a nonhomogeneous Poisson process. That is, failures occur according to a Poisson process, with a time varying rate. The rate is referred to as the *failure intensity* (mean number of failures per unit time).

The fundamental concepts underlying the two models is depicted in Figure 7-1, which is extracted from [Musa87]. In the basic execution time model, the failure intensity λ decreases linearly with the mean failures experienced μ:

$$\lambda(\mu) = \lambda_0 \left[1 - \frac{\mu}{\nu_0}\right],$$

where λ_0 is the initial failure intensity and ν_0 is the total number of failures that would occur in infinite time. In the logarithmic Poisson execution time model, the failure intensity decreases exponentially:

$$\lambda(\mu) = \lambda_0 \, exp(-\theta\mu),$$

where λ_0 again represents the initial failure intensity and θ is the *failure intensity decay parameter.* As explained in [Musa87], the basic execution time model represents the case in which the discovery of each failure (and subsequent repair of underlying faults) leads to a constant reduction (of 1 divided by the total number of failures) in failure intensity. The logarithmic Poisson execution time model, on the other hand, represents the case in which early failures lead to greater reductions in failure intensity than later failures. In other words, the "benefit" accrued by repair processes (initiated in response to failures) decreases exponentially as a function of the number of failures.

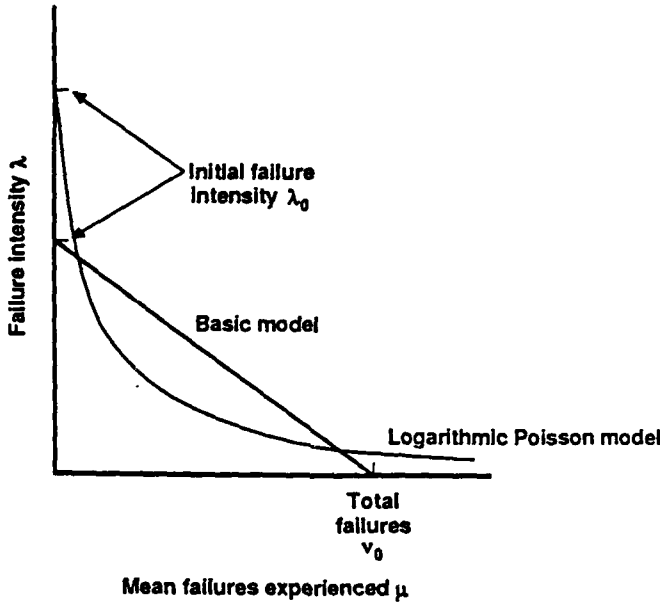

Figure 7-1. Failure Intensity Functions

As indicated in the figure and equations, each model has two parameters, one being the initial failure intensity λ_0 and the other representing failure intensity change (total failures ν_0 in the basic execution time model and failure intensity decay parameter θ in the logarithmic Poisson execution time model). The values of these parameters may be estimated from failure data via standard statistical techniques [Musa87]. In addition, for the basic execution time model, parameter values may be predicted, prior to execution of the software, from characteristics of the software [Musa87].

Several useful measures can be derived from these models. They include the expected number of failures to reach a specified failure intensity objective and the expected execution time to reach a specified failure intensity objective. These measures are also detailed in [Musa87].

7.3 Critique of Current Methodology: Some Fundamental Problems and Limitations

Current software reliability assessment technology suffers from some fundamental problems and limitations. These seem to stem from the fact that software reliability assessment has not established itself as a discipline in its own right; current technology remains bound to the hardware reliability assessment technology from which it evolved.

7.3.1 Evolution from Hardware Reliability Assessment

The field of software reliability assessment evolved from the field of hardware reliability assessment. But there are key differences between software and hardware that limit, or *should* limit, the extent to which the hardware concepts can be applied in the software world. Some of these differences and their implications are described below.

7.3.1.1 Source of Failures

As noted in [Musa87], the source of hardware failures (at least of the hardware failures traditionally

addressed in hardware reliability assessment) is the physical aging and deterioration of hardware components. The source of software failures is software faults, which are manifestations of design and implementation errors. Hardware failures resulting from faults introduced by design errors share more in common with software failures than with age-related hardware failures. Thus, the distinction could be made on the basis of aging versus design instead of on the basis of hardware versus software.[5]

This observation has a critical, but largely unheeded, implication. Hardware reliability is inherently time and usage dependent (as in 5-year, 50,000 mile automobile warranties). Thus, it makes sense for hardware reliability to be defined in terms of failure-free operation over a specified "exposure" time period.

Software reliability, on the other hand, is not directly dependent on time. First, consider software faults, which are the source of software failures. Assuming that the software is not changed, software faults are time-invariant. That is, software faults are introduced during design and implementation; new software faults do not arise because of the passage of time or occurrence of processing. Now consider software failures. Software failures are dependent on inputs. For a given (deterministic) program and a given input state, either the software always operates correctly or it always fails. Time is not a factor, except in the sense the software may be exposed to different inputs over time. Given the same operational profile, the software reliability does not vary with time.

So, while time (in some unit of *exposure*) appears to be the "right" independent variable for hardware reliability assessment, it is not so natural for software reliability assessment. Software reliability is dependent not on time directly, but on (1) the *presence* of faults and (2) the *exposure* to faults, more precisely, the exposure to input states that lead to execution paths on which faults are encountered. Both of these factors have to be taken into account in assessing software reliability.

This view is reflected by Parnas [Parn88]. He suggests two complementary probabilistic measures of software quality: reliability and trustworthiness. Software reliability is defined as "the probability of not encountering an input history that causes a failure"; software trustworthiness is defined as "the probability that no serious design error remains after a set of randomly chosen tests [have been] passed."

Therefore, the emphasis on time domain models of software reliability may not be appropriate, especially in the case of highly reliable software. For highly reliable software, the goal is not to estimate measures such as failure rate, but to assure that critical failures cannot occur.

7.3.1.2 Target of Assessment

In hardware reliability assessment, the basic targets of assessment are relatively low-level components, such as memory chips. The reliability of a given class of components (for example, a given type of memory chip or a given batch of a given type of memory chip) is established by sampling from the population of components in the class. The components in a class are identical in design but are distinct physically. It is assumed that the components of a class have the same reliability, but that failures in the individual components are independent.

The low-level hardware components are used as building blocks in constructing higher level

5. However, in keeping with general practice, this section continues to use the hardware/software categorization, with the understanding that hardware reliability is being used in the traditional sense of age-related reliability.

components, or systems. The reliabilities of the building blocks can be used to estimate the reliabilities of the system [Shoo86]. For example, the reliability of two series-connected components (that is, two components, both of which must operate correctly) is the product of the reliabilities of the two components. Again, the independence of failures in different components is the underlying assumption.

Combinatorial analysis is at the core of hardware reliability assessment. In fact, it is the building-block approach in conjunction with the combinatorial analysis that supports the construction of hardware systems with specified levels of reliability.

In software reliability assessment, on the other hand, the basic targets of assessment are relatively high-level components. As noted in [Musa87],

> In general, it appears that these models can be applied to any type or size of software project, with the following exception. Very small projects (less than about 5000 lines of code) may not experience sufficient failures to permit accurate estimation of execution time component parameters and the various derived quantities.

Moreover, the high-level components are "unique." Instead of being designed as building blocks, they are application-specific. Thus, the combinatorial analysis that is fundamental to hardware reliability assessment does not carry over to software reliability assessment.

This distinction between hardware reliability assessment and software reliability assessment is recognized by McCall, *et al.* Their articulation of the problem, repeated here for emphasis, is as follows [McCa87a]:

> Hardware components consist of separate parts, each of which may be used in many other applications, such as a 1A 250V diode or a 16k dynamic [Random Access Memory] RAM chip. Failure rates can be established for these parts either from test or from analysis of field data. The procedures of MIL-STD-756B [*Reliability Modeling and Prediction*] assume that the reliability of a component is the product of the reliability of its (series-connected) parts. The software analog to this would be to test individual assignment, branching, and [Input/Output] I/O statements and to declare the reliability of a procedure to be the product of the reliability of its individual statements. This analog is faulty because: (a) statements cannot be meaningfully tested in isolation and (b) many software failures arise not from faults in a single statement, but rather from interactions between multiple statements (or from interactions between hardware and software).

The point is that the ultimate objective of traditional hardware reliability assessment — the construction of systems with specified levels of reliability — cannot be accomplished in the software domain in the same way that it is in the hardware domain.

7.3.2 Applicability to Life Cycle Phases

In [Goel85], Goel discusses the applicability of software reliability models to the following phases of the life cycle: (1) design phase, (2) unit testing, (3) integration testing, (4) acceptance testing, and (5) operational phase. Based on assumptions made by the various models, which he enumerates in the article, he comes to the following conclusions. Integration testing is the only phase where all four categories of models — times between failures models, failure count models, fault seeding models, and input domain based models — are applicable. None of the models is applicable during the design phase, because of the lack of test cases and failure history. None is applicable in practice during unit testing, although fault

seeding models and input domain based models are applicable in theory. During acceptance testing, the fault count and input domain based models are applicable, while the others are not. Finally, during the operational phase, only the fault count models are applicable.

The point here is that the current methodologies are applicable only to software that is being executed, and, moreover, only at integration testing and later in the life cycle.

7.3.3 Applicability to Highly Reliable Systems

The acceptance testing of highly reliable software, which is a crucial aspect in the development of the SDS, presents its own unique problem. Highly (or ultrahighly) reliable software should exhibit no (critical) failures during acceptance testing. In this case, a "failure history" does accumulate, but it is one of no failures. Are the models applicable to highly reliable software? Or are they applicable only to "unreliable" software? The problem is captured in the following dilemma posed by Knight [6]:

> Hypothesis 1. For a system that is required to achieve *very* high reliability, if *any* failure occurs during verification testing, then the system will never achieve the required level of reliability.[7]
>
> Hypothesis 2. If a system does not fail during testing no reliability assessment is possible because there is no data.

Hypothesis 2 is clearly an overstatement. Statistically valid conclusions can be drawn from the lack of failures. According to Parnas [Parn88], testing can *in theory* be used to establish trustworthiness in software. However, as he warns, the amount of testing that would be required *in practice* is simply prohibitive.

7.3.4 Traditional Uses of Software Reliability Assessment

At this point, it is appropriate to consider the role that software reliability assessment traditionally plays in the software engineering process. In [Musa87], Musa, Iannino, and Okumoto cite the following four uses of software reliability assessment:

- To (quantitatively) evaluate software engineering technology;
- To evaluate development status during the test phases of a project;
- To monitor the operational performance of software and to control new features added and design changes made to the software; and
- To gain insight into the software product and the software development process, through a quantitative understanding of software quality.

Software reliability modeling has proved to be effective in the cited uses (especially the second and

6. This dilemma was posed at the IDA Testing and Evaluation Workshop that was held in support of this report. While it could of course be more carefully stated, it does make its point.
7. Hypothesis 1 holds only if very high reliability means 100% reliability, or absence of failures.

third), which are valuable in certain environments, such as the American Telephone & Telegraph (AT&T) environment in which Musa and others have applied the models.

7.4 Conclusion

This section presents a summary evaluation of current software reliability assessment technology, based on the above critique. It then suggests directions for future research. Finally, it closes by attempting to put software reliability assessment into a proper perspective.

7.4.1 Summary Evaluation of Current Software Reliability Assessment Technology

Clearly, the preceding critique of software reliability assessment technology raises questions about the applicability of the current failure-history-based methodology to highly reliable systems. As Goel contends in a position statement prepared for the IDA Testing and Evaluation Workshop [Bryk89]:

> The current methodology for evaluating software reliability is based on a very restricted premise, viz, the future error occurrence phenomenon is a stochastic extrapolation of the recent past. This approach is too simplistic and is not likely to be very useful for ultra-high reliability systems, such as [the] SDS.

More significantly, this critique raises questions about the philosophy and assumptions underlying today's software reliability assessment technology. It can be argued that current software reliability assessment methodologies, as well as the definition of *software reliability itself*, are "artifacts" of the evolution of software reliability assessment from hardware reliability assessment. Design faults (be they hardware or software) demand a new approach; they cannot be treated adequately by the same methodology that was developed for age-related faults.

The current definition and methodologies constrain what can be done in the context of software reliability assessment. In particular, current methodologies can be useful only as management tools, in accomplishing purposes such as estimating project schedules, optimizing the allocation of project resources, and optimizing the timing of new releases of software. Granted, these are worthwhile purposes; the acceptance of current software reliability assessment technology stems, in large part, from its success as a management tool in exactly these types of applications.

However, the ultimate goal of any reliability assessment technology should be to facilitate and assure the construction of systems of specified levels of reliability. Because of the fundamental limitations of current software reliability assessment methodologies in this regard, further work on enhancing current methodologies is unlikely to yield satisfactory results.

7.4.2 Future Directions of Software Reliability Assessment Technology

In order for software reliability assessment technology to contribute significantly to SDS development, the software engineering community must support the evolution of the concept of software reliability. Software reliability assessment must move beyond the goal of estimating future failure behavior based on past failure behavior toward the goal of *constructing* reliable software and, ultimately, reliable systems.

7.4.2.1 Assessment of the Software Development Process

Just as it is not possible to "test" correctness into software, it is not possible to test reliability or trustworthiness into software. So, what can be done? As suggested by Parnas [Parn88] and Evangelist [Bryk89], the solution lies in the software development process. Parnas maintains [Parn88]:

> Software can be used in safety-critical applications but extreme discipline in design, documentation, testing and review are needed. Standard practice is not adequate.

Because of the inevitable reliance upon the software development process as the most powerful means of constructing reliable software, software reliability assessment technology should shift focus — from assessment of the reliability of individual software products to assessment of the reliability of software engineering methodologies, practices, tools, and techniques.[8] Then, the software engineering community could begin to approach the ultimate goal of the *construction* of reliable software.

Of course, this is a most challenging task. It involves capturing, recording, and analyzing features of the software development process, throughout the life cycle. In most of the software measurement work that has been done to date, the completed software *product* rather than the entire software development *process* is the target of measurement and analysis. Efforts have concentrated on measuring isolated, low-level features of software products. While the low-level features are readily measurable, the significance of their measured values is questionable.

In undertaking the task of assessing the software development process, the following points should be kept in mind:

- Methodologies and practices, in order to be compared, must be rigorously defined and faithfully followed.
- "Desirable" properties of software, such as reliability, must be identified; then, effective measures, which can quantify these properties, must be defined.
- Effective experimental design must be employed.

7.4.2.2 Assessment of Software Reliability in a System Context

Software reliability must be assessed in a system context. As noted in Section 2.1, the correctness of software for distributed, real-time applications cannot be established in isolation from the underlying computing system, for two distinct reasons. First, in real-time applications, the correctness of software entails not only the values of results, but also the time at which results become available. Timing, of course, is a function of the software, as well as of the underlying computing system.

Second, in complex distributed real-time systems, hardware components are bound to fail, software faults are bound to exist, and unexpected inputs are bound to occur. The software, as the *controlling* element of the system, must be designed to deal with these faults and failures. The software must provide for system fault tolerance and graceful degradation.

---

8. Here, the reliability of a methodology means the reliability *afforded by* the methodology to the software product being developed.

Therefore, just as the correctness of software depends on its ability to meet timing constraints, its correctness also depends on its ability to degrade gracefully in the face of faults, failures, and unexpected inputs. The problem lies in the quantification of "graceful degradation." Intuitively, the concept involves a mapping of the domain of potential faults/failures into a range of "mission impairment." For a specified subset of the domain, the software should be able to assure a mission impairment of zero. Beyond that subset of the domain, the mission impairment should not rise "dramatically," but "gracefully."

The issue of fault tolerance must be addressed in all phases of the life cycle:

- Potential faults and failures must be identified at the outset.
- The required level of fault tolerance must be specified, perhaps in terms of the "graceful degradation" curve.
- Fault tolerant techniques must be incorporated into the software, to provide for the required level of fault tolerance.
- The impact of faults and failures, as well as the effectiveness of fault tolerant techniques, must be assessed during design (possibly through simulation), as well as during testing.
- The reliability afforded by various fault tolerant techniques must be assessed, in the same way that the reliability of other software engineering methodologies should be.

7.4.2.3 Assessment of System Reliability

Finally, software reliability must be taken into account when assessing system reliability. Often, system designers assume a software reliability of 100% when evaluating system reliability [Parn88]. Such an assumption is clearly unreasonable. Innovative approaches are needed here. It is not sufficient to follow the much touted practice of simply casting software reliability in hardware reliability terms and then using combinatorial analysis to derive system reliability. The distinction between design faults and age-related faults must be considered.

7.4.3 Caveat

In closing this section, it is appropriate to place software reliability assessment (and, more generally, software testing and evaluation) in perspective. Specifically, it is important to recognize or acknowledge what software testing and evaluation can *not* accomplish, so that unrealistic expectations do not prevail.

Consider the concept of software reliability. In its broadest sense, software reliability is equated with the probability of "mission success." It is assumed that highly reliable software will successfully perform the intended mission. This sense of the concept is intuitively appealing, as evidenced by the fact that discussions on software reliability, especially in the context of SDS, almost always degenerate into discussions on the feasibility of building a successful system; but, it leads to inflated expectations of what software reliability assessment, as well as software testing and evaluation, can accomplish. The probability of mission success depends on factors that fall outside the scope of software and the traditional software testing and evaluation process. These factors include the following:

- Validity of Input Assumptions: Software is developed to respond to specified inputs, in the case of the SDS, for example, a specified threat. If the specified threat differs from the actual threat, then the fact that the software is "ultrahighly reliable" for the specified threat indicates little or nothing about the reliability of

the software for the actual threat.

- Validity of Hardware Assumptions: Hardware (and, more generally, environmental) assumptions have the same impact as input assumptions. If they are invalid, then the reliability of the software with respect to those assumptions indicates little about reality.
- Effectiveness of Strategy: Strategy is embodied in software. How can the reliability (or effectiveness) of a strategy be quantified? Are probabilities of success computed for non-automated (defensive or military) strategies? Software reliability can hardly be expected to subsume quantification at this level.

In short, traditional software testing and evaluation can offer some assurance that a software product accurately implements a proposed solution to a specified problem, but cannot assure that the problem is adequately specified or that proposed solutions are in some sense "good." As pointed out in Section 2, the most effective approach for dealing with these difficult issues is an iterative development approach, incorporating formal design specifications, simulation, and prototyping. This approach has been adopted by the SDIO. It is important that software testing and evaluation research and development focus on supporting this approach.

8. RECOMMENDED TASKS TO EXPLOIT EXISTING TECHNOLOGY

This section of the report identifies a number of tasks required to ensure that SDS software testing and evaluation attains the maximum effectiveness and efficiency achievable within the current bounds of technology. These tasks do not fall into classifications of dynamic and static analysis, formal verification, measurement, or reliability assessment. Instead, they cut across these distinctions to provide a common framework into which desirable elements of each type of available technology will fit. Tasks to extend technology are discussed separately in Section 9.

The gap between the state-of-the-art and practice in testing and evaluation is very wide. Consequently, it should be possible to effect a substantial improvement in software reliability by requiring the use of a core set of advanced techniques, supported by high-quality, effective automated tools.

To achieve this goal, this section outlines four major tasks. The first task focuses on the critical need to integrate testing and evaluation activities into software development as a whole. It also addresses how software developers should be provided with explicit guidance as to which techniques are appropriate under certain circumstances and how these techniques should be applied to achieve the necessary confidence in results at acceptable cost. The second task discusses the requirement for a SDS Software Data Collection System which will capture and assess information about not only the software being developed, but the technology used to develop, test, and support that software. The third task identifies some of the issues involved in the development of an automated environment to support testing and evaluation of SDS software. In many respects, the final task provides crucial support for the three preceding ones. It concerns the exploitation of process modeling techniques to both explore and specify effective, flexible ways of integrating testing and evaluation into software development activities. While primarily intended to exploit available technology, these tasks do themselves require some advances in technology. For example, although a sophisticated environment will, at least initially, support application of available testing and evaluation techniques, its development requires increased understanding of the ways in which techniques can be cooperatively applied.

Before proceeding to discuss these tasks in more detail, a word of caution is appropriate. There are many important research results which have never had the benefit of significant prototype development and exploration. There is an important need for more thorough experimentation with these research ideas, and this is best accomplished by transferring the ideas from the academic research setting in which they were developed to advanced technology development laboratories. New technology must initially be applied on a few selected software efforts before being required for general practice. This will provide an opportunity to carefully monitor the application of the technology to determine its benefits and costs in both technical and programmatic senses. These testbed sites should be typical software development efforts where software developers work under realistic deadlines to develop "real" code. The introduction of new technology can incur cost and schedule penalties. These risks must be reflected in contracts and software developers provided with incentives to explore the full potential of the new practices. Additionally, before new technology is inserted into SDS practices, the appropriate policies and organizational support must be in place. Technology transition and insertion is a difficult and expensive activity. It is, however, a necessary prerequisite to advancing the state-of-the-practice. A separate, strongly funded technology transfer effort that runs in parallel with R&D efforts must be instituted.

Although primarily intended for SDS software, the technology discussed here offers increased effectiveness for all software testing and evaluation efforts.

8.1 Test Planning and Testing Requirements

An initial model of a possible, promising SDS life cycle development process was given in Section 2.3. The role that a sample set of the various techniques discussed in this report play in the testing and evaluation embedded in this process is shown in Figure 8-1. This figure is not meant to imply that all the identified techniques are those specifically recommended for use, but to illustrate the types of testing and evaluation techniques that can be applied during various development activities. It provides an *initial* picture of how testing and evaluation activities can be effectively integrated into the development process to provide timely feedback on development activities. Many issues require further investigation. For example, what role should simulation play in the testing and evaluation scheme? A specific set of candidate techniques suitable for each stage of software development should be identified. As data on the respective costs and benefits of these candidate techniques becomes available, those required for SDS software testing and evaluation should be determined.

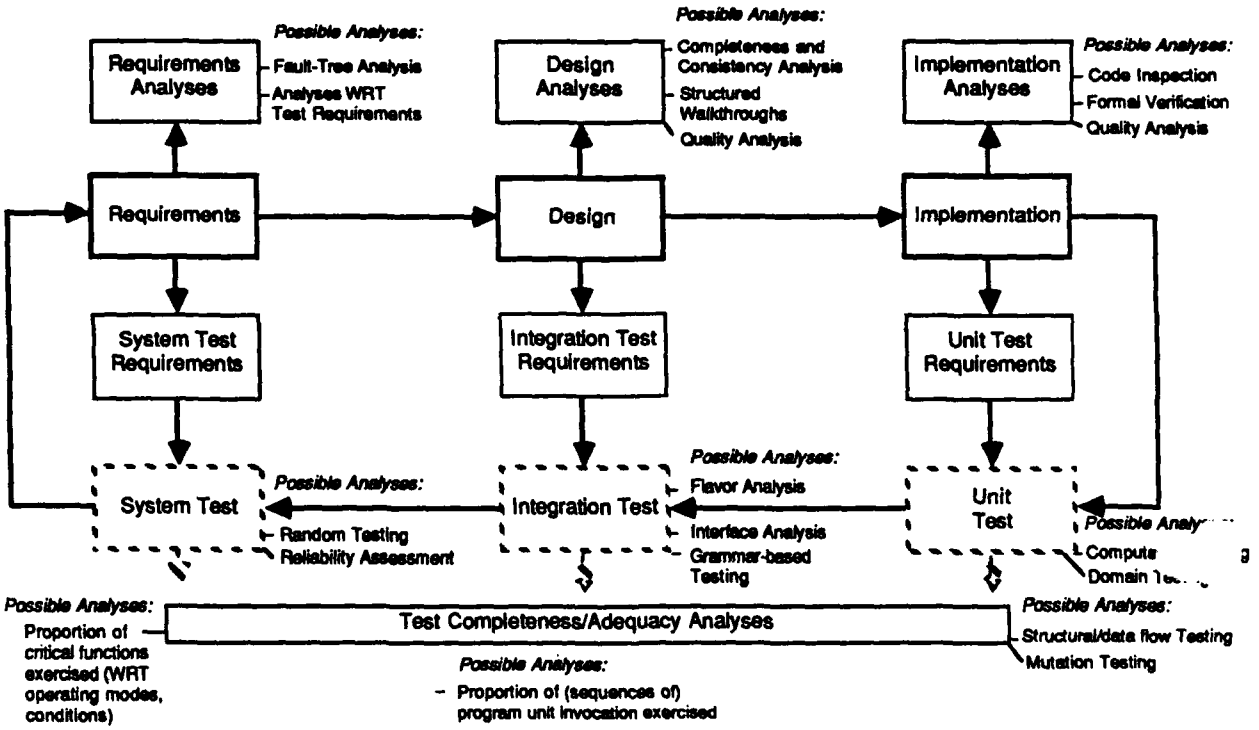

Figure 8-1. Process Model and Candidate Techniques

In particular, for SDS, this process model should be extended to provide support for determining which SDS engineering products must be subject to testing and evaluation, and the level of effort required for different products. Additionally, the development process model must be supported by a model of the improvement process which defines such activities as assessing engineering trade-offs to determine, for example, where testing dollars should be allocated.

One of the mechanisms proposed for implementing this model is the formalized use of system and software test plans. It is vital that the necessary programmatic practices to implement this test plan concept be investigated.

The other mechanism is the use of formal testing requirements imposed at each development stage. These requirements should provide precise specification of testing objectives which enables quantitative assessment of both testing and evaluation progress and outstanding needs. They should not only guide the selection of the appropriate techniques for software under test, but also guide the application of these techniques. Traceability from system requirements, through testing requirements, down to individual testing objects and activities is necessary to support both monitoring of the overall test and evaluation status and to facilitate regression testing.

Notations and practices for specifying and using testing requirements must be developed. Although the requirements will guide the use of existing technology, identifying what information should be captured, how it should be represented, and how used is a difficult problem. For example, one of the simplest ways (though in practice insufficient by itself) to specify the minimum level of dynamic analysis for a piece of sequential software is to provide structural coverage measures. Different types of coverage measures are appropriate for unit, integration, and system testing. Whereas coverage for unit testing is typically assessed based on the control and data elements exercised, the proportion of program unit invocations exercised, or the proportion of possible sequences in which they are invoked, is a better measure for integration testing. A further degree of abstraction is appropriate for assessing the coverage of system testing. Here coverage measures reflecting the system functions exercised should be used, where functions are ranked to reflect their criticality to the system mission and the possible severity of the consequences of their failure.

Even if simple coverage measures were a sufficient means for specifying the required degree of dynamic analysis, hierarchies of coverage measures balancing the extent of required testing against the criticality of the software under test and the cost of achieving different levels of coverage would be needed. There are hierarchies which can be exploited for this purpose, but there is little experience to support mapping levels of criticality to levels of required coverage. What are the factors that determine criticality, what discriminates between different degrees of criticality, and how should criticality be stated? Having determined the coverage required, which dynamic analysis techniques should be employed, how extensive should the test data used for each be?

A related issue concerns integrating the results of testing and evaluation into different stages of development, and using different techniques, to determine the sufficiency of completed testing and evaluation and to provide an overall view of the software status. For example, in the case of measuring software properties, the use of a common, underlying base set of metrics is vital since trade-offs between software properties are inevitable and the overall characteristics of the system will be largely determined from evaluation of individual subsystems and components. How should testing requirements and results be captured as permanent attributes of a program, supported by sufficient details to repeat the testing at need? How can the flexibility necessary to allow evolution of the testing practices embedded in the development process be provided? These, and other, questions must be addressed before the testing requirements mechanism can be institutionalized.

8.2 SDS Software Data Collection System

It is extremely important that a program-wide SDS software data collection system (SSDCS) be established. The SSDCS should be similar in nature to the measurement and collection capabilities of the NASA SEL. As such, it will support analysis of not only SDS software, but serve as a valuable resource for the better understanding and advancement of the technology used to develop, test, and support that software.

The SSDCS will provide the focal point for investigating the composition, effectiveness, and

applicability of the SDS software measurement program. An overall SDS measurement strategy must be developed which provides proper support for the understanding, evaluation, and control of SDS software. This strategy must include a validated set of metrics collected within a measurement methodology which is integrated into development activities. In addition, the measurement strategy must be continuously monitored for effectiveness and applicability. The SSDCS will perform experiments and demonstrations to develop, tailor, and validate those elements of the measurement strategy.

Although primarily intended to improve understanding of existing technology, data which supports the advancement of technology should also be collected. An initial set of technology questions, or goals, which need to be addressed is shown in Figure 8-2. One of the lessons learned in metrics research is to collect only the data that are needed for a specific purpose. A large volume of data often leads to superficial analysis, which does not contribute to understanding. Consequently, the data required to achieve the technology-related goals of the SSDCS must be clearly defined, and the costs associated with collecting and analyzing that data weighted against the potential benefits. While a part of the data collection will, of necessity, require manual procedures, development and testing environments must be instrumented to automate data collection to the maximum extent possible. Instrumentation requirements for these environments must be defined. The specific programmatic support needed to establish and maintain an SSDCS must also be determined; including consideration of the diverse topics of organizational roles and responsibilities, contractual implications, and policy directives.

8.3 Automated Testing and Evaluation Environment

As the field of testing and evaluation technology matures, automated support is no longer merely desirable, but increasingly a prerequisite for the application of today's sophisticated techniques. Software developers need access to a collection of useful testing and evaluation tools with the capability to build an evolving picture of the status of the software under test.

The provision of a comprehensive testing environment which supports all aspects of preparing for, executing, and reporting on testing and evaluation activities is a high-priority goal. Requirements for the testing environment must be determined. Issues of technique integration, generic components, incremental support, language independence, user interaction models, and environment support have been identified previously. Another important concern is the relationship between the different forms of testing and evaluation technology. For example, dynamic analysis approaches traditionally employ inductive methods, whereas formal verification employ deductive methods. This distinction is narrowing as more dynamic analysis techniques use deductive methods to identify the test data necessary to execute selected paths. Thus, the symbolic evaluation that is the front-end of formal verification is becoming a crucial element of many analysis techniques. Similarly, some testing and evaluation activities can borrow from compiler technology, or even be provided through compiler extensions. Test management should be embedded in the environment, leading to proactive tools which guide the user in the application of appropriate techniques for the case in hand. (Process programming should be investigated as a mechanism for achieving this and earlier stated goals.) Finally, the development of efficient testing algorithms which can exploit the capabilities of supercomputers to facilitate the use of computationally intensive testing techniques should also be examined.

Development of such an environment is a significant undertaking, and the possibility of building on

9. Note: These objectives were first identified by Miller in late 1970's [Mill79a], though here are slightly modified.

Dynamic and Static Analysis Objectives:

1. Develop a series of well-understood weighting of programs that distinguish them in terms of their analysis difficulty.[9]
2. Collect data to confirm/refine capability profiles on techniques and tools.

Formal Verification Objectives:

1. Determine model parameters for estimating costs and schedules for development of formally verified software components.

Measurement Objectives:

1. Support development of tailorable metrics for specific application domains.
2. Build metric-based models of the development processes and products and use them for improvement.

Reliability Assessment Objectives:

1. Support development of a model that predicts reliability from the characteristics of the software development and testing process.[9]

General:

1. Gather significant experience with testing and evaluation of large systems that reveals empirical principles which can minimize its cost, or increase its effectiveness at the same cost.[9]
2. Develop a psychology of testing and evaluation which aids in designing and implementing organizations for its effective performance.[9]
3. Evaluate the cost and benefits of the technology introduced.

Figure 8-2. Technology Objectives for the SSDCS

existing efforts should be carefully investigated. The following is a list of only a few of the current efforts that are of particular interest in this respect:

- Dynamic and Static Analysis:
  - — The Arcadia and TEAM environments, and
  - — The system development environment from Stanford University.
- Formal Verification:
  - — The Annotated Verifiable Ada (AVA) system being developed by Computational Logic, Inc.
  - — The Ulysses project at Odyssey Research Associates.
- Measurement:
  - — The Software Metrics Data Collection System,

- — Basili's and Rombach's goal-oriented approach, and
- — AdaMAT.

- Reliability Assessment (the measurement efforts noted above also apply here):
  - — The RADC Software Reliability Measurement Framework,
  - — Revision of MIL-STD-785B (Reliability Program for Systems and Equipment Development and Production), by EIA Committee on System Reliability (G41), Subcommittee on Software Reliability, and
  - — IEEE STDs 982.1 (Dictionary of Measures to Produce Reliable Software) and 982.2 (Guide for the Use of Measures to Produce Reliable Software).

While the testing environment must be available prior to full scale development of SDS software, its availability is unlikely to significantly precede the full scale engineering phase. In the interim, a primitive environment should be assembled from available tools. The primary goals of this interim environment are to (1) provide immediate support to ongoing SDS efforts, (2) start the technology transfer process, (3) support collection of quantitative information on the capabilities of current techniques and tools, and (4) early experimentation with processes aimed at effective integration of testing and evaluation in the development life cycle. Although unable to provide the efficient and effective testing and evaluation expected for a carefully defined environment, such a collection of tools offers a significant improvement over current practices.

Examples of a few candidate tools (taken from those mentioned in this report) for inclusion in such an interim environment are:

- Dynamic and Static Analysis:
  - — The AdaPIC toolset, and
  - — The MOTHRA mutation testing system.
- Formal Verification:
  - — The Gypsy Verification Environment.
- Measurement and Reliability Assessment:
  - — The TAME Environment.
  - — The Ada Test and Verification System (ATVS).

These example candidates have been selected on the basis of (1) support for testing and evaluation of Ada code, and (2) the possibility that they require less than 6 man-months of effort to reach at least advanced prototype status. Additional candidates must be identified and all evaluated with respect to SDS software testing and evaluation needs. The cost to apply the tools and provide them routinely to software developers on a Government Furnished Equipment (GFE) basis should also be investigated. A query to researchers in the different areas of testing and evaluation to identify additional candidates has already been initiated.

8.4 Process Modeling

Testing and evaluation processes must be well-specified. This is necessary to allow such benefits as universally understood testing and evaluation practices and meaningful monitoring of testing and

evaluation activities. It is also necessary to facilitate the adoption of testing and evaluation practices which can expand and grow with technology and to further the understanding of that technology. The tasks to exploit technology just discussed exemplify this need. Indeed, to be effective, they *require* these capabilities as a necessary precursor.

Process modeling, in particular process programming, should be investigated as a mechanism for achieving these goals. Existing techniques and experience in process programming, such as those gained on the Arcadia project, should be exploited for this purpose. An activity to define SDS testing and evaluation processes should be undertaken. An additional activity to investigate the specification of an effective, flexible SDS software development model that fully integrates testing and evaluation activities is also recommended.

9. EXTENDING THE BOUNDARIES OF TECHNOLOGY

The tasks in Section 8 provide a framework for bringing existing technology into SDS practice and promoting a better understanding of the basic interrelationships between development and testing and evaluation activities. This approach to carefully considered innovation holds great promise. Even so, current gaps in testing and evaluation technology preclude confident deployment of a reliable SDS. Fundamental research to resolve critical deficiencies is urgently required. In particular, technology for testing and evaluation of large, concurrent and real-time software is largely nonexistent for practical purposes.

Here again, three major tasks, or more properly task areas, are recommended. First, the SDS faces specific technical problems which require practical solutions in the relatively near-term. A series of technology demonstrations to investigate the capabilities of emerging technology to solve these problems is proposed. The second task area presents a number of areas where fundamental R&D is needed to address shortfalls in technology. Finally, a series of tasks to monitor ongoing testing and evaluation research efforts is recommended. All these task areas concentrate on well-focused research tasks to be conducted over the next 5 years. It is expected that these tasks will lead to the recognition of additional R&D efforts which should be supported over a much longer timeframe.

Unfortunately, the software testing and evaluation research community is too small and too weak at present to rise to the challenges of SDS software testing and evaluation. The community must be strengthened and expanded as quickly as possible. This report has identified the need for expanded research, development, technology transfer and productization. All these require significant infusion of resources. More the just money is needed, however. If contracts were let to perform all of the work that is needed, there are not enough researchers in a position to perform the contracts. The SDIO should consider taking the lead in encouraging other DOD agencies to join with them in building up the testing and evaluation research community to attack the critical problems surrounding highly reliable software.

9.1 Technology Demonstrations

There are a number of areas where the practical use of emerging technology could provide increased understanding of that technology to facilitate its advancement. Similarly, when researchers and software developers are required to address specific problems in a practical arena, they are likely to gain increased understanding of the problem which, in turn, provides valuable insights into possible solutions, or supports the development of a working solution pending necessary theoretical advances.

A series of technology demonstrations which require solutions to specific technical problems is recommended. This proposal is in keeping with the planned development of SDS; current activities, as a whole, are either technology demonstrations or experimental developments. Testing and evaluation technology demonstrations should be conducted on recognizable components of the SDS with a view to *potentially* providing practically useful products. There are several goals which should be applied to all demonstrations, such as requiring increased use of formalism throughout the life cycle.

An initial list of candidate problems to be investigated is given in Figure 9-1. In each case, the specifics of a suitable technology demonstration must be determined so that a decision to pursue a demonstration can be firmly based on projected costs and benefits. Not all of the problems must be addressed individually; the ability to define demonstrations which tackle a combination of problems must be investigated. The possibility of exploiting current software efforts (both SDS and other DOD efforts) to provide vehicles for these demonstrations should be considered.

9.2 Specific Research and Development Tasks

Resolution of the major gaps in testing and evaluation technology requires fundamental advances in the underlying concepts. While the SDIO should not be requiring research simply for research's sake, some *pure* R&D efforts are necessary to find answers to SDS software testing and evaluation problems.

Brief descriptions of an initial set of recommended R&D tasks are given below.

- **Dynamic and Static Analysis Problems:**
  - — Develop techniques for designing testable software and measuring achieved degree of testability.
  - — Develop/investigate specification languages and techniques which can provide an oracle capability to support the dynamic analysis of products from later development activities.
  - — Develop a hierarchy of coverage measures which map against levels of criticality in SDS software.
  - — Perform a fault tree analysis of a sample SDS element and use to determine how to specify impact on testing requirements and the need for fault-tolerance, fail-soft, and fail-safe mechanisms and procedures.
  - — Identify SDS software elements which require permanent runtime self-test, develop methods for specifying self-test requirements and techniques for validating achievement of these requirements.
- **Formal Verification Problems:**
  - — Verified Ada run-time support systems.
  - — Verified secure communications over noisy channels.
  - — Verified distributed and autonomous systems.
- **Measurement and Reliability Assessment Problems:**
  - — Development of a Comprehensive Measurement Methodology.
  - — Integration of Measurement into Software Development.
  - — Automated Tool Support.

Figure 9-1. Candidate Problems to be Addressed in Technology Demonstrations

9.2.1 General R&D Tasks

1. **Increased Formalism for Early Lifecycle Products.** Current specification technology does not support timely feedback to development activities or early identification of errors. The SDIO has recognized this problem and provided one step forward by requiring use of an architectural design language (SADMT) which supports design-to-simulation capabilities. This report itself has identified the need for formal testing requirements. Formal languages for specifying both system and software requirements and designs are also needed. Existing languages should be investigated to identify a minimal subset which can be recommended for use on SDS efforts.

Existing testing and evaluation technology that can be applied to these pre-implementation descriptions should be identified. It is, however, likely that new techniques and tools geared towards these more abstract descriptions will be required. Although it is doubtful that a single, say, system requirements language will be sufficient for the needs of all the diverse types of SDS elements, a small common subset of system requirements languages is desirable so that researchers and software developers can focus on supporting only a few languages. The special needs for each type of language should be identified.

2. **Testing and Evaluation Process Programming.** Process programming [Oste87a] involves encoding software development processes in a rigorous language, such as a programming language. Such formalized specification of testing and evaluation processes offers several benefits. For example, when the necessary interactions between software developers and testing and evaluation tools are captured, it becomes possible for a testing environment to explicitly guide testing activities, rather than merely respond to the commands given by a software developer. Moreover, the act of writing process programs promotes a better understanding of the underlying activities and allows this understanding to be widely disseminated. While high-level testing and evaluation process programs can be independent of particular software efforts, at a more detailed level they must usually be tailored to a specific effort or testing environment. Indeed, the maximum benefits from testing and evaluation process programs will only be obtained when a testing environment is designed to exploit this capability, and vice versa. Consequently, an activity to define and develop process programs to support SDS software testing and evaluation activities should proceed in parallel with the development of an SDS testing environment (see Section 8.3). Over the last few years, the researchers undertaking the development of the Arcadia and TEAM environments have gained much experience in process programming which could be used to facilitate this task.

3. **Regression Testing.** A relatively small task is needed to address the issue of regression testing. As previously discussed, the SDS will be subject to continually changing requirements and operating environments. Thus, considerable effort will be invested in retesting and reevaluating the software. Efforts to facilitate this activity, and reduce its scope, offer potentially enormous payoffs. One of the most important issues is that of *sensitivity focus*; from the earliest stages, all products should be analyzed to determine whether the regression testing required by a change to the software is proportional to the scope of that change. Additional issues particularly pertinent to regression testing include traceability of system requirements through to testing and evaluation objects/activities and the recording of testing and evaluation histories.

9.2.2 Dynamic and Static Analysis R&D Tasks

1. **Techniques and Tools for Analysis of Concurrent and Real-Time Software.** There are large research areas of critical importance to SDS testing and evaluation which are still largely unaddressed. SDS software will be highly concurrent and will have a strong real-time orientation. Research into testing and analysis of such software is barely beginning. There are a small number of early research efforts underway. The limitations of these efforts are well recognized and the need to strengthen them and augment them with others are also well known. Major new research efforts are required to develop the technology needed for adequate testing and analysis of this type of software.

2. **SDS Software Validation Suites.** It is extremely desirable that a base set of tests that are applicable to each component of SDS software be developed. Such a validation suite would play a valuable role in achieving a known level of software assurance across all SDS software, and comprise a major element of acceptance testing. In many respects, this validation suite would be similar to the Ada Compiler Validation Capability (ACVC) established by the Ada Joint Program Office. It could be made available to software developers and all software required to pass the tests as a measure of readiness for acceptance testing. As faults are identified in operational software, the validation suite would be extended with tests which could detect these faults prior to deployment. Substantial resources will be required to develop and apply the validation suite. It will be a major technical challenge to determine the requirements for, design, develop, and maintain an evolving validation suite which provides both a good level of software assurance and efficient utilization of computing resources. The programmatic issues concerned with assigning responsibilities and allocating necessary resources for developing, maintaining, and applying the validation suite must also be investigated.

Validation suites can serve additional roles. For example, the issue of correctness of run-time kernels is crucial to Ada software but unlikely to be fully resolved within the next five years. Meanwhile, validation suites for Ada tasking programs supported by specifications of the expected run-time behavior could be developed. When a specified program is run with a particular kernel and its behavior fails to conform to the specification, this will serve as an indication of some problem with the kernel, compiler or such. This is a non-trivial task since the run-time analysis will generally require the use of formal specifications. Such specifications, however, if carefully planned, will ultimately be useful in the formal verification of the software under examination. This type of role for validation suites has many applications. Another example would be a validation suite for simulators of SDS software.

3. **Develop Methods for Formally Specifying Real-Time, Distributed, and Degraded Systems and Testing Behavior Against Specifications.** The following subsection (Section 9.2.3) identifies the need for methods reasoning about real-time, distributed and degraded systems. To do this will require breakthroughs in three or four separate technologies, including specification languages, proof rules accompanied by formal semantics, proof methods, and automated proof systems. Meanwhile, the path towards formal verification can be exploited to yield early, practical results. To this end, methods for formally specifying the expected behavior of these types of systems, and then testing actual behavior against the specifications, should be developed.

4. **Develop Methods for Building Self-Checking Software in Multiprocessor Systems.** The correctness of software in multiprocessor systems cannot be assured prior to deployment with current technology. Even if this were not the case, self-checking software that can guard against post-deployment corruptions is still advisory. Consequently, self-checking software has a potentially vital importance for SDS software. As with the other R&D tasks listed here, some ear'y work in this area is being performed. Much additional effort is required, however, to produce products which can be applied to SDS.

9.2.3 Formal Verification R&D Tasks

1. **Identify Critical Properties to be Formalized and Verified, Levels of Criticality, Priorities.** Critical properties of a system as a whole must be identified as early as possible in the development process. These properties affect critical design decisions in partitioning and allocating functional responsibilities within a system. Properties that must be proven at the system level imply requirements for components with proven properties and construction techniques that preserve those properties.

As a corollary to identifying critical properties and components, identification of levels of criticality would help in assessing verification requirements, assigning priorities to development and verification effort, and allocating assurance resources between testing and verification.

2. **Identify Critical Life-Cycle Components to be Formalized and Verified.** Critical system components that will require verification also need to be identified as early as possible in the development process. Proving properties of components is much easier when the proof process is made an integral part of their design and implementation. In fact, proofs of *correct* components may be impossible to construct after the fact, because of design decisions and programming practices that increase the complexity of proofs. Early identification of critical components can significantly reduce verification schedules and effort.

3. **Develop Methods for Reasoning about Real-Time Systems.** The critical property to be verified in most real-time systems is that processes meet their deadlines. Results are affected by the algorithms employed, efficiency of object code generated by the compiler, scheduling policies and performance of the run-time system, and target hardware performance. The problem, therefore, spans the design of the entire system. Advances are needed in several areas including real-time specification languages, formal

semantics, proof rules and methods, and automated support.

In the past the deadline problem has been addressed by *worst-case* analysis, which tends to produce overly pessimistic solutions. That is, systems are over built for normal operations to ensure that they can handle the extreme worst-case deadline situations. Specific techniques employed include: algorithms with fixed execution times, assembly language or hand-optimized object code, fixed priorities, deterministic scheduling, and target hardware upgrades. Each of these techniques simplifies the deadline verification problem, but they often restrict capabilities that could be supported under normal (slack) operating conditions.

Adaptive algorithms and scheduling techniques that adjust to system workload have been introduced to gain processing capabilities during slack periods. For example, when system workload is light, slower but more-accurate algorithms can be used and useful background operations can be performed. As the workload picks up, background operations are dropped and required processes can switch to faster algorithms. In addition, as a process nears its deadline, it may try to increase its scheduling priority to assure its completion. Formal methods are needed for reasoning about these techniques that would allow development of proofs of adaptive program behavior.

4. **Develop Methods for Reasoning about Distributed Systems.** Distributed systems are characterized by communication latencies between subsystems. This makes it extremely difficult for subsystems to synchronize their actions. Timing constraints on coordination, for example, may not allow subsystems to fully verify each others actions or readiness. That is, some subsystems may have to proceed on the assumption that the other subsystems are performing their functions at the right time. Formal verification of such systems requires methods for reasoning about system behavior where the timing of state transitions is uncertain.

5. **Develop Methods for Reasoning about "Degraded" Systems.** Fault-tolerant systems are able to recover from or adapt to certain types of component failures. Many such systems may continue to operate in a degraded mode until the failed component can be repaired or replaced. Current verification techniques assume correct operation of underlying hardware and peripheral devices such as sensors. Methods for reasoning about system behavior in the presence of potential component failures is needed to verify fault-tolerant systems.

6. **Develop Support for Proving Attributes Throughout System Development.** Second generation verification tools are improving the utility of earlier tools by assuring the soundness of underlying logic systems, standardizing on programming and annotation languages (Ada and Anna), improving user interfaces, and improving performance. Additional standards and production-quality tools are needed for formal requirements and design notations that can be used as a basis for proofs.

9.2.4 Measurement Technology R&D Tasks

1. **Develop a Comprehensive Measurement Methodology.** Current measurement methodologies do not provide an adequate framework for metric analysis. Methods must be developed which address the requirements of the measurement process, guide the appropriate selection of metrics, and aid in collecting, interpreting, and validating metric results. Such a measurement methodology should encourage the top-down generation of metrics based upon the needs of the particular project/organization (for example, primary emphasis on reliability or portability, cost and time factors). The methodology should support the tailoring of metrics based on the specific needs of the project/organization, and the particular characteristics of the project environment.

2. **Integration of Measurement into Software Development.** The fundamental purpose of software measurement is the generation of knowledge and information which will permit the creation of higher-quality products. The measurement process must produce results which are fed back into the software development process to support improvement and learning.

3. **Automated Tool Support.** There currently exist almost no metric tool support other than those which automate metric analysis. The integrated measurement methodology previously mentioned will require automation at various levels. Requirements determination, metric selection, collection, interpretation, and validation will all require some degree of automated tool support. In addition, a historical database should be established which can be used to validate existing metrics and examine proposed metrics.

9.2.5 Software Reliability Assessment R&D Tasks

1. **Assessment of the Software Development Process.** To support the construction of reliable software, emphasis must be placed on the software development process. That is, the targets of software reliability assessment must be software development methodologies, practices, tools, techniques, and other elements of the process, rather than individual software products. The motivation here is that the best way to construct reliable software is to utilize software development methodologies that have been shown to afford the highest degree of reliability. This task should be accomplished through the Measurement Technology R&D tasks described above, by ensuring proper emphasis on metrics that capture the concepts of reliability. A necessary precondition, of course, is the effective representation of development processes through such a medium as process programming.

2. **Assessment of Software Reliability in a System Context.** The technology must enable software reliability to be assessed in a system context. In distributed real-time systems, the software is responsible for dealing with timing constraints, hardware failures, and software faults. Accordingly, software correctness and reliability depend on whether software meets its requirements with respect to real-time and fault tolerance.

3. **Assessment of System Reliability.** Since software reliability must be taken into account when assessing system reliability, the technology must support *system* reliability assessment. In regard to this issue, the traditional practice of casting software reliability in hardware reliability terms and then using combinatorial analysis to derive system reliability needs to be rethought. In particular, the distinction between design faults and age-related faults needs to be given further consideration.

9.3 Monitor Technology Research

This report does not identify any technology deficiencies which have not previously been recognized. Consequently, some of the problems discussed herein are already being addressed by various researchers. The SDIO must maintain a close awareness of these ongoing R&D efforts so that (1) promising developments are promptly considered for SDS practice, (2) the SDIO helps to fund efforts which indicate solutions to specific SDS problems, and (3) none of the SDIO-sponsored research redundantly duplicates other work.

An initial list of ongoing R&D efforts to monitor can be compiled from those mentioned in this report. Mechanisms for establishing, and maintaining, contact with these efforts must be developed, and appropriate responsibilities assigned. General contact with the research community as a whole is necessary so future research efforts are considered for inclusion in this task as they arise. For example, researchers from the University of California (Irvine), the University of Massachusetts, and Purdue

University are collaborating in the development of a 20 year research plan which is a candidate for future inclusion in the list of R&D efforts to monitor.

REFERENCES

[Albr83] Albrecht, A.J., and J.E. Gaffney. "Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation." *IEEE: Transactions on Software Engineering*, 9/6 (Nov 1983):639-648.

[Ambl76a] Ambler, A.L., et al. 1976. "Gypsy: A Language for Specification and Implementation of Verifiable Programs." *ACM: SIGPLAN Notices*, 12/3 (Mar 1976).

[Ande76a] Anderson, T., and R. Kerr. 1976. "Recovery Blocks in Action: A System Supporting High Reliability." In *Proceedings 2nd International Conference on Software Engineering*, October 13-15, San Francisco, CA:,. Washington, DC: IEEE Computer Society Press.

[Appe88] Appelbe, W.F., R.A. DeMillo, D.S. Guindi, K.N. King, and W.M. McCracken. 1988. *Using Mutation Analysis for Testing Ada Programs*. Purdue University. Technical Report SERC-TR-9-P. Also published in *Proceedings Ada-Europe '88*, June, Munich, Germany.

[Aviz85] Avizienis, A. "The N-Version Approach to Fault-Tolerant Software." *IEEE: Transactions on Software Engineering*, 11/12 (Dec 1985):1491-1501.

[Bake79] Baker, A.L., and S.H. Zweben. "The Use of Software Science in Evaluating Modularity Concepts." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):110-120.

[Basi83a] Basili, V.R., and E.E. Katz. 1983. "Metrics of Interest in an Ada Development." In *Proceedings IEEE Computer Society Workshop on Software Engineering Technology Transfer*, April 25-27, Miami Beach, FL, 22-29. Los Angeles, CA: IEEE Computer Society.

[Basi85a] Basili, V.R., and R.W. Selby Jr. 1985. "Calculation and Use of an Environment's Characteristic Software Metric Set." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 386-391. Washington, DC: IEEE Computer Society Press.

[Basi86a] Basili, V.R., R.W. Selby Jr., and D.H. Hutchens. 1986. "Experimentation in Software Engineering." *IEEE: Transactions on Software Engineering*, 12/7 (Jul 1986):733-743.

[Basi87a] Basili, V.R., and H.D. Rombach. 1987. "TAME: Tailoring an Ada Measurement Environment." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, Arlington, VA, 318-325. Washington, DC: ACM Ada Technical Committee.

[Adam85] Adam, J.A., and P. Wallich eds. "Star Wars SDI: The Grand Experiment." *IEEE SPECTRUM*, 22/9 (Sep 1985):34-64.

[Basi88] Basili, V.R., and H.D. Rombach. "The TAME Project: Towards Improvement-Oriented Software Environments." *IEEE: Transactions on Software Engineering*, 14/6 (Jun 1988):758-773.

[Batt87] Battaglia, M. May 1987. *Integrated Diagnostics Program Plan and Roadmap*. Joint Policy Coordinating Group: Logistics Research, Development Test and Evaluation Integrated Diagnostics Working Panel.

[Boeh81] Boehm, B.W. 1981. *Software Engineering Economics*. Englewood Cliffs, NJ: Prentice-Hall. Also published in *IEEE: Transactions on Software Engineering*, 10/1 (Jan 1984):4-21.

[Boeh84a] Boehm, B.W., T.E. Gray, and T. Seewaldt. "Prototyping Versus Specifying: A Multiproject Experiment." *IEEE: Transactions on Software Engineering*, 10/3 (May 1984):290-303.

[Boug86] Bouge, L., N. Choquet, L. Fribourg, and M.C. Gaudel. "Test Sets Generation from Algebraic Specifications Using Logic Programming." *ACM: Journal of Systems and Software*, 6/4 (Nov 1986):343-360.

[Brow78] Browne, J.C., and D.B. Johnson. 1978. "FAST: A Second Generation Program Analysis System." In *Proceedings 3rd International Conference on Software Engineering*, March 10-12, Atlanta, GA, 142-148. Washington, DC: IEEE Computer Society Press.

[Bryk89] Brykczynski, B., and C. Youngblut. 1989. *Towards SDS Testing and Evaluation: A Collection of Relevant Topics*. IDA Draft Memorandum Report M-513. VA: Institute for Defense Analyses.

[Budd80a] Budd, T.A., R.A. DeMillo, R.J. Lipton, and F.G. Sayward. 1980. "Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs." In *Proceedings 7th ACM Annual Symposium on Principles of Programming Languages*, January 28-30, Las Vegas, NV, 220-233. Baltimore, MD: ACM Order Department.

[Budd85] Budd, T.A, and A.S. Gopal. "Program Testing by Specification Mutation." *IEEE: Computer Language*, 10/1 (Jan 1985).

[Bump87] Sen. Bumpers. "The Software Pitfall Facing an Early-Deployed SDI." *Congressional Record -- Senate*. March 11, 1987:3040-3041.

[Camp79] Campbell, R.H., and R.B. Kolstad. 1979. "Path Expressions in Pascal." In *Proceedings 4th International Conference on Software Engineering*, September 27-29, Munich, Germany, 212-219. Washington, DC: IEEE Computer Society Press.

[Card87a] Card, D.N., and W.W. Agresti. "Resolving the Software Science Anomaly." *ACM: Journal of Systems and Software*, 7/1 (Mar 87):29-36.

[Cava78] Cavano, J., and J.A. McCall. 1978. "A Framework for the Measurement of Software Quality." In *Proceedings ACM Software Quality Assurance Workshop*, November 15-17, San Diego, CA, 133-139. New York: Association for Computing Machinery.

[Cha88] Cha, S.S., N.G. Leveson, and T.J. Shimeall. 1988. *Safety Verification in Murphy Using Fault Tree Analysis*. University of California.

[Chan88] Chandy, K.M. and J. Misra. 1988. *Parallel Program Design: A Foundation*. Reading, MA: Addison Wesley.

[Choq86] Choquet, N. 1986. "Test Data Generation using a Prolog with Constraints." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 132-141. Washington, DC: IEEE Computer Society Press.

[Clar83b] Clarke, L.A., and D.J. Richardson. 1983. "The Application of Error-Sensitive Testing Strategies to Debugging." In *Proceedings ACM SIGSOFT-SIGPLAN Software Engineering Symposium on High-Level Debugging*, March 20-23, Asilomar, CA. Published in *ACM: Software Engineering Notes*, 8/4 (Aug 1983):45-52. Baltimore, MD: ACM Order Department.

[Clar85a] Clarke, L.A., A. Podgurski, D.J. Richardson, and S.J. Zeil. 1985. "A Comparison of Data Flow Path Selection Criteria." In *Proceedings 8th International Conference on Software Engineering*, August 28-30, London, England, 244-251. Washington, DC: IEEE Computer Society Press.

[Clar86a] Clarke, L.A., A. Podgurski, D.J. Richardson, and S.J. Zeil. 1986. "An Investigation of Data Flow Path Selection Criteria." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 23-32. Washington, DC: IEEE Computer Society Press.

[Clar88a] Clarke, L.A., D.J. Richardson and S.J. Zeil. "Team: A Support Environment for Testing, Evaluation, and Analysis." In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, November 28-30, Boston, MA, 121-129.

[Clar88b] Clarke, L.A., and S.J. Zeil. January 1988. *An Advanced Testing System for Ada, System Description and Design*. University of Massachusetts.

[Conr85] Conradi, R., and D. Svanaes. January 1985. *FORTVER - A Tool for Documentation and Error Diagnosis of FORTRAN-77 Programs*. University of Trondheim. Technical Report 1/85.

[Cont86] Conte, S.D., H.E. Dunsmore, and V.Y. Shen. 1986. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/Cummings Publishing Company.

[Curt79a] Curtis, B., S. Sheppard, P. Milliman, M. Borst, and T. Love. "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics." *IEEE: Transactions on Software Engineering*, 5/2 (March 1979):95-104.

[DeMi78] DeMillo, R.A., R.J. Lipton, and F.G. Sayward. "Hints on Test Data Selection: Help for the Practicing Programmer." *IEEE: Computer*, 11/4 (Apr 1978): 34-41.

[DeMi87a] DeMillo, R.A., W.M. McCracken, R.J. Martin, and J.F. Passafiume. 1987. *Software Testing and Evaluation*. Menlo Park, CA: The Benjamin/Cummings Publishing Company.

[DeMi88a] DeMillo, R.A., D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. 1988. "An Extended Overview of the Mothra Software Testing Environment." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 142-151. Washington, DC: IEEE Computer Society Press.

[Dijk76a] Dijkstra, E.W. 1976. *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice Hall.

[DOD86a] DoD Standard STD-2168. 1 August 1986. *Defense System Software Quality Program* (draft).

[DOD88] DoD Standard-2167A. Defense Systems Software Development. 29 February 1988.

[DODD87a] DoD Directive 5000.3-M-3. November 1987. ref:*Software Test and Evaluation Manual*.

[DODD87b] DoD Directive 3405.2. 30 March 1987. *Use of Ada in Weapon Systems*.

[DODD87c] DoD Directive 3405.1. 2 April 1987. *Computer Programming Language Policy*.

[Dunc81] Duncan, A.G., and J.S. Hutchison. 1981. "Using Attribute Grammars to Test Designs and Implementations." In *Proceedings 5th International Conference on Software Engineering*, March 9-12, San Diego, CA, 170-178. Washington, DC: IEEE Computer Society Press.

[Dura84] Duran, J.W., and S.C. Ntafos. "An Evaluation of Random Testing." *IEEE: Transactions on Software Engineering*, 10/4 (Jul 1984):438-444.

[Dyer85a] Dyer, M. 1985. "Software Verification Through Statistical Testing."

[Elme73] Elmendorf, W.R. November 1973. *Cause-Effect Graphs in Functional Testing*. IBM: Technical Report 00.2487.

[Evan83a] Evangelist, W.M. "Software Complexity Metric Sensitivity to Program Structuring Rules." *ACM: The Journal of Systems and Software*, 3/6 (Nov 1983):231-243.

[Faga76] Fagan, M.E. "Design and Code Inspections to Reduce Errors in Program Development." *IBM: Systems Journal*, 15/3 (1976):182-211.

[Farr83] Farr, W.H. September 1983. *A Survey of Software Reliability Modeling and Estimation*. Dahlgren, VA:Naval Surface Weapons Center. Technical Reprt NSWC-TR-82-171.

[Fink83] Finkel, R.A., M.H. Solomon et al. April 1983. *Charlotte: Part IV of the First Report on the Crystal Project*. University of Wisconsin. Technical Report 501.

[Floy67] Floyd, R.W. 1967. "Assigning Meaning to Programs." In *Proceedings American Mathematical Society Symposium in Applied Mathematics*, vol. 19, 19-31. Providence, RI: American Mathematics Society. Also published in *ACM: Communications of the ACM*, 14 (Jan 1971):39-45.

[Fosd76a] Fosdick, L.D., and L.J. Osterweil. "Data Flow Analysis in Software Reliability." *ACM: Computing Surveys*, 8/3 (Sep 1976):305-330.

[Fost80] Foster, K.A. *IEEE: Transactions on Software Engineering*, 6/3 (May 1980):258-264.

[Fran86] Frankl, P.G., and E.J. Weyuker. 1986. "Data Flow Testing in the Presence of Unexecutable Paths." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 4-13. Washington, DC: IEEE Computer Society Press.

[Gann79a] Gannon, C. "Error Detection Using Path Testing and Static Analysis." *IEEE: Computer*, 12/8 (Aug 1979):26-31.

[Gann81] Gannon, J.D., P.R. McMullin, and R.G. Hamlet. "Data Abstraction Implementation, Specification and Testing." *ACM: Transactions on Programming Languages and Systems*, 3/3 (Jul 1981):211-223.

[Gerh80] Gerhart, S.L., et al. 1980. *An Overview of AFFIRM: A Specification and Verification System*. University of Southern California. Technical Report PR-79-81. Also published in *Proceedings IFIP Congress 1980*, 343-347. Amersterdam: North-Holland.

[Germ82a] German, S.M., D.P. Helmbold, and D.C. Luckham. October 1982. "Monitoring for Deadlocks in Ada Tasking." In *Proceedings AdaTEC Conference on Ada*, October, Arlington, VA, 10-25.

[Goel85] Goel, A.L. "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE: Transactions on Software Engineering*, 11/12 (Dec 1985):1411-1423.

[Gogu79a] Goguen, J.A., and J.J. Tardo. 1979. "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications." In *Proceedings Conference on Specification of Reliable Software*, Cambridge, MA, 170-189.

[Good75a] Goodenough, J.B., and S.L. Gerhart. "Toward a Theory of Test Data Selection." *IEEE: Transactions on Software Engineering*, 1/2 (Jun 1975):156-173.

[Good86a] Good, D.I. 1986. *Report on Gypsy 2.05 - January 1986*. University of Texas at Austin.

[Gord88] Gordon, A.J., and R.A. Finkel. "Handling Timing Errors in Distributed Programs." *IEEE: Transactions on Software Engineering*, 12/10 (Oct 1988):1525-1535.

[Gorl87] Gorlick, M.M., C.F. Kesselman, D.A. Marotta, and D.S Parker. May 1987. *Mockingbird: A Logical Methodology for Testing*. Computer Science Laboratory.

[Grie81] Gries, D. 1981. *The Science of Programming*. New York: Springer-Verlag.

[Gutt78a] Guttag, J.V., E. Horowitz, and D.R. Musser. "Abstract Data Types and Software Validation." *ACM: Communications of the ACM*, 21/12 (Dec 1978):1048-1064.

[Hals77a] Halstead, M.H. 1977. *Elements of Software Science*. New York: Elsevier North-Holland Publishing.

[Hame82] Hamer, P.G., and G.D. Frewin. 1982. "M.H. Halstead's Software Science - A Critical Examination." In *Proceedings 6th International Conference on Software Engineering*, September,

Tokyo, Japan, 197-206. Washington, DC: IEEE Computer Society Press.

[Helm83] Helmbold, D.P. and D.C. Luckham. November 1983. *Runtime Detection and Description of Deadness Errors in Ada Tasking*. Stanford University. Program Analysis and Verification Group Report no. 22. Technical Report CSL-TR-83-249.

[Helm85] Helmbold, D.P., and D.C. Luckham. 1985. *TSL: Task Sequencing Language*. Stanford University Technical Report. Also in *Proceedings SIGAda International Conference*, May, Paris, France. Published in *ACM: Ada Letters*, V/2 (Sep-Oct 1985):255-274.

[Hoar69] Hoare, C.A.R. "An Axiomatic Basis for Computer Programming." *ACM: Communications of the ACM*, 12/10 (1969):576-583.

[Hoar85] Hoare, C.A.R. "Communicating Sequential Processes," Prentice-Hall International, 1985.

[Howd76c] Howden, W.E. "Reliability of the Path Analysis Testing Strategy." *IEEE: Transactions on Software Engineering*, 2/3 (Sep 1976):208-215.

[Howd77a] Howden, W.E. May 1977. *Symbolic Testing - Design Techniques, Costs, and Effectiveness*. Gaithersburg, MD: National Bureau of Standards. Technical Report NBS-GCR-77-89.

[Howd78a] Howden, W.E. "Theoretical and Empirical Studies of Program Testing." *IEEE: Transactions on Software Engineering*, 4/4 (Jul 1978):293-298.

[Howd78b] Howden, W.E. "Algebraic Program Testing." *ACTA Informatica*, no. 10 (1978):53-66.

[Howd82a] Howden, W.E. "Weak Mutation Testing and Completeness of Test Sets." *IEEE: Transactions on Software Engineering*, 8/4 (Jul 1982):371-379.

[Howd83] Howden W.E. 1983. "A General Model for Static Analysis." In *Proceedings 16th Annual Hawaii International Conference on System Sciences*, 163-169.

[Howd87] Howden, W.E. 1987. *Functional Program Testing and Analysis*. New York: McGraw-Hill.

[Howd89] Howden, W.E. 1989. "Current Validation Research and Development Activities." In *Towards SDS Testing and Evaluation: A Collection of Relevant Topics*. IDA Memorandum Report M-513. Alexandria, VA: Institute for Defense Analyses. Draft.

[Kafu85a] Kafura, D.G., and J.T. Canning. January 1985. *A Validation of Software Metrics Using Many Metrics and Many Resources*. Virginia Polytechnic Institute. TR-85-6.

[Kemm80] Kemmerer, R.A. 1980. *FDM - A Specification and Verification Methodology*. System Development Corp. Technical Report SP-488.

[Kemm86] Kemmerer, R.A. 1986. *Verification Assessment Study Final Report, Volume I: Overview, Conclusions and Future Directions*. National Computer Security Council. Technical Report C3-CR01-86.

[Kern74a] Kernighan, B.W., and P.J. Plauger. 1974. *The Elements of Programming Style*. New York: McGraw-Hill.

[Kieb83] Kieburtz, R.B., and A. Silberschatz. "Access-Right Expressions." *ACM: Transactions on Programming Languages and Systems*, 5/1 (Jan 1983):78-96.

[Knig86a] Knight, J.C., and N.G. Leveson. "An Experimental Evaluation of the Assumption of Independence in Multiversion Programming." *IEEE: Transactions on Software Engineering*, 12/1 (Jan 1986):96-109.

[Lamp83] Lamport, L. "Specifying Concurrent Program Modules." *ACM: Transactions on Programming Languages and Systems*, 5/2 (Apr 1983):190-222.

[Leve83a] Leveson, N.G., T. Shimeall, J. Stolzy, and J. Thomas. 1983. "Design for Safe Software." In *Proceedings AIAA Space Science Meeting*, January, Reno, NV.

[Lin85] Lin, H. June 1985. *Software for Ballistic Missile Defense*. Center for International Studies. Massachusetts Institute of Technology.

[Linn88] Linn, J.L, C.D. Ardoin, C.J. Linn, S.H. Edwards, M.R. Kappel, and J. Salasin. April 1988. *Strategic Defense Initiative Architecture Dataflow Modeling Technique: Version 1.5*. Alexandria, VA: Institute for Defense Analyses. IDA Paper P-2035.

[Long88] Long, D.L., and L.A. Clarke. 1988. "Task Interaction Graphs for Concurrency Analysis." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 132-133. Washington, DC: IEEE Computer Society Press.

[Luck84a] Luckham, D.C., and F.W. von Henke. September 1984. *An Overview of ANNA A Specification Language for Ada*. Stanford University. Technical Report CSL-TR-84-265. Also published in *IEEE: Software*, 2/2 (Mar 1985):9-24.

[Luck86a] Luckham, D.C. 1986. *Anna: A Language for Specifying and Debugging Ada Software*. University of Stanford. Draft manuscript, 180 pages.

[Luck87] Luckham, D.C., D.P. Helmbold, S. Meldal, D.L. Bryan, and M.A. Haberler. July 1987. "Task Sequencing Language for Specifying Distributed Ada Systems." In *Proceedings of CRAI Workshop on Software Factories and Ada*, Capri, Italy, eds. A.N. Habermann and U. Montanari, 249-305. Springer-Verlag LNCS No. 275. Also published as Stanford University Technical Report CSL-TR-87-334.

[Math88a] Mathur, A.P., and E.W. Krauser. April 1988. *Mutant Unification for Improved Vectorization*. Purdue University. Technical Report SERC-TR-14-P.

[McCa76] McCabe, T.J. "A Complexity Measure." *IEEE: Transactions on Software Engineering*, 2/4 (Dec 1976):308-320.

[Meld88] Meldal, S., D. Luckham, and M. Haberler. 1988. "Specifying Ada Tasking Using Patterns of Behavior." In *Proceedings IEEE 21st Hawaii International Conference on System Sciences*, January, 129-134.

[MIL83] Military Standard MIL/ANSI-STD-1815A. January 1983. *Ada Programming Language*.

[Mill79a] Miller, E. 1979. "Program Testing Technology in the 1980's." The Oregon Report: In *Proceedings Conference on Computing in the 1980's*, 72-79. Washington, DC: IEEE Computer Society Press.

[More84] Morell, L. 1984. *A Theory of Error-Based Testing*. Ph.D. thesis, University of Maryland. Also University of Maryland, Technical Report TR-1395.

[More88] Morell, L.J. 1988. "Theoretical Insights into Fault-Based Testing." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 45-62. Washington, DC: IEEE Computer Society Press.

[Musa87] Musa, J., A. Iannino, K. Okumoto. 1987. *Software Reliability: Measurement, Prediction, Application*. New York: McGraw Hill.

[Muss79] Musser, D.R. 1979. "Abstract Data Type Specification in the AFFIRM System." In *Proceedings Conference on Specification of Reliable Software*, 47-57. Also published in *IEEE: Transactions on Software Engineering*, 6/1 (Jan 1980):24-32.

[Myer78a] Myers, G.J. "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections." *ACM: Communications of the ACM*, 21/9 (Sep 1978):760-768.

[Myer79] Myers, G.J. 1979. *The Art of Software Testing*. New York: John Wiley and Sons.

[Neum87] Neumann, P.G., and contributors. "Risks to the Public." *ACM: Software Engineering Notes*, 12/1 (Jan 1987):3-33.

[Ntaf81a] Ntafos, S.C. 1981. *On Testing with Required Elements*. University of Texas at Dallas. Technical Report 90. Also published in *Proceedings 5th International Computer Software and Applications Conference*, November 18-20, Chicago, IL, 132-139. Los Alamitos, CA: IEEE Computer Society Press.

[Olen86] Olender, K.M., and L.J. Osterweil. 1986. "Specification and Static Evaluation of Sequencing Constraints in Software." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 14-22. Washington, DC: IEEE Computer Society Press.

[Oste87a] Osterweil, L.J. March 1987. "Software Processes are Software Too." In *Proceedings 9th International Conference on Software Engineering*, March 30 - Arpil 2, Monterey, CA, 2-13. Washington, DC: IEEE Computer Society Press.

[Parn85] Parnas, D.L. "Software Aspects of Strategic Defense Systems." *ACM: SIGSOFT Software Engineering Notes*, 10/5 (Oct 1985):15-23.

[Parn88] Parnas, D.L., A.J. van Schouwen, and S.P. Kwan. May 1988. *Evaluation Standards for Safety Critical Software*. Queens University. Technical Report 88-220.

[Perk86] Perkins, J., D.M. Lease, and S.E. Keller. 1986. "Experience Collecting and Analyzing Automatable Software Quality Metrics for Ada." In *Proceedings 4th Annual National Conference on Ada Technology*, March, Atlanta, GA, 67-74.

[Pete77] Peterson, J. "Petri Nets." *ACM: Computing Surveys*, 9/3 (Sept 1977):223-252.

[Pnue77] Pnuenuli, A. 1977. "The Temporal Logic of Programs." In *Proceedings of the 18th Annual Symposium on Foundations of Computer Sciences*, Oct 31 - Nov 2.

[RADC83a] Rome Air Development Center. July 1983. *Software Interoperability and Reusability, Vols. I and II.* Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-83-174.

[RADC83b] Rome Air Development Center. July 1983. *Software Quality Measurement for Distributed Systems, Vols I, II and III.* Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-83-175.

[RADC86] Rome Air Development Center. 1986. *Ada Test and Verification System (ATVS)*. Griffiss Air Force Base, NY: Rome Air Development Center. RADC Contract F30602-86-C-0192.

[McCa87a] McCall, J.A., W. Randall, C. Bowen, N. McKelvey, R. Senn, J. Morris, H. Hecht, S. Fenwick, P. Yates, M. Hecht, and R. Vienneau. November 1987. *Methodology for Software Reliability Prediction, Vol. I.* Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-87-171.

[Rama82] Ramamoorthy, C.V., and F.B. Bastani. "Software Reliability – Status and Perspectives." *IEEE: Transactions on Software Engineering*, 8/4 (Jul 1982):354-367.

[Rand75] Randell, B. "System Structure for Software Fault Tolerance." *IEEE: Transactions on Software Engineering*, 1/2 (Jun 1975):220-232.

[Rich85a] Richardson, D.J., and L.A. Clarke. 1985. "Testing Techniques Based on Symbolic Evaluation." In *Software: Requirements, Specification, and Testing*, T. Anderson (ed.), 93-110. Blackwell Scientific Publications.

[Rich86a] Richardson, D.J., and M.C. Thompson. December 1986. *An Analysis of Test Data Selection Criteria Using the RELAY Model of Error Detection*. University of Massachusetts. Technical Report 86-65.

[Rich88] Richardson, D.J., and M.C. Thompson. 1988. "The RELAY Model of Error Detection and Its Application." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 223-230. Washington, DC: IEEE Computer Society Press.

[Robi79] Robinson, L., K.N. Levitt, and B.A. Silverberg. 1979. *The HDM Handbook*. SRI International. Project No. 4628.

[Romb87a] Rombach, H.D. 1987. "A Controlled Experiment on the Impact of Software Structure on Maintainability." *IEEE: Transactions on Software Engineering*, 12/3 (Mar 1987):344-354.

[Romb87b] Rombach, H.D., and V.R. Basili. 1987. "A Quantitative Assessment of Software Maintenance: An Industrial Case Study." In *Proceedings of the Conference on Software Maintenance*, September 21-24, Austin, TX, 134-144.

[Rose85b] Rosenthal, L.S. January 1985. *Guidance on Planning and Implementing Computer System Reliability*. Gaithersburg, MD: National Bureau of Standards. NBS Special Publication 500-12.

[SDIO87] Strategic Defense Initiative Organization. 30 June 1987. *Strategic Defense System Test and Evaluation Master Plan (TEMP)*.

[SDIO88a] Strategic Defense Initiative Organization. 16 November 1988. *Strategic Defense System Software Policy*.

[SDIO88b] Strategic Defense Initiative Organization. 16 November 1988. *Software Policy*. SDIO Management Directive No. 7.

[Selb87a] Selby, R.W. Jr. 1987. "Incorporating Metrics into a Software Environment." In *Proceedings Joint Conference of 5th National Conference on Ada Technology and Washington Ada Symposium*, March 16-19, Arlington, VA, 326-331. Washington, DC: ACM Ada Technical Committee.

[SERC87] Software Engineering Research Center. 1987. *The Mothra Testing Environment, User's Manual*. Purdue University. Technical Report SERC-TR-4-P.

[Shen83] Shen, V.Y., S.D. Conte, and H.E. Dunsmore. "Software Science Revisited: A Critical Analysis of the Theory and Its Empirical Support." *IEEE: Transactions on Software Engineering*, 9/2 (Mar 1983):155-165.

[Shaw78] Shaw, A.C. "Software Descriptions with Flow Expressions." *IEEE: Transactions on Software Engineering*, 4/3 (May 1978):242-254.

[Shoo86] Shooman, M.L. 1986. *Probabilistic Reliability: An Engineering Approach*. New York: McGraw-Hill, 1968. Updated and reprinted, Malabar, FL: Krieger, 1986.

[Tayl80a] Taylor, D.J., D.E. Morgan, and J.P. Black. "Redundancy in Data Structures: Improving Software Fault Tolerance." *IEEE: Transactions on Software Engineering*, 6/6 (Nov 1980):585-594.

[Tayl85] Taylor, R.N., and T.A. Standish. "Steps to an Advanced Ada Programming Environment." *IEEE: Transactions on Software Engineering*, 11/3 (Mar 1985):302-310.

[Tayl86a] Taylor, R.N., and C.D. Kelly. 1986. "Structural Testing of Concurrent Programs." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 164-169. Washington, DC: IEEE Computer Society Press.

[Tayl88] Taylor, R.N., F.C. Belz, L.A. Clarke, L. Osterweil, R.W. Selby Jr. , J.C. Wileden, A.L. Wolf, and M. Young. 1988. "Foundations for the Arcadia Environment Architecture." In *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, November 28-30, Boston, MA, 1-13.

[Weis88a] Weiss, S.N. 1988. "A Formal Framework for the Study of Concurrent Program Testing." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 106-113. Washington, DC: IEEE Computer Society Press.

[Weyu80c] Weyuker, E.J., and T.J. Ostrand. "Theories of Program Testing and the Application of Revealing Subdomains." *IEEE: Transactions on Software Engineering*, 6/3 (May 1980):236-246.

[Weyu84a] Weyuker, E.J. "The Complexity of Data Flow Criteria for Test Data Selection." *Information Processing Letters*, 19/2 (Aug 1984):103-109.

[Weyu88] Weyuker, E.J. 1988. "An Empirical Study of the Complexity of Data Flow Testing." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 188-195. Washington, DC: IEEE Computer Society Press.

[Whit78a] White, L.J., F.C. Teng, H. Kuo, and D. Coleman. 1978. *An Error Analysis of the Domain Testing Strategy.* Ohio State University. Technical Reprt CISRC-TR-78-2.

[Whit86] White, L.J., and I.A. Perera. 1986. "An Alternative Measure for Error Analysis of the Domain Testing Strategy." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 122-131. Washington, DC: IEEE Computer Society Press.

[Whit88a] White, L.J., and B. Wiszniewski. 1988. "Complexity of Testing Iterated Borders for Structured Programs." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 231-237. Washington, DC: IEEE Computer Society Press.

[Wolf86c] Wolf, A.L., L.A. Clarke, and J.C. Wileden. September 1986. *The AdaPIC Toolset: Supporting Interface Control and Analysis Throughout the Software Development Process.* University of Massachusetts. COINS Technical Report 86-51. To appear in *IEEE: Transactions on Software Engineering*.

[Wood80b] Woodward, M.R., D. Hedley, and M. Hennell. "Experience with Path Analysis and Testing of Programs." *IEEE: Transactions on Software Engineering*, 6/3 (May 1980):278-286.

[Wood88] Woodward, M.R., and K. Halewood. 1988. "From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues." In *Proceedings 2nd Workshop in Software Testing, Verification, and Analysis*, July 19-21, Banff, Canada, 152-158. Washington, DC: IEEE Computer Society Press.

[Youn86a] Young, M., and R.N. Taylor. 1986. "Combining Static Concurrency Analysis with Symbolic Execution." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 170-178. Washington, DC: IEEE Computer Society Press.

[Youn88a] Youngblut, C., B. Brykczynski, K. Gordon, R.N. Meeson, and J. Salasin. *Bibliography of Testing and Evaluation Reference Material*. Alexandria, VA: Institute for Defense Analyses. Draft IDA Memorandum M-496.

[Yu88a] Yu, T.J., B.A. Nejmeh, H.E. Dunsmore, and V.Y. Shen. "SMDC: An Interactive Software Metrics Data Collection and Analysis System." *ACM: Journal of Systems and Software*, no. 8 (1988).

[Zeil83a] Zeil, S.J. "Testing for Perturbations of Program Statements." *IEEE: Transactions on Software Engineering*, 9/3 (May 1983):335-346.

[Zeil86] Zeil, S.J. 1986. "The EQUATE Testing Strategy." In *Proceedings Workshop on Software Testing*, July 15-17, Banff, Canada, 142-151. Washington, DC: IEEE Computer Society Press.

[Zweb79] Zweben, S.H., and M. H. Halstead. "The Frequency Distribution of Operators in PL/I Programs." *IEEE: Transactions on Software Engineering*, 5/2 (Mar 1979):91-94.

APPENDIX A: GLOSSARY OF TERMS

While attempting to define the majority of terms used in this paper, the reader is assumed to be familiar with general software-related terms and, therefore, this glossary focuses on testing and evaluation terms. Where necessary, the reader is referred to the IEEE Standard Glossary of Software Engineering Terminology, for additional definitions.

Many of the following definitions are taken from the IEEE and other existing glossaries. In each such case, the definition is followed by a reference to the relevant source. These sources are listed at the end of the glossary.

A1 CERTIFICATION: The distinguishing feature of systems in this class is the analysis derived from formal design specifications and verification techniques and the resulting high degree of assurance that the Trusted Computing Base is correctly implemented. This assurance is developmental in nature, starting with a formal model of security policy and a formal top-level specification of the design.

ABSTRACTION: (1) A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information. (2) The process of forming an abstraction. [IEEE83].

ABSTRACT MACHINE: A representation of the characteristics of a process or machine. [IEEE83]

ABSTRACT MODEL SPECIFICATIONS: Also called the Predicate Transform Method. For syntax it employs the basic precondition/post-condition format developed by Hoare. It defines functions in terms of an underlying abstraction selected by the specifier.

ACCEPTANCE TESTING: Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable a customer to determine whether or not to accept the system. *See also QUALIFICATION TESTING, SYSTEM TESTING.*

ACCESSIBILITY: Code possesses the characteristic accessibility to the extent that it facilitates selective use of its parts. [RADC83].

ACCURACY: (1) Those attributes of the software which provide the required precision in calculations and outputs, or (2) a measure of the degree of freedom from error; the degree of exactness possessed by an approximation or measurement. [RADC83].

ADAPTABILITY: A measure of the ease with which a program can be altered to fit differing user images and system constraints. [RADC83].

ADAPTIVE ALGORITHMS: Multiple algorithms that are selected at run-time depending on program conditions such as workload and required accuracy.

ADAPTIVE PROGRAMS: Programs that employ adaptive algorithms.

ADEQUATE TEST DATA: A test data set T is adequate for a program P if P behaves correctly on T but all incorrect programs behave incorrectly.

AFFIRM: An automated verification system developed at the University of Southern California, Information Sciences Institute.

ALGEBRAIC SPECIFICATION: An algebraic specification is made up of a list of functional symbols (constructors and defined functions) on a set of sorts and of a set of axioms defining properties of the defined functions.

ALGEBRAIC TESTING: A testing approach in which program correctness is treated as a program equivalence problem.

ALGORITHM: (1) A finite set of well-defined rules for the solution of a problem in a finite number of steps; for example, a complete specification of a sequence of arithmetic operations for evaluating sin x to a given precision. (2) A finite set of rules that gives a sequence of operations for performing a specific task. [IEEE83].

ALGORITHMIC NOTATION: Use of an algorithm to express a proof.

ALPHA TESTING: Testing of a software product or system conducted at the developer's site by the customer. *See also BETA TESTING.*

ALTERNATE-SUFFICIENT: As used in Morell's model of fault-based testing, the case where either the original program or one of the alternate programs must be correct.

ALTERNATIVES: As used in Morell's model of fault-based testing, the set of alternative programs derived by making a series of small, predefined changes to the original program.

ANNA: A language used to annotate Ada programs by making assertions on statements, variables, and program units which can be transformed into both static and dynamic checks for certain types of faults and failures.

ANNOTATION LANGUAGE: A language which defines assertions that can be used to annotate a product expressed in some other language, thereby facilitating either static or dynamic checking of particular properties of the annotated product.

ARC: In a directed graph, the oriented connection between two nodes. Also called an edge. [Mill81]

ARCADIA: A software development environment under development by a consortium of academic and commercial organizations. The principal characteristics of the Arcadia design revolve around the use of process programming and tool fragments to yield a highly flexible and extensible architecture.

ARCHITECTURE: *See SYSTEM ARCHITECTURE.*

ARCHITECTURAL DESIGN: (1) The process of defining a collection of hardware and software components and their interfaces to establish a framework for the development of a computer system. (2) The result of the architectural design process. [IEEE83]

ARIES: The generic interpreter developed for the TEAM testing environment.

ARITHMETIC EXPRESSION: A formula which defines the computation of a value.

ARRAY: A composite object consisting of components that have the same type.

ASSERTION: A logical expression specifying a program state that must exist or a set of conditions that program variables must satisfy at a particular point during program execution; for example, A is positive and A is greater than B. *See also INPUT ASSERTION, OUTPUT ASSERTION.* [IEEE83]

ASSIGNMENT STATEMENT: An instruction used to express a sequence of operations, or used to assign operands to specified variables, or symbols, or both. [IEEE83]

ASYMPTOTICAL NORMAL ESTIMATOR: An estimator is called asymptotical normal if its distribution is almost normal for sufficiently large sample sizes.

AUDIT: (1) An independent review for the purpose of assessing compliance with software requirements, specifications, baselines, standards, procedures, instructions, codes, and contractual and licensing agreements. (2) An activity to determine through investigation the adequacy of, and adherence to, established procedures, instructions, specifications, codes, and standards or other applicable contractual and licensing requirements, and the effectiveness of the implementation.

AUGMENTABILITY: Those attributes of the software which provide for expansion of

capability for functions and data. Code possesses the characteristic augmentability to the extent that it can easily accommodate expansion in component computational functions or data storage requirements. [RADC83].

AUTONOMOUS PROOFS: In networks of processes, an autonomous proof treats a process like an independent entity which, therefore, requires an independent specification. The specifications (but not the code) of the component processes are used in the proof.

AVAILABILITY: The probability that a specified function or capability can be initiated or invoked when the system is operated in its intended environment for a specified period of time. [DeMi88]

AXIOMATIC CORRECTNESS PROOF: A proof that employs the Axiomatic Method to verify correctness of a program.

AXIOMATIC METHOD: *See INVARIANT ASSERTION METHOD.*

BASIC EXECUTION TIME MODEL: Software reliability model in which the failure process is assumed to be a nonhomogeneous Poisson process with linearly decreasing failure intensity. [Musa87].

BETA TESTING: Testing conducted at one or more customer sites by the end-user of a delivered software product or system. *See also ALPHA TESTING*

BINDING: The assigning of a value or referent to an identifier; for example, the assigning of a value to a parameter or the assigning of an absolute address, virtual address, or device identifier to a symbolic address or label in a computer program. See also *DYNAMIC BINDING, STATIC BINDING.* [IEEE83]

BLACK BOX TESTING: *See FUNCTIONAL TESTING.*

BLOCK: See *PROGRAM BLOCK.*

BREADTH: The breadth of a fault-based technique reflects the number of potential faults considered. It may be finite or infinite.

BLOCKED: A process is blocked when it is waiting for an event to occur before execution can proceed.

BLOCKING FREEDOM: When a process can never get into a blocked state.

BOTTOM UP TESTING STRATEGY: A systematic testing philosophy that seeks to test those modules at the bottom of the invocation structure first. [Mill81]

BOUNDARY VALUE ANALYSIS: A selection technique in which test data are chosen to lie along boundaries of input domain (or output range) classes, data structures, procedure parameters, etc. Choices often include maximum, minimum, and trivial values or parameters. This technique is often called stress testing. [Adri82]

BOYER-MOORE THEOREM PROVER: A tool that mechanizes proofs in a logical theory developed by Boyer and Moore. Primarily an induction machine, it incorporates many ad hoc proof strategies and expression simplifiers.

BRACKETED SECTIONS: Regions of text, immediately surrounding an input/output statement in which the global invariant need not hold.

BRANCH TESTING: A test method satisfying coverage criteria that require that for each decision point each possible branch be executed at least once. [IEEE83]

BUG: See *FAULT.*

BUILT-IN-TEST: Hardware with built-in-test capabilities is hardware in which diagnostic probes are designed into electronic components to facilitate easier detection and investigation of faults.

BULK CONSTANT: The proportion of faults that cause failures.

CALENDAR TIME: Chronological time, including time during which a computer may not be running. [Musa87]

CAUSE-EFFECT GRAPH: A combinatorial logic network representing causes (distinct input conditions or equivalence classes of input conditions) and effects (output conditions or a system transformation) and the logical relations between them.

CAUSE-EFFECT GRAPHING: A test data selection technique. The input and output domains are partitioned into classes and analysis is performed to determine which input classes cause which effect. A minimal set of inputs is chosen that will cover the entire effect set. [Adri82]

CERTIFICATION: Acceptance of software by an authorized agent usually after the software has been validated by the agent, or after its validity has been demonstrated to the agent. [Adri82]

CHANNEL NAME: Symbolic name assigned to a communication channel.

CLOCK TIME: Elapsed time from start to end of program execution, including wait time, on a running computer. [Musa87]

CODE: (1) A set of unambiguous rules specifying the manner in which data may be represented in a discrete form. (2) To represent data or a computer program in a symbolic form that can be accepted by a processor. (3) To write a routine. (4) Loosely, one or more computer programs, or part of computer program. [IEEE83]

CODE AUDITOR: An automated tool which checks for conformance to prescribed programming standards and practices.

CODE INSPECTIONS: *See INSPECTIONS.*

COHESION: A relative measure of the strength of relationships among the internal components of a module insofar as they contribute to the variation in assumptions made by the outside program concerning the role the module plays in the program. [RADC83].

COINCIDENTAL CORRECTNESS: Program testing detects an fault by discovering the effect of that fault. It is possible, however, that an fault on an executed path may not produce erroneous results for some selected test data; this is referred to as coincidental correctness.

COLLATERAL TESTING: That testing coverage which is achieved indirectly, rather than as a direct object of a testcase generation activity. [Mill81]

COMMUNICATING SEQUENTIAL PROCESSES (CSP): A technique in which the synchronization between concurrent processes is explicit and, as a result, the semantics of message passing can be expressed formally.

COMMUNICATION AXIOM: Used in the CSP proof to verify that the assertions made in sequential proofs after communication points are valid.

COMMUNICATION CHANNELS: The logical or physical means by which data is transmitted between devices and/or processes.

COMMUNICATION HISTORY: The history of communication events that occur during the execution of a distributed program.

COMMUNICATION SPACE: Consists of those symbols, known within the module, by which information can be passed to or from the module from outside it. Communication space mechanisms consist of formal parameters, global variables, and return parameters. [Mill81]

COMMUNICATION EVENT TRACING: The process of keeping a history of the communication events as they occur in running a distributed program.

COMPETENT PROGRAMMER ASSUMPTION: An assumption used in mutation testing that competent programmers produce programs, which, if not actually correct, are close to being correct.

COMPILE-TIME CHECK: Checking performed when a computer program expressed in a

problem-oriented language is translated into the assembly code or machine code of a particular computer.

COMPILER: A computer program used to translate a higher order language program into its relocatable or absolute machine code equivalent. Contrast with *INTERPRETER*.

COMPLEXITY: (1) The degree of complication of a system or system component, determined by such factors as the number and intricacy of interfaces and conditional branches, the degree of nesting, the types of data structures, and other system characteristics. [IEEE83] (2) A characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software.

COMPONENT: A basic part of a system or program. [IEEE83]

COMPUTATIONAL FAULT: An incorrect path computation, such as an fault caused by missing or inappropriate assignment statements.

COMPUTATION TESTING: A testing technique which analyzes path computations and selects test data aimed at revealing computation faults.

COMPUTATIONALLY EQUIVALENT: Two programs, or portions of a program, are said to be computationally equivalent if they produce the same results.

COMPUTATIONALLY INTENSIVE: Software where the majority of processing occurs in computing required functions rather than handling inputs and outputs.

COMPUTATIONAL INDUCTION: An inductive method for proving things about recursively defined functions.

COMPUTER SYSTEM: A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designed programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during their execution. A computer system may be a standalone unit or may consist of several interconnected units. Synonymous with ADP system, computing system. [IEEE83]

CONCURRENCY HISTORY: The sequence of concurrency states beginning with the initial state of a concurrent system. A proper history is a finite history in which all elements are unique, save possibly the final element of the sequence. A complete history of a program S is the set of all proper histories of S.

CONCURRENCY STATE COVERAGE: Member of a series of successively more stringent testing coverage measures analogous to structural and data flow testing criteria for sequential programs. *See also STATE TRANSITION COVERAGE, SYNCHRONIZATION COVERAGE.*

CONCURRENCY STATE GRAPH: A graphical representation of a complete concurrency history where each node represents a unique concurrency state and each edge represents a transition from one concurrency state to another.

CONCURRENCY STATE: A concurrency state summarized the control state of each of the concurrent processes at some point in an execution, including synchronization information, while omitting other information such as data values.

CONCURRENT PROCESSES: Processes that may execute in parallel on multiple processors or asynchronously on a single processor. Concurrent processes may interact with each other, and one process may suspend execution pending receipt of information from another process or the occurrence of an external event. [IEEE83]

CONFIGURATION: (1) The collection of interconnected objects which make up a system or subsystem. (2) The total software modules in a software system or hardware devices in a hardware system and their interrelationships. [DACS79]

CONSISTENCY: Those attributes of the software which provide for uniform design and

implementation techniques and notation. [RADC83].

CONSISTENT ESTIMATOR: An estimator is said to be consistent if its variance tends to zero and its expectation tends to the true population parameter as the sample size tends to infinity.

CONTROL FLOW: The sequence of operations performed in the execution of an algorithm. [IEEE83].

CONTROL PATH: The sequence of control statements that affect the execution of a particular program path.

CONSTRUCTIVE ASSERTION: An assertion used by the constructive program verification method.

CONSTRUCTIVE PROGRAM VERIFICATION: Correctness proofs are established constructively by interweaving the generation of program statements and their accompanying assertions with proof justifications.

CONTROL FLOW: The sequence of operations performed in the execution of an algorithm. [IEEE83]

CONTROL LOCATION PREDICATES: A set of axioms which state how control behaves in a construct.

CONTROL STRUCTURE: (1) A construct that determines the flow of control through a computer program. [IEEE83]. (2) The sequence of control constructs performed in the execution of a program.

COOPERATION PROOF: A proof in which processes cooperate. That is, the process interactions maintain the global assertion and all local assertions which are made.

CORRECTNESS: *See PROGRAM CORRECTNESS.*

CORRECTNESS PROOF: *See PROOF OF CORRECTNESS.*

CORRECTNESS SPECIFICATIONS: In the constructive method, a sublanguage which formally describes the specifications of the program.

CORRELATION PRINCIPLE: There exists a narrow correlation between specification and implementation structure.

COUPLING: A measure of the strength of data interconnection among modules.

COUPLING EFFECT: An assumption used in mutation testing which states that test data that can distinguish between programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes from more complex errors.

COVERAGE ANALYZER: A software tool which determines and assesses measures associated with the invocation of program structural elements to determine the adequacy of a test run.

COVERAGE CRITERIA: Usually applied to coverage of a program's logic, coverage criteria specify that each statement, branch, or path must be executed at least once during program testing.

COVERAGE MEASURE: *See TESTING COVERAGE MEASURE.*

COVERAGE MONITOR: See *COVERAGE ANALYZER.*

CRITICAL RANGE: Metric values used to classify software into categories of acceptable, marginal and unacceptable. [IEEE88]

CRITICAL SECTION: A segment of code to be executed mutually exclusively with some other segment of code is called a critical section. Segments of code are required to be executed mutually exclusively if they make competing uses of a computer resource or data item. [IEEE83]

CRITICAL VALUE: Metric value of a validated metric which is used to identify software which has unacceptable quality. [IEEE88]

CROSS-REFERENCER: (1) A computer program that provides cross-reference information

on system components. For example, programs can be cross-referenced with other programs, macros, parameter names, etc. This capability is useful in problem-solving and testing to assess impact of changes to one area or another. (2) A utility program which provides cross-reference data concerning a program written in a higher level language. These utility programs analyze a source program and provide as output such data as follows: 1. Statement label cross-index, 2. Data name cross-index, 3. Literal usage cross-index, 4. Inter-subroutine call cross-index, 5. Statistical counts of statement types.

CYCLOMATIC COMPLEXITY: The cyclomatic complexity of a program is equivalent to the number of decision statements plus 1. [Adri82]

DATA: A representation of facts, concepts, or information in a formalized manner suitable for communication, interpretation, or processing by human or automated means. [IEEE83]

DATA ABSTRACTION: The result of extracting and retaining only the essential characteristic properties of data by defining special data types and their associated functional characteristics, thus separating and hiding the representation details. [IEEE83]

DATA FLOW ANALYSIS: Consists of the graphical analysis of collections of (sequential) data definitions and reference patterns to determine constraints that can be placed on data values at various points of executing the source program.

DATA FLOW ANOMALY: A sequence of the events reference (r), definition (d), and of variables in a program that is either erroneous in itself or often symptomatic of an error.

DATA FLOW TESTING: A testing technique which provides a set of successively more stringent path selection criteria that guide the selection of test data to examine the relationships between variable definitions and variable uses.

DATA STRUCTURE: A formalized representation of the ordering and accessibility relationships among data items without regard to their actual storage configuration. [IEEE83]

DATA TYPE: A class of data characterized by the members of the class and the operations that can be applied to them; for example, integer, real, logical. [IEEE83]

DATA-ABSTRACTION IMPLEMENTATION, SPECIFICATION, AND TESTING SYSTEM (DAISTS): An compiler-based testing system which supports testing the implementation of abstract data types against user-defined algebraic axiomatic specifications of those data types.

DATA-INTERFACE ANALYSIS: A form of interface analysis which examines the transformations of one type of data into another type based on available definitions of allowable transformations.

DE-EUTROPHICATION MODEL: A reliability model, based on exponential failure intensity in terms of time, developed by Jelinski and Moranda.

DEADLOCK: The state in which two or more processes are waiting for a resource that is held by the other.

DEADNESS FAULT: A fault which occurs when part of a concurrent computation can no longer proceed due to a task communication failure.

DEBUGGER: A software tool intended to assist the user in software fault localization and, potentially, fault correction.

DEBUGGING: The process of correcting syntactic and logical faults detected during testing. Debugging shares with testing certain techniques and strategies, but differs in its usual ad hoc application and local scope. [Adri82]

DECENTRALIZED (SDS) ARCHITECTURE: A Strategic Defense System architecture in which important battle management decisions are made locally on a platform. (Note that command and control decisions may still be made in a global, centralized fashion.)

DECISION NODE: A node in the program

digraph which corresponds to a decision statement within the program. [Mill81]

DECISION STATEMENT: A statement in which an evaluation of some predicate is made that (potentially) affects the subsequent execution behavior of the module. [Mill81]

DECISION-TO-DECISION PATH: *See SEGMENT.*

DEDUCTIVE SYSTEM: A deductive system is composed of axioms and rules of inference by which valid statements, or theorems, may be derived from the axioms and other theorems.

DEGRADED SYSTEM: A degraded system is one in which some functionality has been surrendered in order to allow continued processing of critical functions after a failure has occurred.

DENOTATIONAL MODEL OF PROGRAMMING NOTATION: The semantics of programming constructs of an abstract programming language are defined by semantic valuation functions.

DENOTATIONAL SEMANTIC DESCRIPTION OF Ada: A description of Ada using a denotational model.

DEPLOYMENT: The operational employment of a system in its intended, target environment.

DESIGN: (1) The process of defining the software architecture, components, modules, interfaces, test approach and data for a software system to satisfy specified requirements. (2) The results of the design process. [IEEE83]

DESIGN SPECIFICATION: A specification that documents the design of a system or system component; for example, a software configuration item. Typical contents include system or component algorithms, control logic, data structures, data set-use information, input/output formats, and interface descriptions. [IEEE83]

DESIGN-BASED FUNCTIONAL TESTING: The application of test data derived through functional analysis (*see FUNCTIONAL TESTING*) extended to include design functions as well as requirement functions. [Adri82]

DESK CHECKING: The manual simulation of program execution to detect faults through step-by-step examination of the source code for faults in logic or syntax. *See also STATIC ANALYSIS.* [IEEE83]

DETAILED DESIGN: (1) The process of refining and expanding the preliminary design to contain more detailed descriptions of the processing logic, data structures, and data definitions, to the extent that the design is sufficiently complete to be implemented. (2) The result of the detailed design process. [IEEE83].

DETERMINISM: The property of a transformation process that the same outputs are always produced for a given set of inputs. [DACS79].

DETERMINISTIC PROGRAMS: Those programs in which control flow is deterministic.

DEVELOPMENTAL TEST AND EVALUATION: Test and evaluation that focuses on the technological and engineering aspects of the system, or equipment items. [DACS79]

DIFFERENTIAL MODEL: A reliability model proposed by Littlewood to account for the possibility that some faults are more likely to occur than others.

DIGRAPH: Short name for directed graph. [Mill81]

DIRECTED GRAPH: Consists of a set of nodes interconnected with oriented arcs. An arbitrary directed graph (digraph) may have many entry nodes and many exit nodes. A program digraph has only one entry and one exit. [Mill81]

DIRECT METRIC: A metric that represents and defines a software quality factor, and which is valid by definition (e.g., mean-time to software fault of 1000 operating hours for the factor reliability). [IEEE88]

DISTRIBUTED ARCHITECTURE: A system architecture in which there is not a global address

space and which is often geographically distributed.

DISTRIBUTED PROCESSING SYSTEM: (1) A cooperative distributed processing system is defined as a collection of interconnected processing elements with decentralized control that permits cooperation among processors for the execution of a single task. (2) Distributed systems are an appropriate response to distributed functions to be performed. The functions may be distributed geographically, operationally, or managerially. The important characteristic is that they be functionally independent of one another and have weak, well-defined data flow oriented interactions. (3) A cooperative arrangement of interconnected computers whose quasi-autonomous operations are coordinated by a reassignable executive program. [DACS79]

DOMAIN ERROR: Is an incorrect path domain that occurs due to path selection or missing path faults.

DOMAIN TESTING: A testing technique that generates test data to detect domain errors in a program. Detection of domain errors is guaranteed within a quantifiable error bound.

DYNAMIC ALLOCATION: The allocation of addressable storage and other resources to a program while the program is executing. [IEEE83]

DYNAMIC ANALYSIS: The process of evaluating a program based on execution of the program. [IEEE83]

DYNAMIC ASSERTION: A technique which inserts assertions about the relationship between program variables into the program code. The truth of the assertions is determined as the program executes. [Adri82]

DYNAMIC BINDING: Binding performed during execution of a program. Contrast with *STATIC BINDING*. [IEEE83]

DYNAMICALLY RECONFIGURING: See *DYNAMIC RESTRUCTURING*.

DYNAMIC RESTRUCTURING: The process of changing software components or structure while a system is running. [IEEE83]

EDGE: In a digraph, the oriented connection between two nodes. Also called an arc. [Mill81]

EFFICIENCY: The extent to which the software performs its intended functions with a minimum consumption of computing resources. [IEEE83]

EFFICIENT ESTIMATOR: If two different estimators have the same expectation, then the one with the smaller variance is said to be more efficient.

ELEMENTARY COMPUTATIONAL STRUCTURES: In a program, those objects such as references to variables, arithmetic expressions and relations, and Boolean expressions that may appear independently or as part of a more complex component.

EMULATOR: Hardware, software, or firmware that supports the imitation of all or part of one computer system by another. [IEEE83]

ENTRY NODE: In a program digraph, a node which has more than one outway and zero inways. An entry node has an in-degree of zero and a non-zero out-degree. [Mill81]

ENVIRONMENT: The combination of all external or extrinsic conditions that affect the operation of an entity. [DACS79]

ENVIRONMENT SIMULATOR: An automated replication of the external world constructed for testing.

EQUATE: An automated testing system which merges weak mutation testing and perturbation testing to find faults in the execution of paths in an Ada program.

EQUIVALENCE PARTITIONING: A test data selection technique based on considerations of (1) partitioning the input domain of a program into a finite number of equivalence classes such that a test of a representative value of each class is equivalent to a test of any other value and (2)

each test case should invoke as many different input conditions as possible in order to minimize the total number of test cases necessary.

ERROR: (1) A discrepancy between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. [IEEE83] (2) A mental mistake made by a programmer which may result in a program fault.

ERROR CHECKLIST: A list of errors that must be looked for during an inspection. The list is compiled from errors that have frequently been found during prior inspections.

ERROR CORRECTION MODEL: A model to estimate the mean correction time.

ERROR CORRECTION RATE: Number of errors corrected per unit of time.

ERROR COUPLING EFFECT: The assumption, used in mutation testing, that test data on which all simple mutants fail is so sensitive that it is highly likely that all complex mutants must also fail.

ERROR GUESSING: A test data selection technique. The selection criteria is to pick values that seem likely to cause [failures]. [Adri82]

ERROR OPERATOR: A transformation applied to a program to produce a mutation of the program that contains a specific type of fault. Test data that can distinguish between the original and mutated programs is said to be adequate for detecting that fault.

ERROR QUEUE LENGTH: Number of errors detected waiting to be processed by the fault-correction personnel.

ERROR SEEDING: The process of intentionally adding a known number of faults to those already in a program for the purpose of estimating the number of indigenous faults in the program. [IEEE83]

ERROR SENSITIVE TEST CASE ANALYSIS (ESTCA): A testing technique that provides rules for generating test data sensitive to commonly occurring faults.

ERROR-BASED TESTING: Testing where information about programming style, error-prone language constructs, and other programming knowledge, is applied to select test data capable of detecting faults, either a specified class of faults or all possible faults.

EVALUATION: The process of examining a system or system component to determine the extent to which specified properties are present.

EVOLUTIONARY DEVELOPMENT AND DEPLOYMENT: A paradigm for constructing computer systems where the system is developed and deployed in a series of versions with increasing functionality.

EXCEPTION: An event that causes suspension of normal program execution. [IEEE83]

EXCEPTION HANDLING: A set of programming techniques for recognizing and acting upon exceptions.

EXECUTABLE SPECIFICATION: A specification which is given in a sufficiently formal notation to allow its execution by a computer.

EXECUTABLE STATEMENT: A statement in a module which is executable in the sense that it produces object code instructions. [Mill81]

EXECUTION: The process of carrying out an instruction or the instructions of a computer program by a computer. [IEEE83]

EXECUTION ENVIRONMENT: See *ENVIRONMENT*.

EXECUTION PATH: *See Path.*

EXECUTION TIME: (1) The amount of actual or central processor time used in executing a program. (2) The period of time during which a program is executing. See also *RUN TIME*. [IEEE83]

EXHAUSTIVE TESTING: Executing the program with all possible combinations of values for

program variables. [Adri82]

EXIT NODE: In a digraph, a node which has more than one inway, but has zero outways. An exit node has zero out-degree, and a non-zero in-degree. [Mill81]

EXPECTED VALUE: Mean of a random variable. [Musa87]

EXPRESSION ANALYSIS: A form of static error analysis that detects certain commonly occurring faults associated with the evaluation of expressions; for example, incorrect or incomplete parentheses.

EXPRESSION SET: In the EQUATE system, the set of all expressions and subexpressions from the abstract syntax tree of the module under test.

EXTENT: The extent of a fault-based testing technique reflects the scope of information used to determine the absence of a predefined set of possible faults. It may be local or global.

EXTREMAL TEST DATA: Test data that is at the extreme or boundary of the domain of an input variable or which produces results at the boundary of an output domain. [Adri82]

FACTOR SAMPLE: A set of factor values which is drawn from the metrics data base and used in metrics validation. [IEEE88]

FACTOR VALUE: A value (see metric value of the direct metric that represents a factor. [IEEE88]

FAIL-SAFE: A fail-safe system limits the amount of damage caused by a failure and may not strive to continue functionality.

FAIL-SOFT: A fail-soft system continues operation but provides only degraded performance or reduced functional capabilities until the fault is removed.

FAILURE: The inability of a system or system component to perform a required function within specified limits. A failure may be produced when a fault is encountered. [IEEE83]

FAILURE COUNT MODEL: Software reliability model in which the failure process is represented as a stochastic process with a time dependent failure rate. [Goel85].

FAILURE DETECTION RATE: Number of failures detected per unit of time.

FAILURE HISTORY: In software reliability modeling, the record of software failures, in terms of failure times or failure counts per interval.

FAILURE INTENSITY: Failures per unit of time, the derivative with respect to time of the mean value function of failures. [Musa87]

FAILURE INTENSITY DECAY PARAMETER: In the logarithmic Poisson execution time model, the parameter that represents the rate of exponential decay of the failure intensity as a function of mean failures experienced. [Musa87].

FAILURE INTERVAL: Time between failures. [Musa87]

FAILURE PROBABILITY: The probability of failure under specified conditions.

FAILURE RATE: The ratio of the number of failures to a given unit of measure; for example, failures per unit of time, failures per number of transactions, failures per number of computer runs. [IEEE83]

FAILURE SEVERITY: Classification of a failure by its operational impact. [Musa87]

FALSE ASSUMPTION DECOMPOSITION ERROR: Errors resulting from incorrect assumptions about the meaning or usage of data.

FAULT: A manifestation of an error in software. A fault, if encountered, may cause a failure. [IEEE83]

FAULT CORRECTION RATE: Number of failures corrected by the failure-correction personnel per unit of time.

FAULT CORRECTION PROFILES: The profiles (in terms of number of people, and fault correction rate) of the failure correction-personnel assumed for a particular model.

FAULT DENSITY: Probability density of the failures.

FAULT REDUCTION FACTOR: Net reduction in faults per failure experienced. [Musa87]

FAULT SEEDING MODEL: Software reliability model in which the number of indigenous faults in a program is estimated from the number of seeded faults and indigenous faults that are detected. [Goel85].

FAULT-BASED TESTING: Testing which employs a test data selection strategy designed to generate test data capable of demonstrating the absence of a prespecified set of faults; typically frequently occurring faults.

FAULT-TOLERANCE: The probability that a system detects, recovers, and insulates itself from the effects of specified component faults or failures in order to maintain a high degree of availability when operated under stated conditions for a specified period of time. [DeMi88]

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. [DACS79]

FAULT-TOLERANCE TECHNIQUES: Programming techniques which increase the fault-tolerance of a system or system component; for example, N-version programming, recovery blocks.

FAULT TREE: The tree built during (software) fault tree analysis which is developed using backward reasoning to the identify the causes and conditions which may lead to a critical failure.

FAULT TREE ANALYSIS: A form of safety analysis that assesses hardware safety to provide failure statistics and sensitivity analyses which indicate the possible effect of critical failures.

FIDELITY: Fidelity is defined as the accuracy with which a given algorithm is mechanized for a given operating system and hardware system. [DACS79]

FILE COMPARATOR: A software tool which compares two files to identify discrepancies between them.

FINITE INPUT SEQUENCES: A finite set of symbols which forms a string of characters used as the input for some process.

FINITE STATE MACHINE: A computational model consisting of a finite number of states, and transitions between these states. [IEEE83]

FIRM MUTATION TESTING: A version of mutation testing which merges the strengths of strong and weak mutation testing by using components with more extensive scope than weak mutation testing and allowing several mutants to be applied in a single program execution.

FIRST-ORDER LOGIC: *See PREDICATE CALCULUS.*

FIRST-ORDER PREDICATE CALCULUS: *See PREDICATE CALCULUS.*

FLAVOR ANALYSIS: A form of analysis used in testing large scale software systems to detect incorrect assumptions about the meaning and usage of data.

FLEXIBILITY: The effort to extend the software mission, functions, or data to satisfy other requirements. [RADC83]

FLOWCHART: A graphical representation of the definition, analysis, or solution to a problem in which symbols are used to represent operations, data, flow, and equipment. [IEEE83]

FOLLOW-ON OPERATIONAL TEST AND EVALUATION (FOT&E): Operational test and evaluation conducted after deployment of a system. For example, to validate assumptions made in previous operational testing.

FORMAL DEVELOPMENT METHODOLOGY (FDM): An automated verification system, developed by the System Development Corporation and employing the Ina Jo language.

FORMAL LANGUAGE: A language whose rules are explicitly established prior to its use. Synonymous with artificial language. Examples include programming languages, such as FORTRAN and Ada, and mathematical or logical languages, such as predicate calculus. [IEEE83]

FORMAL SEMANTICS: The mathematical definition of the semantics of a language.

FORMAL SOFTWARE DEVELOPMENT: The use of formal methods to specify, verify, and test software.

FORMAL SPECIFICATION: In proof of correctness, a description in a formal language of the externally visible behavior of a system or system component. Generally, a specification written and approved in accordance with established standards. [IEEE83]

FORMAL VERIFICATION: *See VERIFICATION.*

FUNCTION: (1) A specific purpose of an entity or its characteristic action. (2) A subprogram that is invoked during the evaluation of an expression in which its name appears and that returns a value to the point of invocation. [IEEE83]

FUNCTIONAL ABSTRACTION: A design strategy in which programs are viewed as a hierarchy of abstract functions.

FUNCTIONAL REQUIREMENT: A requirement that specifies a function that a system or system component must be capable of performing. [IEEE83]

FUNCTIONAL SPECIFICATION: A set of behavioral and performance requirements which, in aggregate, determine the functional properties of a software system. [Mill81]

FUNCTIONAL TESTING: Application of test data derived from the specified functional requirements without regard to the final program structure. [Adri82]

GENERATOR: In the EQUATE system, the expression set term whose subexpression was modified in the derivation of operand substitution terms.

GENERIC COMPONENT: A generic component is one which can be instantiated in a number of predefined ways so that each occurrence of the component can be tailored to suit a particular usage. For example, a generic component which provides a set of queue handling routines might be designed so that it can be instantiated to operate on queues with different message formats.

GEOMETRIC MODEL: A reliability model proposed by Moranda as a variation of the De-Eutrophication model.

GEOMETRIC POISSON MODEL: A reliability model proposed by Moranda as an alternative to the Geometric model.

GLOBAL ASSERTION: Those assertions which are valid for the whole program being validated.

GLOBAL INVARIANT: Those assertions which do not change for the whole program.

GOAL/QUESTION/METRIC PARADIGM: A measurement approach which aids in determining and specifying the goals of a software development project.

GRAMMAR-BASED TESTING: A testing method that generates test cases from a formal specification of a system or system component.

GRAPH: A model consisting of a finite set of nodes having connections called edges or arcs. [IEEE83]

GYPSY SPECIFICATION LANGUAGE: A language consisting of two intersecting components: a formal specification language and a verifiable high-level programming language.

GYPSY VERIFICATION ENVIRONMENT (GVE): An automated verification system developed at the University of Texas at Austin and employing the Gypsy language.

HARDWARE: Physical equipment used in data processing, as opposed to computer programs, procedures, rules, and associated documentation. Contrast with *SOFTWARE*. [IEEE83]

HARDWARE-IN-THE-LOOP SIMULATION: A simulation that includes one or more physical elements of the system being simulated, interacting with the software models of the remaining system elements.

HAZARD: A set of conditions (state) that has an unacceptable risk of leading to an accident, given certain environmental conditions.

HAZARD FUNCTION: (1) The probability that an error occurring in a given infinitesimal time interval given that no error has occurred previously to that interval. (2) Instantaneous failure rate of a system. (Musa's model.) (3) The error-rate relationship. [DACS79]

HAZARD RATE: Probability density (per unit of time) of failure given that failure has not occurred up to the present. [Musa87]

HEURISTIC: An exploratory method of problem solving in which solutions are discovered by evaluation of the progress made toward the final result. Contrast with *ALGORITHM*. [DACS79]

HIERARCHICAL DEVELOPMENT METHODOLOGY (HDM): An automated verification system developed by SRI International and employing the SPECIAL state-machine language.

HOARE LOGIC: A logic in which the behavior of a statement S is specified by an assertion P describing possible states before execution of S, and a second assertion Q describing possible states after the execution of S.

HOMOGENEOUS: Processing characteristics that do not vary with time.

HOST MACHINE: A computer used to develop software intended for another computer. [IEEE83]

IMPLEMENTATION: (1) The implementation of a program is either a machine executable form of the program, or a form of the program that can be automatically translated (e.g., by compiler or assembler). (2) That process by which an architectural design is turned into a delivered program. It includes the detailed functional and procedural design, coding, testing, and documentation necessary to meet program requirements, either for new or modified software. [DACS79]

IMPROVEMENT PARADIGM: A paradigm which guide activities necessary to better understand and learn from the software construction process.

INA JO: A non-procedural specification language based on an extension of first-order predicate calculus, used in the Formal Development Methodology automated verification system.

INCREMENTAL ANALYSIS: Occurs when (partial) analysis may be performed on an incomplete product to allow early feedback on the development of that product.

INCREMENTAL TESTING: See *INCREMENTAL ANALYSIS*.

INDEPENDENT VALIDATION AND VERIFICATION (IV&V): Verification and validation of a software product by an organization that is both technically and managerially separate from the organization responsible for developing the product. [IEEE83]

INDETERMINISM: Inverse of *DETERMINISM*.

INDUCTION: The use of a mathematical technique that employs reasoning from a part to a whole.

INDUCTIVE ASSERTION METHOD: A proof of correctness technique in which assertions are written describing program inputs, outputs and intermediate conditions, a set of theorems is developed relating satisfaction of the input

assertions to satisfaction of the output assertions, and the theorems are proved to be true. [IEEE83]

INEVITABILITY: A program will inevitably establish predicate *R* in the computation started in state *S* if and only if for every sequence *t* in the computational history ot *S* there is an initial section *r* of *t* such that *R* holds in the last state of *r*.

INFEASIBLE PATH: A sequence of program statements that can never be executed. [Adri82]

INFERENCE RULES: The basic building blocks of formal proof. They generally consist of a number of hypotheses and a conclusion, the idea being that the validity of the conclusion can be inferred from the validity of all the hypotheses.

INFERENCE SYSTEM: A set of inference rules.

INFORMATION HIDING: The technique of encapsulating software design decisions in modules in such a way that the module's interfaces reveal as little as possible about the module's inner workings; thus, each module is a "black box" to the other modules in the system. The discipline of information hiding forbids the use of information about a module that is not in the module's interface specification. [IEEE83]

INITIAL OPERATIONAL TEST AND EVALUATION: The first phase of operational test and evaluation conducted on preproduction items, prototypes, or pilot production items and normally completed prior to the first major production decision. It is conducted to provide a valid estimate of expected system operational effectiveness and suitability.

INITIAL VALUE SET: In the EQUATE system, the set of values first taken on by each expression set term at each test location during testing.

INPUT: *See PROGRAM INPUT.*

INPUT ASSERTION: A logical expression specifying one or more conditions that program inputs must satisfy in order to be valid. [IEEE83]

INPUT CLAUSE: See *INPUT ASSERTION.*

INPUT CONDITION: See *INPUT ASSERTION.*

INPUT DOMAIN: *See INPUT SPACE .*

INPUT DOMAIN BASED MODEL: Software reliability model in which reliability is estimated from the fraction of test runs resulting in failure. Failures are weighted according to the operational profile of the software. [Goel85]

INPUT SPACE: Consists of that subset of a module's communication space which can be (1) altered externally to the module, and (2) which is (potentially) used within the module in such a way that affects its execution. [Mill81]

INPUT-SPACE PARTITIONING TESTING TECHNIQUES: Testing techniques which employ a test data generation strategy based on partitioning the input space of a program.

INSPECTION: A formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems. [IEEE83]

INSTRUMENTATION: See *PROGRAM INSTRUMENTATION.*

INTEGRATION: The process of combining software elements, hardware elements, or both into an overall system. [IEEE83]

INTEGRATION TESTING: An orderly progression of testing in which software elements, hardware elements, or both are combined and tested until the entire system has been integrated. [IEEE83]

INTEGRITY: (1) The probability that stored information and data will not be modified by unauthorized means. [DeMi88] (2) The extent to which unauthorized access to the software or data can be controlled. [RADC83].

INTERFACE: 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.

[IEEE83]

INTERFACE ANALYSIS: Checks the interfaces between program elements for consistency and adherence to predefined rules or axioms.

INTERFACE CONTROL: Interface control requires that input/output specifications must be controlled as engineering configuration items at system design, implementation, integration, and operation times. [DACS79]

INTERFERENCE-FREE: *See NON-INTERFERENCE.*

INTERMITTENT ASSERTION METHOD: A formal verification method that proves properties of programs by induction on their space. The method only applies to *while* statements.

INTEROPERABILITY: The effort to couple the software of one system to the software of another system. [RADC83].

INTERPRETER: (1) Software, hardware, or firmware used to interpret computer programs. Contrast with *COMPILER*. [IEEE83]

INTERPROCESS COMMUNICATION: The sending and receiving of messages by the processes/entities within an operating system. [DACS79]

INVARIANCE: A predicate *R* is invariant in the state *S* if and only if *R* is true in every state of every sequence in the computational history of *S* unless the state is blocked or empty.

INVARIANT ASSERTION METHOD: A proof method in which one deduces the correctness of complex statements from the correctness of their components.

INVOCATION: (1) The transfer of control to an entity causing it to be activated. (2) The linking to or insertion of a procedure body by means of a named reference within a procedure. Subroutine linking is sometimes referred to as a "call." Code insertion is referred to as a "macro call." [DACS79].

LEMMA: An intermediate conclusion in the development of the proof of a theorem.

LIFECYCLE: See *SOFTWARE LIFECYCLE.*

LINEAR CODE SEQUENCE AND JUMP PROGRAM UNITS: Sections of the code through which the flow of control proceeds sequentially until terminated by a jump in the control flow.

LIVENESS: A program property that states that a desired state, such as termination, can be reached.

LOGARITHMIC POISSON EXECUTION TIME MODEL: Software reliability model in which the failure process is assumed to be a nonhomogeneous Poisson process with exponentially decreasing failure intensity. [Musa87].

LOGICAL ASSERTIONS: Logical postulates usually used to characterize legitimate program input and output states and hence the effect (semantics) of the program.

LOOP: A set of instructions that may be executed repeatedly while a certain condition prevails. [IEEE83]

LOSS EVENT: In fault tree analysis, the critical failure which is assumed to have occurred and which forms the root of the fault tree.

MAINTAINABILITY: (1) The probability that specified unavailable functions can be repaired or restored to their operational state in the system's intended maintenance environment during a specified period of time. [DeMi88] (2) The average effort to locate and fix a software failure. [RADC83].

MANUAL REVIEWS: *See INSPECTION and WALKTHROUGH.*

MATHEMATICAL INDUCTION: *See INDUCTION.*

MEAN TIME BETWEEN FAILURES (MTBF): The sum of mean time to failure and mean time to repair.

MEAN TIME TO FAILURE (MTTF): (1) A measure of the time elapsed before a failure occurs where units of time may reflect either execution time or calendar time. Used as an indication of software reliability. (2) Expected value of the failure interval. [Musa87]

MEAN TIME TO REPAIR (MTTR): Expected value of the time required to restore to normal operation. [Musa87]

MEASURE: To ascertain or appraise by comparing to a standard; to apply a metric. [IEEE88]

MEASUREMENT: 1) the act or process of measuring; 2) a figure, extent, or amount obtained by measuring. [IEEE88]

METRIC: A measure of the extent or degree to which a product possesses and exhibits a certain quality, property, or attribute. [IEEE83]

METRICS DATA BASE: An organized collection of factor values and corresponding metric values. [IEEE88]

METRICS FRAMEWORK: A tool used for organizing, selecting, communicating and evaluating the required quality attributes for a software system; a hierarchical breakdown of factors, sub-factors, and metrics for a software system. [IEEE88]

METRICS METHODOLOGY: A systematic approach to establishing quality requirements and identifying, implementing, analyzing and validating software quality metrics for a software system. [IEEE88]

METRICS PLAN: A document that contains a complete software quality metrics framework for a system, the set of documented metrics, and the set of documented data items. [IEEE88]

METRICS SAMPLE: A set of metrics values which is drawn from the metrics data base and used in metrics validation. [IEEE88]

METRICS VALIDATION: The act or process of ensuring that a metric correctly predicts a quality factor. [IEEE88]

METRIC VALUE: An element from the range of a metric; a metric output. [IEEE88]

MISSING PATH FAULT: Occurs when a special case requires a unique sequence of actions, but the program does not contain a corresponding path. This type of fault is cause by missing conditional statements.

MODE: A way of operating a program to perform a certain subset of the functions that the entire program can perform, as selected by control data or operating conditions. Often, the mode of a program will be defined as program states, with transitions annotated to delineate events causing the passages between modes of operation. [DACS79]

MODULARITY: Those attributes of the software which provide a structure of highly cohesive modules with optimum coupling. [RADC83]

MODULE: A module is a separately invocable element of a software system. [Mill81]

MODULE-INTERFACE ANALYSIS: A form of interface analysis which examines the interfaces between system components for consistency, completeness, and redundancy.

MOTHRA: An automated testing system which applies mutation testing, structural testing, and a form of functional testing to FORTRAN programs.

MULTI-LEVEL SECURITY: A mode of operation permitting data at various security levels to be concurrently stored and processed in a computer system, when at least some users have neither the clearance nor the need-to-know for all data contained in the system. [IEEE83]

MULTI-TASKING: A method of describing concurrent programs as collections of separate tasks.

MULTI-UNIT TEST: Consists of a unit test of a single module in the presence of other modules. It includes (1) a collection of settings for the input space of the module and all the other modules invoked by it, but (2) precisely one invocation of the module under test. [Mill81]

MUTANT: A mutated form of a program produced by applying an error operator which introduces a predefined fault into a program statement.

MUTATION ANALYSIS: See *MUTATION TESTING.*

MUTATION TESTING: A method to determine test set thoroughness by measuring the extent to which a test set can discriminate the program from slight variants of the program. [Adri82]

MUTATION TRANSFORMATION: *See ERROR OPERATOR.*

MUTUAL EXCLUSION: Mutual exclusion occurs when each process accessing shared data excludes all others from doing so simultaneously.

N-VERSION PROGRAMMING: The independent generation of N>2 functionally equivalent programs from the same initial specification. The N programs possess all the necessary attributes for concurrent execution, during which comparison vectors are generated by the program at certain points. [DACS79]

NETWORK OF PROCESSES: A set of processes executing in parallel and communicating via a communication channel.

NODE: 1) A number assigned to a place within a program text. Generally, nodes are assigned only to executable statements. [Mill81] 2) A vertex in a graph.

NON-EXECUTABLE PATH: See *INFEASIBLE PATH.*

NON-EXECUTABLE STATEMENT: A declaration or directive within a module which does not produce (during compilation) object code instructions directly. [Mill81]

NON-INTERFERENCE: In verification of parallel programs, those assertions which will be valid regardless of the manner in which the programs interact.

NON-PROCEDURAL LANGUAGE: Those languages which do not have procedure call statements in their syntax.

NONHOMOGENEOUS POISSON PROCESS MODEL: A reliability model developed by Goel and Okumoto.

NP-COMPLETE: A problem for which all known solutions do not have a polynomial-time solution.

OPERAND SUBSTITUTION TERMS: In the EQUATE system, the set of expressions that can be formed by substituting any member of the expression set for any subexpression of another expression set member.

OPERATING MODE: See *MODE.*

OPERATING SYSTEM: Software that controls the execution of programs. An operating system may provide services such as resource allocation scheduling, input/output control, and data management. Although operating systems are predominantly software, partial or complete hardware implementations are possible. An operating system provides support in a single spot rather than forcing each program to be concerned with controlling hardware. [IEEE83]

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. [DACS79]

OPERATIONAL ENVIRONMENT: The environment in which a system or system component will be deployed and operate.

OPERATIONAL PROFILE: The expected run time distribution of inputs to a program.

OPERATIONAL RELIABILITY: The reliability of a system or software subsystem in its actual use environment. Operational reliability may differ considerably from reliability in the specified or test environment. [IEEE83].

OPERATIONAL SOFTWARE: See

OPERATIONAL.

OPERATIONAL REQUIREMENTS: Qualitative and quantitative parameters which specify the desired operational capabilities of a system and which will serve as a basis for determining the operational effectiveness and suitability of a system prior to deployment.

OPERATIONAL TEST AND EVALUATION: Formal testing conducted prior to deployment to evaluate the operational effectiveness and suitability of a system with respect to its mission.

OPERATIONAL TESTING: Testing performed by the end user on software in its normal operating environment. (DOD usage) [IEEE83]

OPERATOR: (1) In symbol manipulation, a symbol that represents the action to be performed in an operation. Examples of operators are +,-,*,/. (2) In the description of a process, that which indicates the action to be performed on operands. [IEEE83]

OPERATOR-INTERFACE ANALYSIS: A form of interface analysis which examines the usage of operators applied to data structures.

ORACLE: A mechanism to produce the "correct" responses to compare with the actual responses of the software under test. [Adri82]

OUTPUT: See *PROGRAM OUTPUT.*

OUTPUT ASSERTION: A logical expression specifying one or more conditions that program outputs must satisfy in order for the program to be correct. [IEEE83]

OUTPUT CLAUSE: See *OUTPUT ASSERTION.*

OUTPUT CONDITION: See *OUTPUT ASSERTION.*

OUTPUT SPACE: Consists of the collection of variables, including file actions, which are (or could be) modified by some invocation of the module. [Mill81]

OUTSIDE-IN TESTING: A strategy for integration testing where units handling program inputs and outputs are tested first, and units which process the inputs to produce outputs being incrementally included as the system is integrated.

PAGING: The technique of repeatedly using the same areas of internal storage during different stages of program execution. [IEEE83]

PARAMETER ESTIMATION: The process of establishing parameter values for a model.

PARAMETER PREDICTION: Determination of parameter values from characteristics of the software product and the development process. [Musa87]

PARTIAL CORRECTNESS: Conditional or partial correctness of a program (as opposed to total correctness) is obtained when proving the correctness of a program is based on the assumption that the program terminates.

PARTITION ANALYSIS: A program testing and verification technique which employs symbolic evaluation to provide common representations of a program's specification and implementation. *See also PARTITION ANALYSIS TESTING, PARTITION ANALYSIS VERIFICATION.*

PARTITION ANALYSIS TESTING: The test data selection process used in partition analysis to generate test data based on analysis of both the program specification and implementation.

PARTITION ANALYSIS VERIFICATION: The verification process used in partition analysis which attempts to determine the consistency properties that hold between a program specification and its implementation.

PATH: A sequence of segments. [Mill81]

PATH ANALYSIS: Program analysis performed to identify all possible paths through a program, to detect incomplete paths, or to discover portions of the program that are not on any path. [IEEE83]

PATH COMPUTATION: The function that is computed by the sequence of executable statements along a path. Symbolic evaluation gives a path computation as a vector of the algebraic expressions for the output values.

PATH DOMAIN: Corresponds to a particular execution path in a program and consists of the input data points that cause the path to be executed.

PATH DOMAIN BOUNDARY: The boundary of a path domain determined by the predicate interpretations in the path condition.

PATH EXPRESSION: A logical expression indicating the input conditions that must be met in order for a particular program path to be executed. [IEEE83]

PATH SELECTION ADEQUACY CRITERIA: Criteria which can be used to assess the adequacy of executing a given set of program paths for detecting a specified set of potential faults.

PATH SELECTION CRITERIA: Criteria which specify the set of paths to be executed during program testing.

PATH SELECTION ERROR: Occurs when a program incorrectly determines the conditions under which a path is executed. This may be due to an incorrect conditional statement or an incorrect assignment statement that affects a conditional statement.

PATH TESTING: A test method satisfying coverage criteria that each logical path through the program be tested. Often paths through the program are grouped into a finite set of classes; one path from each class is tested. [Adri82]

PERFORMANCE: The ability of a computer system or subsystem to perform its functions. [IEEE83]

PERFORMANCE REQUIREMENT: A requirement that specifies a performance characteristic that a system or system component must possess; for example, speed, accuracy, frequency. [IEEE83]

PERTURBING FUNCTION: A term added to arithmetic expressions to introduce a known fault. Used in perturbation testing to determine whether particular potential faults may go undetected by a given test path.

PERTURBATION TESTING: A test path adequacy measurement technique that proposes using the reduction of the space of undetectable faults as a criterion for test path selection and is intended to reveal faults in arithmetic expressions.

PETRI NET: A method of analyzing state transitions.

PIECE-WISE EXPONENTIALLY DISTRIBUTED: Applied to the distribution of the execution time between failures means that the hazard rate is a constant that changes only at each error correction.

PORTABILITY: The ease with which software can be transferred from one computer system or environment to another. [IEEE83]

POST-ASSERTION: An assertion attached to the end of a program being verified which is expected to be satisfied whenever execution passes this point.

POTENTIALITY: A program has the potential to establish predicate R in the computation started in state S if and only if there exists a finite sequence r such that r is an initial section of same sequence in the computation history of state S and R holds in the last state of r.

PRECISE INTERFACE CONTROL: An approach to interface analysis which uses requisition of access and provision of access concepts to extend the traditional notion of visibility.

PREDICATE: A logical formula involving variables/constants known to a module. [Mill81]

PREDICATE CALCULUS: A first-order language in which one can make general statements about all objects in a fixed set called the universe. The formulae in this language are constructed out of names for relations and names for

individual objects in the universe.

PREDICATE INTERPRETATION: A constraint equivalent to a program predicate where program variables are replaced by their symbolic values in terms of input variables.

PREDICATE TRANSFORMER: A function that maps an assertion and a syntactic unit into another assertion.

PREDICTIVE ASSESSMENT: The process of using a predictor metric(s) to predict the values of another metric. [IEEE88]

PREDICTIVE METRIC: A metric which is used to predict the values of another metric. [IEEE88]

PROBABILITY: The fraction of occasions on which a specified value or set of values of a quantity occurs, out of all possible values for that quantity. [Musa87]

PROBABILITY DENSITY: Probability per unit variation of random variable. [Musa87]

PROBABILITY DISTRIBUTION: The set of probabilities corresponding to the values that a random variable can take on. [Musa87]

PROCEDURE SUBDOMAIN: A partition of a program's input data such that the elements of the subdomain are treated uniformly by the specification and processed uniformly by the implementation. Used in Partition Analysis.

PROCESS: In a computer system, a unique, finite course of events defined by its purpose or by its effect, achieved under given conditions. [IEEE83]

PROCESS AUGMENTED FLOWGRAPH: An annotated graphical representation of communicating concurrent processes formed by connecting the graphs representing the individual processes with special edges indicating all synchronization constraints.

PROCESS CONTROL SYSTEM: A system embedded in some larger system that interacts with external devices or objects to control ongoing external processes.

PROCESS STEP: Any task performed in the development, implementation or maintenance of software (e.g., identify the software components of a system as part of the design). [IEEE88]

PROCESS METRIC: Metric used to measure characteristics of the methods, techniques, and tools employed in acquiring, developing, verifying, and operating the software system. [IEEE88]

PROCESS PROGRAMMING: The specification of software development processes in a procedural manner (for example, a programming language) which serves to formalize and communicate these processes, facilitate their analysis, and define the necessary interactions and interfaces between automated and manual actions.

PROFILE: A compendium of information which contributes to the definition of an environment. [DACS79]

PRODUCT METRIC: Metric used to measure the characteristics of the documentation and code. [IEEE88]

PROGRAM: *See Module.*

PROGRAM BLOCK: In problem-oriented languages, a computer program subdivision that serves to group related statements, delimit routines, specify storage allocation, delineate the applicability of labels, or segment paths of the computer program for other purposes.

PROGRAM COMPONENT: *See COMPONENT.*

PROGRAM COUNTER: A variable which indicates the program statement currently being executed.

PROGRAM CORRECTNESS: (1) The extent to which software is free from design defects and coding defects; that is, fault free. [IEEE83]. (2) Extent to which the software satisfies its specifications and fulfills the user's mission objects. [RADC83]. (3) If for all initial states that belong to the set of legitimate initial states, the program P terminates with a final state that

belongs to the set of legitimate final states, then program *P* exhibits program correctness.

PROGRAM DEBUGGING: *See DEBUGGING.*

PROGRAM GRAPH: Graphical representation of a program. [Adri82]

PROGRAM INSTRUMENTATION: (1) Probes, such as instructions or assertions, inserted into a computer program to facilitate execution monitoring, proof of correctness, resource monitoring, or other activities. (2) The process of preparing and inserting probes into a computer program. [IEEE83]

PROGRAM PATH: *See PATH.*

PROGRAM PREDICATE: *See PREDICATE.*

PROGRAM PROVING: The act of demonstrating that a program is correct.

PROGRAM SPECIFICATION: The formalization that precisely states the requirements and objectives which the program is to satisfy.

PROGRAM TESTING: *See TESTING.*

PROGRAM TEXT: The set of statements, executable and non-executable, which make up a module. Program text is expressed in a programming language. [Mill81]

PROGRAM TRACE: A record of the execution of a computer program; it states the sequence in which the instructions were executed.

PROGRAM TRANSFORMATION: To replace one segment of a program description by another, equivalent description. [DACS79]

PROGRAM VERIFICATION: The act of demonstrating that a program achieves some intended purpose.

PROGRAM UNIT INVOCATION: See *INVOCATION.*

PROGRAMMING LANGUAGE: An artificial language designed to generate or express programs. [IEEE83]

PROOF: A structure of valid applications of inference rules to obtain a conclusion (proof).

PROOF JUSTIFICATION: Concerns establishing, in a precise algorithmic notation, the reasoning required to determine the validity of the assertions.

PROOF CHECKER: A program that checks formal proofs of program properties for logical correctness.

PROOF OF CORRECTNESS: A formal technique used to prove mathematically that a program satisfies its specifications. [IEEE83]

PROOFS OF PROGRAMS: *See PROOF OF CORRECTNESS.*

PROOF OF SOUNDNESS: Proof that all statements in the theory that are derived from theorems (true statements) by rules of inference of the theory must be true.

PROOF RULES: The inference rules that permit the derivation of more complex theorems, including theorems about the semantics of a complete program.

PROTOTYPE: A limited implementation of a system built in order to capture or validate some aspects of a system design. The fundamental concept is that a prototype of a system is more cheaply or more quickly constructed than the actual system. Hence, some aspects of function or execution speeds are typically sacrificed.

PROTOTYPING: A discipline of system design where the function of the actual system is captured in a series of increasingly accurate prototypes.

PROVISION OF ACCESS: In the AdaPIC system, provision of access occurs when a entity grants the right of reference, or use, to some set of entities.

PURIFICATION DEGREE: The ratio of change in the hazard rate function from the beginning of

testing to the end versus what it was at the beginning.

QUANTIFIER-FREE FORMULAE: Formulae in which there are no operations that bind the variables in a logical formula by specifying their quantity.

QUALIFICATION TESTING: Formal testing, usually conducted by the developer for the customer, to demonstrate that the software meets its specified requirements. See also *SYSTEM TESTING*.

QUALITY: The degree to which a program possesses a desired combination of attributes that enable it to perform its specified end use.

QUALITY ASSURANCE: A planned and systematic pattern of all actions necessary to provide adequate confidence that the item or product conforms to established technical requirements. [IEEE83]

QUALITY ASSURANCE METHOD: An approach for reducing the risk associated with a software system and one or more properties.

QUALITY ATTRIBUTE: A characteristic of software; a generic term applying to factors, subfactors, or metric values. [IEEE88]

QUALITY SUB-FACTOR: A decomposition of a quality factor or quality sub-factor. [IEEE88]

QUALITY FACTOR: An attribute of software that contributes to its quality. [IEEE88]

QUALITY REQUIREMENT: A requirement that a software attribute be present in software to satisfy a contract, standard, specification, or other formally imposed document. [IEEE88]

RANDOM: Possessing the property of having more than one value at one time, each occurring with some probability. [Musa87]

RANDOM TESTING: An essentially black-box testing approach in which a program is tested by randomly choosing a subset of all possible input values. The distribution may be arbitrary or may attempt to accurately reflect the distribution of inputs in the application environment.

RANDOM VARIABLE: A variable that possesses the property of randomness (*see RANDOM*). [Musa87]

REAL TIME CONSTRAINTS: Those constraints imposed by the environment in which the system is going to operate.

REAL-TIME: Pertaining to the processing of data by a computer in connection with another process outside the computer according to time requirements imposed by the outside process. This term is also used to describe systems operating in conversational mode, and processes that can be influenced by human intervention while they are in progress. [IEEE83]

REASONING SYSTEMS: Systems capable of performing the deduction of logical expressions from other logical expressions.

RECONFIGURATION: Adjustment of the relationships between the software modules in a software system or hardware devices in a hardware system.

RECOVERY BLOCK: Software fault tolerance mechanism. A recovery block consists of a conventional [program] block which is provided with a means of error detection (an acceptance test) and zero or more stand-by spares (the additional alternates). [Rand75].

RECURSION: An initial condition is defined, and the transformation from one condition to the next is defined in terms of the previously defined conditions.

RECURSION INDUCTION: To prove that $g=h$ (a) show that g and h satisfy the defining equation for some other function f, and (b) show that f holds over the domain of interest.

RECURSION THEOREM: A theorem about primitive and partial recursive functions due to Kleene.

RECURSIVE FUNCTION THEORY: Each recursive function is defined by combining some initial functions using composition, recursion, and minimalization.

RECURSIVE PROGRAMS: Those programs that have or use recursive procedures or functions.

REFERENCE ANALYSIS: A form of static error analysis which can detect reference anomalies; for example, when a variable is referenced along a program path before it is assigned a value along that path.

REGRESSION TESTING: Selective retesting to detect faults introduced during modification of a system or system component, to verify that modifications have not caused unintended adverse effects, or verify that a modified system or system component still meets its specified requirements. [IEEE83]

REGULARITY HYPOTHESIS: The regularity hypothesis for a level n consists in assuming that if the test is successful for data of complexity less than n, then the program behaves correctly for any value.

RELAY: A fault-based test data selection technique based on defining revealing conditions that guarantee that a fault originates failure during execution and that the failure transfers through computations and data until it is revealed.

RELIABILITY: (1) The probability that the software will perform as intended under stated conditions for a specified period of time. [DeMi88] (2) The probability that the software will perform its logical operations in the specified environment without failure. [RADC83]

RELIABILITY ASSESSMENT: The process of determining the achieved level of reliability of an existing system or system component.

RELIABILITY MODEL: A model used for predicting, estimating, or assessing reliability. [IEEE83]

RELIABILITY GROWTH MODEL: A reliability model which takes account of improvements in reliability that result from correcting faults in the software.

RELIABLE TEST DATA: A set of test data T is reliable for program P if it reveals that P contains an fault whenever P is incorrect.

RELIABLE TEST DATA SELECTION STRATEGY: A test data selection strategy is reliable if it guarantees to generate test data capable of detecting every fault in a program.

RENDEZVOUS: The interaction that occurs between two parallel tasks when one task has called an entry of the other task, and a corresponding accept statement is being executed by the other task on behalf of the calling task. [IEEE83]

REQUIREMENT: A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed document. The set of all requirements forms the basis for subsequent development of the system or system component. [IEEE83]

REQUIREMENTS LANGUAGE: A language used to provide a succinct and unambiguous specification of the required system capabilities.

REQUIREMENTS SPECIFICATION: A specification that sets forth the requirements for a system or system component; for example, a software configuration item. Typically included are functional requirements, performance requirements, interface requirements, design requirements, and development standards. [IEEE83]

REQUISITION OF ACCESS: In the AdaPIC system, requisition of access occurs when an entity requests the right to refer to, or make use of, some set of entities.

RESOURCE ASSIGNMENT: The process of granting the request for a resource by a task.

RETESTING: *See REGRESSION TESTING.*

REUSABILITY: The effort to convert a software component for use in another application. [RADC83].

REUSABLE THEORIES: Formal reasoning rules that can be used by more than one system.

REVEALING SUBDOMAIN: A subset of a program's input domain is revealing if the existence of one incorrectly processed input implies that all of the subset's elements are processed incorrectly. Conversely, if one input is processed correctly, all elements in the subdomain are processed correctly.

ROBUSTNESS: The extent to which software can continue to operate correctly despite the introduction of invalid inputs. [IEEE83]

RUN-TIME: The instant at which a program begins to execute. [IEEE83]

RUN-TIME ENVIRONMENT: The environment in which a program executes, either the host or target environment.

RUN-TIME SYSTEM: A set of software routines added to a compiled program, typically at link time, to implement the semantics intended by the compiler.

RUN-TIME SCHEDULER: Software which allocates processing resource to parallel tasks.

SAFE SYSTEM: A system which prevents unsafe states from producing safety failures.

SAFETY: The extent to which the program is protected from exposure to a specified set of hazards. [DeMi88]

SAFETY ANALYSIS: Identification of the possible causes, and evaluation of the possible consequences, of critical system failures. Intended to determine the necessary fault tolerance or other mechanisms needed to ensure safe operation in the system under various operating conditions and modes.

SAFETY FAILURE: A failure which leads to casualties or otherwise serious consequences.

SAFETY PROPERTY: A program property that is satisfied if conditions or actions that should never happen within a program never occur.

SATISFIABILITY: Concerns the existence of an interpretation that satisfies the verification conditions in a proof of correctness.

SCHEDULER: A computer program which allocates resources to waiting processes to allow them to execute in an efficient or prioritized manner.

SCHEDULING ALGORITHM: A set of rules used to determine how available processing resources should be allocated to parallel tasks based on priorities of the tasks.

SCOPE: The range within which an identified unit displays itself. Scope of activity refers to the boundaries within which a data structure or program element remains an integral unit. Scope of control refers to the submodules in a program that potentially may execute if control is given to a cited module. Scope of error denotes the set of submodules that are potentially affected by the detection of a fault in a cited module. [DACS79]

SECURITY: The extent to which computer hardware, software, and resident information and data are protected from specified threats such as unauthorized access, use, modification, destruction, transmission, or disclosure. [DeMi88]

SEGMENT: A (logical) segment or decision-to-decision path, is the set of statements in a module which are executed as a result of the evaluation of some predicate (conditional) within the module. It begins at an entry or decision statement and ends at a decision statement or exit, and should be thought of as including the sensing of the outcome of a conditional operation and the subsequent statement execution up to and including the computation of the next predicate value, but not including its evaluation. [Mill81]

SELF-CHECKING SOFTWARE: Software which makes an explicit attempt to determine its own correctness and to proceed accordingly.

SEMANTICS: (1) The relationship of characters

or groups of characters to their meanings, independent of the manner of their interpretation and use. (2) The relationships between symbols and their meanings. [IEEE83]

SEMANTIC CHARACTERIZATION: Determining the approach used in the formal definition of the semantics of programming language constructs.

SEMANTIC VALUATION FUNCTIONS: Semantic valuation functions map programming constructs to the values (numbers, truth values, function, and so on) that they denote.

SENSITIVITY ANALYSIS: In safety analysis, that analysis which assesses the potential impact of a potentially critical failure on the ability of the system to perform its mission.

SENSITIVITY FOCUS: In the context of regression testing, the concern that the amount of retesting required after a software change is proportional to the extent of that change.

SEQUENTIAL PROCESSES: Processes that execute in such a manner that one must finish before the next begins. [IEEE83]

SEQUENTIAL PROOF: A formal proof made for sequential processes.

SHARED VARIABLE: A variable shared by more than one process.

SHARED VARIABLE COMMUNICATION: A variable shared by more than one process and used to communicate between processes.

SIDE EFFECT: Processing or activities performed, or results obtained, secondary to the primary function of a program, subprogram, or operation. [IEEE83]

SIMULATION: Use of an executable model to represent behavior of an object. The computational hardware, external environment, and even code segments may be simulated. [Adri82]

SOFTWARE: (1) Computer programs, procedures, rules, and possibly associated documentation and data pertaining to the operation of a computer system. (2) Programs, procedures, rules, and any associated documentation pertaining to the operation of a computer system. Contrast with *HARDWARE*. [IEEE83]

SOFTWARE ASSURANCE: *See QUALITY ASSURANCE.*

SOFTWARE COMPONENT: General term used to refer to an element of a software system, such as module, unit, etc. [IEEE88]

SOFTWARE ENGINEERING: The systematic approach to the development, operation, maintenance, and retirement of software. [IEEE83]

SOFTWARE FAULT TREE ANALYSIS: A form of fault tree analysis used for analyzing the safety of software designs or code.

SOFTWARE LIFE CYCLE: The period of time that starts when a software product is conceived and ends when the product is no longer available for use. The software life cycle typically includes a requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and sometimes, retirement phase. [IEEE83]

SOFTWARE PRODUCT: A software entity designated for delivery to a user. [IEEE83]

SOFTWARE QUALITY: (1) The totality of features and characteristics of a software product that bear on its ability to satisfy given needs; for example, conform to specifications. (2) The degree to which software possesses a desired combination of attributes. (3) The degree to which a customer or user perceives that software meets his or her composite expectations. (4) The composite characteristics of software that determine the degree to which the software in use will meet the expectations of the customer. [IEEE83]

SOFTWARE QUALITY INDICATORS: Process guidelines in the form of detailed data, derived from scheduled surveys, inspections, evaluations, and tests, that provide insight into the condition of a product or process.

SOFTWARE QUALITY METRIC: A function whose inputs are software data and whose output is a single (numerical) value that can be interpreted as the degree to which software possesses a given attribute that affects its quality. [IEEE88]

SOFTWARE RELIABILITY: (1) The probability that software will not cause the failure of the system for a specified time under specified conditions. The probability is a function of the inputs to and use of the system as well as a function of the existence of faults in the software. The inputs to the system determine whether existing faults, if any, are encountered. (2) The ability of a program to perform a required function under stated conditions for a stated period of time. [IEEE83]

SOFTWARE RELIABILITY MODEL: *See RELIABILITY MODEL.*

SOFTWARE REQUIREMENT: See *REQUIREMENT.*

SOFTWARE TOOL: A computer program used to help develop, test, analyze, or maintain another computer program or its documentation; for example, automated design tool, compiler, test tool, maintenance tool. [IEEE83]

SOURCE LANGUAGE: (1) A language used to write source programs. (2) A language from which statements are translated. [IEEE83]

SPECIAL: The state-machine specification language employed by the Hierarchical Development Methodology verification system.

SPECIAL VALUES: Special values have special mathematical properties; for example, zero, one, a very small value, a very large value.

SPECIAL VALUES TESTING: Testing to ensure proper handling of all special values.

SPECIFICATION: A document that prescribes in a complete, precise, verifiable manner, the requirements, design, behavior, or other characteristics of a system or system component. [IEEE83]

SPECIFICATION LANGUAGE: A language, often a machine-processable combination of natural and formal language, used to specify the requirements, design, behavior, or other characteristics of a system or system component. [IEEE83]

SPECIFICATION MODEL: A model used to give a formal specification of a program.

SPECIFICATION MUTATION: A form of mutation testing which is applied to program specifications to determine the absence or presence of a predefined set of potential faults in the implementation of the specification.

STACK FRAMES: A stack element of a pushdown stack automaton.

STANFORD PASCAL VERIFIER: A tool which reasons in quantifier-free first-order predicate calculus.

STARVATION FREEDOM: Occurs when a process (which is not blocked or deadlocked) cannot get into a state where a request for a resource will never be granted.

STATE TRANSITION: A change from one program state to another.

STATE TRANSITION COVERAGE: Member of a series of successively more stringent testing coverage measures for concurrent programs analogous to structural and data flow testing criteria for sequential programs. *See also CONCURRENCY STATE COVERAGE, SYNCHRONIZATION COVERAGE.*

STATE-MACHINE LANGUAGE: A language accepted by a finite state automaton.

STATE-MACHINE SPECIFICATION: Defines a set of functions that specify transformations on input. The set of functions may be viewed as defining the nature of the abstract data type or describing the behavior of an abstract machine.

STATEMENT COMPLEXITY: A complexity value assigned to each statement which is based on (1) the statement type, and (2) the total length of postfix representations of expressions within

the statement (if any). These values are intended to represent a statement's potential execution time. [Mill81]

STATEMENT TESTING: A test method satisfying the coverage criterion that each statement of a program be executed at least once during program testing. [Adri82]

STATIC ANALYSIS: The process of evaluating a program without executing the program. [IEEE83]

STATIC ANALYZER: A software tool that aids in the evaluation of a computer program without executing the program. Examples include syntax checkers, compilers, cross-reference generators, standards enforcers, and flowcharters. [IEEE83]

STATIC BINDING: Binding performed prior to execution of a program and
not subject to change during execution. Contrast with *DYNAMIC BINDING*.

STATIC CONCURRENCY ANALYSIS: A technique for determining all the possible synchronization patterns in a concurrent program, without program execution.

STATIC ERROR ANALYSIS: Analysis of a program to determine whether certain kinds of faults or dangerous conditions are present. *See TYPE AND UNITS ANALYSIS, REFERENCE ANALYSIS, EXPRESSION ANALYSIS, INTERFACE ANALYSIS*.

STATICALLY LINKED: See *STATIC BINDING*.

STATISTICAL TESTING: A testing approach which employs the probability distributions of the product inputs and randomized sampling techniques to organize test material. The randomization supports statistical inferences about the product's operational characteristics and an estimate of its expected reliability (MTTF).

STRESS TESTING: *See BOUNDARY VALUE ANALYSIS*.

STRONG MUTATION TESTING: *See MUTATION TESTING*.

STRONG TYPING: A programming language feature that requires the data type of each data object to be declared, and that precludes the application of operators to inappropriate data objects and, thereby, prevents the interaction of data objects of incompatible types. [IEEE83]

STRUCTURAL COVERAGE MEASURE: A measure of the structural coverage accomplished during testing activities. Usually given as the percentage of program statements, branches, or paths which have been executed.

STRUCTURAL INDUCTION: A formal proof method using recursive induction upon the structure of the data manipulated by a program.

STRUCTURAL TESTING: A testing method where the test data are derived solely from the program structure. [Adri82]

STRUCTURED PROGRAMMING: (1) A well-defined software development technique that incorporates top-down design and implementation and strict use of structured program control constructs. (2) Loosely, any technique for organizing and coding programs that reduces complexity, improves clarity, and facilitates debugging and modification. [IEEE83]

STRUCTURED WALKTHROUGH: *See WALKTHROUGH*.

STUB: Special code segments that, when invoked by a code segment under test, will simulate the behavior of designed and specified modules not yet constructed. [Adri82]

STUB ANALYSIS: In the AdaPIC system, stub analysis checks the consistency of multiple views of the same stub, and the consistency of each of these views against some authorized specification of that module.

SUBGOAL INDUCTION: A proof method that is applicable to *while* statements as the output structure, and assumes that the functional abstraction of the loop body is available.

SUBSYSTEM: A group of assemblies or components or both combined to perform a single

function. [IEEE83].

SUPERFLUOUS CODE ERROR: An error which occurs when a program contains code which is never executed or is redundant for some reason.

SURVIVABILITY: (1) The probability that the software will perform and support critical functions in its intended environment without failure when a specified portion of the system is inoperable. [DeMi88]. (2) The probability that the software will continue to perform or support critical functions when a portion of the system is inoperable. [RADC83].

SYMBOL CROSS-REFERENCER: A software tool that produce dictionaries relating the symbols used in a program by logical name.

SYMBOLIC ALTERNATIVE: Used in a modified form of symbolic execution to prepresent the effect of several mutation transformations.

SYMBOLIC DATA: Symbols used to represent actual input data.

SYMBOLIC DEBUGGING: The process of examining a path computation and path domain in order to obtain information about the cause of a known fault.

SYMBOLIC EVALUATION: *See SYMBOLIC EXECUTION.*

SYMBOLIC EVALUATION SYSTEM: A software tool that accepts symbolic values for some of the program inputs and algebraically manipulates these symbols according to the expressions in which they appear. It can be used to support test data generation, assertion checking, path analysis, and detection of data flow anomalies.

SYMBOLIC EXECUTION: A verification technique in which program execution is simulated using symbols rather than actual values for input data, and program outputs are expressed as logical or mathematical expressions involving these symbols. [IEEE83]

SYMBOLIC INPUTS: See *SYMBOLIC DATA.*

SYMBOLIC INTERPRETATION: Where the values taken on by variables are represented as algebraic expressions that denote the computational history of those variables.

SYMBOLIC TESTING: A method of examining the path computation and path condition to ascertain the correctness of a program path.

SYMBOLIC VALUES: Values which are maintained as algebraic expressions given in terms of the symbolic names assigned to input values.

SYNCHRONIZATION: The exchange of signals used when certain processes must be stopped at a given point until some event under the control of another process has occurred.

SYNCHRONIZATION COVERAGE: Member of a series of successively more stringent testing coverage measures for concurrent programs analogous to structural and data flow testing criteria for sequential programs. *See also CONCURRENCY STATE COVERAGE, STATE TRANSITION COVERAGE.*

SYNCHRONIZATION FAULT: A fault which results from incorrect sequencing and communications between concurrent processes.

SYNTACTIC UNIT: A unit that corresponds to a set of statements in a program which define an operation upon some object.

SYNTAX: (1) The relationship among characters or groups of characters, independent of their meanings or the manner of their interpretation and use. (2) The structure of expressions in a language. (3) The rules governing the structure of a language. See also *SEMANTICS.* [IEEE83]

SYSTEM: A collection of people, machines, and methods organized to accomplish a set of specified functions. [IEEE83]

SYSTEM ARCHITECTURE: The structure and relationship among the components of a system. A system architecture may also include the system's interface with its operational

environment. [IEEE83]

SYSTEM COMPONENT: *See COMPONENT.*

SYSTEM DESIGN: The process of defining the hardware and software architectures, components, modules, interfaces, and data for a system to satisfy specified system requirements. [IEEE83]

SYSTEM HAZARD: See *HAZARD.*

SYSTEM INTERFACE: *See INTERFACES.*

SYSTEM PERFORMANCE: *See PERFORMANCE.*

SYSTEM REQUIREMENTS: *See REQUIREMENTS.*

SYSTEM SAFETY: The ability of the system to prevent critical failures leading to unacceptable consequences. Examples of unacceptable consequences include the failure of the system mission, and loss of life or property.

SYSTEM SECURITY: *See SECURITY.*

SYSTEM ROBUSTNESS: *See ROBUSTNESS.*

SYSTEM SPECIFICATION: *See SPECIFICATION.*

SYSTEM TESTING: The process of testing an integrated hardware and software system to verify that the system meets its specified requirements. [IEEE83]

TAP: A debugger designed to detect timing faults caused by the misordering of events in a distributed system.

TARGET ENVIRONMENT: SEE *TARGET MACHINE.*

TARGET MACHINE: The computer on which a program is intended to operate. Contrast with *HOST MACHINE.* [IEEE83]

TASK: *See PROCESS.*

TASK COMMUNICATION: *See PROCESS COMMUNICATION.*

TASK ENTRY FAMILY: An entry declaration for a task which includes a discrete range and so declares a family of distinct entries.

TASK SEQUENCING FAULT: A fault which occurs when a program's tasks interact in a different order than anticipated.

TASK SEQUENCING LANGUAGE: A language used to annotate Ada programs by specifying constraints to be satisfied by sequences of task events. These constraints can be transformed into dynamic checks for certain types of faults and failures.

TEMPORAL LOGIC: A logic theory with temporal quantifiers (for example *henceforth* and *eventually*), which permits statements about temporal conditions to be made.

TERMINATION: The act of finishing a program or a proof.

TEST: A unit test of a single module consists of (1) a collection of settings for the input space of the module, and (2) exactly one invocation of the module. A unit test may or may not include the effect of other modules which are invoked by the module undergoing testing. [Mill81]

TEST AND EVALUATION (T&E): A formal testing process used to evaluate the technical and operational characteristics of a system. Performed in a number of stages, for example, *QUALIFICATION TESTING, DEVELOPMENTAL TEST AND EVALUATION, INITIAL OPERATIONAL TEST AND EVALUATION, OPERATIONAL TEST AND EVALUATION, FOLLOW-ON OPERATIONAL TEST AND EVALUATION.*

TEST BED: (1) A test environment containing the hardware, instrumentation tools, simulators, and other support software necessary for testing a system or system component. (2) The repertoire of test cases necessary for testing a system or system component. [IEEE83]

TEST CASE: *See TEST DATA SET.*

TEST DATA: *See TEST DATA SET.*

TEST DATA ADEQUACY: *See ADEQUATE TEST DATA.*

TEST DATA GENERATOR: An automated tool that accepts as input a computer program and test criteria, generates test input data that meet these criteria, and, sometimes, determines the expected outputs. [IEEE83]

TEST DATA SELECTION STRATEGY: Provides guidance for selecting test data for a program; for example, a branch testing test data selection strategy selects data that cause those program paths to be executed such that each branch is executed at least once.

TEST DATA SET: A specific set of input and output values for the variables in the communication space of a module that are used in a test. Also called a test case.

TEST DRIVER: A program that directs the execution of another program against a collection of test data sets. Usually the test driver also records and organizes the output generated as the tests are run. [Adri82]

TEST GRAMMAR: A context-free grammar which describes those aspects of a program to be tested, as well as the assumptions as to which test cases are considered equivalent. The grammar generates test data in levels of ever increasing complexity of test cases. At each level the programmer may use the results of testing at previous levels to strengthen the assumptions on the test grammar, thereby reducing the number of test cases generated at subsequent levels. [DACS79]

TEST HARNESS: *See TEST DRIVER.*

TEST INSTRUMENTORS: Automated tools that produce an altered version of a program or component that is logically equivalent to the unmodified program but contains calls to special data collection routines that record information pertaining to the execution behavior of the program.

TEST MANAGEMENT: Management procedures designed to control in an ordered way a large and evolving amount of pieces of information on system features to be tested, on system implementation plans, and on test results. [DACS79]

TEST PATH: The specific (sequence) set of segments that is traversed as the result of a unit test operation on a set of test data. A module can have many test paths. [Mill81]

TEST PLAN: A document prescribing the approach to be taken for intended testing activities. The plan typically identifies the items to be tested, the testing to be performed, test schedules, personnel requirements, reporting requirements, evaluation criteria, and any risks requiring contingency planning. [IEEE83]

TEST POINT: A tuple containing a value for each program input.

TEST REPEATABILITY: An attribute of a test indicating whether the same results are produced each time the test is conducted. [IEEE83]

TESTABILITY: (1) The extent to which software facilitates both the establishment of test criteria and the evaluation of the software with respect to these criteria. (2) The extent to which the definition of requirements facilitates analysis of the requirements to establish test criteria. [IEEE83]

TESTING: The process of exercising or evaluating a system or system component by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results. Contrast with *DEBUGGING*. [IEEE83]

TESTING COVERAGE MEASURE: In general, a measure of the testing coverage achieved as a result of a test, often expressed as a percentage of the number of statements, branches, or paths that were traversed. [Mill81]

TESTING ENVIRONMENT: A collection of software tools to assist the user in planning, conducting, and reporting on testing activities.

TESTING, EVALUATION, AND ANALYSIS MEDLEY: A testing environment under development at the University of California (Irvine) which will support incremental and integrated application of a number of different dynamic and static analysis techniques to Ada programs. It is expected to become part of the Arcadia software development environment.

TESTING TARGET: The current module (system testing) or current segment (unit testing) upon which testing effort is focused. [Mill81]

THEOREM PROVERS: Tools to mechanize the process of producing a formal proof.

THRESHOLD VALUES: Values of technical or operational properties and parameters below which the overall system worth will be unacceptable.

TIME DOMAIN MODELS: Software reliability models in which reliability is considered a function of time. Include times-between-failures models and failure-count models.

TIMES BETWEEN FAILURES MODEL: Software reliability model in which the time between failures is treated as a random variable whose parameters depend on the number of faults remaining in the program. [Goel85].

TIMING GRAPH: A directed acyclic graph representing the partial ordering of events for a distributed program.

TOOL: (1) See *SOFTWARE TOOL*. (2) A hardware device used to analyze software or its performance.

TOP DOWN TESTING STRATEGY: A systematic testing philosophy which seeks to test those modules at the top of the invocation structure earliest. [Mill81]

TOTAL CORRECTNESS: In proof of correctness, a designation indicating that a program's output assertions follow logically from its input assertions and processing steps, and that, in addition, the program terminates under all specified input conditions. [IEEE83]

TRACE: *See PROGRAM TRACE.*

TRACE MUTATION TESTING: A form of mutation testing where certain classes of program traces rather than output values are used for distinguishing between a program and its mutants. This eliminates the need for assumptions such as the Coupling Effect and allows repeated applications of mutation transformations.

TREE: An abstract hierarchical structure consisting of nodes connected by branches, in which: (a) each branch connects one node to a directly subsidiary node, and (b) there is a unique node called the root that is not subsidiary to any other node, and (c) every node besides the root is directly subsidiary to exactly one other node. [IEEE83]

TRUSTWORTHINESS OF SOFTWARE: Probability that no serious [software] design error remains after a set of randomly chosen tests [have been] passed. [Parn88].

TYPE ANALYSIS: A form of static error analysis involving the determination of correct use of named data items and operations. Usually, type analysis is used to determine whether or not the domain of values (functions, etc.) attributed to an entity are done so in a correct and consistent manner.

UNIFORM: All possible values or selections occur with equal probability. [Musa87]

UNIFORMITY HYPOTHESIS: The uniformity hypothesis consists in assuming that if the test is successful for one datum in a subdomain then the program behaves correctly for any data in this subdomain.

UNIT TEST: *See TEST.*

UNIT TESTING: The process of testing each unit in isolation. *See also INTEGRATION TESTING AND SYSTEM TESTING.*

UNITS ANALYSIS: Units analysis determines whether or not the units or physical dimensions attributed to an entity are correctly defined and consistently used.

UNREACHABILITY: A statement (or segment) is unreachable if there is no logically obtainable set of input-space settings which can cause the statement (or segment) to be traversed. [Mill81]

UNSAFE STATE: A state which may lead to a safety failure unless some specific action is taken to avert it.

UPDATE ANALYSIS: In the AdaPIC system, update analysis compares two versions of the same submodule to look for changes in declarations, requisition/ provision specifications, or references to non-local entities.

USER INTERACTION MODEL: A model which defines the possible user interaction with a software system or tool.

VAL: A formal language for specifying the behavior of hardware designs whose architectures are specified in VHDL. It provides the capability for automatic comparison of behavior of different levels of a VHDL hierarchical design during simulation.

VALIDATED Ada COMPILER: An Ada compiler that has been determined by the Ada Joint Program Office to compile Ada source code in accordance with the language specification given in the Ada Language Reference Manual.

VALIDATED METRIC: A software quality metric whose values have a specified association with the corresponding values of a designated quality factor or with the values of a valid metric of that factor, when the two sets of metric values are obtained from the same domain (e.g., the same software components). [IEEE88]

VALIDATION: (1) The process of evaluating software at the end of the software development process to ensure compliance with software requirements. [IEEE83] (2) Static and dynamic analysis of a software product to ensure it attains the features and performance attributes prescribed by its requirements.

VARIABLE: (1) A quantity that can assume any of a given set of values. (2) In programming, a character or group of characters that refers to a value and, in the execution of a computer program, corresponds to an address. [IEEE83]

VARIABLE ASSIGNMENT: An expression which assigns a value to a variable.

VARIABLE DEFINITION: A program statement which defines a variable and its allowable usage.

VARIABLE NAME: An identifier allocated to a variable for purposes of reference. See also *VARIABLE*.

VARIABLE REFERENCE: Accessing a value from a variable.

VARIABLE UNDEFINITION: Causing the value of a variable to become undefined; for example, when the program control flow passes beyond the scope of a variable.

VARIABLE USAGE ERROR: A programming anomaly arising from the erroneous usage of variables; for example, a reference to an undefined variable, the definition of a variable which is never referenced, or a dead variable definition where a variable is defined twice without an intervening reference.

VERIFIABILITY: The adequacy with which a given algorithm represents the requirements of the physical world. [RADC83].

VERIFICATION: (1) The process of determining whether or not the products of a given phase of the software development cycle fulfill the requirements correctness. (3) The act of reviewing, inspecting, testing, checking, auditing, or otherwise establishing and documenting whether or not items, processes, services, or documents conform to specified requirements. [IEEE83]

VERIFICATION CONDITION GENERATOR: A program that generates sets of logical conditions that must be proven in order to verify software.

VERIFICATION SYSTEM: A software tool that accepts as input a computer program and a representation of its specification, and produces,

possibly with human help, a correctness proof or disproof of the program. [IEEE83]

VHDL: A high-level, wide-spectrum hardware design language designed to support the development of distributed systems.

VISIBILITY: The visibility of a variable refers to those locations within a program where the variable is available for reference. The visibility is determined by the declaration, scope, and binding rules given in a programming language.

WALKTHROUGH: A manual review process in which the designer or programmer leads one or more other members of the development team through a segment of design or code that he or she has written, while the other members ask questions and make comments about technique, style, possible errors, violation of development standards, and other problems. [IEEE83]

WATERFALL SOFTWARE DEVELOPMENT LIFE CYCLE: A discipline of software development which proceeds in a series of discrete steps. The standard DOD ordering of these steps is as follows: software requirements analysis, preliminary design, detailed design, coding and unit testing, integration testing, and system testing.

WEAK MUTATION TESTING: A form of mutation testing where mutation transformations are applied to program components rather than a program as a whole mutation testing in its original form, this technique does not guarantee exposure of all faults in the class of faults associated with mutation transformations but does allow repeated application of mutation transformations for a single test.

WHITE BOX TESTING: Testing approaches which examine the program structure and derive test data from the program logic.

WIDE-SPECTRUM LANGUAGE: A language which can serve several purposes; for example, can be used in a series of successive software development phases.

WORST CASE ANALYSIS: Analysis that assumes the worst-case conditions for every parameter under study.

[Cont86] Conte, S.D., H.E. Dunsmore, and V.Y. Shen. 1986. *Software Engineering Metrics and Models*. Menlo Park, CA: Benjamin/Cummings Publishing Company.

[Adri82] Adrion, W.R., M.A. Branstad, and J.C. Cherniavsky. "Validation, Verification, and Testing of Computer Software." *ACM: Computing Surveys*, 14/2 (Jun 1982):159-192.

[DeMi88b] DeMillo, R.A., R.J. Martin, and R.N. Meeson. September 1988. *Strategy for Achieving Ada-Based High Assurance Systems*. Alexandria, VA: Institute for Defense Analyses. Draft IDA Paper P-2143.

[DACS79b] Data and Analysis Center for Software (DACS). October 1979. *The DACS Glossary: A Bibliography of Software Engineering Terms*. Rome Air Development Center: Griffiss Air Force Base.

[IEEE83a] *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Standard 729. February 18, 1983.

[IEEE88] IEEE Draft Standard 1061/D15, Standard for a Software Quality Metrics Methodology. New York: The Institute of Electrical and Electronics Engineers.

[Goel85] Gocl, A.L. "Software Reliability Models: Assumptions, Limitations, and Applicability," *IEEE: Transactions on Software Engineering*, 11/12 (Dec 1985):1411-1423.

[Mill81a] Miller, E., and W.E. Howden, eds. 1981. *Tutorial: Software Testing and Validation*, 2nd Edition. Los Alamitos, CA: IEEE Computer Society Press.

[Musa87] Musa, J., A. Iannino, K. Okumoto. 1987. *Software Reliability:*

Measurement, Prediction, Application. New York: McGraw Hill.

[Parn88] Parnas, D.L., A.J. van Schouwen, and S.P. Kwan. May 1988. *Evaluation Standards for Safety Critical Software*. Queens University. Technical Report 88-220.

[RADC83b] Rome Air Development Center. July 1983. *Software Quality Measurement for Distributed Systems, Vols I, II and III*. Griffiss Air Force Base, NY: Rome Air Development Center. Technical Report RADC-TR-83-175.

[Rand75] Randell, B. "System Structure for Software Fault Tolerance." *IEEE: Transactions on Software Engineering*, 1/2 (Jun 1975):220-232.

APPENDIX B: ACRONYMS

ACM	Association for Computing Machinery
ACVC	Ada Compiler Validation Capability
AdaMAT	Ada Metrics and Analysis Tool
AI	Artificial Intelligence
AJPO	Ada Joint Program Office
ANNA	ANNotated Ada
AT&T	American Telephone & Telegraph
ATVS	Ada Test and Verification System
AVA	Annotated Verifiable Ada
BIT	Built-in-test
BM/C3	Battle Management/Command, Control, and Communication
C3	Command, Control, and Communication
COCOMO	COnstructive COst MOdel
CPU	Central Processing Unit
CSC	Computer Software Component
CSCI	Computer Configuration Item
CSP	Communicating Sequential Processes
CSU	Computer Software Unit
DACS	Data and Analysis Center for Software
DAISTS	Data-Abstraction Implementation, Specification, and Testing System
DARPA	Defense Advanced Research Projects Agency
DBMS	DataBase Management System
DIANA	Descriptive Intermediate Attributed Notation for Ada
DOD	Department of Defense
DT&E	Developmental Test and Evaluation
IEEE	Institute of Electrical and Electronics Engineers
EIA	Electronics Industries Association
ESTCA	Error Sensitive Test Case Analysis
FDM	Formal Development Methodology
FOT&E	Follow-on Operational Test and Evaluation
FSD	Full Scale Development
GFE	Government Furnished Equipment
GVE	Gypsy Verification Environment
HDL	Hardware Design Language
HDM	Hierarchical Development Methodology
HOL	High Order Language
IDA	Institute for Defense Analyses
IEEE	Institute for Electrical and Electronics Engineers
I/O	Input/Output
IOT&E	Initial Operational Test and Evaluation
IV&V	Independent Verification and Validation
LCSAJ	Linear Code Sequence and Jump
LOC	Lines of Code
MIMD	Multiple-Instruction, Multiple-Data streams
MTBF	Mean Time Between Faults
MTTF	Mean Time to Failure
MTTR	Mean Time to Repair
NASA	National Aeronautics and Space Administration

NCSC	National Computer Security Center
NOSC	Naval Ocean Systems Center
NTB	National Test Bed
NTDS	Naval Tactical Data System
NTF	National Test Facility
PDL	Program Design Language
OT&E	Operational Test and Evaluation
OSD	Office of the Secretary of Defense
QT	Qualification Testing
R&D	Research and Development
RADC	Rome Air Defense Center
RAM	Random Access Memory
SADMT	Strategic Defense Initiative Architecture Dataflow Modeling Technique
SCA	Static Concurrency Analyzer
SDI	Strategic Defense Initiative
SDIO	Strategic Defense Initiative Organization
SDS	Strategic Defense System
SEL	Software Engineering Laboratory
SIMD	Single-Instruction, Multiple-Data streams
SMDC	Software Metrics Data Collection
SOIF	System Operational and Integration Function
SSDC	SSDS Software Data Collection System
SSDS	Software Data Collection System
STEP	Software Test and Evaluation Project
T&E	Test and Evaluation
TAME	Tailoring A Measurement Environment
TEAM	Testing, Evaluation, and Analysis Medley
TEMP	Test and Evaluation Master Plan
TSL	Task Sequencing Language
VAL	VHDL Annotation Language
WIS	WWMCCS Information System
WWMCCS	World Wide Military Command and Control System

APPENDIX C: WORKSHOP PARTICIPANTS

The following identifies the individuals who participated in each of the workshop panels. In addition to these, the welcoming remarks were given by LtCol J. Price, SDIO, and the workshop was attended by the SDIO sponsor of this effort, LtCol C. Lillie.

C.1 Participants in the Validation Panel

Mr. Lou Chmura
Naval Research Laboratory
Code 5533
Washington, DC 20375-5000
(202) 767-3249
chmura@nrl-css.arpa

Dr. Lori Clarke
COINS Dept.
University of Massachusetts
Amherst, MA 01003
(413) 545-1328
clarke@cs.umass.edu

Dr. Rich DeMillo
Computer Science Dept.
Purdue University
West Lafayette, IN 47907
(317) 494-7823
rad@purdue.edu

Dr. Laura K. Dillon
Computer Science Dept.
University of California
Santa Barbara, CA 93106
(805) 961-3411
dillon@aslan.ucsb.edu

Dr. William Howden
Dept. of Computer Science and Engineering
University of California
Mail Code C-014
La Jolla, CA 92093-0114
(619) 755-3359
howden@odin.ucsd.edu

Dr. Virginia Kobler
US Army Strategic Defense Command
CSSD-H-SBY
PO Box 1500
Huntsville, AL 35807
(205) 895-3857

Dr. David Luckham	Computer Systems Lab, ERL 456 Stanford University Stanford, CA 94305 (415) 497-1242 dcl@sail.stanford.edu
Dr. Leon Osterweil	Dept. of Information and Computer Science University of California Irvine, CA 92717 (714) 856-4048 ljo@ics.uci.edu
LtCol James Price	SDIO TE 1E149 Washington, DC 20301-7100 (202) 693-1600
Dr. Debra J. Richardson	Dept. of Information & Computer Science University of California Irvine, CA 92717 (714) 856-7353 richardson@ics.uci.edu
Mr. Ken Rowe	NCSC ATTN: C33 9800 Savage Rd. Ft. Meade MD 20755-6000 (301) 859-4491 rowe@dockmaster.arpa
Dr. John Salasin	GTE Government Systems Corporation 1700 Research Boulevard Rockville, MD 20850 (301) 294-8400 salasin_jj%ncsd.decnet@gtewd.arpa
Dr. Richard N. Taylor	Dept. of Information & Computer Science University of California Irvine, CA 92717 (714) 856-6429 taylor@ics.uci.edu
Dr. Dolores Wallace	National Institute of Standards and Technology Technology Bldg. B266 Gaithersburg, MD 20899 (301) 975-3340 wallace@swe.icst.nbs.gov

Dr. Lee White

Dept. of Computer Engineering and Science
CASE Western Reserve University
509 Crawford Hall
Cleveland, OH 44106
(216) 368-2802
leew@alpha.ces.cwru.edu

Mrs. Christine Youngblut

Institute for Defense Analyses
Computer and Software Engineering Division
1801 N. Beauregard St.
Alexandria, VA 22311
(301) 948-8391
youngb@ida.org

Dr. Steven Zeil

Dept. of Computer Science
Old Dominion University
Norfolk, VA 23508
(804) 683-4832
zeil@cs.odu.edu

C.2 Participants in the Verification Panel

Dr. William Easton

Peregrine Systems, Inc.
P.O. Box 192
Bluemont, VA 22012
(703) 689-1168
easton@ida.org

Dr. Warren Hunt, Jr.

Computational Logic, Inc.
1717 West Sixth Street,
Suite 290
Austin, TX 78703-4776
(512) 322-9951
hunt@cli.com

Dr. C. Terrence Ireland

NCSC
7800 Savage Rd.
Ft. Meade, MD 20755
(301) 859-4371
cti@mimsy.umd.edu

Dr. Richard Kemmerer

Dept. of Computer Science
University of California
Santa Barbara, CA 93106
(805) 961-4232
kemm@hub.ucsb.edu

Dr. Timothy Lindquist
Computer Science Dept.
Arizona State University
Tempe, AZ 85287
(602) 965-2783
lindquis@ajpo.sei.cmu.edu

Mr. Terry Mayfield
Institute for Defense Analyses
Computer and Software Engineering Division
1801 N. Beauregard St.
Alexandria, VA 22311
(703) 824-5524
mayfield@ida.org

Dr. Reg Meeson
Institute for Defense Analyses
Computer and Software Engineering Division
1801 N. Beauregard St.
Alexandria, VA 22311
(703) 845-3541
meeson@ida.org

Dr. Richard Platek
Odyssey Research Associates
301A Dates Dr.
Ithaca, NY 14850-1313
(607) 277-2020
oravax!richard@cu-arpa.cs.cornell.edu

Mr. Andy Moore
Naval Research Laboratory
Washington, D.C. 20375-5000
(202) 767-6698
moore@nrl-css.arpa

Mr. David Nielson
Titan Systems, Inc.
Test and Evaluation Dept.
2000 WestPark Dr.
Westboro, MA 01581
(508) 870-0006

Mr. Karl Nyberg
Grebyn Corporation
P.O. Box 1144
Vienna, VA 22180
(703) 281-2194
karl@grebyn.com or nyberg@ajpo.sei.cmu.edu

C.3 Participants in the Software Measurement Panel

Dr. Victor Basili
Dept. of Computer Science
University of Maryland
College Park, MD 20742
(301) 454-8742
basili@mimsy.umd.edu

Ms. Sabrina Beckman	Titan Systems, Inc 2705 Artie St. Wuite 25 Huntsville, AL 35805
Mr. Bill Brykczynski	Institute for Defense Analyses Computer and Software Engineering Division 1801 N. Beauregard St. Alexandria, VA 22311 (703) 824-5515 bryk@ida.org
Mr. David Card	Computer Sciences Corp. 8728 Colesville Rd. Silver Spring, MD 20910 (301) 650-3245
Mr. Joseph Cavano	RADC/COE Griffiss Air Force Base, NY 13441 (315) 330-4476
Dr. Michael Evangelist	MCC 3500 W. Balcones Center Dr. Austin, TX 78759 (512) 338-3479 wme@mcc.com
Dr. Kathy Holland	National Computer Security Center ATTN: C33 9800 Savage Rd. Ft. Meade, MD 20755-6000 (301) 859-4491
LtCol Charles Lillie	SDIO TE 1E149 Washington, DC 20301-7100 (202) 693-1600 lillie@hc.dspo.gov
Dr. Cathy Jo Linn	Institute for Defense Analyses Computer and Software Engineering Division 1801 N. Beauregard St. Alexandria, VA 22311 (703) 824-5520 clinn@ida.org
Dr. Dieter Rombach	Dept. of Computer Science University of Maryland College Park, MD 20742 (301) 454-8974 rombach@mimsy.umd.edu

LtCol Anthony Shumskas	OSD DT&E 3D1075 Washington, DC 20301-7100 (202) 695-4421
Dr. Richard Selby	Dept. Of Information & Computer Science University of California Irvine, CA 92717 (714) 856-6326 selby@ics.uci.edu
Dr. Vincent Shen	MCC 3500 W. Balcones Center Dr. Austin, TX 78759 (512) 338-3345 shen@mcc.com

C.4 Participants in the Reliability Assessment Panel

Dr. Frank Ackerman	AT&T Bell Labs Room 6E 110 Whippany, NJ 07981 (201) 386-3377 attunix!whuxr!afa
Mr. Jim Baldo	Institute for Defense Analyses Computer and Software Engineering Division 1801 N. Beauregard St. Alexandria, VA 22311 (703) 824-5516 baldo@ida.org
Dr. Amrit Goel	Syracuse University ECE Dept. 111 Link Hall Syracuse, NY 13244 (315) 443-4350 goel@suvm.acs.syr.edu
Dr. Carlos Gonzalez	Institute for Defense Analyses Computer and Software Engineering Division 1801 N. Beauregard St. Alexandria, VA 22311 (703) 323-3818 cgonzalez@gmuvax.gmu.edu

Dr. Karen Gordon

Institute for Defense Analyses
Computer and Software Engineering Division
1801 N. Beauregard St.
Alexandria, VA 22311
(703) 845-3591
gordon@ida.org

Dr. John Knight

Software Productivity Consortium
1880 Campus Commons Dr. North
Reston, VA 22091
knight@software.org or jck@virginia.edu

Dr. Andre M. Van Tilborg

Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217-5000
avantil@nswc-wo.arpa

CDR David Vaurio

USN
Computer Security R&D
National Security Agency
Ft. Meade, MD 20755
(301) 859-4485

Dr. Charles Waespy

Institute for Defense Analyses
Operational Evaluation Division
1801 N. Beauregard St.
Alexandria, VA 22311
(703) 845-2587

LAST PAGE OF IDA PAPER P-2132

Distribution List for IDA Paper P-2132

NAME AND ADDRESS	NUMBER OF COPIES
Sponsor	
LtCol James Price SDIO TE 1E149 The Pentagon Washington, DC 20301-7100 (202) 693-1600	2
Other	
Dr. Frank Ackerman AT&T Bell Labs Room 6E 110 Whippany, NJ 07981 (201) 386-3377	1
Dr. Vic Basili Dept. of Computer Science University of Maryland College Park, MD 20742 (301) 454-8742	1
Ms. Sabrina Beckman Titan Systems, Inc 2705 Artie St. Suite 25 Huntsville, AL 35805	1
Mr. David Card Computer Sciences Corp. 8728 Colesville Rd. Silver Spring, MD 20910 (301) 650-3245	1
Mr. Mike Carrio Teledyne Brown Engineering Ada PDL West Oaks Executive Park Suite 200 3700 Pender Drive Fairfax, VA 22030	1

NAME AND ADDRESS	NUMBER OF COPIES
Mr. Joseph Cavano RADC/COE Griffiss Air Force Base, NY 13441 (315) 330-4476	1
Mr. Lou Chmura Naval Research Laboratory Code 5533 Washington, DC 20375-5000 (202) 767-3249	1
Dr. Lori Clarke COINS Dept. University of Massachusetts Amherst, MA 01003 (413) 545-1328	1
Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dr. Rich DeMillo Computer Science Dept. Purdue University West Lafayette, IN 47907 (317) 494-7823	1
Dr. Laura K. Dillon Computer Science Dept. University of California Santa Barbara, CA 93717 (805) 961-3411	1
Dr. Bill Easton Peregrine Systems, Inc. P.O. Box 192 Bluemont, VA 22012 (703) 689-1168	1
Dr. Michael Evangelist MCC 3500 W. Balcones Center Dr. Austin, TX 78759 (512) 338-3479	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Amrit Goel Syracuse University ECE Dept. 111 Link Hall Syracuse, NY 13244 (315) 443-4350	1
Dr. Kathy Holland National Computer Security Center ATTN C33 9800 Savage Rd. Ft. Meade, MD 20755-6000 (301) 859-4491	1
Dr. William Howden Dept. of Computer Science and Engineering University of California Mail Code C-014 La Jolla, CA 92093-0114 (619) 755-3359	1
Dr. Warren Hunt, Jr. Computational Logic, Inc. 1717 West Sixth Street Suite 290 Austin, TX 78703-4776 (512) 322-9951	1
Dr. C. Terrence Ireland NCSC 7800 Savage Rd. Ft. Meade, MD 20755 (301) 859-4371	1
Dr. Dick Kemmerer Dept. of Computer Science University of California Santa Barbara, CA 93106 (805) 961-4232	1
Dr. John Knight Software Productivity Consortium 1880 Campus Commons Dr. North Reston, VA 22091	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Virginia Kobler US Army Strategic Defense Command CSSD-H-SBY PO Box 1500 Huntsville, AL 35807 (205) 895-3857	1
Dr. Timothy Lindquist Computer Science Dept. Arizona State University Tempe, AZ 85287 (602) 965-2783	1
Dr. Dave Luckham Computer Systems Lab, ERL 456 Stanford University Stanford, CA 94305 (415) 497-1242	1
Mr. Andy Moore Naval Research Laboratory Washington, D.C. 20375-5000 (202) 767-6698	1
Mr. John R. Nickels Sparta Corporation 7926 Jones Branch Drive Suite 1070 McLean, VA 22102 (703) 448-0210	1
Mr. David Nielson Titan Systems, Inc. Test and Evalutation Dept. 2000 WestPark Dr. Westboro, MA 01581 (508) 870-0006	1
Mr. Karl Nyberg Grebyn Corporation P.O. Box 1144 Vienna, VA 22180 (703) 281-2194	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Leon Osterweil Dept. of Information and Computer Science University of California Irvine, CA 92717 (714) 856-4048	1
Dr. Richard Platek Odyssey Research Associates 301A Dates Dr. Ithaca, NY 14850-1313 (607) 277-2020	1
Dr. Debra J. Richardson Dept. of Information & Computer Science University of California Irvine, CA 92717 (714) 856-7353	1
Dr. Dieter Rombach Dept. of Computer Science University of Maryland College Park, MD 20742 (301) 454-8974	1
Mr. Ken Rowe NCSC ATTN C33 9800 Savage Rd. Ft. Meade MD 20755-6000 (301) 859-4491	1
Dr. John Salasin GTE Government Systems Corporation 1700 Research Boulevard Rockville, MD 20850 (301) 294-8400	2
Dr. Richard Selby Dept. Of Information & Computer Science University of California Irvine, CA 92717 (714) 856-6326	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Vincent Shen MCC 3500 W. Balcones Center Dr. Austin, TX 78759 (512) 338-3345	1
LtCol Anthony Shumskas OSD DT&E 3D1075 Washington, DC 20301-7100 (202) 695-4421	1
Dr. Richard N. Taylor Dept. of Information & Computer Science University of California Irvine, CA 92717 (714) 856-6429	1
Dr. Andre M. Van Tilborg Office of Naval Research 800 N. Quincy St. Arlington, VA 22217-5000	1
CDR David Vaurio USN Computer Security R&D National Security Agency Ft. Meade, MD 20755 (301) 859-4485	1
Dr. Dolores Wallace National Institute of Standards & Technology Technology Bldg. B266 Gaithersburg, MD 20899 (301) 975-3340	1
Dr. Lee White Dept. of Computer Engineering and Science CASE Western Reserve University 509 Crawford Hall Cleveland, OH 44106 (216) 368-2802	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Steven Zeil Dept. of Computer Science Old Dominion University Norfolk, VA 23508 (804) 683-4832	1
Dr. John Gannon Department of Computer Science University of Maryland College Park, MD 20742	1
Mr. Chuck Horswill GE Government Systems P.O. Box 8555 Bldg. 98 Philadelphia, PA 19101 (215) 354-6317	1
LtCol John Morrison NTB/JPO MS 82 Falcon AFB, CO 80912-5000 (719) 380-3267	1
Dr. Theodore F. Elbert Professor and Chairman Department of Computer Science University of West Florida Pensacola, FL 21514-5752 (904) 474-2232	1
LtCol Fred J. Foster, USAF Staff Assistant Director, Operational Test and Evaluation The Pentagon Washington, D.C. 20301-1700 AV 224-2776 (202) 694-2153	1
Dr. William L. Kilmer Professor, Dept. of Electrical and Computer Engineering University of Massachusetts Amhert, MA 01003 Office (413) 545-0672 Messages (413) 545-2442	1

NAME AND ADDRESS	NUMBER OF COPIES
Mr. Richie Platt 428 Gabe Ct. Denton, TX 76201 (817) 387-7255	1
Dr. Charlie Richter Software Technology Program MCC 3500 W. Balcones Center Dr. Austin, TX 78759-6509	1
LtCol Charles W. Lillie SDIO Phase 1 Program Office The Pentagon Washington, D.C. 20301-7100 (202) 693-1600	2
CSED Review Panel	
Dr. Dan Alpert, Director Center for Advanced Study University of Illinois 912 W. Illinois Street Urbana, Illinois 61801	1
Dr. Barry W. Boehm TRW Defense Systems Group MS R2-1094 One Space Park Redondo Beach, CA 90278	1
Dr. Ruth Davis The Pymatuning Group, Inc. 2000 N. 15th Street, Suite 707 Arlington, VA 22201	1
Dr. John M. Palms, Vice President Academic Affairs & Professor of Physics Emory University Atlanta, GA 30322	1

NAME AND ADDRESS	NUMBER OF COPIES
Dr. C.E. Hutchinson, Dean Thayer School of Engineering Dartmouth College Hanover, NH 03755	1
Mr. A.J. Jordano Manager, Systems & Software Engineering Headquarters Federal Systems Division 6600 Rockledge Dr. Bethesda, MD 20817	1
Mr. Keith Uncapher University of Southern California Olin Hall 330A University Park Los Angeles, CA 90089-1454	1
Mr. Robert K. Lehto Mainstay 302 Mill St. Occoquan, VA 22125	1
Mr. Oliver Selfridge 45 Percy Road Lexington, MA 02173	1
IDA	
General W.Y. Smith, HQ	1
Mr. Philip Major, HQ	1
Dr. Robert E. Roberts, HQ	1
Dr. John F. Kramer, CSED	1
Dr. Robert I. Winner, CSED	1
Ms. Anne Douville, CSED	1
Dr. Richard Wexelblat, CSED	1
Mr. Terry Mayfield, CSED	1
Mr. Bill Brykczynski, CSED	15
Ms. Sylvia Reynolds, CSED	2
IDA Control & Distribution Vault	2
Mr. Jim Baldo, CSED	1
Dr. Carlos Gonzalez, CSED	1
Dr. Karen Gordon, CSED	2
Ms. Christine Youngblut, CSED	2
Dr. Cathy Jo Linn, CSED	1
Dr. Reginald Meeson, CSED	2

NAME AND ADDRESS	NUMBER OF COPIES
Dr. Charles Waespy, OED	1