RADC-TR-75-161
Interim Report
June 1975

METHODOLOGY FOR COMPREHENSIVE SOFTWARE TESTING

General Research Corporation

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York 13441

METHODOLOGY FOR COMPREHENSIVE SOFTWARE TESTING

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

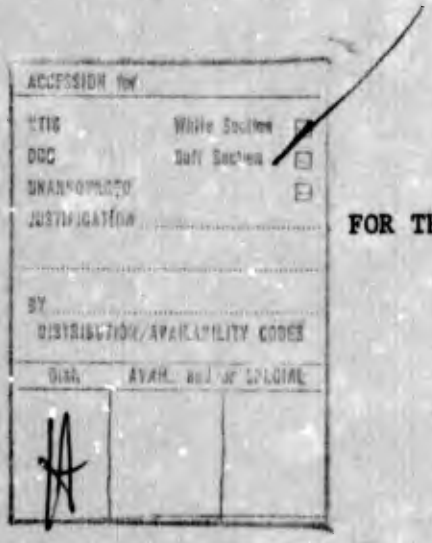
This technical report has been reviewed and approved for publication.

APPROVED: Richard A Robinson

RICHARD A. ROBINSON
Project Engineer

APPROVED: Robert D Krutz

ROBERT D. KRUTZ, Col., USAF
Chief, Information Sciences Division



FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-75-161	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): METHODOLOGY FOR COMPREHENSIVE SOFTWARE TESTING	5. TYPE OF REPORT & PERIOD COVERED: Interim Report, July 1973 — February 1975
6. PERFORMING ORG. REPORT NUMBER: GRC-CR-1-465	
7. AUTHOR(s): E. F. Miller, Jr.	8. CONTRACT OR GRANT NUMBER(s): F30602-73-C-0344
9. PERFORMING ORGANIZATION NAME AND ADDRESS: General Research Corporation, P. O. Box 3587, Santa Barbara CA 93105	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55810219, AF-5581
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISIS), Griffiss AFB NY 13441	12. REPORT DATE: June 1975
	13. NUMBER OF PAGES: 150
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

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

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer:
Richard A. Robinson (ISIS)

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

Software Systems
Software Engineering and Quality
Software Reliability
Software Testing

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

This report describes the methodology which underlies and is supported by the Automated Verification System (AVS) which is scheduled for delivery to the Air Force in June 1975. The methodology is tailored to be largely independent of implementation and language. The AVS is intended to reduce the cost of assuring that software systems written in the JOVIAL J3 dialect, are comprehensively tested. The methodology is intended to engineer workable and practical first-level solutions to automating the measurement of

OVER

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

402 754

computer program testing effectiveness, assistance in manual testcase design and selection, and increased mechanization of certain aspects of software system maintenance.

PREFACE

As part of its program for applying advanced technology[1] to the important issue of software quality and reliability, Rome Air Development Center has contracted with General Research Corporation (Program Validation Project) for the design, development, installation and documentation of a JOVIAL Automated Verification System (JAVS). This system is intended to reduce the cost of assuring that software systems written in the JOVIAL/J3 dialect[2] are comprehensively tested. The work involves application of existing General Research proprietary algorithms, techniques, and methodologies to the general problem of JOVIAL-language software testing verification. The specific tasks are to engineer workable and practical first-level solutions to the following problems:

1. Automation in the measurement of computer program testing effectiveness
2. Assistance in manual testcase design and test data selection for increased testing effectiveness
3. Initial mechanization of certain aspects of software system maintenance

This report describes the methodology which underlies and is supported by the JOVIAL Automated Verification System, which is scheduled for delivery to the Air Force in June 1975. The methodological descriptions have been tailored to be largely independent of implementation and language. The discussion in the text is intended to be intuitive and demonstrative; technical details of certain concepts are presented in Appendices.

CONTENTS

CONTENTS (Cont.)

ILLUSTRATIONS

ILLUSTRATIONS (Cont.)

SUMMARY

This report describes a newly developed methodology for systematically and comprehensively testing computer software. At present, computer software is tested only according to its developers' intuitions, if it is tested at all. The overall reliability of a software product is at least partially dependent on the thoroughness of its testing; measurably increased testing therefore contributes to increased reliability.

Simple computer programs can be comprehensively tested without difficulty, by inspection. When computer software becomes complex, so that human intuition is inadequate to deal with its subleties, the testing activity must be based on a systematic and rigorous methodology. One purpose of such a methodology is to reduce the overall "cost" of testing to tolerable levels.

The reader of this report is assumed to be conversant with the main concepts of computer science, although not necessarily with the issues affecting software reliability. This summary presents the high points of the report, and is intended to provide the reader with a general feeling for the form and content of the material that follows.

ISSUES IN PROGRAM TESTING

In many applications which involve a digital computer, the difficult problem of developing the software portions of the system has been of increasing concern to system managers. Computer hardware reliability problems can be attacked with conventional engineering techniques, and positive results can be achieved. For all practical purposes, computer hardware can be made as reliable as necessary through multiple redundancy and other techniques.

The situation is very different for computer software--particularly in critical applications. There have been no truly effective ways to make software a low-risk element of a system implementation, regardless of the effort applied.

Approaches to Software Quality

The computer science community has recognized this important problem, and has been developing systematic approaches which seek to increase the reliability of software and, if possible, simultaneously reduce the overall cost of producing it.

Some of the "synthesis" approaches to doing this are the following:

- Structured Programming disciplines, which seek to minimize the complexity of software (and thereby enhance its overall quality and reliability) by constraining the control structures of the programming language used.
- Chief Programmer Teams, a management technique which assigns a talented person (the Chief Programmer) sole responsibility for all aspects of a software system, including its ultimate effectiveness and reliability.
- Technologically sophisticated software design methodologies such as "top down" and "bottom up" design and implementation disciplines, which attempt to systematize the software production process and thereby enhance the quality of programs.

These "synthesis" techniques generally try to increase software quality by keeping software problems from happening in the first place.

The alternative, to deal with software which has already been developed (or is in the final stages of development), involves two primary "analysis" approaches:

- Program proofs, which demonstrate the correctness of programs by treating them as if they were mathematical theorems. A mechanical theorem prover is often used to assist in the proof construction.
- Automated Verification Systems (AVSs), which attempt to increase the practical reliability of software by increasing the achieved level of testedness.

Automatable Methodology

The methodology described in this report addresses only the software testing area. Two distinct viewpoints of program testing can be taken:

- Single-module testing, wherein the objective is to devise methods for comprehensively testing single "modules" (individually invokable segments of program text).
- System testing, in which the whole software system is considered as if it were a single module. In this case, techniques closely related to those used for single-module testing are applied on a more global basis.

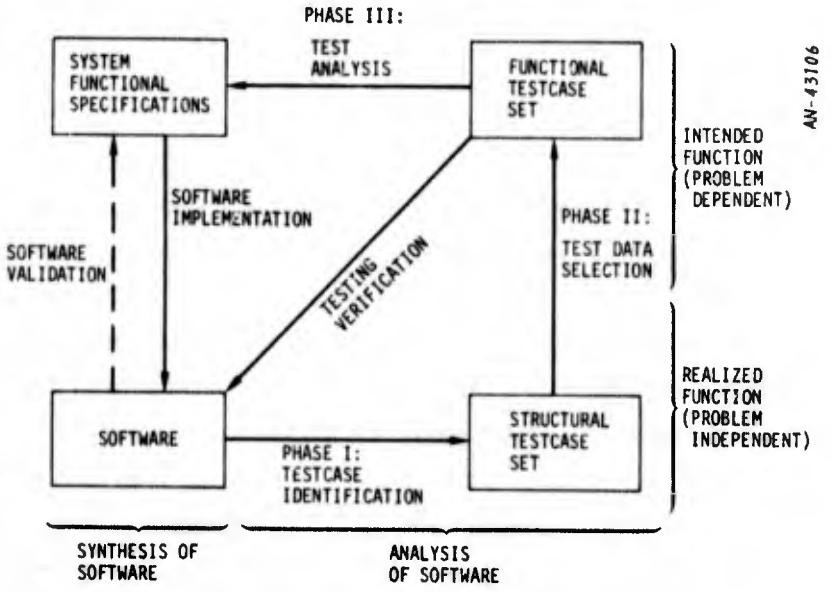
The need to assure a sound theoretical basis for a systematic general testing methodology is clear. It is equally important to develop a methodology that can be automated, because the combinatorial difficulties encountered in testing large-scale software systems can be so great that purely manual use of any systematic testing methodology would be of questionable value. Only methodologies which can be supported by an AVS are considered here; the analytical mechanisms employed are those which (by design) meet the dual requirements of generality and automatability.

Automating the process of single- and multiple-module testing has other advantages. When supported by an AVS, a program testing activity can rely on the fact that AVS instructions and advice are exhaustive and comprehensive. With the AVS, there should be no errors of omission. Furthermore, the AVS can be used to demonstrate its own veracity, since it is a software system itself.

The Meaning of "Verification"

The following diagram shows the relationships between a software System Functional Specification, the software, and the process by which an AVS seeks to invert, or "validate", the software implementation phase, as shown by a dashed line. In Phase I, the embodied software is analyzed by the AVS to produce a set of "structurally indicated" testcase patterns. In Phase II, structural indications of appropriate testcases are used to select actual testcase data. The aggregate of testcase data can be used to exercise the software system--that is, to assure that all portions of the software system have been exercised against some acceptance criteria. This is called Testing Verification.

The testcase set can also be used in another important way. In Phase III the relationship between the set of functional tests and the Functional Specification is used as an indicator of the veracity of the original imple-



Relation Between Testing and Validation

mentation activity. If the functional testcase set does not match one-for-one with the requirements stated in the Functional Specification, then one must conclude that either the implementation is imperfect, or the specification is imperfect (assuming, of course, that the functional testcase set has been correctly generated). The absence of a mismatch leads to a general increase in the program testers' belief in the software as a genuine implementation of the requirements stated in the functional specification.

Limitations of Testing

Although program testing is a powerful tool in this restricted sense, the current state-of-the-art will not support a fully automatable analysis of the correspondence between sets of testcases and the functional specifications of software (Phase III of the process just defined). Instead, the role of the AVS is to assure that the testing verification meets some criterion of comprehensiveness. Testing verification is the primary outcome of operation of the AVS and is an indirect indicator of the degree to which the real objective (matching tests with functional behavior) is actually met.

Comprehensive exercise of a software system does not guarantee that it is error-free. However, practical experience indicates that thorough exercise will locate a very high proportion of errors. Hence, the use of testing verification as an approximation to full program verification seems to be reasonable and practical.

SINGLE-MODULE TESTING

Single-module testing is oriented toward a particular well-defined testing goal, based on the internal properties of a program's control structure: to assure that each statement in the program has been exercised at least once, and that each decision in the program has been exercised at least once to each possible outcome (but not necessarily in every possible combination).

The smallest executable piece of a program is the sequence of activities the program performs in using the outcome of a decision to determine the

program's future course of action. We call this a "decision-to-decision path", or DD-path for short: a DD-path is diagrammed in the next illustration. If every DD-path in a program has been exercised at least once by a testcase set for that program, then both of the testing criteria stated above have been met.

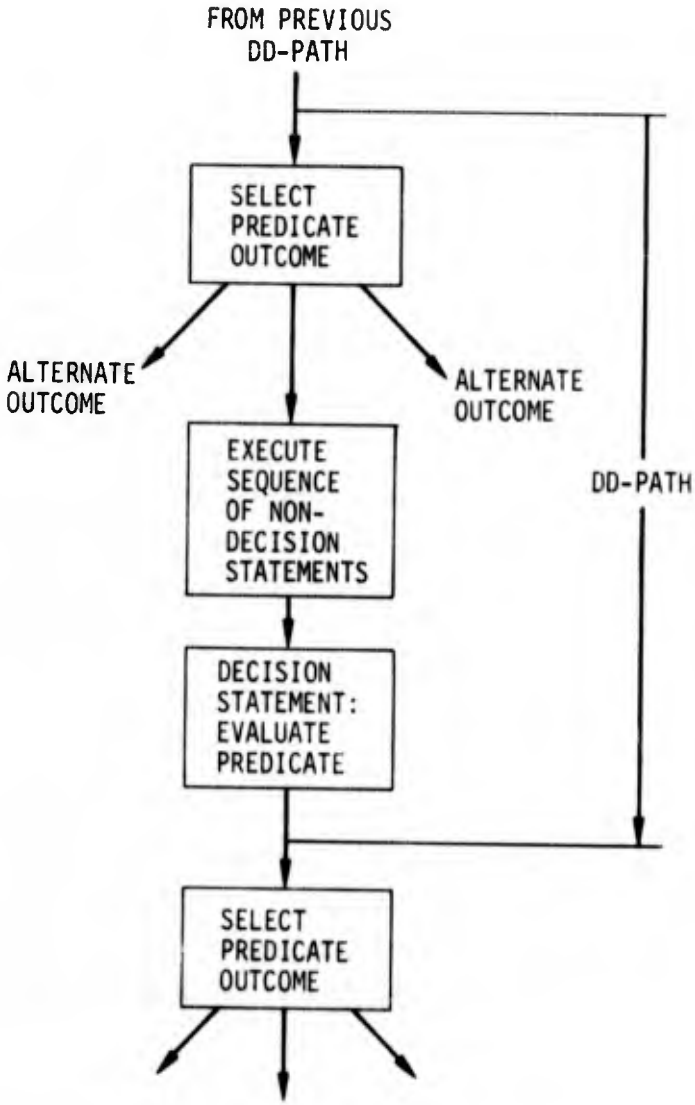
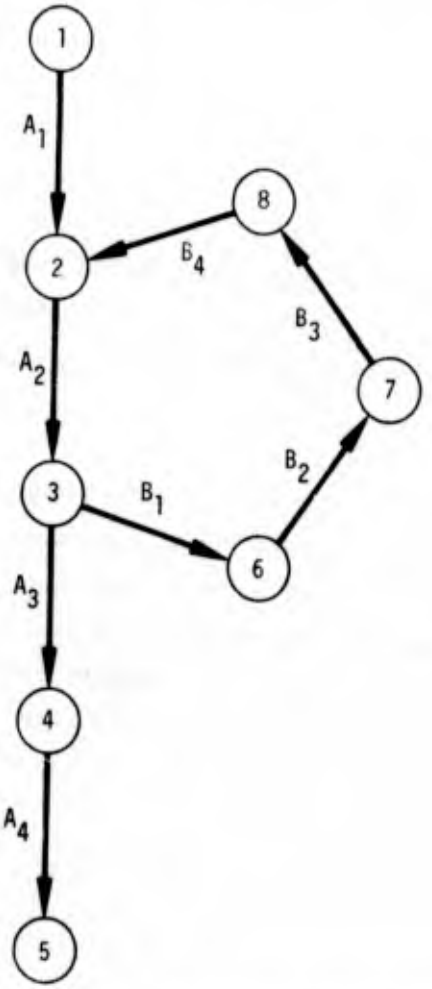


Diagram of Decision-to-Decision Path (DD-Path)

The Iteration Structure

During testing (after some initial set of tests), the situation is the following. Some as-yet-untested DD-path is selected as the subject of the question: "What means can be used to construct new testcase data which will cause this DD-path to be executed?"

The answer to this question is provided in part by the iteration structure of a module. Briefly, the iteration structure of the module is a tree of inter-dependent sets of DD-paths arranged in a way that makes it easy and straightforward to identify potential program flows. The pattern needed to deal with any particular program flow is like that shown below.



Iterative and Non-Iterative Flow Patterns

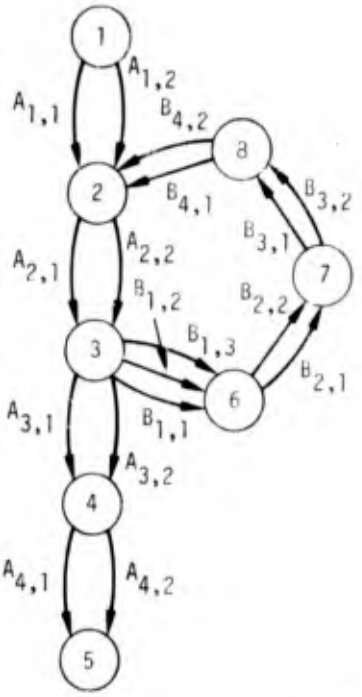
The diagram shows a single non-iterative flow pattern modified by a single iterative flow pattern. The non-iterative flow consists of DD-paths with the labels A_1, A_2, . . .; the iteration (in this case, a simple cycle), consists of DD-paths B_1, B_2, ...

This pattern is a prototype of all possible patterns of program flow because it incorporates the two necessary "types" of program execution: (1) selection of future program action and (2) repetition (iteration) of previously executed actions. For any computer program, and for a particular fixed set of program "input data," the actual flow pattern which results can be represented as a collection of such patterns. Although the actual pattern for a large program may be very complex, it is always composed only of selection and iteration operations.

Single-module testing is the process of identifying a particular pattern of program flow, and then constructing input data for the module which makes that program flow pattern actually happen. A test consists of a single invocation of a module, operating in a data environment which is sufficient to satisfy the data-input needs for that invocation. The set of data needed for a test is called the testcase dataset.

Program Flow Classes

Program flow patterns which do not differ significantly can be grouped into an equivalence class of program flow, a technique that simplifies the iteration structure significantly. Two actual flows are considered as members of the same class if their differences do not affect the overall iteration structure. The example shown opposite indicates how a number of program flows are grouped into classes such as those shown in the preceeding sketch. There are 16 possible flows in the "A" class, and 24 possible flows in the "B" class.



"Equivalent" Program Flows

The $\{A_i\}$ sequence forms what is called a "level-0 path class;" the $\{B_i\}$ sequence forms a "level-1 path class." In general, a level-i path class is a sequence of DD-paths which cause repetition of some DD-path present on some level-(i-1) path class. The dependence relationships between level-i path classes form a tree; the set of all possible classes of program flow is represented by the set of all subtrees of this tree.

Reach Sequences

The level-i path class tree forms the basis for single-module testcase data generation. Extracting a particular program text sequence to be analyzed for specific data value settings is accomplished with reach sequences. A reach sequence is a non-iterative flow pattern derived from the level-i path class tree which "reaches" a particular DD-path. Presumably, the DD-path reached is one which has not been exercised by any of the existing testcases. In the case illustrated, the DD-path sequence $A_{1,1} \rightarrow A_{2,2} \rightarrow B_{1,2} \rightarrow B_{2,2}$ is

a reach sequence which makes the program execute the DD-path $B_{2,2}$. Similar reach sequences are defined for multi-level iteration structures, and for the cycles which are formed by the intersection of two or more level-i path classes.

Testcase Data Generation

The testcase dataset which forces execution of a particular DD-path can be derived by examining the program text which corresponds to a reach sequence which employs the DD-path. This analysis process is relatively straightforward manually, but is not completely automatable even with state-of-the-art techniques.

Selecting the Testing Target

In the typical situation there are several untested DD-paths that need the program tester's attention. The testing methodology does not explicitly decide which of those to concentrate on, but it does provide for a program complexity measure to help rationalize the selection process. The untested DD-path chosen as the focus of testing is called the testing target.

If all DD-path attributes are otherwise equal, the program tester should choose a target DD-path which is as "high up" in the iteration structure (i.e., on a high-order level-i path class) and as "far down into the code" as possible. By concentrating on a difficult testcase, the tester will tend to maximize the amount of collateral testing. Executing a testcase designed for some particular DD-path often causes a large number of other DD-paths to be executed also. This phenomenon can reduce the overall effort needed to achieve the 100% testing coverage desired.

SYSTEM TESTING

The techniques used for single-module testing are employed systematically in achieving full system testing. A large software system is typically composed of subsystems, components, and single modules. Software system testing is performed in terms of the dynamic invocation structure (dynamic

organization) of the system; i.e., the tree of inter-module dependencies in which the execution of the software system occurs. Choosing the particular module to which the current testing effort should be directed is based on the dynamic organization, but this choice can also be made by analyzing the single-module coverages already achieved.

System Testing Strategies

Two possible system testing strategies are: (1) bottom-up testing and (2) top-down testing. The bottom-up testing strategy tests single modules at the bottom of the invocation chains first, followed by higher-up elements of the software system in their turn. The bottom-up strategy tends to maximize the individually achievable levels of testing at the possible expense of significant problems in constructing the testing environment.

The top-down testing strategy concentrates first on the "topmost" modules of the software system and operates on successively deeper modules within the invocation hierarchy. This method of system testing is likely to produce a large amount of collateral testing. It has the disadvantage, however, that it may be difficult to construct new testcase data when the testing target is far from the apex of the invocation hierarchy. This may result from a program's protection of data used by lower-level modules, often a normal and desirable attribute of large-scale software systems.

Coverage and Target Selection

A testing coverage measure consolidates individual module testing coverages into a value applicable to an entire software system. One simple measure is to consider the software system to be tested as much as its least-tested module, expressed in terms of the exercised percentage of DD-paths. Other measures, which take module and system complexity into account, are also possible.

The testing strategy can use explicit or implicit rules to select the appropriate target for continuing testing efforts. With the simple measure described above, the next target is always either (1) the least tested module at the current level within the hierarchy (top down), or (2) the least tested module within the subset of modules currently under analysis or invoked by those modules (bottom up).

AUTOMATED VERIFICATION SYSTEM DESIGN

An Automated Verification System (AVS) supports the testing methodology by providing various support facilities. The AVS database contains all relevant information about the source text and the structure of the software being analyzed. This information is generated and stored once for each module.

Instrumentation

Individual modules whose source text and related structural information have already been added to the AVS database can be instrumented before execution in a testing environment. The instrumentation techniques are designed to (1) produce program texts which are logically equivalent to the originals, and (2) provide low-overhead invocations to a special instrumentation module which intercepts the program's flow of control as it passes through each DD-path.

Data Collection and Reduction

During tests, the program performs instrumentation invocations that carry with them the name of the module and the number of the DD-path currently being executed. The AVS data collection and reduction facility records these invocations and produces reports which indicate the testing coverage attained for each module.

Test Case Data Generation Assistance

The testcase generation assistance facility of the AVS is used when DD-paths within a module are found not to have been executed. Under user-specified commands, the AVS uses the information about the level-i path

class tree to generate appropriate combinations of level-i paths (reach and cycle sequences). The user analyzes the program source text that corresponds to these sequences to determine appropriate testcase data.

Retesting Guidance

In addition to its role in software testing, the AVS provides rudimentary answers to two questions important in maintaining software:

1. If a particular module has been changed, what other modules will have to be retested?

2. If a series of changes has been made throughout a software system, which modules will have to be retested to restore the system's testedness level?

The first question is answered by having the AVS refer to the invocation structure of the software system it has analyzed. The second question is answered by using the instrumentation facility of the AVS: those modules which have less than the required level of exercise must be re-examined in detail.

THE IMPACT OF THE AVS-BASED METHODOLOGY

It is difficult to assess the exact impact of widespread Air Force use of AVS testing technology, because there is only a limited experience basis from which estimates of decreased life-cycle software costs could be made. Based on our initial experience, and admittedly our intuition, we can expect the AVS testing technology to:

- Decrease the overall cost of comprehensively testing software or, equivalently, increase the level of testing coverage achieved for the same cost.

- Identify programming constructs which are difficult to handle from a testing point of view and eventually eliminate their use in critical software systems.

- Assure that software systems subjected to the AVS-based testing methodology will have a significantly decreased likelihood of failure in actual use, particularly if the software failures could result from lack of comprehensive testing.
- Provide a strong technological basis for continued development of the methodologies and the tools which support them.

In the long run, the benefits of AVS technology will occur (1) through increased understanding of the general problem of software testing; and (2) through development of better automated tools.

SUMMARY

The main points made in this section (which are needed to understand the perspective of the remainder of the report) are the following:

- Among a variety of approaches to improving software quality, systematic program testing offers near-term benefits not available with other techniques.
- The most difficult problem in testing is the construction of testcase data which exercises previously unexercised parts of programs.
- Detailed analysis of a program's control structure can be used to construct testcase data.
- The methodology for single-module testing can be extended to system (multi-module) testing.
- The simplest measures of testing coverage, if achieved, have a positive effect on installed software system quality.
- Systematic software testing can be supported by the facilities of an Automated Verification System (AVS).
- Systematizing the program testing process can reduce software quality enhancement costs appreciably.

SECTION 1

IMPROVING SOFTWARE THROUGH TESTING

In applications involving digital computers, achieving high-quality, reliable, and easily maintainable computer software is an increasingly pivotal activity. Computer hardware now attains capabilities undreamed of as recently as a decade ago, and hardware costs continue to decrease in both absolute and relative (cost per machine instruction executed) terms. But the technology for generating and testing software has not kept pace, and software costs continue to increase as the level of sophistication in computer applications grows. Software, its design and construction, its installation and use, and its maintenance can be characterized as expensive, time-consuming, and perilous to ultimate system performance.

While there are many causes of this unfortunate and potentially dangerous situation, it is more important to ask: "What can be done to improve the quality of computer software?" The testing methodology described in this report is intended to provide some preliminary answers to this question.

This section addresses some general aspects of the software quality problem, places some of the "quality enhancement techniques" in perspective with one another, and identifies the way in which we hope to improve software quality by increasing its "testedness."

1.1 SOFTWARE QUALITY PROBLEMS AND THEIR SOLUTIONS

Like any manufactured product, software is the result of a sequence of production activities: design, implementation, checkout, and installation. Computer software, however, has special characteristics which make each of these steps difficult. Software is largely an intellectual product and therefore is subject to whatever vagaries plague the minds of its creators. Although no one seriously advocates complete elimination of the human in the software development process, nearly everyone agrees that there must be some diminution of the role that human creativity plays in producing software. Instead, human creativity (an unquestionably valuable resource) should be

controlled, monitored, and focused on production of software of high quality primarily, and of internal elegance only secondarily.

Approaches that increase software quality can be categorized as "synthesis" or "analysis" approaches. Examples are given below.

1.1.1 Synthesis Approaches to Quality Software

In synthesis approaches, high quality is pursued through psychological or technological disciplines under which the software is generated.

Structured Programming. This discipline seeks to assure quality software by guaranteeing that programs are expressed only with simple and powerful program control structures, and in terms of highly organized data structures. Structured programming is the current vogue, and some spectacular results have been attributed to the technique.[3,4] While it is still possible to make mistakes with structured programming, it has been argued that it is very much more difficult for a programmer to do so. The outcome, it is argued, is inherently "high quality" program text and, consequently, inherently better software.[4]

Top-Down Design and Implementation. This discipline is based on the belief that good design does not just happen, but is engineered into a software system from its beginning. A software system is built by coding in successive stages of refinement, beginning with the topmost (most general) element and proceeding downward until the job is complete.[4] Whenever necessary, uncompleted program elements are "stubbed" (simulated with dummy versions) as a mechanism to permit evaluation of the whole system's behavior well before the implementation is complete.

Chief Programmer Teams. This management-oriented concept is based on the belief that good software systems are assembled by a small, cohesively-run programming team under the leadership of a single Chief Programmer, whose responsibility is to assure good design (and effective operation) in all phases of a software development project.[3,4]

1.1.2 Analysis Approaches to Quality Software

Analysis techniques of assuring high-quality software are concerned less with means of production than with the specific techniques used to guarantee its quality. Because the techniques used generally assume that a fully-implemented software system is available for detailed examination, there is sometimes an emphasis on automation of the techniques whenever that is demonstrably cost-effective.

Program Proving. This process uses the actual program texts to prove precise mathematical theorems about program behavior,[5,6] an activity which is sometimes very complicated. Practical application of program proving techniques to realistic computer programs is at least a decade away.[6]

Software Quality Analyzers. These analyzers attempt to recognize common forms of errors systematically (and automatically when possible), and thus identify how to improve quality. Typical analyzers identify such faults as conflicts in data specification and usage, expressions which could cause run-time exceptions (such as denominators that might equal zero), text that could be rearranged to increase clarity, and common subexpressions that can be eliminated or simplified.

Automated Verification Systems. A typical Automated Verification System (AVS) supports enhancement of software quality and reliability by attempting to impose rigorous and systematic constraints on the means chosen to exercise the software. Software which has been thoroughly exercised can be expected not to contain any overt (and some types of covert) mistakes. In addition, the exercising may demonstrate adherence to functional behavior requirements that can satisfy system designers that the software is working well.

1.1.3 Systematic Methodologies

This report deals specifically with a systematic testing methodology that meets these criteria:

- The methodology must be amenable to automation, so that it can be supported by an AVS.

- The methodology should be founded strongly in a theory of program behavior and, if possible, must have a strong relation to techniques of program proving.*
- The methodology must be applicable to very large software systems, because it is most important--and difficult--to attain high quality in them.
- The methodology must be implementable with state-of-the-art software technology, so that it can be applied immediately to existing software systems.

Meeting all of these constraints is a difficult problem both in arriving at a good methodological design and from the AVS designer's viewpoint. The methodology and analysis techniques described here meet these important criteria.

1.2 ROLE OF TESTING IN SOFTWARE DEVELOPMENT PROCESS

Various forms of software "test and evaluation" pervade the entire development process. In this section we place these test and evaluation activities in perspective in terms of a reasonably comprehensive model of the "software development process."**

1.2.1 Overview of Software Development Process

Figure 1.1 exhibits the software development process as being composed of five sequential phases:***

1. System Requirements. The process begins when the needs of the [computer-based parts of the] system are identified in terms of the intended system behavior.

---

*This criterion is included in the expectation that, when the projected decade for maturation of program proving techniques has passed, only a straightforward adaptation of AVS concepts will be necessary to close the loop between program proving and program testing.

**This section is based largely on the work of Reifer.[7]

***The computer is usually only part of a system, but we concentrate here on the computer-related aspects of the whole system.

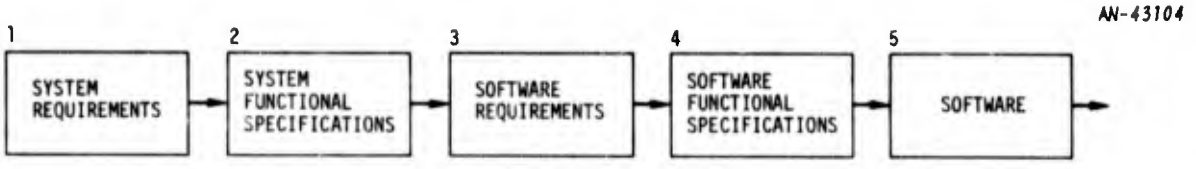


Figure 1.1. Software Production Process (after Ref. 7)

---

2. System Functional Specifications. After the System Requirements are known, the next step is to develop a detailed System Functional Specification, which defines the individual functions of each software element.

3. Software Requirements. After the function of each software element is identified, it is then possible to design specific Software Requirements which the computer software must meet to fulfill the system's needs.

4. Software Functional Specifications. These specifications contain the software design, and include detailed information (sufficient to support implementation) about the organization and expected functional behavior of each software element.

5. Software. This is the outcome of the implementation activity, during which the detailed design is converted into programs.

These five steps may be repeated a number of times during the software production process.

1.2.2 Verification Functions

Each stage of the software development process can be subjected to one or more forms of verification, intended to assure the quality of the "product" at that stage. The verification activities, shown schematically in Fig. 1.2, are as follows:[7]

System Verification. This is the process of determining that the System Requirements are clearly and correctly expressed at the next phase of the process. This form of verification is intended to insure that unintended system capabilities are not included by error, and that all of the required system capabilities are actually included.

Requirements Verification. This verification step is intended to demonstrate that the Software Requirements reflect the needs expressed in the

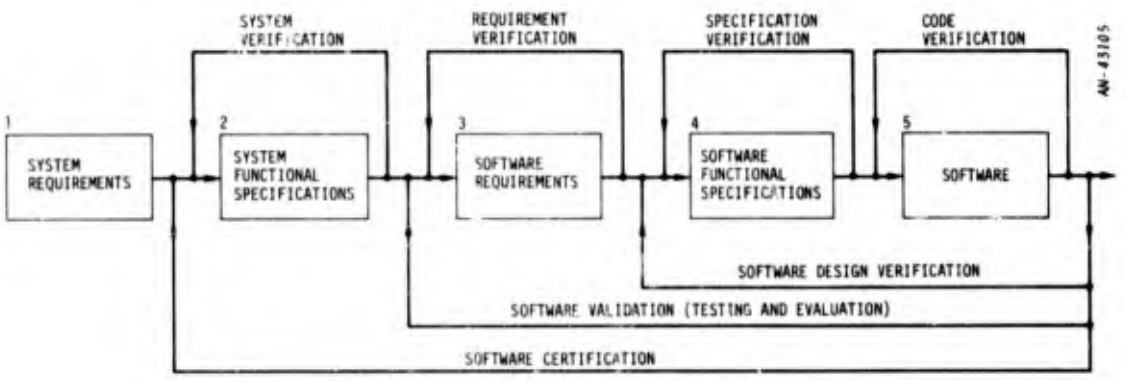


Figure 1.2. Verification, Validation, and Certification Process Model (from Ref. 7)

System Functional Specifications. Requirements verification addresses the question: "Will the conceptual software design work?"

Specification Verification. This is the process of determining whether the Software Functional Specifications (the basic software design) meet the goals set forth in the Software Requirements. In other words, this verification step attempts to assure that the conceptual design actually "does the job" it is intended to do.

Code Verification. This verification step is based upon the implemented software, and consists of determining that the appropriate relationship exists between the "embodied solution" and the "intended solution" as expressed in the Software Functional Specifications. Code verification is an analytic step which involves detailed examination of software source texts for their consistency, veracity, and "correctness". During this phase, one asks such questions as:

1. Is the program logic flow correct?
2. Are program interfaces properly designed and implemented?
3. Are the program performance measures (execution time and program text size) within acceptable bounds?
4. Is the software sufficiently error-tolerant?
5. Is the software adequately error-resistant?

These five verification activities, in aggregate, assure successful passage from System Requirements identification to properly working, adequately resilient, embodied software. Some of the verification steps may be grouped together, as shown at the bottom of Fig. 1.2 and described next.

1.2.3 Software Design Verification

This activity analyzes embodied software (code) and attempts to assure that it faithfully implements the Software Requirements Specifications. This form of verification is very difficult because it requires simultaneous knowledge of the actual implementation and thorough understanding of the needs

the software is supposed to satisfy. Software Design Verification requires two activities:

1. Examine the algorithms used in the Software and determine, independently of the Software Functional Specifications, that each algorithm adequately supports the Software Requirements. Doing this requires knowledge of the interaction of the software with the computer hardware, and detailed "system-level understanding" of the Software Requirements and their interactions with the software as it was actually implemented.

2. Examine the "needs" expressed in the Software Functional Specifications and assure that all of those needs are met by the embodied software. In this case, it is necessary to know which functional algorithms are the appropriate ones to have chosen, to understand which implementations are "good" ones to have used, and to appreciate what constitutes "good design" with respect to the peculiarities of the application.

Computer science is not yet sufficiently advanced to provide other than very general guidelines for dealing with questions of this nature. Therefore, this form of verification is usually subsumed by other forms for which there are some reasonably practical and effective methodologies and tools.

1.2.4 Software Validation (Testing and Evaluation)

As shown in Fig. 1.2, the Software Validation process attempts to relate the embodied software with the system-level Functional Specifications. The general objective of this verification activity is to determine whether the software truly mechanizes the specified design and performance criteria. This is accomplished by testing the software, and by analyzing the results of the tests in terms of the Functional Specifications of the system. Such features as execution time and program text size are often as important as the "correctness" of the results of each software test.

When the software cannot be subjected to live tests with real data,* then alternative means must be found to support the testing and evaluation activity. In many cases, it is possible to simulate reality sufficiently well to ascertain these same forms of information without live tests. There have been significant technological breakthroughs relevant to this issue. The interested reader is advised to consult Refs. 8 and 9 for some comments on the problems of testing certain kinds of real-time systems which cannot be tested in the "real world".

Because Software Validation is accomplished primarily by analyzing executions of the completed software, there are many opportunities for significant interaction between the testing activity and detailed analysis of the System Functional Specifications. Combining Software Validation with Software Design Verification forms the central thrust of the testing methodology elaborated in Secs. 2 through 4.

1.2.5 Software Certification

Software Certification is the full extension of the Software Validation process, and deals with the issues that arise in trying to assure that the implemented code satisfies the explicit System Requirements. After certification--which might be accomplished with combinations of full-system tests and very detailed examinations of the outcomes of such tests--official endorsement of an acceptable operational software capability can be given.

1.2.6 Role of Testing

All forms of verification involve testing. A testing activity can be characterized as follows:

- Statement of Testing Objective. The desired outcome of a test is stated, and the means to decide whether the outcome has been met are designed.
- Execution of the Test. The test is performed in an environment in which it is possible to collect the information needed to determine success or failure.

*Any computer system responsible for assuring the safety of human life falls into this category.

- Analysis of the Test. After execution of the test, the information necessary to determine success or failure is analyzed to determine the actual test outcome.

- Redesign of Test. If the outcome of the test is not the one sought in the test objective, then one redesigns the test in a way which improves the usefulness of the next outcome.

The view that all stages of the software development process involve some form of "testing" is important in subsequent discussions, where we describe systematic means for devising "good" tests for specific parts of the entire multi-step verification activity. (See Secs. 2 and 3.)

1.3 RELATION BETWEEN TESTING AND VALIDATION

There is an important relationship between software testing and software validation. The program testing methodology described in this report provides a bridge between what is desired (a generalized Software Validation capability) and what is possible and practical now.

1.3.1 An Overview

Figure 1.3 describes the relationship between software testing and software validation, in terms of special value for developing a program testing methodology. The implementation phase of the software development process has performed the transformation:

SYSTEM FUNCTIONAL SPECIFICATIONS ⟶ SOFTWARE

The objective of the validation phase is to reverse this process; that is, to provide (by an alternate route) the effect of the transformation:

SOFTWARE ⟶ SYSTEM FUNCTIONAL SPECIFICATIONS

As shown in Fig. 1.3, this can be accomplished by dealing with sets of "test-cases" for the software. This activity divides into three phases (shown in the figure):

- Phase I: Testcase Identification. The software is analyzed for specific kinds of information (explained in detail in Secs. 2 and 3) that yield a collection of structural tests.

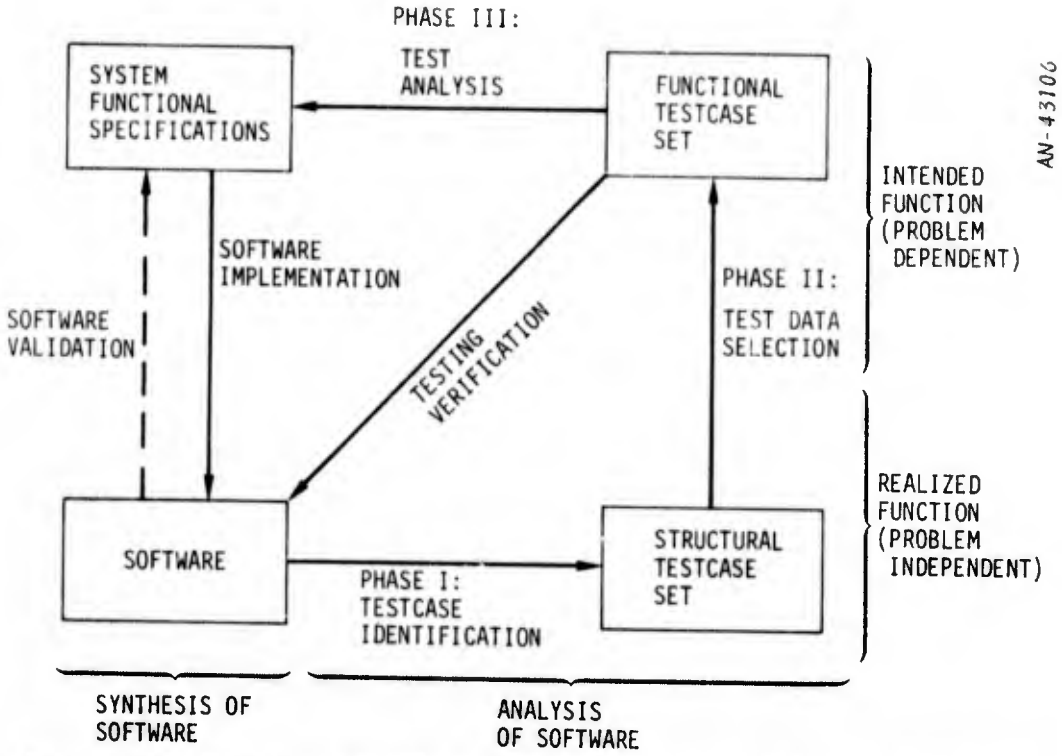


Figure 1.3. Relation Between Software Testing and Software Validation

---

- Phase II: Test Data Selection. Supplying specific input values for a structural test converts it into a functional test.
- Phase III: Test Analysis. The set of functional testcases for the software is then analyzed for its relationship to the System Functional Specifications. If the software actually corresponds to the requirements, there will be one functional testcase for each separable functional requirement. If there are more testcases than functional requirements, or if there is no testcase for some functional requirement, then one can conclude that the software is deficient in some manner.

The methodology described in this report addresses only the first two phases shown in Fig. 1.3; the third phase is assumed to be handled by other means.

The AVS which supports the methodology performs the analysis functions needed in Phases I and II of the process. One would expect that in the long

run the functions of the AVS would be extended to encompass even more of the process.

1.3.2 Testing Verification

Some interesting and important symmetries are evident in the Software Validation process shown in Fig. 1.3.

First, note that there is a clear division between the parts of the process that deal with the intended function of the software, and the parts which deal with the realized function. For example, structural analysis solely of the realized software is used to construct appropriate sets of testcases based on whatever is present in the software.

Second, note that all of the activities other than the software implementation process (over which we have no control) involve analysis of some kind. It is an analysis function to choose values of input-space variables to convert a structural test into a functional test, for example.

The bridge between functional testcase sets (the outcome of Phase II) and the behavior these functional tests evoke from the software system is called testing verification, shown in Fig. 1.3 as a diagonal line. Testing verification binds the problem-dependent and problem-independent aspects of the process together by providing the means to observe the actual behavior of a functional testcase set derived purely from structural analysis. In addition, these executions of the software provide the measures of testing coverage that are necessary for subsequent application of the testing methodology. If the actual behavior of the software system differs in any way from what is expected during testing verification, then one has located either a fault in the software or a fault in the functional specification. Thus, testing verification provides a useful approximation to comprehensive test analysis (Phase III).

1.3.3 An Example

The distinction drawn in Fig. 1.3 can be understood by considering a simple example (Fig. 1.4). The program text in this example satisfies the

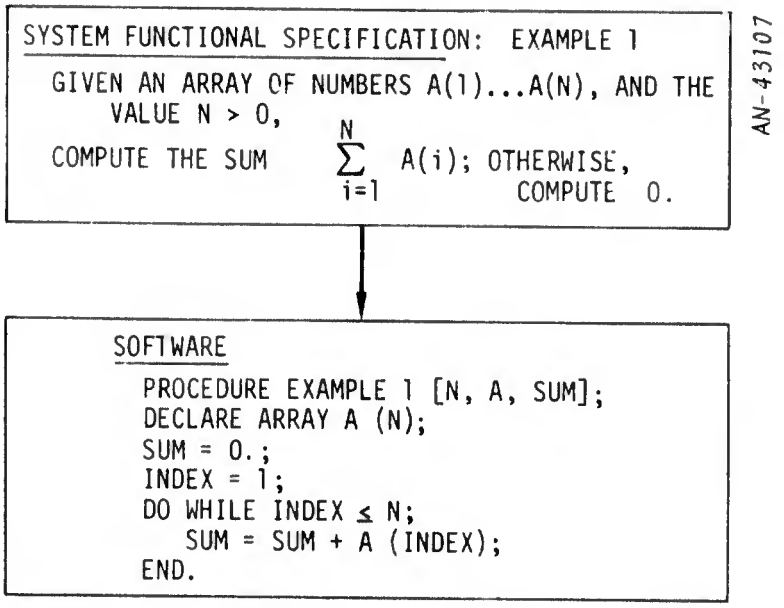


Figure 1.4. Example Specification and Software

System Functional Specification, as is easy to see. In terms of this program text, we can describe the three steps in the validation process represented in Fig. 1.3.

1. Inspection of the program text reveals that two distinct tests are required:

   a. For the case $N \leq 0$, there is no data to be summed.

   b. For the case $N \geq 1$, there is data to be summed.

   These two structural testcases are sufficient to exercise all the statements within the program text.

2. The two testcases must now be fleshed out by generating data which will cause them to execute as planned. For this example, this amounts to choosing appropriate values of the array A and, as indicated in (1.a) and (1.b), selecting the appropriate values of N.

3. Analysis of these two functional testcases, showing that they meet the Functional Specifications, must rely on two observations:

   a. The fact that $\sum_{i=1}^{N} A(i)$ is not meaningful for $N \leq 0$

   b. The fact that successful generation of $\sum_{i=1}^{N} A(i)$ for an arbitrarily selected set of values of the $A(i)$ is a clear indication of the capability of the program text to perform its intended function.*

These three steps validate that the program text meets its Functional Specifications. This is also indicated by the following facts:

1. The structurally indicated tests (1.a and 1.b) operate and produce meaningful results.
2. There are no structurally indicated tests which do not correspond to the Functional Specifications.
3. There are no Functional Specifications which are not represented in the Functional Testcase set.

The program testing methodology is largely an extrapolation of these simple techniques, expressed in a systematic manner and designed to be applicable to large software systems.

1.3.4 Advantages of Testing

As an alternative to direct Software Validation (the dashed arrow in Fig. 1.3), a systematic program testing methodology offers a number of important advantages. Some of these are:

- Systematic and automatable methodologies for generating the Structural Testcases and the Functional Testcases can be developed, and assurance of the quality of software is therefore on much firmer theoretical ground.

*The general question of the choice of data to guarantee correctness of a program text is a considerably more difficult problem.

- If designed properly, the systematic methodology can operate independently of the size and complexity of the software, even though Phase III of the process (matching Functional Specifications with Functional Tests) is difficult to automate.
- There is a valuable chance to select functional tests which individually "consume" a high proportion of the structurally indicated testcase set. If "good functional tests" achieve some well-understood criterion of testing coverage, then one has increased confidence in the ability of the software to meet its Functional Specifications, even for instances not included in the testcase set.

1.3.5 Disadvantages of Testing

As already noted, it is not presently possible to draw mathematically provable conclusions about software quality solely by systematic testing. Even if a software system is thoroughly and systematically tested, there is no guarantee that it will not fail. An affirmative statement about testing is: a system which has been thoroughly tested will not fail for any of the cases included in the testcase set. Thus, testing should continue to be performed by software engineers who have an understanding of the benefits and limitations of systematic testing.

1.4 THE GENERAL METHODOLOGICAL BASIS

Systematic testing of computer software is based on knowledge of its internal structure. There are several kinds of structure in a computer program: data structure is the organization of the data on which the program operates; computation structure describes the program's operations on the data; and control structure is the program's means of organizing the computations. The kind of structure dealt with by the program testing methodology described here is the control structure. The other kinds of structure are tested only insofar as they interact with the control structure.

In this section we set down the general characteristics of software control structure to which the general program testing methodology will be applied.

1.4.1 Static Organization

A software system is assumed to be organized according to the pyramidal structure shown in Fig. 1.5, which indicates the general dependence of a software system on its underlying substructures. The natural organization for such a system is to consider it as partitioned into the following categories:

- Individual units of code, called modules*
- Collections of modules, called components
- Collections of components, called subsystems
- A collection of subsystems, which compose the entire software system

There is no requirement that any of the categories exist in fact. For example, a software system could consist of only one or two modules. The important point to recognize is that large-scale software systems are organized in this manner, although they may not be described in this fashion explicitly.

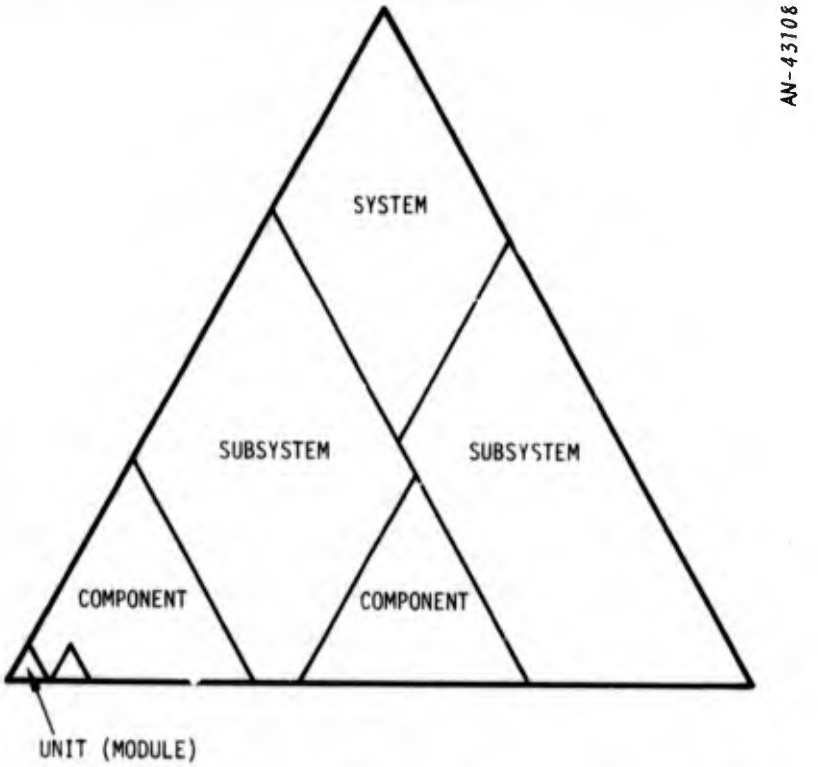


Figure 1.5. Static Organization of Software System

*Henceforth, a module will always mean a "smallest invokable element of program text."

The static organization of a software system describes the way the individual elements of the system depend on one another, but without regard to the way program control flows up and down. The static organization can be thought of as corresponding to the compile-time listings of the program texts.

1.4.2 Dynamic Organization

The dynamic organization of a software system may largely coincide with its static organization, to the extent that major components and subsystems are independent of one another. Individual modules within each component or subsystem are invoked (often by means of a standard "subroutine invocation procedure") as the means to achieve significant economies in software size. From a behavioral standpoint, the difference between the static organization and the dynamic organization lies in the manner in which individual modules are invoked. If every element (module, component, or subsystem) can be invoked from only one location within a higher-level element of the software system, then the static and the dynamic structures are identical. In effect, there is only one copy of each module, statically and dynamically. This will not be the case for most large software systems, however.

The dynamic organization of a software system is the structure which results from considering the effects of all of the invocations between modules, components, and subsystems within the software system. This dynamic organization is the "invocation hierarchy" of the software system. The systematic testing methodology deals with software systems exclusively within their invocation hierarchy.

1.4.3 A Unified Hierarchical Structure

Besides the organizational hierarchies described in Secs. 1.4.1 and 1.4.2, there is a third structural hierarchy which will be important. This is called the iteration hierarchy; it is included within the framework of the invocation hierarchy. This concept is described in detail in Sec. 2, but it is important to have some intuitive feeling for it now.

Computer programs can be considered as being composed of only two distinct types of operations:

> Selection Operations. Conditional actions taken to select future program actions.
>
> Iteration Operations. Conditional actions taken to decide which program actions to repeat.

Selection operations occur in program text as detailed selection logic, and are generally embedded in iteration operations, which occur in program text as logical operations specifically intended to control iterations within the code. The selection operations are the ones which cause the immense variety in execution flow patterns possible in large program texts. The iteration operations are the ones which make it possible for programs to perform repetitive operations of great power and diversity.

The invocation structure of a software system can be expanded into its "merged" iteration structure. This is explained as follows (see Fig. 1.6). Suppose that each invocation of each module within the software system is replaced with a complete copy of the module invoked (with parametric symbols replaced in the appropriate manner). The iteration structure for the resulting "program" is the one for the software system, after the individual iteration structures are copied together according to the invocation structure. Figures 1.6d and 1.6e show the individual and aggregate iteration structure for two invocations of module B by module A. The iteration structure describes the way a program text controls iteration within itself; the selection functions can also contribute to the total execution complexity, as should be evident.

The points of control transfer between modules--the dynamic invocations between modules--are natural cut-points for the merged iteration structure. Thus we can:

- Treat the iteration structures of modules separately, without loss of generality in terms of the entire iteration structure.
- Treat large collections of modules without necessarily having to deal with the fine details of the iteration structure of each module.

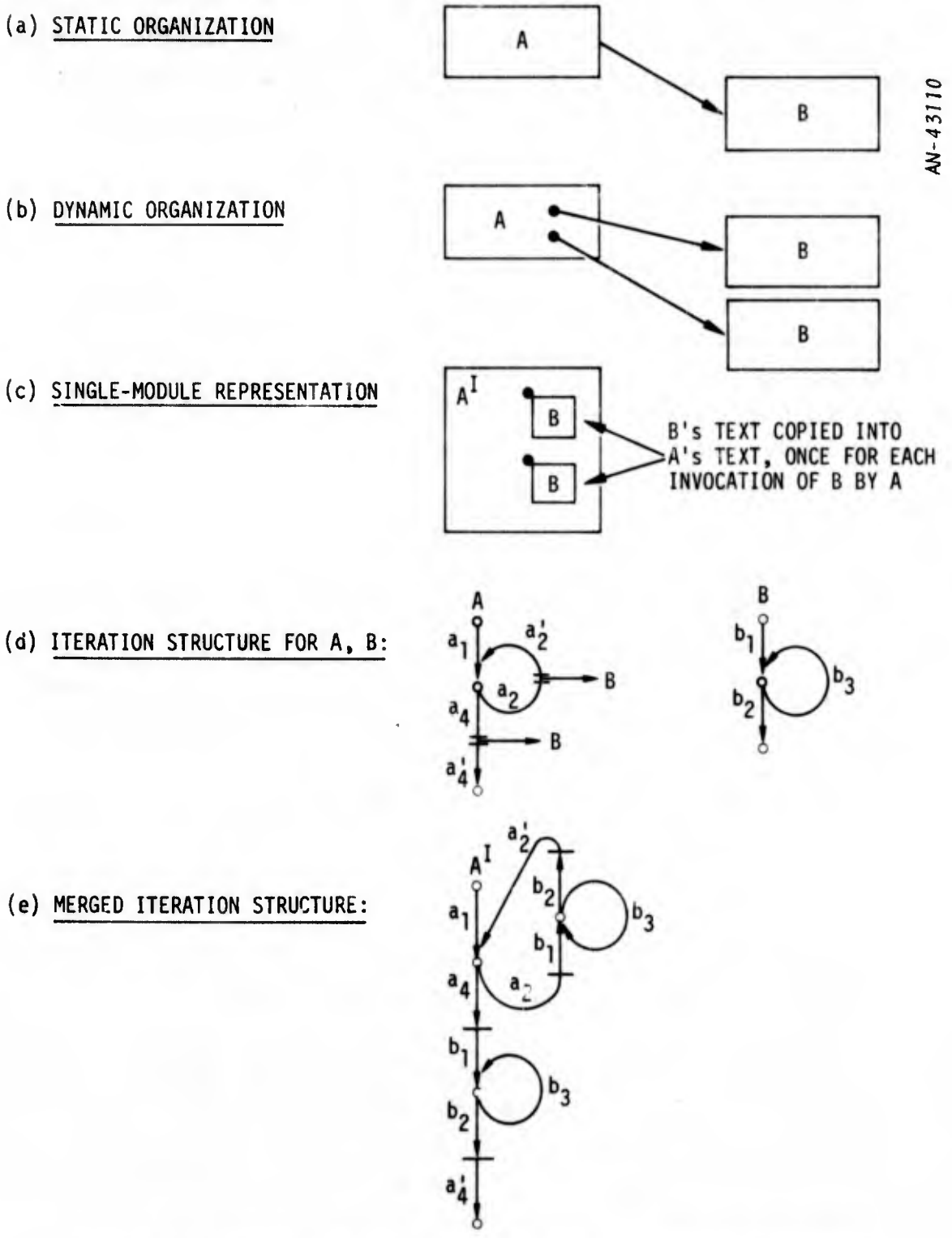


Figure 1.6. Representation of Software System Iteration Structure

These facts are central to the testing methodology. The invocation structure can be thought of as the skeleton of the merged iteration structure, because the single-module iteration structures are joined within the framework of the invocation structure. Details about the local iteration structure of any part of a software system are available when needed, but can be ignored when necessary and advantageous.

1.4.4 An Overview of the Methodology

Software system testing is organized according to two hierarchies: the invocation structure and the iteration structure. System-level testing is performed primarily in terms of the invocation structure of the software system; single-module testing is based wholly on the iteration structure. Systematically testing a software system requires judiciously chosen use of the two distinct testing disciplines:

1. Testing of single modules, regardless of where they lie in the invocation structure, using the properties of the module's iteration structure. (The specific mechanisms for doing this are described in detail in Sec. 2.)

2. Testing of components and subsystems, regardless of where they lie in the iteration structure, based on their positions within the invocation structure. This testing form deals with collections of modules which invoke one another, but considers the actions of all but the topmost module within any component as primitive activities, to be tested according to (1). (This is discussed in Sec. 3.)

The selection between (1) and (2) is based, at least in part, on measuring the level of testing coverage achieved as testing proceeds.

Although the testing methodology and its principles apply to any software system--the methodology is fully general--certain features of the supporting technology make it necessary to add some restrictions. There is every expectation that these limitations can be relaxed at some time in the future. The restrictions are·

1. The software system must not be implemented with recursion, since the properties of the iteration structure in the presence of recursive calls are not well understood.

2. The software system cannot involve time-dependent operations, since the methodology relies on an instrumentation technique to collect testing coverage data, and any form of instrumentation will always invalidate time-dependent processing.

3. The software system cannot involve concurrent operations, since no provision has been made for co-executing processes.

It is perfectly reasonable to apply the testing methodology to software systems which include such features, provided that separate steps (not part of the methodology nor supported by the AVS) are taken to deal with the issues of recursion, time dependence, and concurrency.

SECTION 2

SYSTEMATIC SINGLE-MODULE TESTING

The general approach to systematic testing of computer software, which was identified in its broadest outlines in the preceding section, involves two levels of attack on the testing problem. In some phases of the testing activity, a software tester may choose to concentrate on software-system-wide testing matters; in other phases of the effort, the primary concern is to achieve comprehensive testing of a single module.

Techniques for dealing with system-wide problems are presented in Sec. 3. This section is devoted to the methodology for single module testing.

2.1 AN OUTLINE OF THE METHODOLOGY

2.1.1 Fine Structures of Software: the DD-Path

A useful control structure testing goal--one which is easily measurable and attainable in practice--is one that deals with the largest immutably executable sequences of program text. These "segments", or "Decision-to-Decision Paths" (DD-paths) represent the coarsest granulation of program structure which still permits comprehensive representation of a program's iteration structure. The notion of a DD-path is discussed in detail in Sec. 2.2; a useful intuitive definition is:

> A Decision-to-Decision Path (DD-path) is the sequence of program actions which result from a control-flow decision based on evaluation of a program predicate (logical-valued expression used in the program control structure), up to and including the evaluation of the next such predicate, but excluding that predicate's use in deciding future program behavior.

A DD-path represents the operations performed between evaluations of decision statements within a program text. When the program begins operation it proceeds to the first decision point, which is the end of the first DD-path. Thereafter, the program action is like the operation of an escapement in a clock: a decision outcome determines the next activities until the program

comes to another decision, and so on. The last DD-path executed results in program exit.

Some observations about the properties of actual programs, expressed in terms of DD-paths, are necessary before discussing specific testing goals:

- Even very short programs can have many DD-paths.
- There may be a very large number of distinct sequences of DD-paths from entry to exit of a program.
- For this reason, thorough exercise of "all possible program paths" is not possible for typical programs.
- It can be very difficult to identify testcase data which will cause a particular DD-path, or sequence of DD-paths, to be executed. The difficulty grows as the length of the DD-path sequence increases.*

2.1.2 Measures of Testedness

Having stated these difficulties, one might naturally ask, "What hope is there for testing programs?" What is needed is a measure of testing that (1) can be achieved for all programs, large and small, and (2) when achieved implies something about the quality of the software system.

A first-level measure of testedness is the following:

Goal 1: Provide usefully small sets of testcases for single modules which, in aggregate, exercise each DD-path at least once.**

---

*This problem is recursively unsolvable in general, as is well known.[10] Much research effort is, nonetheless, being devoted to devising practical solutions to this problem. At the present writing, no useful techniques have been demonstrated, and the methodology described subsequently is specifically designed to take this (unfortunate, but true) fact into account.

**Real programs may have DD-paths that cannot be exercised, no matter what input values are used. The goal might be revised to read "...exercise each DD-path that can be exercised, at least once; and, for each DD-path that cannot be exercised, provide a detailed explanation of why it cannot."

A program which has been tested to this level will meet the following criteria:

1. Each statement in the program text will have been executed at least once.

2. Each decision in the program text will have been brought to each of its possible outcomes at least once, although not necessarily in every combination.

Meeting this restricted measure of testedness has been demonstrated to be capable of catching a high proportion of program mistakes. This simple measure of testing quality is automatable, relatively straightforward to measure, and indicative of general software quality. This last statement is true because of the following additional observations:

- Programs which cannot be tested to this simple measure contain, as a minimum, some superfluous program text.

- Programs which are comprehensively tested according to the testing measure can be assumed to harbor no "surprises" in future executions, at least insofar as such surprises would result from single-DD-path causes (as opposed to multiple-DD-path interactions).

The limitations of this first-level measure are apparent: there is no guarantee, even when the testing goal is achieved, that errors resulting from interactions between DD-paths (either within a single module or in different modules or even different components or subsystems) are discovered. To lay the foundation for future testing methodologies, it is valuable to describe some possible higher-order measures of program testedness:

Goal 2: Provide testcase sets which exercise all DD-paths and, in addition, exercise all pairs of DD-paths which can be executed in a single testcase.

Goal 3: Provide testcase sets which meet Goal 2, and in addition, exercise at least one sequence of DD-paths within each group of equivalent program flows (see Sec. 2.2).

Goal 4: Provide testcase sets which meet Goal 3 and, in addition, exercise all possible levels of iteration within the program.

.
.
.

Goal n: Provide testcase sets which exercise all possible program flows (in general not possible).

Additional research must be done, however, before any of these measures can be achieved systematically in a general methodology.

The testcases applied to a module should be designed so that each of them executes as many different DD-paths as possible. For other than the first testcase, each testcase should also be designed to reach as many as yet unexecuted DD-paths as possible; doing this assures that a small number of testcases will be necessary.

One of the objectives of the Automated Verification System (AVS) is to support the testing activity in the following ways:

1. By providing instrumented, but logically equivalent, copies of programs to record the DD-paths that are executed.
2. By analyzing the execution-time data produced by the instrumented programs.
3. By providing a report that identifies the DD-paths within a module which were not executed.

2.1.3 Testing Environment

The environment for testing a single module is diagrammed in Fig. 2.1. The instrumented version of the module's program text is considered to be executed in the presence of its supporting modules (if any), and the execution is shown as being controlled from a "Test Control Program". The Test Control Program may, in fact, be one of the invoking modules elsewhere in the software system, or it may be a specially constructed program.

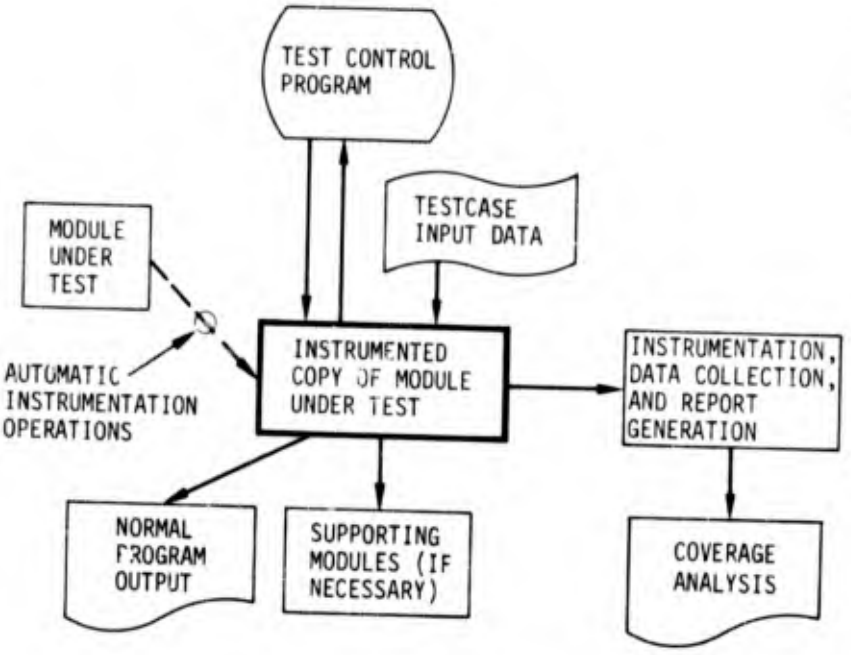


Figure 2.1. Testing Environment

During execution, data emitted by the instrumentation is accumulated and then analyzed in an activity offline to the normal program execution. The output of this activity, after each test execution, is a report on the amount of testing achieved (expressed in terms of percentages of DD-paths) in each test, and cumulatively in all tests.

A single test (unit test) consists of passing control to a module exactly once, and supplying the module with all the testcase input data which it uses during its natural operation. The testcase data may consist of a single value passed as an actual parameter, or it may consist of an entire file of information which is read from within the module. The exact data which makes up a testcase depends on the internal characteristics of a module, and on the needs of other modules which the module being tested may call upon. The set of variables within a module which must have values assigned prior to a test is called the input space of the module.

The input space for a module may be empty. That is, the module may have no input parameters, the module may not rely on communication via globally defined data, and the module may not read data directly from a file or indirectly via some other module. Such modules fall into two categories:

Memory Modules. A module may have purely local storage in which values generated within the module are remembered (stored) from execution to execution. Modules with memory can be tested by modifying the module, temporarily, so that it takes the values in its memory from the effect of an invocation of an auxiliary module. The input space of the module is then accessible (via the space of the auxiliary module) and testing proceeds normally.

One-Shot Modules. A module which has an empty input space and which does not have memory is termed a "one-shot" module because it can be executed in only one manner. A one-shot module is completely tested by a single invocation.

2.1.4 Phases of Single-Module Testing

Single-module testing can be considered as falling into four phases:

- Beginning Phase. "Natural" testcases are assumed to comprise the first testcases. The testcase data may arise from basic software system input data, or it may arise as the result of prior testing of an invoking module; in the second case, it is assumed that some mechanism has been employed for recording the data passed to the current module or, equivalently, it is assumed that the modules which invoke the current module are treated as part of the test control program (Fig. 2.1).

- Continuing Phase. This phase begins after the initial testcases have been executed and the coverage is analyzed. Either all DD-paths have been tested at least once, or some set of DD-paths has not been executed yet. If all DD-paths have been executed, or if the percentage executed is above some system-wide threshold setting, there is no more work to do. If there are untested DD-paths, the tester must:

1. Select the next DD-path towards which testing effort will be directed

2. Identify the data which will force this DD-path to be executed

- Concluding Phase. This phase of testing brings the DD-path coverage to 100%. The concluding phase of testing is likely to require somewhat more effort than the earlier phases.

- Retesting Phase. This situation occurs when the module has been modified (during software system maintenance, for example). An instrumented execution with the existing testcases will reveal the degree to which modification of the module has affected the testing level achieved previously. It is likely that only the continuing and concluding phases of testing will have to be repeated.

2.1.5 Practical Considerations

The number of testcases certainly need not exceed the number of DD-paths within the module, since each testcase could be designed to test a different DD-path. Because the testing goal does not require independent tests of each DD-path--only that they all be tested at least once--common sense suggests trying to exercise a fairly large number of previously untested DD-paths with each new testcase. Execution of many more DD-paths than a testcase was designed to execute is called collateral testing.

Thus, a secondary objective of the test data generation strategy is to provide high testing payoff. Although it is impossible to know whether this will always be achieved--it depends on the internal form of each module--a good testcase set for a single module typically covers at least 10% of the DD-paths in each test. Hence, the total number of tests should not exceed 10% of the number of DD-paths.

Achieving the 100%-exercise testing goal is substantially more difficult than stating it. Some procedures useful in deciding how best to reach the 100% testing goal are the following:

- The untested set of DD-paths can be ranked according to some uniform criterion (based on a DD-path complexity measure, for example). The "next DD-path testing target" is the DD-path among those as yet untested which has the greatest complexity.* Any ranking scheme should take collateral testing benefits into account if possible.

- Having selected an untested DD-path, there should be a systematic method to assist in constructing testcase data. This data must cause execution of the target DD-path and simultaneously maximize collateral testing.

The first capability can be supplied by analyses based on the testing data collection and reduction activity. How to achieve the second objective has been the subject of a great deal of investigation, resulting in the approaches detailed in the remainder of this section.

A natural question at this point is, "How can a minimum testcase set be found?" Obviously, at least one test is required for each module; and we have already noted that one does not need to have more tests than there are DD-paths. It seems reasonable that there is some "minimum" number of tests, and if possible the methodology should seek this minimum.

Unfortunately, with present knowledge there is no basis for algorithmic selection of a minimum testcase set. The reasons for this are:

1. The effectiveness of any one testcase is known only after it is executed--this permits collateral testing to occur.

2. The structure information on which testcase generation is based is not sufficient to permit identifying a true minimum testcase set. Testcase construction is indicated, but not guaranteed, using structural information alone; the success of a structurally

*Actually, one should choose to construct a testcase for the most complex DD-path sequence for which it is possible to construct a testcase. Obviously, there is a tradeoff between the ease of constructing a testcase and the return it gives (by collateral testing).

suggested testcase depends on the details of the program text along that path.

Finding a minimum testcase set is equivalent to the optimum cover problem, which is known to be unsolvable in general.*

2.2 ITERATION STRUCTURE

The iteration structure of a single module provides mechanisms for developing testcases and identifying testcase data which will cause a program execution to include a specific DD-path.

The material in this section is designed to clarify the concept of a module's iteration structure, and to give reasonably precise but understandable explanations of how to use the iteration structure to construct testcase data.

2.2.1 Definition of DD-Path

The iteration structure is a collection of sequences, or equivalence classes of sequences, of DD-paths. We now explain the precise meaning of a DD-path, and suggest some of the reasons for using this particular model in describing a module's internal control structure.

The control structure in a program is contained in the control statements within it. Program control actions are the result of the program's computational behavior interacting with the program's decision statements. Achieving the testing goal (Goal 1) guarantees that each of the decision statements has been exercised at least once to each of its possible outcomes, and that each statement expressing the program's computational behavior has been exercised at least once.

---

*Given a set S, a collection $\{S_i\}$ of subsets of S, and a weighting function which assigns a weight to each element of S, the optimum cover problem is to select subsets of $\{S_i\}$ such that (1) each element of S appears in at least one S_i, and (2) the weighting function is minimized. In testing, S corresponds to the set of DD-paths, and the S_i correspond to the testcases.

The definition of a DD-path is motivated by two desires:

- We want the testing goal to be reflected easily and clearly in the model we choose for the control structure, since this will simplify the analyses considerably.
- We want the model of the program's control structure to be as simple as possible, so that we have the smallest possible number of objects to deal with.

Not only does a DD-path meet these needs, but it is a primitive concept. A DD-path is the longest sequence of statements which have the property that when any one of them is executed, all of them are executed. Each such statement sequence is an immutably executable sequence of program text. In analyzing the structure of a module we need consider only the collection of DD-paths and nothing finer-grained.

Figure 2.2 diagrams an elementary program control flow. We have separated the constituents of each decision within the program text in the following way:

1. The computation of the value of a predicate (program decision element) is thought of as occurring before the program flow of control passes "through" the particular predicate.
2. The "middle" of the control/selection process has an assigned node number. The node number is merely a device for unambiguously associating a "place name" with each important point within the program text. The node number is necessary because the program text may contain statement forms for which any other numbering scheme--such as the "statement numbers"--would not be unique.
3. The value of the predicate is used to determine future program action. Because a predicate may have two possible outcomes (shown as the TRUE and FALSE outcomes in Fig. 2.2), there are two possible sequences of computation.*

*n-valued predicates, or a set of relational expressions, are used when a node has more than two outways.

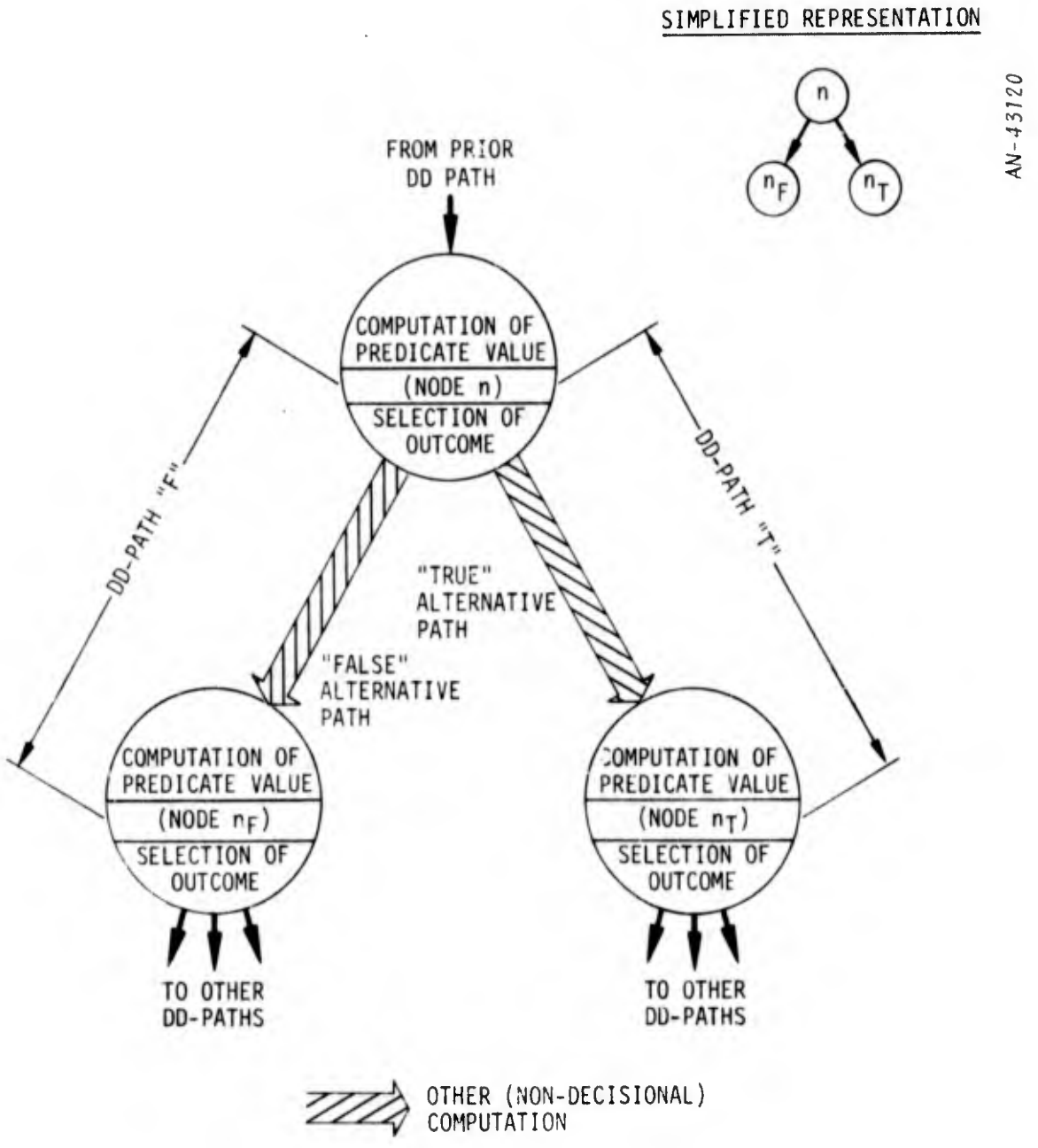


Figure 2.2. General Diagram of Program Flow

A predicate outcome, the set of statements executed as the result of that predicate outcome, and the computation of the subsequent predicate, is considered a DD-path. The TRUE and FALSE DD-paths for the single decision statement, assigned node n , are shown in Fig. 2.2. Also shown in the figure are some connecting DD-paths--the DD-paths which ensue as the result of having "taken" one or the other of the two DD-paths shown. DD-paths do not necessarily begin and end at different nodes; a self-loop results when the beginning and ending nodes of a DD-path are the same.

2.2.2 Special DD-Paths

Not every DD-path in a program text has a preceding DD-path, and not every DD-path in a program text has a succeeding DD-path. Two important special forms of DD-paths are treated in the same manner as above, but one of them involves use of a special predicate form.

- An entry DD-path is one which has no predecessors. It is the DD-path executed when the program is invoked. Because it is always executed at least once for every invocation of the module (that is, because there is no program-internal cause which can keep it from being executed if the program is invoked) the predicate for an entry DD-path is always TRUE. By convention, an entry DD-path originates at node number 1. When a program text has more than one entry name, then there is one DD-path originating at node 1 for each entry name, and in this case the entry DD-path predicate reflects the entry point selected.

- An exit DD-path is one which has no successors. There may be more than one exit DD-path within a program text if there is more than one statement within the program which has the logical effect of terminating the original invocation of the program. It is convenient to consider all exit DD-paths as having the same (possibly fictitious) terminal node number.

If a program has "alternate exits" (used for error processing, for example), then the categories of output state are distinguished by the (possibly fictitious) node number which has no successor DD-paths.

2.2.3 DD-Path Descriptions

The information sufficient to describe a DD-path, including those which are entry DD-paths or exit DD-paths, is the following:

- The initial node number. This identifies the program text statement from which the predicate arises.
- The final node number. This identifies the program text statement upon which the DD-path terminates.
- The predicate outcome, always associated with the text of the predicate residing within the statement which carries the (assigned) DD-path initial node number.
- The ordered sequence of statements executed as the result of the DD-path having been "selected" by the program. This set of statements will never involve any decision-making statement forms for, if they did, then additional DD-paths would be included. Each DD-path is unique within a program, and is always derivable unambiguously.

By convention, an exit DD-path ends on a node at which there is no decision with respect to the local text (i.e., the invocation is over with). One statement can belong to several DD-paths, as should be clear.

We often use a simpler notation for a DD-path, diagrammed in the insert of Fig. 2.2. This simplified representation shows the outways of a decision node and arrows which point to subsequent DD-paths' initial nodes.

To better understand the concept of a DD-path, some example constructions are given in Fig. 2.3. The reader should study these program texts and the pictorial DD-path representations before proceeding.

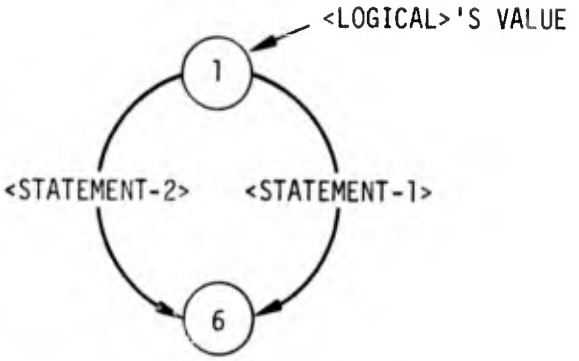
2.3 DD-PATH PARALLELISM

The total number of possible flows in a program which contains iteration is no smaller than countably infinite.* Even for programs which involve

*This statement is based on examining a program's control structure in the abstract; for all actual programs the total number of flows is finite.

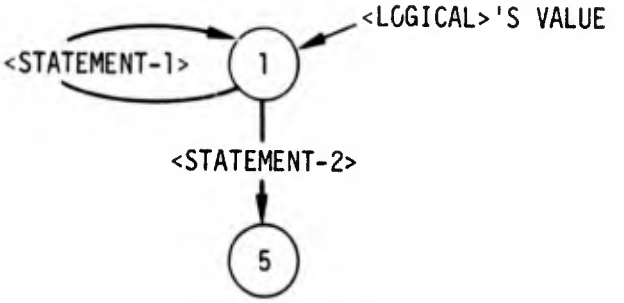
(a)

```
1    IF <LOGICAL>
2       <STATEMENT-1>
3    ELSE
4       <STATEMENT-2>
5    END
6    ...
```



(b)

```
1    WHILE <LOGICAL>
2       <STATEMENT-1>
3    END
4       <STATEMENT-2>
5    ...
```



(c)

```
1    CASE <EXPRESSION>
2    CASE <a>
3       <STATEMENT-a>
4    CASE <b>
5       <STATEMENT-b>
        .
        .
        .
n-2  CASE ELSE
n-1     <STATEMENT-ELSE>
 n   END
n+1  ...
```

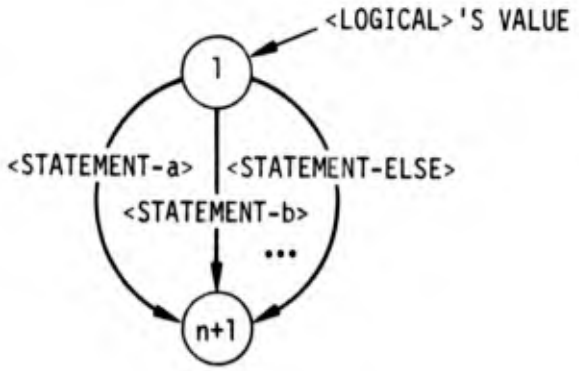


Figure 2.3. DD-Path Structures for Typical Control Statements

no iteration, the total number of possible legitimate combinations of DD-paths can be a very large number. Many programs contain parallel DD-paths, a important notion in dealing with the potentially severe problems of program combinatorics, described next.

2.3.1 Parallelism Identification

A typical DD-path pattern that often occurs within a program is shown in Fig. 2.4; pairs of DD-paths emanate successively from nodes n_1, n_2, n_3 ... . Disregarding the effect of the program statements which make up each of these DD-paths, there are eight possible different program flows, each touching at least three DD-paths. (Recall that the outgoing lines from each node represent alternative, or disjunctive, program flows.) Each of the three sets of DD-paths shown in Fig. 2.4 is a parallel DD-path set, since the program control flow is "parallel" between successive nodes.

When such a sequence of DD-paths occurs in the context of a large number of other DD-paths, it is advantageous to deal with some more compact representation of all eight possible combinations of program flow. Such a representation is diagrammed on the right of Fig. 2.4, where each pair of parallel DD-paths is considered as belonging to a DD-path class and only the sequential interconnection of the classes is considered subsequently. The number of actual program flows within a DD-path class is called the DD-path class cardinality.

For this simple case, an eight-to-one reduction in the complexity of the representation of the flow possibilities of the three DD-path classes results. In actual programs, this reduction in combinatoric complexity can be very important; in practice, a reduction of well over 2500 to 1 has been observed in some circumstances.

This kind of representational reduction does not relieve a program tester from having to exercise each DD-path at least once. In the example shown in Fig. 2.4, at least two testcases, and possibly as many as four testcases, will be required to exercise each DD-path at least once. (The maximum number of testcases required is less than the number of possible flows. The

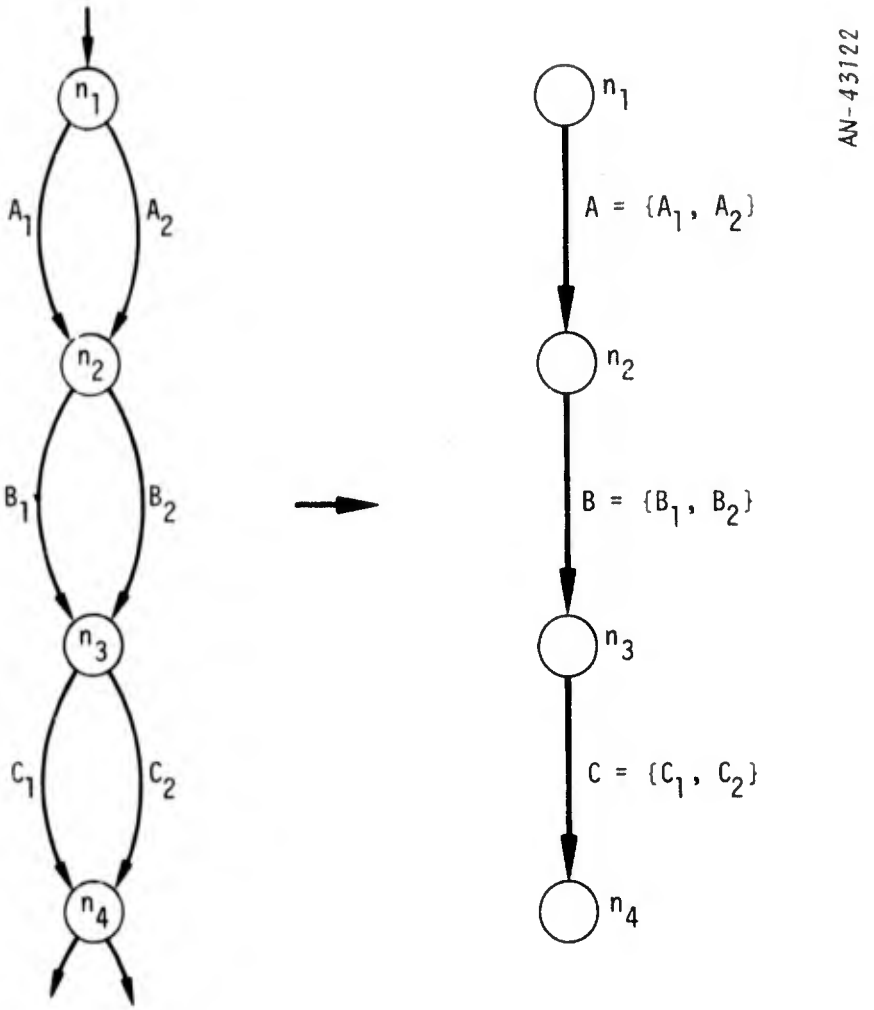


Figure 2.4. Simple Form of DD-Path Parallelism and DD-Path Class Representation

first testcase exercises at least three of the six DD-paths, leaving only three to be exercised. At most three other testcases will be required. The total number of possible program flows is always larger than the maximum number of testcases required to exercise each DD-path at least once.)

2.3.2 Parallelism Reduction

An AVS can identify DD-path structural parallelism automatically. Other forms of combinatorial parallelism could also be identified, but the present methodology does not attempt to deal with any advanced types of program parallelism.

2.4 LEVEL-i PATH CLASSES

After a program text is decomposed into the set of DD-paths, we can group them in ways specifically meaningful to the program testing problem. The general testing problem reduces to the issue of supporting a user in answering the following question:

> How can input data to a module be designed to force execution of a DD-path which heretofore has not been executed?

The concepts of level-i path class, and the associated level-i path class tree, provide the means to automate assistance in answering this important question. All of the methodology's tools for single-module testing are based on the representation of program control flow by the tree of level-i path classes.

2.4.1 Classification of Program Structures

Program structures can be classified as iterative, non-iterative, or mixed:

> Iterative programs have at least one DD-path that can execute more than once per invocation.
>
> Non-iterative programs are ones in which it is not possible to execute any DD-path more than once per invocation.
>
> Mixed programs contain parts which are iterative and parts which are non-iterative.

Figure 2.5a illustrates a non-iterative program structure. There is a single collection of DD-path classes. (They are DD-path classes rather than DD-paths, because each A_i may represent two or more parallel DD-paths.) We say that this program has one level-0 path class, which involves the DD-path classes $A_1, A_2, \ldots, A_6$. The number of possible program flows is a function of the cardinality of each of the DD-path classes A_i.

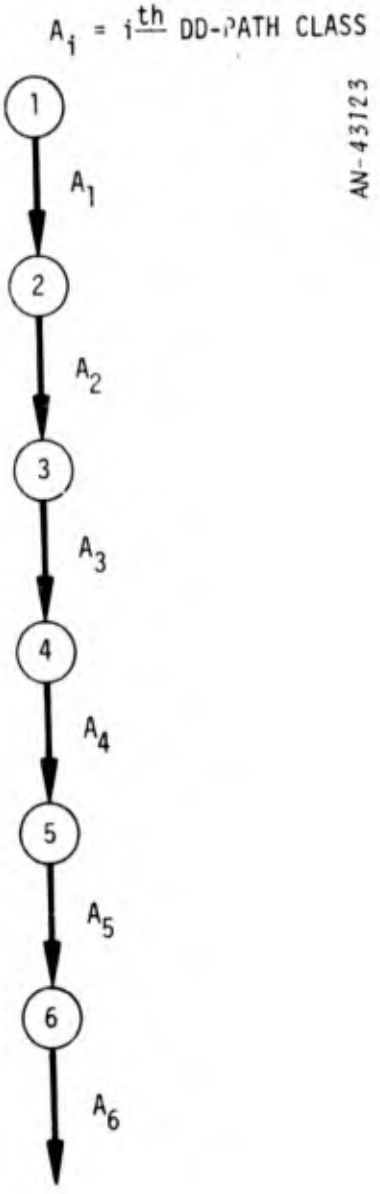
Figure 2.5b illustrates an iterative program structure: at least one DD-path class which causes reexecution of some program text predicate. One of the DD-paths in the DD-path class emanating from node n_I causes program flow to repeat the program predicate at node n_I. This program has two level-i path classes: a level-0 path class involving DD-paths $A_1, A_2, \ldots, A_4$, and a level-1 path class involving DD-path classes $B_1, B_2, \ldots, B_5$. The number of possible program flows is a function of the cardinalities of the two level-i path classes, and of the nature of the iteration represented in the corresponding text.

2.4.2 Testcase Generation Principles

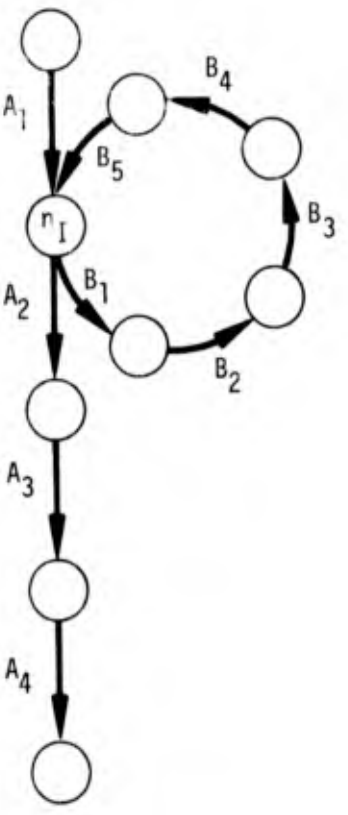
Testcase data generation relies partially on the ability of a tester to analyze program computation along a control path--in terms of the actual program statements--and to select testcase data which causes the required flow to occur. As we shall see in Sec. 4, there are tools and techniques which can simplify this analytic process.

Because program flow can always be classified as either iterative or non-iterative, we can now understand the motivation for dealing with program texts in terms of the "iteration structure". Analysis of a non-iterative flow pattern, such as that shown in Fig. 2.5a, is possible directly, merely by enumerating the associated program text. For an iterative flow, such as that shown in Fig. 2.5b, analysis of the program structure is performed in terms of the two "series" of DD-path classes (the A_i series and the B_i series).

In practice, iterative and non-iterative flow will be mixed, and DD-paths may be used in both senses (and possibly in a number of instances).



a. Non-Iterative



b. Iterative

Figure 2.5. Basic Program Structures

There may be a large number of structures of the forms shown in Figs. 2.5a and 2.5b.

2.4.3 Level-i Path Class Definition

The value of the concepts of non-iterative and iterative program flow is only realized if it is possible to identify these types of possible program behavior before comprehensive testing. Doing so carries with it one problem: without considering a program's computational and functional behavior, it is possible to identify only potential program flow classes. This means, in general, that the iteration structure will include program flows that are not actually possible. All possible forms of program flow are represented in the iteration structure, however.

A level-i path class is a sequence of DD-paths which occur on the ith level of the iteration structure of a program text. A level-0 path class represents non-iterative flow. A level-1 path class represents non-iterative flow "over" some level-0 non-iterative flow; the combination of a level-0 and a level-1 path class yields an iterative path. In general, a level-i path class describes non-iterative flow "over" some level-(i-1) non-iterative flow. Whenever two level-i path classes, at different levels, are related in this way, we refer to the higher-level path class as a descendant of the lower-level path class, and the lower-level path class as an ancestor of the higher-level path class.

2.4.4 Level-i Path Class Generation

Generating the level-i path classes, and identifying the ancestry relations between them, is fairly straightforward. The algorithms generate level-i path classes by selectively adding successor DD-paths.* At each stage of this process, a check is made to determine if the level-i path class repeats a decision node which resides on some other level-i path class; if so, a complete level-i path class has been found, and this fact, as well as information describing the ancestry of the level-i path, is recorded. This process

*The detailed algorithms are described in Ref. 11; we have not included them here because they do not contribute directly to understanding the testing methodology.

terminates when no new level-i path classes are "started" and when all of the DD-paths have been employed at least once.

Equivalent level-i path classes are identified and pruned from the tree of level-i path classes. That is, a single level-i path class may have more than one ancestor; the tree need not show the level-i path class again for each ancestor.

2.4.5 Level-i Path Class Tree

The collection of level-i path classes forms a tree structure rooted at the invocation point(s) of the program. A typical level-i path class tree is of the form shown in Fig. 2.6. The "root" of the tree is shown as L, the level-0 path classes are labeled $L_{0,1}, L_{0,2}, \cdots, L_{0,n_0}$. The higher-level path classes are labeled similarly.

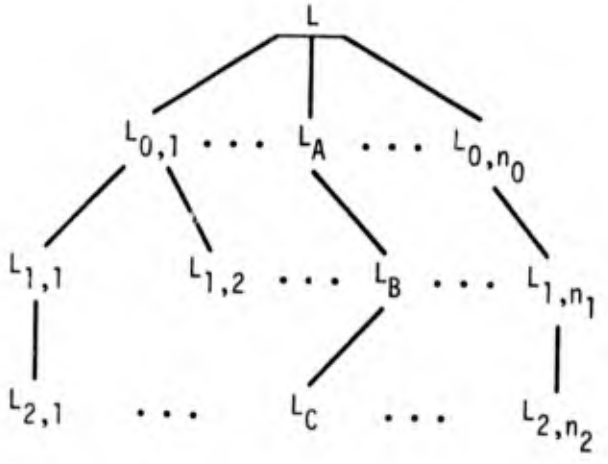


Figure 2.6. Tree of Level-i Path Classes

Not every $L_{i,j}$ need be distinct in the level-i path class tree. Each level-i path class may appear many times; the total number of occurrences of each level-i path class is an important measure of the complexity of a program's iteration structure.

Each DD-path belongs to one or more level-i path classes, possibly at different levels. Hence, the level-i path class tree also describes the ways in which individual DD-paths can be composed into actual execution sequences. Every DD-path that can be executed belongs to at least one level-i path, so that the level-i path class tree "covers" the set of DD-paths.

2.4.6 Use of Level-i Path Class Tree

Testing can be based entirely on partial traversals of the level-i path class tree. For example, one such traversal,

$$L \rightarrow L_A \rightarrow L_B \rightarrow L_C$$

describes a particular form of program iteration, since L_A describes a level-0 flow, which is modified by the level-1 flow L_B, which is in turn modified by the level-2 flow L_C. Each possible class of iteration (but not each possible specific form of iteration) is represented by the set of tree-traversals similar to the one given above.

Because each DD-path is represented in at least one level-i path class within the level-i path class tree, a set of testcases that traverses all the distinct level-i path classes of the tree is guaranteed to execute each DD-path at least once. It is therefore natural to concentrate on the terminal branch level-i path classes, that is, those which have no descendants. A set of testcases which touches each terminal branch of the tree will also have executed each DD-path at least once. In practice, this method for generating testcases is not effective, since a large number of program flows may be represented by each of the terminal branch path classes.

Each program invocation traces a particular program flow pattern. This pattern is dependent on the structure of the program and the way it employs

data in its input space. Regardless of the specifics, the program flow will be represented by some sub-tree of the level-i path class tree. Although it is possible to use the level-i path class tree to enumerate all the possible program flows, this is generally absurd because of the large numbers of traversals involved.

The program tester turns these observations to his advantage, since his objective is only to find ways to test particular parts of a program. The question is "How can the program be made to execute a particular DD-path?" For each DD-path there are many instances of its use within the level-i path class tree; this happens because a DD-path can belong to more than one level-i path class. However, for each DD-path there is a finite sub-tree which represents all possible ways to involve that particular DD-path. This sub-tree, and mechanisms for dealing with it, form the structural basis for generating testcases.

2.5 ITERATION STRUCTURE TOOLS

The iteration structure of a program text, expressed as the collection of level-i path classes, is the principal basis for the single-module testing strategy. The level-i path classes represent all possible patterns of non-iterative program flow, as well as the relations between those patterns. Because the information used to generate the level-i path classes did not take into account anything other than the explicit control structure of a program text, this set of flow patterns actually includes many structurally legal but computationally illegal flows. Including potentially superfluous flow information does not create a problem, however, because the main use of the iteration structure is for identifying ways to make particular program flows occur--rather than attempting to make all legal flows occur. Describing how to take advantage of the information represented in the set of level-i path classes is the main objective of this section.

2.5.1 Realistic Testing Situations

Testing of a single module hardly ever occurs in total ignorance of the program's general behavior. We postulated in Sec. 2.1 that there would be some "initial testcase data" that, when executed, would result in at least

partial coverage of the set of DD-paths. The continuing and concluding phases of testing can be focused on the following objective:

> Continuing Testing Objective: Given a partially exercised single module, devise additional testcases which increase the testing coverage.

Typically some number of DD-paths in a module will not yet have been exercised. The "Continuing Testing Objective" translates into a desire to minimize this number; or correspondingly, maximize the fraction of DD-paths which have been exercised. Repetition of this process will either result in full exercise of every DD-path in the program, or lead to explanations of the reasons why certain of the DD-paths cannot be tested.

Two questions arise in dealing with the techniques needed to achieve comprehensive module testing:

1. Of the unexercised DD-paths, towards which one should the available testing effort be targeted?
2. Once the target DD-path is selected, what methods can be used to assist in developing testcase data which exercises that DD-path?

Some possible answers to the first question are discussed in Sec. 2.6. The remainder of this section is devoted to describing two mechanisms for dealing with the second question.

2.5.2 Reach Sequences

We assume that some DD-path has been selected as the target of the testing effort. The particular technique used to assist a tester in constructing testcase data which exercises that DD-path depends on the role that DD-path has within the iteration structure of the module. We will illustrate the concepts with a simple example.

Figure 2.7 shows a typical multi-level iteration structure. There are three level-i path classes, with the dependence shown in the figure. The individual DD-paths classes are the following:

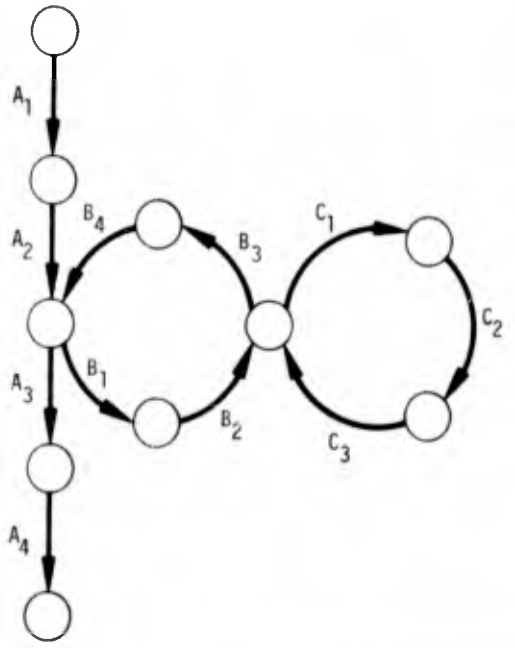


Figure 2.7. Multi-Level Iteration Structure

Level-0 path class: A_1, A_2, A_3, A_4

Level-1 path class: B_1, B_2, B_3, B_4

Level-2 path class: C_1, C_2, C_3

Testcase design is dependent on which DD-path class is as yet untested. There are eleven DD-path classes shown, but we will examine only three of them:

1. If DD-path class A_2 is untested, then we need to analyze the information along DD-path class A_1 to determine how to force execution of the DD-paths in class A_2. (The fact that DD-path class A_2 is the one not yet executed indicates that this program's iteration structure has not been exercised at all.)

2. If DD-path class B_2 is untested, then we need to analyze the computations performed along the path $A_1 \rightarrow A_2 \rightarrow B_1$. (The fact that this particular DD-path class has not been tested indicates that the "B" iteration of the program has not yet been exercised.)

3. If DD-path class C_2 has not been executed, then we need to analyze the computations performed along the path $A_1 \rightarrow B_1 \rightarrow B_2 \rightarrow C_1$. (The fact that this particular DD-path has not been tested shows that we have never entered the program's "C" iteration.)

The sequences of DD-path classes just described are called reach sequences relative to a particular DD-path in class A_2, B_2, or C_2. In practice there may be many different reach sequences relative to a particular DD-path; they will involve, in general, all of the level-i path tree traversals it is possible to make which arrive at a level-i path class which contains the specified DD-path.

2.5.3 Analyzing Reach Sequences

Any particular reach sequence describes an important form of information about the program which gave rise to it, and it is a basis for building a testcase which "reaches" the specified unexercised DD-path. A reach sequence represents a kind of derived level-i path class, since it is composed of an ordered sequence of DD-path classes. As for a level-i path, a particular reach sequence may represent a large number of different logical flows--depending on the cardinalities of the DD-path classes along it. When this number is large--as it typically is--it is better to analyze the indicated flow of the reach sequence as an aggregate.

For testcase assistance, we need to consider the program statements that correspond to the sequence of DD-path classes which make up the reach sequence. Assuming that a DD-path can be executed, then there will be at least one logically possible flow present in the set of reach sequences for it.* To construct a testcase which "reaches" the untested DD-path, the program tester examines the program statements which correspond to this set of structures, attempting to construct input-space variable settings which make at least one flow occur.

*This is not precisely true, because of the effect of cycles (Sec. 2.5.4).

It is not necessary to consider every flow possible, since the first legitimate one will do. Furthermore, it is not necessary to consider what the program does after it has reached the target DD-path, since that is unimportant in terms of generating a testcase.* The analysis performed must explicitly consider the actions of program statements that correspond to the reach sequence. These actions will determine whether a testcase within the class of flows represented by a particular reach sequence is logically possible.

As suggested earlier, in real testing situations there are many possible reach sequences for a particular DD-path. Each of these is examined in turn as a possible indicator for legitimate testcase data. Experience indicates, however, that only a few reach sequences must be analyzed in any great detail; the reason seems to lie in the practicalities of computer programs and the high degree of interdependence between parts of actual computer programs. In other words, one can expect to find legitimate testcase data for a particular untested DD-path by analyzing only one or two reach sequences in detail.

The concluding phase of testing may require somewhat more detailed analysis of the set of reach sequences for the yet untested DD-paths. In addition, it may be necessary to deal directly with the program's cycles (described in the next section) in order to achieve full DD-path coverage.

The use of reach sequences derived from the iteration structure is highly effective in dealing with one of the primary issues of comprehensive program testcase generation. For a man-machine interactive methodology, this concept is an important and powerful analysis tool.

2.5.4 Cycle Sequences

The cycles of a program contain information about the specifics of the program's iteration structure. When analysis of a simple non-iterative reach sequence does not yield testcase data which achieves coverage of some

*Here, we assume that the program terminates; if it does not, then the testcase will have identified the causes of an internal "infinite loop", which can then be eliminated.

particular DD-path, the program tester may have to perform his analyses directly in terms of the cycles along the path. This problem can be illustrated in terms of the iteration structure shown in Fig. 2.7. Suppose that it is DD-path class C_1 which has not yet been tested, and analyzing all the reach sequences which led to that DD-path class has not led to construction of a testcase. One possible reason is because the required logical flow is of the form:

$$A_1 \rightarrow A_2 \rightarrow (B_1 \rightarrow B_2 \rightarrow B_3 \rightarrow B_4 \rightarrow)^k \rightarrow B_1 \rightarrow B_2 \rightarrow C_1$$

In this flow expression we have shown part of the sequence "raised to the kth power." The meaning of this notation is that that part of the sequence must be repeated k times before the program can actually flow into DD-path class C_1, as we want it to. The value of k is dependent on the computations performed in the cycle itself; that is, within the DD-path classes B_1, B_2, B_3, or B_4. We also know that the number of times around this simple cycle is a function only of the statements in this subset of the program.

Unfortunately, there are no general rules for dealing with the program computations performed in a cycle and the way they force exit from the cycle (via DD-path C_1 or DD-path A_3). Certain features of iteration cycles are often worth looking for:

- The cycle's "Iteration Variable"--the variable which counts the number of times around the cycle, or controls continuation of the cycle. In the former case, this variable corresponds to the counter-superscript, n. In many cases, however, the iteration variable is not a simple function of the computations performed in the cycle.
- The cycle's "Invariant"--the general logical property which is true independent of the number of times the cycle is traversed. It is often very difficult to discover the invariant "function" of a cycle. In some cases, when the cycle is structurally possible but not logically realizable, it may not exist at all.

A program tester should keep in mind that a reach sequence is only an approximation of a program's more general iteration structure. It may be necessary to analyze the cycles around which a reach sequence is built in order to arrive at effective testcase data in every situation.

2.6 TREATMENT OF NON-ITERATIVE FLOW

The concept of the iteration structure, and the notions of reach and cycle sequences, provide the unifying basis for our methodology. Even in the presence of program iteration, testing problems are reduced to analysis of non-iterative program flow patterns. For a program with mixed flow structure, the non-iterative flows can be separated from the iterative flows. In turn, iterative flows can be considered as being composed of combinations of non-iterative flow. Reach and cycle sequences are intrinsically non-iterative.

The process of selecting appropriate non-iterative flows to analyze is the subject of the next section. We deal here with some details of analyzing any particular non-iterative flow.

2.6.1 The Combinatorics Problem

A non-iterative flow can be primitive to a program (that is, arise from a level-0 path class), or it can be one derived from a reach or a cycle sequence. At least one and possibly a very large number of actual program flows will be represented. This results from parallelism between DD-paths, and the fact that all of the level-i path classes, and, correspondingly, the reach and cycle sequences, are expressed in terms of DD-path classes. The total number of actual flows represented by a non-iterative flow is called the cardinality of the flow. The cardinality of a flow is always at least one, since there is always at least one connected sequence of DD-paths present. For flows in real programs, the cardinality may be very large. A sequence with, say, 15 two-member DD-path classes in sequence would have $2^{15} = 32,768$ different flows.

Not all of these flows may be logically possible. DD-paths model the capability of a program's control structure, based on the values that predicates within it can have. The actions performed along prior DD-paths determine whether a particular decision is taken one way or the other. One DD-path

in a chain of parallel DD-path classes may constrain the actions which can subsequently occur. DD-paths are incompatible when a variable that must have a particular value in the predicate of one DD-path is always set to a different value by a prior DD-path, and is not changed on any of the intervening DD-paths. DD-path incompatibility may be very difficult to identify, particularly when the DD-paths themselves are long and the number of DD-path classes to handle is large.

2.6.2 Practical Program Flows

In real situations a program tester may have to deal with flows with cardinalities in the range 10^2 to 10^4, or possibly even larger. How can flows that complex be treated without requiring impossible amounts of analyst time?

The answer to this question comes from an observation about the program tester's objective. In practice, the program tester is not interested in considering all possible program flows, but instead wants to construct test datasets which make one flow occur, namely, one that tests some previously untested DD-path. In other words, so long as one logically realizable program flow can be found, the cardinality of the flow class from which it was derived makes little difference. Technically, what this means is that the program tester needs to consider program flow patterns only in aggregate to accomplish a specific testing objective, rather than having to consider every combination represented by the flow.

To see how this observation applies, consider the flow that was shown in Fig. 2.5a, in which there are six successive DD-path classes. The analysis of the corresponding program text should proceed in the following manner:

1. The particular untested DD-path from the set represented by A_6 should be identified. The predicate for this DD-path resides at node 6 of this sequence; this predicate must have a particular value for the chosen DD-path to be executed.

2. The program computations along the DD-paths which belong to the set A_5 are analyzed for their relationship with the chosen DD-path. This analysis determines a set of consistent program

values, or a set of constraints, at node 5 which permits the predicate at node 6 to take on the required value.

3. This process is repeated, possibly using some simple calculations to keep track of the requirements, until the statements on the DD-paths within the set A_1 have been analyzed. If at any point an inconsistency between program values is found, then the program tester should discard some of the information already generated and repeat some of the analyses.

4. The outcome is a set of requirements for the data which is used along this particular path. A testcase dataset for this particular flow consists of data values which satisfy these constraints.

The constraints arrived at by this process will consist, in general, of a set of nonlinear inequalities involving program variables and the particular set of program predicates involved in the flow. Mathematical solutions for nonlinear inequalities do not necessarily exist in closed form; in fact, the techniques for solving such a set of inequalities are not well understood. As a result, automating the solution of the set of inequalities is beyond the present state of the art.

On the other hand, experience suggests that hand solution of the set of constraints is relatively straightforward, although it may require some skill on the part of the program tester as well as some knowledge of what the program is supposed to be doing. The AVS which supports the methodology can provide some forms of assistance to the program tester; see Sec. 4.

From a complexity point of view, however, we see that the specific and limited objective of a program tester--to test a single untested DD-path by whatever means are possible--is conceptually simpler by this approach than by dealing with all combinations of program flow individually. In effect, the program tester must deal only with linear combinations of DD-paths and the program statements which correspond to them; an automated approach would have to deal with exponentially increasing numbers of program flows. The reduction can be significant; for a non-iterative flow consisting of fifteen

two-member DD-path classes, only 105 cases must be analyzed to arrive at a testcase for a DD-path in the fifteenth DD-path class.* By comparison, there are 32,768 different (potential) program flows.

A second saving occurs when the DD-paths are parallel as the result of being alternative outcomes of the same predicate (e.g., in a simple IF statement). The analysis steps outlined above need contend only with the effect of the conditioned statement on the other DD-paths, and need not consider the value of the predicate at all. This permits a further reduction in the complexity of analyzing a high-cardinality flow.

2.7 SUMMARY: A UNIFIED METHODOLOGY

The ingredients of the methodology for testing single modules have been described in the preceding sections. The analytic tools needed include the following:

- Division of a program text into immutably executable pieces, called DD-paths, which arise from each program predicate's outcomes.
- Analysis of the relations of these DD-paths and identification of the iteration structure. The iteration structure captures all possible forms of interconnection of iterative and non-iterative flow in a common representation, and also captures the dependencies between all of the flows (the level-i path class tree).
- Generation of reach and cycle sequences to linearize the analysis of complex program iterations.
- Reduction of the combinatoric difficulty of dealing with complex program structures by changing program testing into, primarily, a problem of analyzing separate non-iterative program flow classes.

These techniques can be employed to develop comprehensive testcase data for single modules.

---

*The 15th class must be analyzed against its 14 predecessors; the 14th against its 13 predecessors; and so on. $14 + 13 + \ldots + 1 = 105$.

2.7.1 Structure of the Methodology

As we have already mentioned, the testing process for a single module can be divided into a beginning phase, a continuing phase, and a concluding phase. In all three phases, there is a single objective: to construct testcases which cause execution of as yet unexecuted DD-paths within the program. Testing is over when all DD-paths have been exercised, or when those which have not been exercised are shown by the program tester to be logically unexecutable.

This process is diagrammed in Fig. 2.8. The methodology begins by executing whatever testcases for the module already exist; this initial test, performed with the assistance of the instrumentation facility of the AVS (as described in Sec. 4), results in a Testing Coverage Report. This report identifies the DD-paths which have not been exercised. If there are none, then testing is finished.

The next step is to choose a likely DD-path upon which to concentrate the testing. After this choice is made (see below), the tools already described are employed, as appropriate, to assist in generating testcases which increase the achieved DD-path exercise percentage. These additional testcases are added to the previously generated ones, additional test executions are made, and the AVS facilities are used to provide the updated Testing Coverage Report.

2.7.2 DD-Path Choice Mechanisms

The effectiveness of this scheme for program testing depends to some extent on the mechanism used to select the next target DD-path for testcase generation. This testing target should be one of the unexecuted DD-paths; when there is more than one untested DD-path to choose from, making the choice among them can affect the collateral testing, and thereby influence the efficiency of the testcase set.

The DD-path selection criteria used should attempt to maximize collateral testing. On the other hand, maximum collateral testing coverage may pose a very difficult testcase data generation problem. The selection function

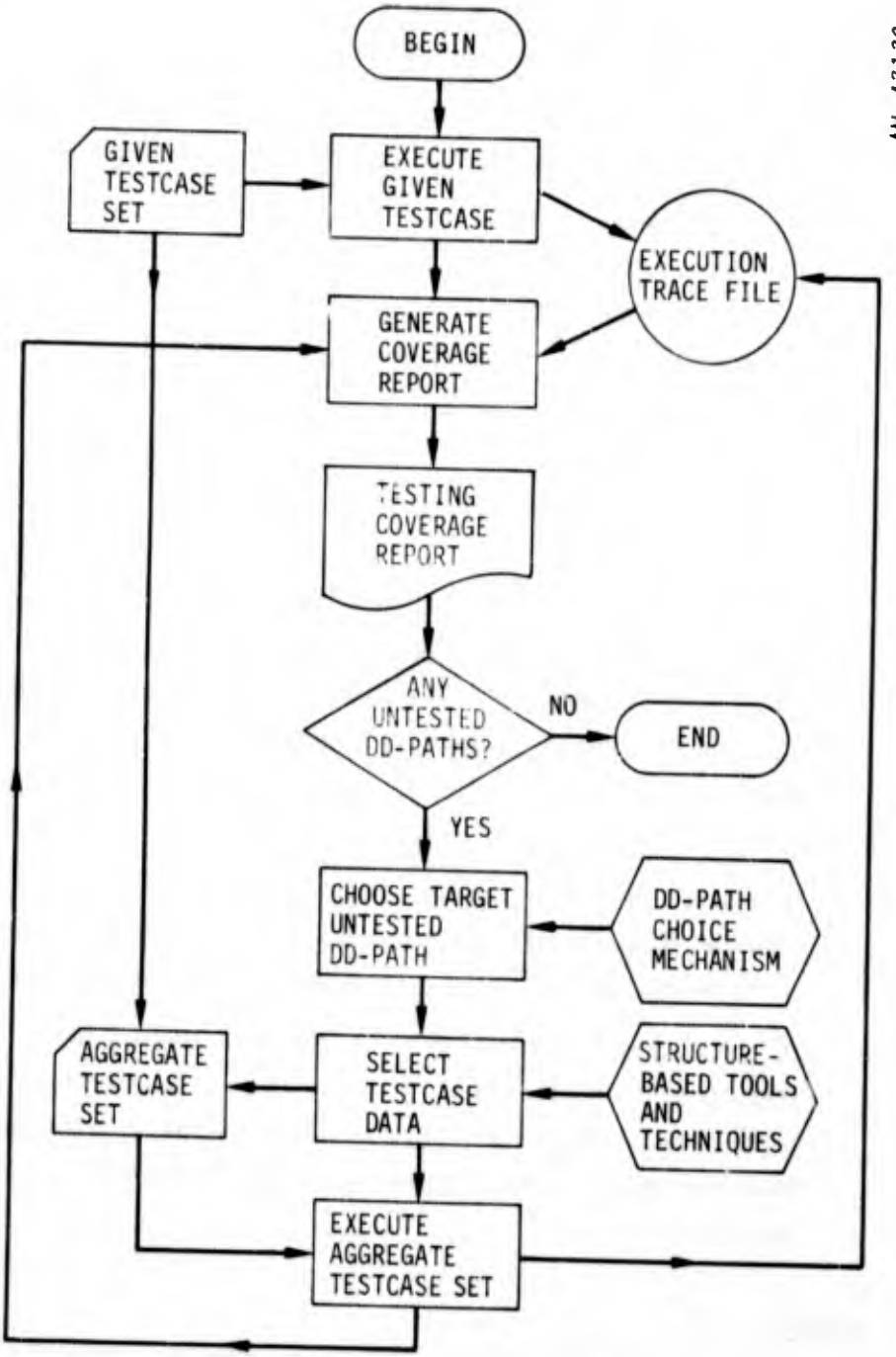


Figure 2.8. Methodology for Single-Module Testing

actually used should depend on the nature of the program being analyzed. The following guidelines may be of value:

1. Choose a DD-path which is on a terminal-branch level-i path class; that is, one that resides on the highest-possible iteration level.
   a. This assures there will be a high degree of collateral testing, since after the target DD-path is executed the program must still finish executing and, in the process, may hit a large number of other untested DD-paths.
   b. If such a DD-path has not yet been executed, and the program tester intends to satisfy the basic testing goal, then it will have to be dealt with at some time anyway. Better sooner than later.
2. Choose a DD-path which is at the end of a fairly long reach sequence. The reasons for this are similar to (1.a), but involve an additional observation: the more complex a set of logical conditions dealt with in generating a testcase, the more likely that the resulting testcase dataset will resemble data which corresponds to the functional nature of the program being analyzed.
3. If a prior testcase carries the program execution near one of the untested DD-paths, it may be more economical to determine how that testcase can be modified to exercise the untested DD-path.
4. If the analyses required for a particular DD-path selection are difficult, then attempt to choose a DD-path which lies along the lower-level portions of its reach sequence(s). Doing this simplifies the analysis problem, but may still achieve a high degree of collateral testing.

2.7.3 Summary

The single-module testing methodology uses structurally-based analytic tools which are selectively applied by the program tester. The techniques are specifically designed so that they can be automated in an AVS. The analysis relies on decomposing the control structure of individual programs into DD-paths, and on identifying the iteration structure for the resulting se of

DD-paths. The iteration structure forms the base for (1) devising program flows which are likely to test previously untested DD-paths, and (2) helping to select testing target DD-paths which will generate substantial collateral testing coverage.

The methodology places single-module testing on a rational basis which is insensitive to the size of the program analyzed, and which is inherently automatable. While full automation in program testing is impossible at the present state of the art, this semi-automatic methodology systematizes program tester effort in a way which minimizes the difficulty of individual program analyses, and is likely to minimize overall program testing effort.

SECTION 3

SYSTEMATIC SOFTWARE SYSTEM TESTING

The previous section discussed the testing methodology for a single module--that is, a "subroutine" or "procedure" embedded and invoked in some manner in a large-scale software system. This section describes the methodology for extending single-module testing coverage to all modules in a large software system.

Large-scale software systems vary enormously in the way they are organized and used. For example, a software system intended for real-time command and control applications may consist of relatively concise programs organized in a rigid manner. Individual real-time program modules may have many separate functions to perform, and invocation and selection of these functions may take place in a complex control program. On the other end of the spectrum, a software system for generalized database management may consist of somewhat expansive and loosely organized programs. Such a system may have a large number of modules each performing a specific function, and combined with relatively simple control programs.

3.1 TYPICAL SOFTWARE SYSTEM ORGANIZATION

Partitioning a software system into subsystems, components, and modules (as was done in Sec. 1) provides the structural basis for discussing techniques for system-wide software testing. Figure 1.5 showed a software system composed of elements of this kind, with each element represented by a pyramid, the top of which may be thought of as the point at which the element is invoked by the remainder of the system.

3.1.1 Individual Modules

Describing a software system in these terms makes treating it equivalent to dealing with the interelement invocations.* The dynamic organization is important because it is a form of structural information which can be found automatically (by an AVS, see Sec. 4). Depending on the language used to

*If there are no interelement invocations, then the software system is a single module, and the techniques of Sec. 2 apply.

express the software system, the individually invokable modules or elements* may be defined explicitly or implicitly. In typical procedure-oriented languages, for example, the elements are separately compilable and invokable. For other programming languages, module identification may be more difficult, since the boundaries at which invocations occur may not be so plain.** We assume, in what follows, that interelement invocation boundaries have already been identified.

An invocation point is a place within a software system at which control is passed to an invoked module. Invocation may include more than merely passing control; for example, the invoking and the invoked elements may share data by means of some data-sharing mechanism within the programming language. Data can be shared between individual statements, between individual modules, and between other software system elements. What characterizes an invocation (in the presence of shared data) is the fact that there is some explicit indication that control has been given to some other element, and that the control will be returned when that element has finished its work. Invokable elements can be invoked in this way many times.

3.1.2 Components and Subsystems

Groups of modules exist for functional purposes; the computational facilities provided by several modules may be combined into more powerful functional structures. The dividing line between a "component" and a "subsystem" depends on the structure of the software system being analyzed. For simplicity, we can think of a component as a set of modules which is invoked, as a component, at some point within a software subsystem. Similarly, a subsystem is a collection of modules which control invocation of a set of components; a system consists of a series of invocations of subsystems. All, some, or none of these organizational classifications may exist in any particular software system.

---

*Subsequently, when we use the term "element" we refer to a module, component, or subsystem. An element consists of some subset of the modules which make up a software system.

**Appendix A describes the techniques which apply to JOVIAL/J3.

3.1.3 Invocation Structure

The invocation structure of a software system is a description of the way in which individual software elements depend on one another. This description takes the form of a tree-like hierarchy; the "top" module (the one not invoked by any other element) is connected to the modules it invokes, which are connected to the modules they invoke, and so forth. Figure 3.1 shows a typical invocation tree describing the dynamic organization of a software system. The tree begins at the highest level in module $P_{1,1}$; this module invokes a set of other modules $P_{2,1}, \ldots, P_{2,n_2}$. These in turn invoke other modules at successively lower levels within the invocation hierarchy. $P_{1,1}$, the topmost module, is not invoked by any other module.

The boundaries identifying software components and subsystems have not been identified because:

- It is not necessary for all of the $P_{i,j}$ shown in Fig. 3.1 to be distinct; for example, some module may be invoked at two different levels within the tree.

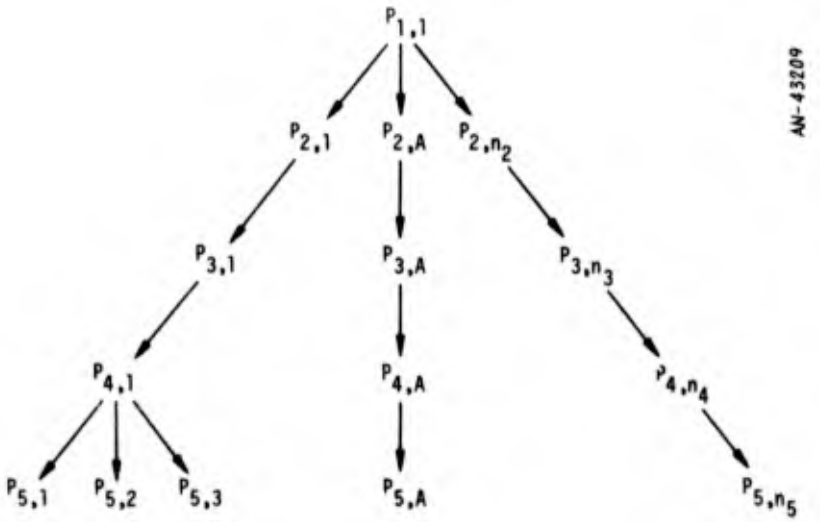


Figure 3.1. Tree Representation of Invocation Hierarchy

- The assignment of functions to components and subsystems may not necessarily have been done in the same way in which the invocation tree is organized.

We can illustrate these points in terms of the typical invocation chain shown in Fig. 3.1 as $P_{1,1} \rightarrow P_{2,A} \rightarrow P_{3,A} \rightarrow P_{4,A} \rightarrow P_{5,A}$; we refer to this invocation chain as the A invocation chain. All of the $P_{i,A}$ are distinct, because of the manner in which we have defined the invocation tree.* The individual modules in the A invocation chain may reside, successively, in the "system", the first-level "subsystem", the second-level "subsystem", a particular component, and so forth. The system need not have such an elegant internal organization, however. What is important is that the invocation chain is unique within the invocation tree, regardless of the software system's organization.

In most large-scale software systems, the invocation dependence between modules will tend to be localized to some extent. Finding such localizations of dependence will be important in minimizing the difficulty of achieving comprehensive software system testing; we expand on this in the sections which follow. An example of localization is the set of invocations made by the $P_{4,1}$ module in the tree shown in Fig. 3.1. This module (and the three other modules it invokes) can be thought of as a component of the software system. A software system with a complex invocation tree will not necessarily be dense with interconnecting links, but instead will probably have many natural clusters of dependence.

3.2 TESTING OBJECTIVES

The testing goal for a single module was stated in Sec. 2; this testing goal was chosen as a reasonable balance between attempting to exercise all of the paths possible within a single module, and making a few disorderly stabs in the dark. If met, the testing goal assures that every statement in a module is exercised at least once, and that every decision statement in a module is exercised at least once to each of its possible outcomes, although not

*As before, we assume that the software system is non-recursive and contains no coroutines. Thus, circular invocation chains will not occur.

necessarily in every possible combination. This testing goal is relatively easy to achieve, primarily because there are good techniques for constructing test data. It provides strong assurances to the program tester that the code is comprehensively exercised and that all (simple) unintended behaviors have the opportunity to reveal themselves.

The testing objective applied to single modules is equally valuable when applied to testing an entire software system. It would be desirable to certify mathematical correctness of the software system, but this is impossible at the present state of the art.[6] The lesser testing objective--which assures comprehensive exercise of the software system--is the most practical one possible with present technology. The overall system-testing goal is the following:

> Goal S1: Provide a usefully small set of testcases for a software system which exercises as many DD-paths as possible, out of the aggregate set of DD-paths in all modules of the system. The coverage measure is the percentage of DD-paths exercised.

This testing goal is a straightforward extension of Goal 1 (Sec. 2) to encompass all of the DD-paths in the software system.

Although this basic testing goal is a desirable one from the whole-system testing standpoint, it may not be completely achievable in practice. (Some reasons for this are discussed in Sec. 3.4.) Moreover, the basic single-module testing goal may be too granular to serve as a general means to guide the system testing process. That is, it would probably be uneconomical to allocate system testing effort for a complete software system in the same way it was allocated when testing a single module--that is, by attempting to test a module's "most complex" untested DD-path. Adhering blindly to a simple testing measure may not take advantage of collateral testing achievable with a different measure, particularly when the invocation structure is complex and varied.

Two alternative system testedness measures have the property that they converge to the overall testing effort's goal of 100% system-wide DD-path coverage.

Goal S2: The system testedness value is expressed in terms of the percentage of testing achieved in each single module. The system is considered to have been tested only to the level attained by the least tested module.

Goal S3: The system testedness value is expressed in terms of the percentage of testing achieved in each element (assuming that the element boundaries are known). The system is considered to have been tested only to the level attained by the least tested element.

Goal S2 will assure that no module escapes some testing exercise until the concluding phase of system testing. This would not be possible with the S1 measure of system coverage: a relatively large number of DD-paths could have been exercised throughout the software system, but some module may have no DD-paths exercised. Goal S3 extends this concept to system elements (components and subsystems) in the obvious way. A testing strategy based on software system element coverage assures that the concluding phase of system testing begins with some minimum per-element level of testing.

These measures are still based only on the percentage of DD-path executions attained in a set of system tests. A more complicated (and possibly more effective) measure of the degree of testing of a whole software system could be based on either of the following:

Goal S4: If there is some measure of single-module (static) complexity, then the testing coverage measure for each module can be multiplied by the complexity to yield a whole-system coverage-complexity product. The coverage measure is then the achieved coverage-complexity product, as a percentage of the total complexity.

Goal S5: A weighting scheme, possibly based on the degree of interaction between components and subsystems, could be used to emphasize testing the most important parts of the software system. The coverage measure is the weighted percentage of coverage achieved.

For the purposes of this methodology, we assume that only the simple S1 or S2 measure is used. Extension of the methodology to other measures is beyond the scope of this report.

3.3 TESTING PHASES AND STRATEGIES

As with testing of single modules, testing of a software system proceeds in four distinct phases: beginning, continuing, and concluding testing and retesting. The system testing effort, however, can be organized according to two fundamentally distinct strategies: (1) bottom-up system testing, and (2) top-down system testing.

Bottom-Up Testing: This testing strategy attempts to provide comprehensive system testing coverage by building testcases from the bottom of the system invocation hierarchy first, and extending these testcases upward during the continuing and concluding testing phases. Bottom-up testing may require the use of special testing environments (see below), but is likely to achieve the best overall testing coverage.

Top-Down Testing: This testing strategy deals with an entire software system first, and, after subsystem (or component) testedness is measured, proceeds downward through the software system's invocation structure. Testcase data is added only at the topmost level and, as a result, a set of system-wide testcases are developed directly. A possible disadvantage of this strategy is that certain lower-level elements of the software system may not be testable because system communication paths to them are blocked (see Sec. 3.4).

The optimum system testing strategy for a particular system may combine the two strategies. The choice is based on the level of coverage achieved, the difficulty of proceeding upward or downward in the system organization, and the effort required to establish a testing environment in each case.

3.3.1 Testing Environment

For top-down system testing, the testing environment at each stage is the obvious one: the topmost element of the software system will control some data and will selectively pass it downward in the invocation structure to the

subsystems, to the components, and eventually, to individual modules. New testcase data is added at the points where the software system normally accepts input data. These normal data input points are not necessarily part of the topmost program; there may be special "data entry" subsystems or components which are invoked by the topmost program specifically for this purpose.

In bottom-up testing, a system tester has two choices: where to concentrate the testing effort, and where to provide testcase data.

1. Testcase data can be supplied through the existing data input points. The system tester must be aware of data transformations performed prior to delivery to the module on which he is currently working. These transformations may make it difficult, or impossible, to exercise the current testing target.

2. The testcase data can be supplied through a separate testing environment designed and implemented specifically to provide for testing of a single system element. At the system testing level the testing would be performed with the normal data input mechanism. This testing environment is like that shown in Fig. 2.1 for a single module.

The choice between these two mechanisms for providing testcase data must be based on the specific internal features of the software system.

3.3.2 Collateral Testing

In system testing, the overall effort needed to achieve the desired testing coverage can be significantly reduced by collateral system testing. This phenomenon is similar to that for single modules, but at the system level collateral testing has a much larger potential benefit in achieving overall testing coverage rapidly and efficiently. Collateral system testing is illustrated in Fig. 3.2. A system test invocation is intended to produce an invocation chain leading to some testing target module, A. As a result of the choice of the chain used (if there is any choice), an entire subsystem, B, is executed collaterally. Thus, a test intended for one objective may achieve increased coverage of other components or subsystems in the software system.

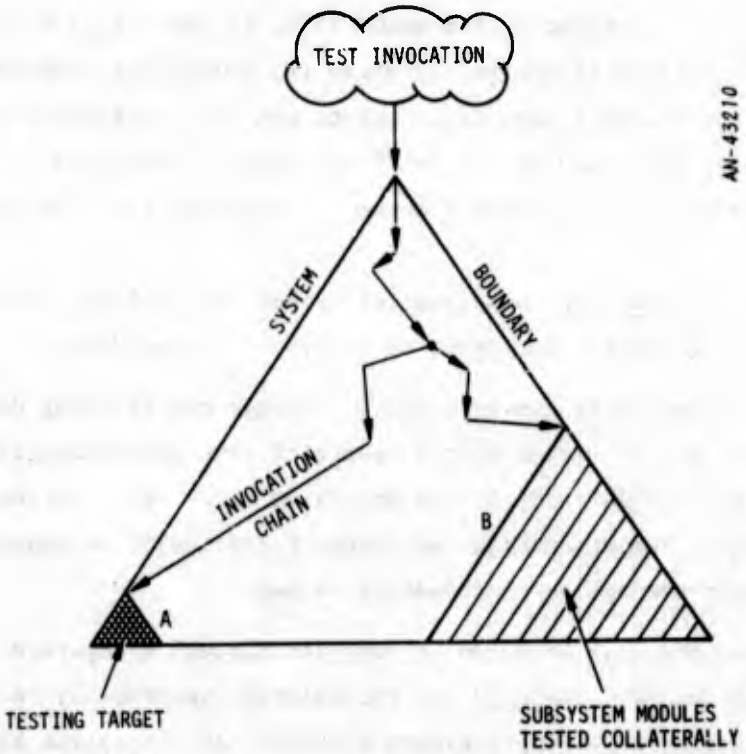


Figure 3.2. Collateral System Testing

The situation is analogous to single-module testing only in the sense that the overall testing coverage is increased by whatever collateral testing occurs. In system testing the program tester typically has considerably more control over the form of collateral testing achieved than he is likely to have when dealing with a single module. Single-module tests are primarily designed to achieve narrow testing objectives (e.g., an untested DD-path), and testcase data which meets that testing objective is chosen without regard for the later consequences of executing the test. For a system-wide test, the potential effects of subsystem invocation are already evident from the invocation structure.

3.4 MODULE INTERCOMMUNICATION

An important feature of a software system is the nature of the intercommunication between modules, between components, and between subsystems; this is important for two reasons:

- The communication spaces between elements of a software system may interfere with systematic system testing. Restricted between-element communication may make system testing substantially more difficult by requiring construction of special-purpose testing environments.
- When two system elements communicate with each other, the communication paths are the obvious ones to intercept when an element must be removed from its system context.

We discuss each of these problems in detail in the next sections.

3.4.1 Blocked Communication

Blocked communication is illustrated in Fig. 3.3. A test invocation, which carries with it some testcase data, is made to an element under test; in turn, this element invokes some other element one or more times. In the illustration, there are two invocations (A_1 and A_2) of the "invoked" element, B. The data conditioning boundary makes the two invocations to the B element different in the following way:

1. The A_1 invocation to B is made with full knowledge of the intercommunication space available from A at the invocation point A_1.
2. The A_2 invocation to B is made with a restricted set of intercommunication data.

The restricted set arises because the invocation of B at point A_2 occurs after certain conditions have been checked by A. Data conditioning was performed as a normal part of A's operations.

In the unblocked case, essentially all of the control information needed by B is available to it as the result of the invocation at A_1. It should be possible to construct values for the intercommunication context which would make the B element do anything at all. We should be able to achieve full testing coverage of B by means of invocations of the A_1 type, assuming that the appropriate test data can actually be constructed.

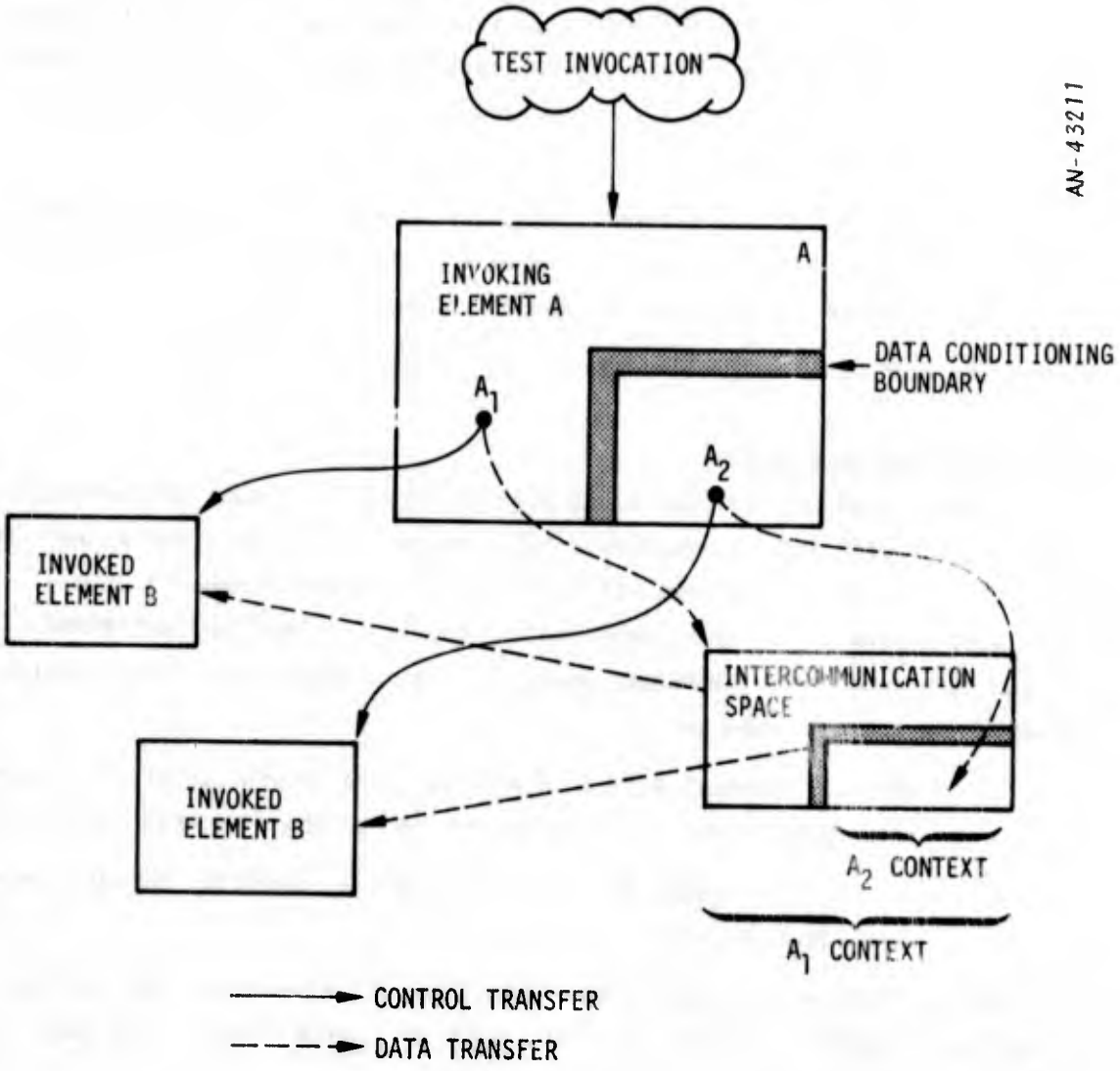


Figure 3.3. Blocked Intercommunication

The blocked case restricts that freedom. We can only use that part of the input communication space which is left unconditioned at the invocation point A_2. This restriction may make it impossible to exercise the B element comprehensively via the A_2 invocation. If there were no A_1 invocation a local testing environment for B would be necessary.

This situation is reversed if the testing objective is freestanding; e.g., the B element shown in Fig. 3.3 independent of its invocations by the A element. It will then be necessary to identify the appropriate intercommunication space for B and to provide testcase data for B in terms of that specialized testing environment. This will have to be done whenever B's relationship with the remainder of the system under test contains only blocked intercommunication. The alternative, employing the natural intercommunication space for B, may require lowering the acceptable testedness level for B in its system context.

In the light of these two alternatives, there is an interesting question about the level of system testedness: "If some elements of a system have a blocked intercommunication space, then why is it necessary to test those elements?" In other words, when it is impossible to reach certain parts of the software, those parts of the system can be considered superfluous. For example, consider that portion of the B element which cannot be exercised by means of data settings within the constrained communication space of the A_2 invocation in Fig. 3.3. This situation can occur for the following reasons:

- Portions of lower-level system elements cannot be tested as the result of some prior level of subsystem integration. In other words, there are:
  a. Extra capabilities in the software specifically included to provide for lower-level testing (i.e., on the module or component level).
  b. Some upper-level software elements that do not make use of all of the capabilities of some lower-level elements.

- The software system was built with defensive coding techniques that seek to increase a system's reliability by protecting some system elements through explicit conditioning of the data they are fed. Defensive coded software is protected against forms of data which cannot occur in normal operation of the system; the increased protection is obtained by decreasing the direct testability of the system.*

The system testers' decisions on whether or not to provide local testing environments, or possibly to extend the topmost levels of a software system to permit otherwise illegal communication paths purely for testing purposes, must be made in cooperation with system designers and implementors.

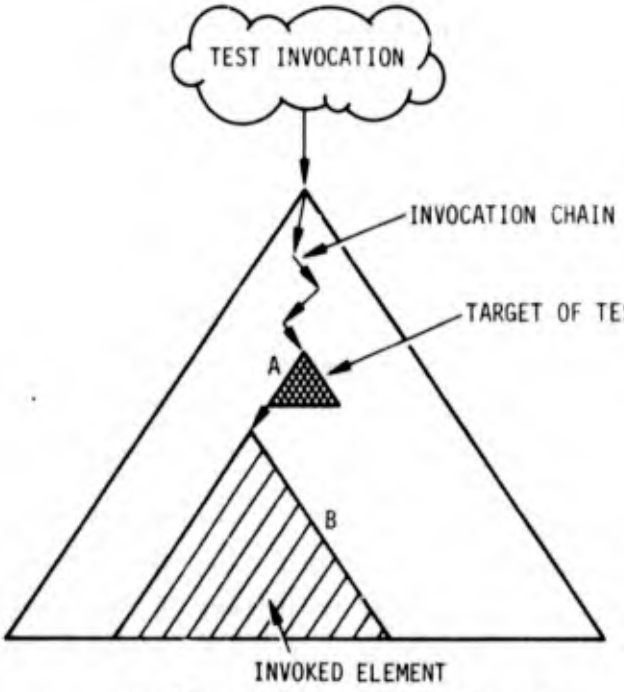
3.4.2 Stubbing

Even when the intercommunication paths are not blocked, it may not be desirable to invoke an entire system element merely for testing purposes. For example, the invoked component may be long-running, or it may require a large execution region. The process of providing a functional alternative to a system element, when it does not contain the current testing target, is called stubbing.

The stubbing process is illustrated in Fig. 3.4. In most situations, the invocation chain reaches first to the testing target and subsequently to the invoked subsystem. Stubbing breaks these latter communication lines and provides a testing stub which simulates the functional effects of the invoked subsystem. The normal path, shown as the invocation of the element B by the testing target A, is replaced with a simulated intercommunication path which replaces B by its stub, S_B. This may be accomplished by modifying the link/edit/load process during which instrumented copies of the appropriate modules are readied for test execution.

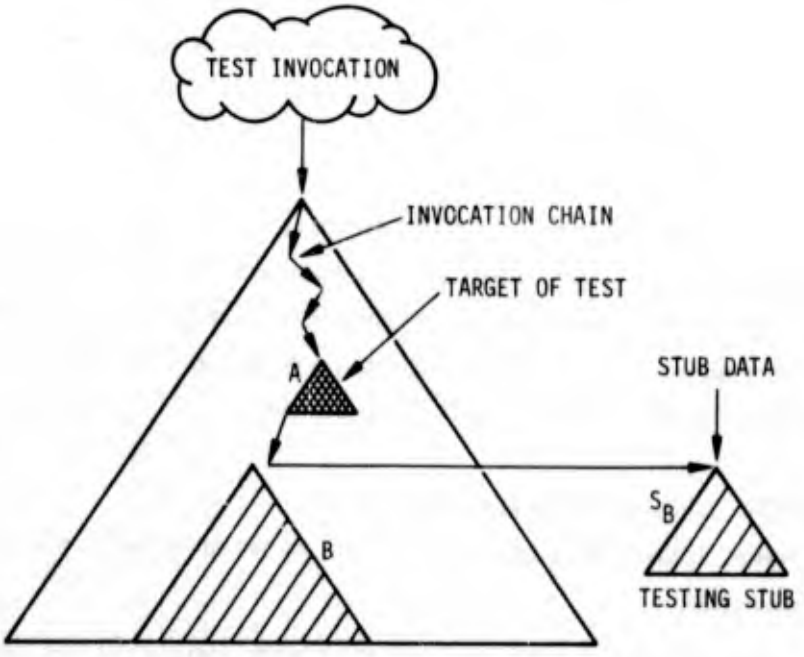
The testing stub, S_B, must provide the same responses as the element it simulates. This simulation may require little more than setting a few

*Indirect means of testing may be used; typically this involves the construction of special testing environments.



AN-43212

a. Normal Testing Environment



b. Stubbed Testing Environment

Figure 3.4. Stubbing Process

parameters, or it may involve as much work as B entails itself; in the latter case the use of a stub for B is equivalent to using B itself, and there is no advantage in using a stub.

Typical situations encountered when setting up testing stubs are categorized as follows:

1. The stub for the invoked module requires only a means to capture control and return it to the invoking element. This occurs when the computations performed by the stubbed element are not needed during the testing operations of the invoking element.

2. The stub for the invoked module must compute certain values in the intercommunication space. This situation occurs when some but not all of the functional activities of the invoked element are employed in the operations performed by the testing target subsequently to the stubbed invocation. The other operations of the invoked element, being unnecessary, may be eliminated by appropriate secondary stubbing.

3. All of the invoked element's functions are necessary to the activities subsequent to the invocation. In this case stubbing is ineffective.

Identifying the intercommunication space for two system elements can be a very difficult task, particularly if non-standard methods have been used. In most cases, however, one must only note the appropriate actual/formal parameter correspondences. Automatic identification of elements' intercommunication space is beyond the scope of this report.

3.5 SUMMARY: SYSTEM TESTING METHODOLOGY

The basic ingredients of the general software system testing methodology are the following:

- The ability to perform comprehensive single-module testing for each invokable module
- Knowledge of the system's invocation structure

- Previous (and initial) system testing coverage measures
- A next-testing-target selection function to allocate testing effort

The general form of the system testing methodology is shown in Fig. 3.5, which emphasizes continuous use of a system testing coverage measure. The interaction between the system testing coverage measure and the process of selective application of the single-module testing methodology is described next.

3.5.1 System Testing Coverage Measure

The measures of system-wide testing discussed in Sec. 3.2 provide overall assessments of system test coverage. The coverage value can also be used to select the best next testing target.

The simple per-module coverage measure, S2, will direct testing effort toward the module which is the least tested. The per-element form of composite testing coverage, S3, allocates testing effort toward the element which has undergone the least testing. Other measures, such as S4 or S5, can be employed in a similar manner.

The measure actually used should depend on the internal structure, and possibly the functional requirements, of the software system as a whole. The measure should unambiguously identify the module(s) least tested, but should tend to identify a number of possible testing targets. The choice between them should be made within the confines of the invocation hierarchy, and by considering the two important variations of testing strategy: top-down testing, and bottom-up testing.

3.5.2 Bottom-Up Testing

Figure 3.6 shows the use of a bottom-up testing target selection mechanism. The selector emphasizes comprehensive testing of single modules before components, components before subsystems, etc. The selection rule is the following:

> Bottom-Up Testing Selector. Begin the testing effort with modules which are invoked at the end (i.e., the lowest level) of the invocation chain (see Fig. 3.1). Advance upward in the hierarchy only after all terminal-branch elements have been exercised comprehensively. The testing target

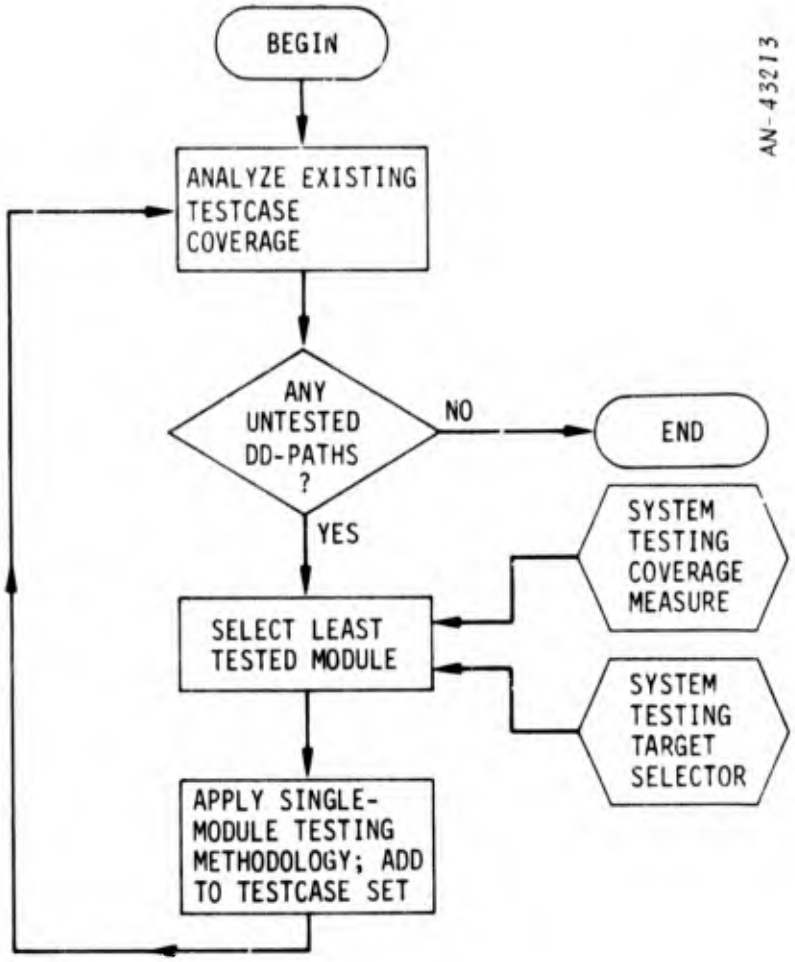


Figure 3.5. Overview of System Testing Methodology

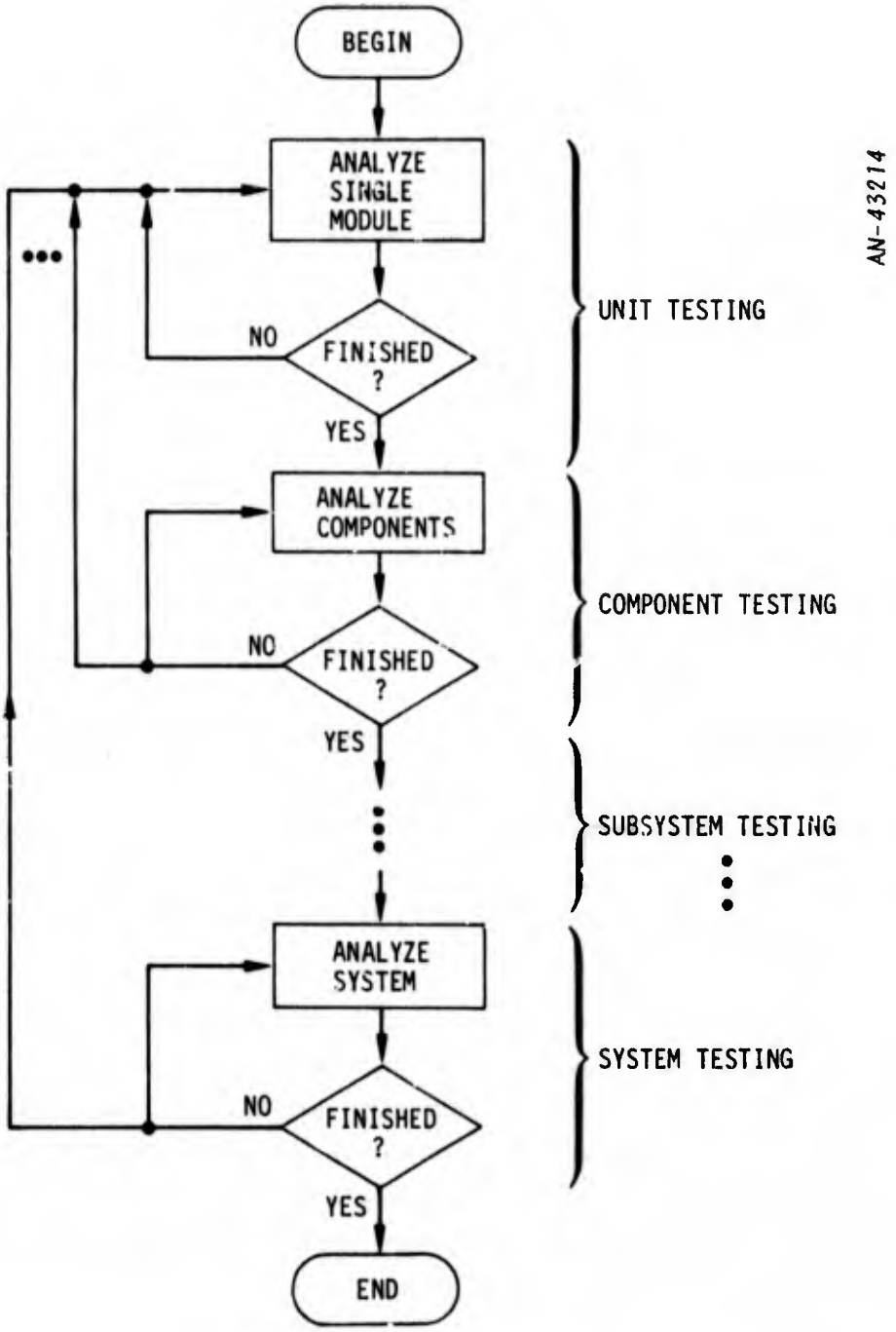


Figure 3.6. System Testing Methodology (Bottom Up)

is always the least tested module that is furthest down in the invocation hierarchy.

It may be necessary to accept less-than-full exercise of each module as a reasonable testing strategy. This amounts to assigning a minimum threshold of testedness for each module. Bottom-up testing will assure that all possible single-module testing will have occurred first; the technique has a high likelihood of transmitting this high level of exercise to the topmost levels of the software system. Special testing environments may be required along the way, however.

3.5.3 Top-Down Testing

Top-down testing is almost the reverse of bottom-up testing, and involves selecting for further testing effort the largest elements of the system (i.e., modules which control entire subsystems or components) before attempting testing of modules at the lower levels. The selection rule that corresponds to this testing strategy is:

> Top-Down Testing Selector: Begin testing at the beginning of the invocation structure (i.e., at the highest level). Advance downward only after the highest-level modules have been comprehensively exercised (to a preset threshold). The testing target selected is always the least tested module which resides at the level in the invocation hierarchy at which testing is currently proceeding.

This selector operates strictly in terms of the chain depth within the invocation structure tree. (This depth is the initial-subscript value for the $P_{i,j}$ shown in Fig. 3.1.) It is generally undesirable to accept less than 100% testing coverage at any stage of the process, however, since doing so may mean that an important invocation chain is missed. That is, any invocation chain which is the only one that permits testing some lower-level module must not be skipped in the early levels of testing.

The top-down method generally assures maximum collateral testing prior to attacking any particular lower-level module. Stubbing of system elements may be necessary to conserve limited testing resources, however.

3.5.4 General Strategy

The best approach for systematically testing a large software system will depend on the specifics of that system's elements; it is not possible to state a universally applicable strategy. Mixtures of the top-down and bottom-up approaches may well cost the least, and may result in the greatest testing coverage. The program tester must be prepared to make some *ad hoc* decisions on "where to go next" independent of the values of the system coverage measures and the outcome of the chosen selector function.

SECTION 4

AUTOMATED VERIFICATION SYSTEM DESIGN REQUIREMENTS

The previous two sections have discussed single-module testing and software system testing. The discussion assumed that the methodological tools described would be largely automatable. Mechanization of the methodology in an Automated Verification System (AVS) is important for the following reasons:

- An automated system is less prone to failure by errors of omission.
- An automated system can arrive at fully verified code for significantly less cost; or equivalently, higher-quality software for the same cost.

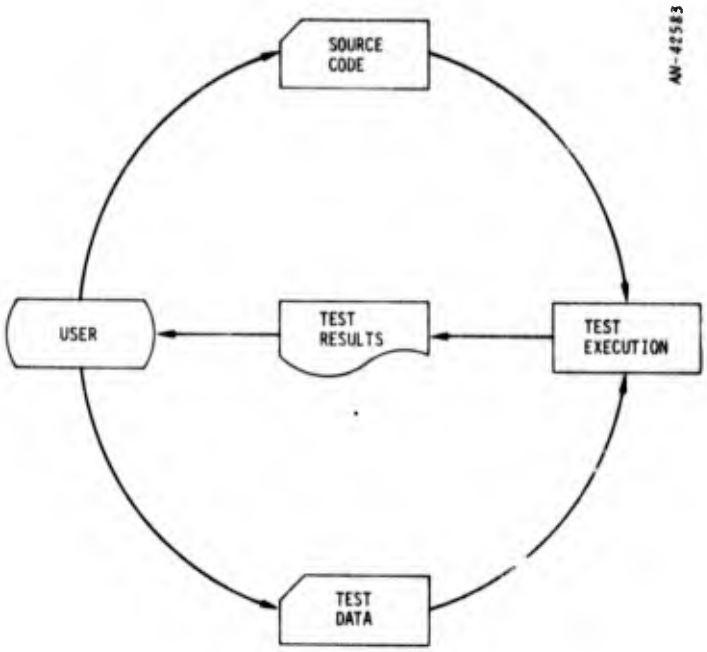
These criteria have guided the development of the methodology and have contributed to the design of the associated Automated Verification System.

In this section we see how the principal elements of the methodology are automated. Both single-module and system-wide testing methods are treated. Specific AVS design criteria already found to be valuable in program testing are also discussed, as are the features of the AVS needed in program maintenance support activities.

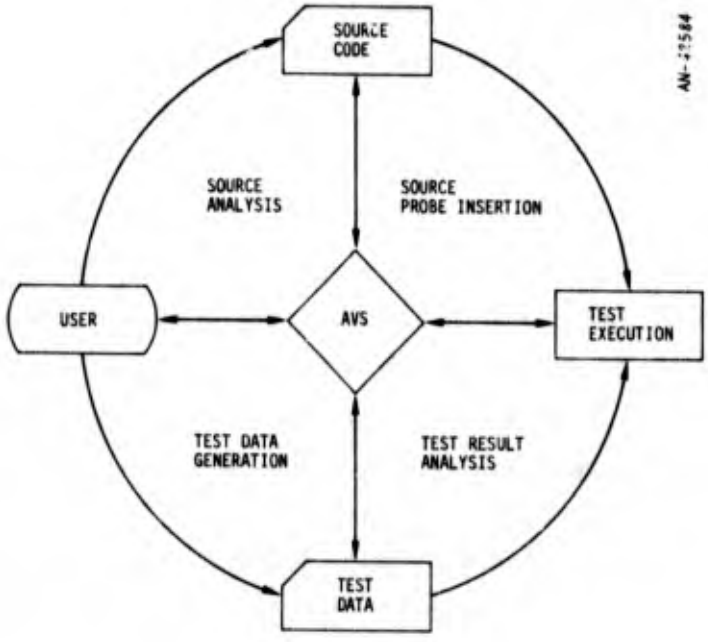
4.1 AVS FUNCTIONS AND FACILITIES

Since the activities that take place during program testing and verification are fairly general, it will be valuable to compare and contrast conventional (non-AVS) testing with automated testing.

Conventional testing is diagrammed in Fig. 4.1a. The program tester analyzes a series of executions of the software being tested. His knowledge of the internal operation of the software system, combined with analysis of the software's normal execution-time outputs, permits constructing appropriate test data. These testcases are added to the existing testcase set and additional executions of the software system are made. The process continues until the user is satisfied with the outcomes of the tests, or until some other constraint (such as cost overrun) signals an end to the process.



a. Conventional (Manual)



b. AVS

Figure 4.1. Conventional and AVS Software Testing

AVS-based testing is diagrammed in Fig. 4.1b. The processing capabilities of the AVS enable the user to (1) analyze tests, (2) generate testcase data by analyzing portions of the source text of the software system, and (3) automatically instrument selected elements of the software system for subsequent executions. The AVS forms the operational framework and support tool for all of the analyses performed by the software system tester.

4.1.1 Typical AVS Facilities

AVS facility design is based partly on experience with fully-manual program testing activities (Fig. 4.1a), and on the AVS-based extrapolation of these same activities (Fig. 4.1b). The AVS must provide the following capabilities:

- Producing logically equivalent but specially instrumented copies of individual program texts. This instrumentation (software probes) is used to provide execution-time feedback to the user so that he can know how much of the code he has tested and what portions of it remain to be tested.
- Capturing the data emitted by the software probes and analyzing it in a way which tells the user the testing coverage.
- Displaying a selected portion of a program so that it can be analyzed in detail. This capability is important for generating testcase data.
- Managing the testcase aggregation process by providing straightforward mechanisms for indexing and cataloging the testcases already run or being constructed.

All of these capabilities should be implemented in a uniform, user-oriented manner. To design an AVS which is more difficult to use than the corresponding manual testing process would be a step backward. The interface with the user (via AVS commands) should be simple, powerful, and unambiguous. The AVS should act as the program tester's partner, and the partner should not be capricious.

4.1.2 Limitations of Automation

The critical elements of the capabilities indicated above can be automated entirely. Certain capabilities, however, fall outside the range of straightforward and direct implementation in a state-of-the art AVS--partly because of the very nature of the processes involved. It is valuable to identify features which are not considered as part of an AVS in this section, but would be desirable additions to an AVS in the future. The reasons which make each additional capability impractical to implement now, as part of a general AVS capability, are also stated.

- Automatic Testcase Generation. This capability would identify an untested portion of a module and automatically find values for the variables in its input space that will cause the particular portion to be exercised. Desirable as this capability is, it is not possible to consider providing it at the current state of the art. The reasons are highly technical,[12] but boil down to the following: to build testcases automatically, in every instance where it is desirable to do so, requires that the AVS be able to find a solution to a set of nonlinear inequalities which correspond to the set of backtracked logical conditions which would make the desired program flow occur. While heuristic techniques may provide some capability in the future (see Ref. 13), they are not well developed at this time.

- AVS-Supported Testcase Data Maintenance. This capability would assign to the AVS the role of keeping a file of testcase data (natural and generated) and retrieving selected elements whenever necessary. The wide range of system input mechanisms makes this a difficult facility to design, and virtually impossible to implement in a general way which serves every possible need. Existing file storage and retrieval systems are more effective for this function than any which could reasonably be mechanized in an AVS at this time.

- Complexity Measures. The single-module and multiple-module testing methodologies described in Secs. 2 and 3 allude to the existence of a measure to select the most complex of a set of equally

untested DD-paths or system elements. Such a measure is needed when single-module or system testing reaches the point at which the program tester must decide where to focus his testing effort, and when all other factors affecting the desirability of a particular DD-path or system element are equal. The descriptions of this selection function have avoided the discussion of any specific complexity measure. Research on measuring the complexity of computer software is still in its infancy (see Ref. 14). When software research yields measures that make sense mathematically and intuitively, they can be incorporated as part of the AVS's functions.

4.2 AVS ORGANIZATION

The AVS functions identified above have varying degrees of interdependence. The data flow among them is shown in Fig. 4.2, in which the AVS is central to all operations. We discuss each of the major categories of information flow in detail, and in terms of the five basic processing steps shown in Fig. 4.3. The reader should consult both figures while studying this section.

4.2.1 Module Analysis

The user can provide commands to the AVS for basic analysis of source text modules. This initial analysis includes three major kinds of activities:

1. Scanning the source text and converting it into an internal format. The source text is then stored, statement by statement, within a special AVS database for later use.

2. Analyzing the internal structure of individual source modules, and storing the result in the database. This analysis consists of identifying the DD-paths and level-i path classes.

3. Analyzing the individual statements in the program for their internal semantics. This analysis, which may be optional in some AVSs, generally results in a complete symbol table for each module. When the language processed by the AVS requires it, this

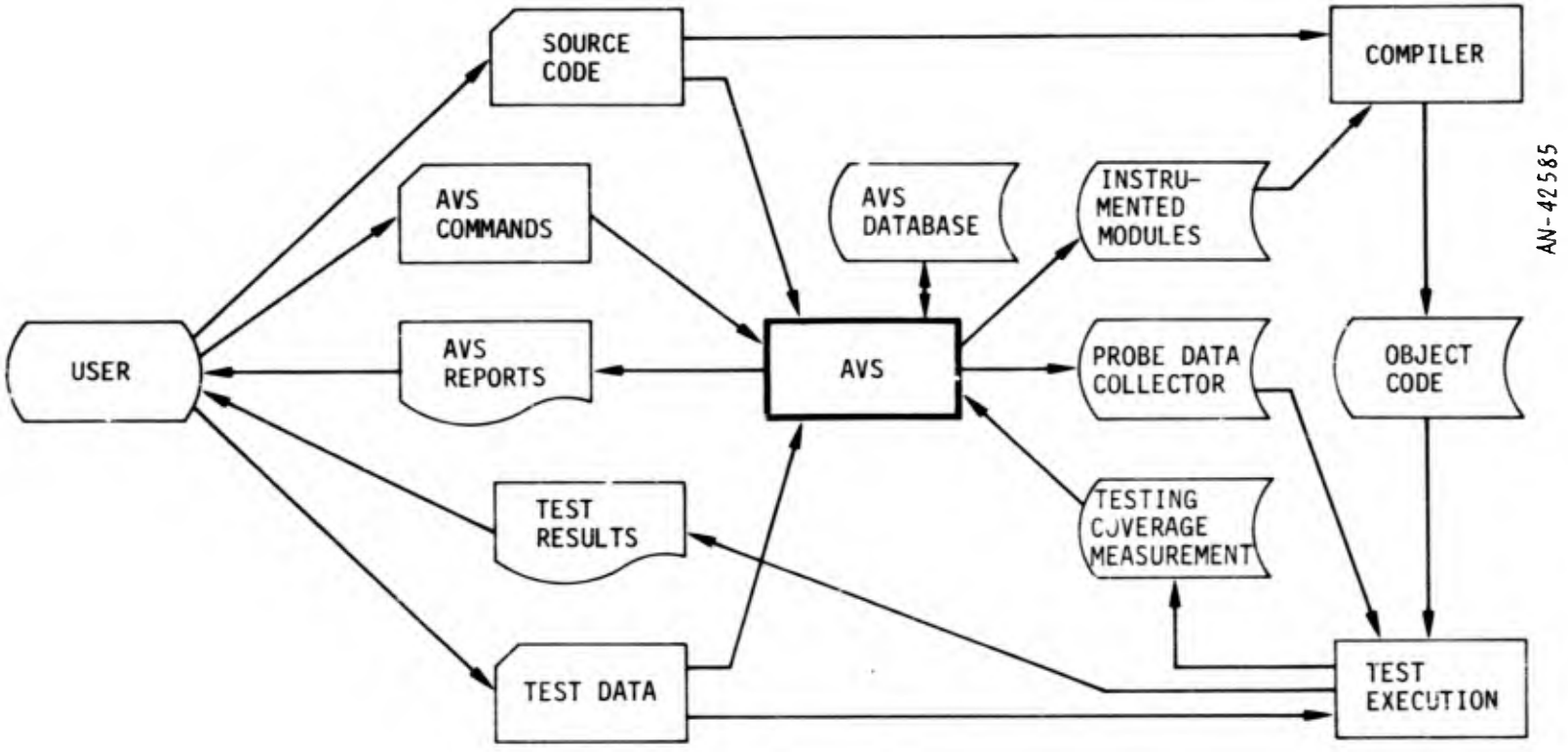


Figure 4.2. Data Flow in AVS Testing

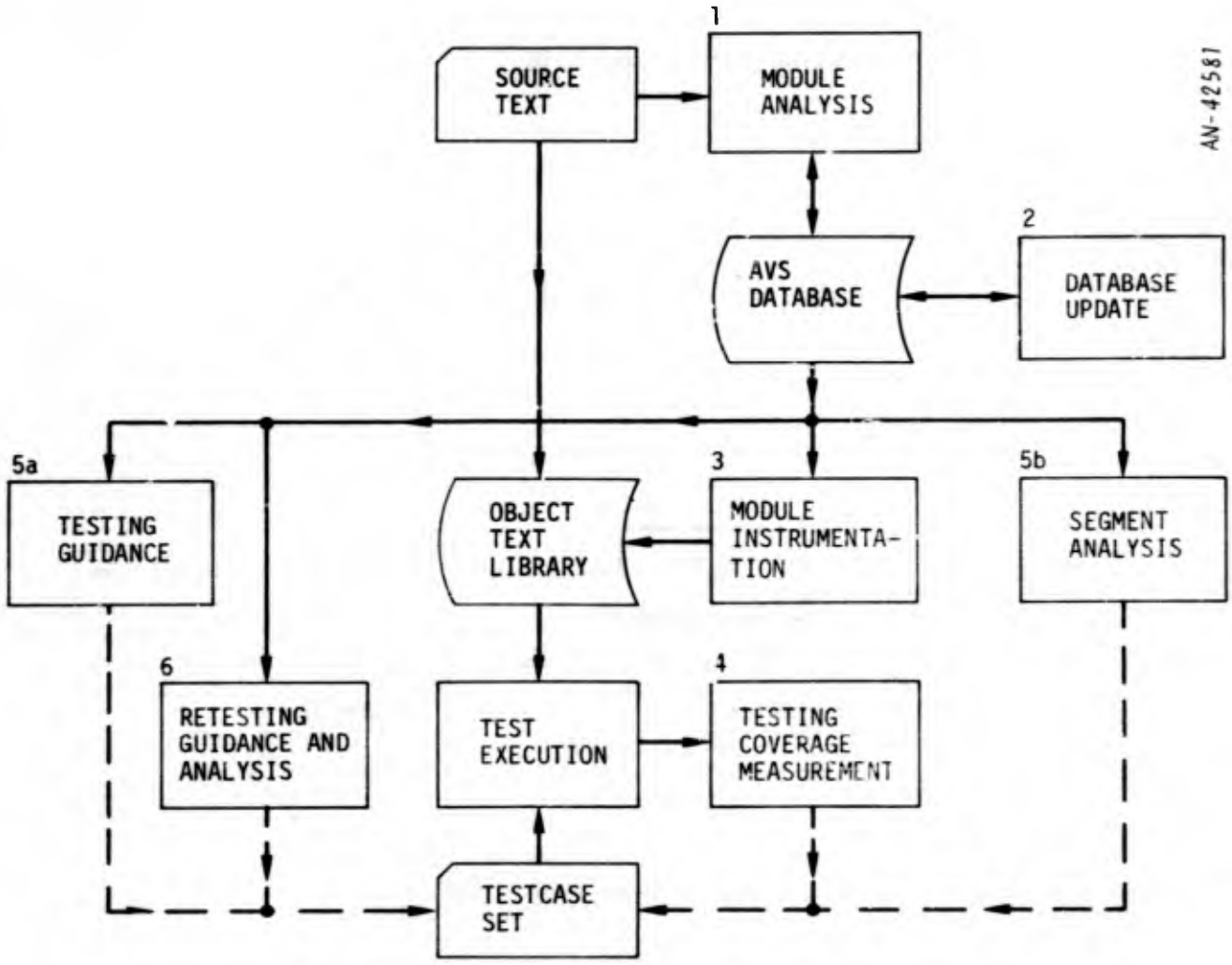


Figure 4.3. AVS Standard Processing Steps

symbol table is organized to help identify the definitional chain of each symbol.

The result of the Module Analysis step is a database containing essentially all the information which will be needed for the other operations of the AVS.

4.2.2 Database Update

Each module is added to the database as it is processed. The AVS has a separate processing step to merge the added data into the database; a built-in priority system assures that only the most recent copy of each module is in the updated database. Database update is separated for convenience in using the AVS; separate AVS executions can be used to treat a small number of modules at a time, so that a large system database can be constructed incrementally.

4.2.3 Module Instrumentation

Testing of individual modules or system elements (components, subsystems, etc.) requires collecting execution-time information. Module Instrumentation is the process of modifying the individual source texts so that they make the appropriate invocations to the data-collection routine. The AVS accepts user commands which indicate the names (or numbers) of the modules which are to be instrumented. The instrumentation process itself (see Sec. 4.3) involves rewriting the selected modules so that the instrumentation probe text is inserted automatically at appropriate points. This operation is accomplished through reference to the source texts stored in the AVS database.

Modules which have been instrumented are delivered to the compiler for translation into machine-executable form in the normal manner. In addition to instrumented modules the user may supply the compiler with other, uninstrumented, source codes. This capability is also available with appropriate user commands.

The compiler produces relocatable (binary) object modules; these can be executed directly, or saved for later use on an object library. Saving the

instrumented versions of the object modules is desirable when they can be selected (by an operating system facility) for use individually. This avoids regenerating instrumented texts each time they are needed.

4.2.4 Testing Coverage Measurement

During test execution, the software system object code is combined with the special-purpose modules to which the instrumented modules pass data. The data collector reports the results of program execution back to the AVS, which then analyzes the data and provides appropriate DD-path testing coverage reports to the user. These reports indicate the degree of testing achieved for any or all of the modules which were instrumented for the particular test execution. In addition to the special instrumentation output, the user has at his disposal the normal execution-time outputs of the software system under test.

The Testing Coverage Measurement step centralizes the processing of data collected by the software probes, and generates coverage analyses for the user in terms of the coverage measures described in Secs. 2 and 3.

4.2.5 Testing Guidance and Segment Analysis

The Testing Guidance and Segment Analysis facilities of the AVS address specific individual testing situations and specifically selected modules. The AVS database is sufficient to support all the guidance functions needed for single-module and system testing. Some of the forms of output produced by these processing steps include:

- Identification of the properties of DD-paths specified by the user for detailed analysis. These are generally DD-paths which have not been executed in the testcase dataset thus far.
- Generation of appropriate reach and cycle sequences based on the known iteration structure of the module, as selected by specific commands from the user.
- Display of the program text which corresponds to reach and cycle sequences identified and commanded for display by the user.

- Exposition of the properties of individual level-i path classes, of the dependence tree of level-i path classes, and related information pertaining to the details of the iteration structure of each module.

These analytic tools, combined with the coverage reports and the normal execution output of the program, assist the user in generating testcase datasets which exercise previously unexercised DD-paths.

4.2.6 Retesting Guidance

The Retesting Guidance processing step is invoked only when the AVS is used as the vehicle for controlling maintenance of a software system. The purpose of this step is to assess the significance of a change in the software as it affects the level of testing coverage previously achieved. The modules which are changed are identified by the user in a set of commands, and the AVS analyzes the impact of the changes. This information is presented in the following types of reports:

1. The set of modules which invoke the changed module
2. The set of modules which the changed module invokes
3. The set of modules which share data with the changed module*

This dynamic dependence information lets the program tester choose the appropriate point at which to reapply the system-testing methodology. Only those portions of the system-testing methodology which concern the modules that change or are affected by the change need to be repeated.

4.3 AVS FACILITIES FOR SINGLE-MODULE TESTING

How to best use the AVS's facilities varies with the current testing objective. This section describes the AVS facilities required to support the single-module testing methodology; the facilities applicable to system testing are described in Sec. 4.4.

---

*This capability is included here for completeness. It is not known at this time whether this shared data space can be identified automatically and unambiguously. This capability is not included in the current implementation of the AVS.

The single-module testing goal, as we have already discussed, is to generate sets of testcases which, in aggregate, exercise each DD-path within a module at least once. The AVS supports the single-module testing methodology by providing three major categories of processing:

1. Automatic Instrumentation of Single Modules. This instrumentation intercepts each DD-path in a way which permits normal execution of the module in a testing environment, but which provides for capture of data transferred by the instrumentation probes.

2. Analysis of Testing Coverage. These analyses are performed using the instrumented code, and provide reports to the program tester which identify the DD-paths which have not yet been exercised in the testing activity.

3. Analysis to Support Testcase Generation. This collection of analytic tools is based on manipulation of the iteration structure of selected modules (as described in Sec. 2), and is the basis for selection of appropriate testcase data.

We discuss each of the required AVS facilities in turn.

4.3.1 Automatic Instrumentation

The AVS facility which provides flow instrumentation must implement a set of algorithms which automatically insert "software probes" into source text in a way that insures:

1. That the resulting source text is, with the exception of the effect of the inserted probes, logically equivalent to the original source text.

2. That each DD-path is intercepted with an instrumentation probe in a way that makes determining which DD-paths were executed in a test as simple as possible.

The technology for meeting these two objectives is well understood. Logical integrity within the individual program text is assured by providing additional variables in which to capture the effect of the evaluation of logical expressions prior to their use within the control structure of the module. In this way, a single probe can be inserted for each DD-path; the invocation to

the data collection routine can pass the DD-path number (and other information if necessary) in its actual parameter list.

This instrumentation process is illustrated in Fig. 4.4, for a simple IF statement. These are two DD-paths, one for each of the possible outcomes of the decision. The probe insertion processor must implant an invocation to the probe routine (called PROBE in the example) for each DD-path. This is accomplished by first reassigning the value of the logical expression to a local variable which can then be checked as the means to decide which PROBE invocation to make. The two PROBE invocations each carry with them (1) the name of the module to which the DD-path belongs, and (2) the number of the DD-path (either n_T or n_F).

The logical properties of the instrumented code are equivalent to the uninstrumented version, but the execution time and execution length increase because it was necessary to add a variable to temporarily hold the value of the logical expression, and because it was necessary to add text which controls the invocation of the PROBE routine. The overhead for this type of instrumentation is highly dependent on the nature of the program text processed; space increases on the order of 20-50% and execution-time increases on the order of 25-75% are typical.

4.3.2 Execution-Time Data Collection

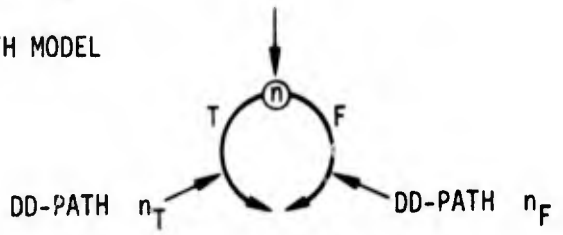
Because the inserted instrumentation carries with it both the name of the module and the DD-path number, the execution-time data collection and reduction can be relatively simple. A test execution for one or more instrumented modules will result in a stream of invocations to the data-collection routine, each invocation carrying with it the DD-path number and the name of the module to which the DD-path belongs.

The sequence of instrumentation calls is analyzed by conventional techniques. The forms of output needed for dealing with a particular module typically fall into the following categories:

(A) SOURCE TEXT

```
                MODULE <NAME>
                    •
                    •
                    •
          n     IF (<LOGICAL-EXPRESSION>)<CONSEQUENCE>
                    •
                    •
                    •
```

(B) DD-PATH MODEL



(C) INSTRUMENTATION TECHNIQUE

```
                MODULE <NAME>
                    •
                    •
                    •

ADDITIONAL      { <LOCAL-VARIABLE = <LOGICAL-EXPRESSION>
INSTRUMENTATION   IF (<LOCAL-VARIABLE> = TRUE) PROBE (<NAME>, nT)
TEXT              IF (<LOCAL-VARIABLE> = FALSE) PROBE (<NAME>, nF)

MODIFIED
ORIGINAL TEXT   { IF (<LOCAL-VARIABLE> = TRUE)<CONSEQUENCE>
```

AN-43317

Figure 4.4. Example of DD-Path Instrumentation

1. The total number of DD-paths executed, a rough measure of the total length of the test

2. The total number of distinct DD-paths executed and, possibly, the number of times each was executed

3. The set of DD-paths which were not executed, the "feedback" to the tester to identify untested DD-paths

4.3.3 Testcase Generation Assistance

Given that a particular DD-path has not been executed in a set of testcases already developed for a single module, the AVS must support analyses which will assist the program tester in constructing testcase data which will cause execution of that DD-path. The AVS functions required in this role are of two kinds:

1. Assistance in constructing reach and cycle sequences which employ the DD-path in a variety of ways

2. Production of excerpts from the program text which display these sequences of DD-paths in a manner which makes it straightforward for a program tester to develop appropriate testing data

The first requirement is met by incorporating a series of commands to the AVS to construct designated reach and cycle sequences. This set of commands has two types of operations:

- For a given untested DD-path, the AVS constructs a list of reach or cycle sequences (in terms of level-i path class numbers) which apply to that DD-path.

- For a specified sequence of level-i path class numbers, the AVS constructs the program flow which represents the reach or cycle sequences.

At the AVS command level, the user first specifies a DD-path and receives a report indicating all the possible reach and/or cycle sequences it is possible to consider. This list is then used to systematically try each possible reach sequence until a testcase dataset can be constructed; if necessary, the cycle sequences can be analyzed with the same facility.

The second requirement is met by providing an output of the form illustrated in Fig. 4.5. This output consists of a series of program statements organized to preserve the execution order implied by the ordering of the DD-path numbers which correspond to a reach or cycle sequence. This output listing identifies these form of information:

1. The sequence of DD-paths for which program text is to be displayed.

2. The program texts for each DD-path, in their normal execution order. Each program text is displayed with its original statement number, and the output is delineated with heavy dashed lines indicating the beginning and end of each DD-path text.

It is important in this output format that non-sequential statement sequences be shown in the DD-path order; this is illustrated for DD-path No. m in which statements $n+3$ and $n+n_1$ (which are actually logically contiguous) are displayed adjacent to one another.

This basic form of display of non-iterative program flow is the one which program testers analyze in order to construct appropriate testcase data. This analysis goes backward along the indicated path and usually begins at the last program text decision statement in the sequence.

4.4 AVS FACILITIES FOR SYSTEM TESTING

System-wide testing requires special AVS facilities. The primary mechanisms used in the system testing methodology are sets of strategies for selecting the "next target of single-module testing effort". In dynamic application, system-wide testing consists of a series of single-module test generating activities applied in some rational order. Means for selecting the next target of this activity, as well as general strategies for applying these selection mechanisms, were discussed in Sec. 3.

The AVS provides system-level instrumentation and data collection, and implements the system "next target" selection function. We discuss each of these facilities in turn.

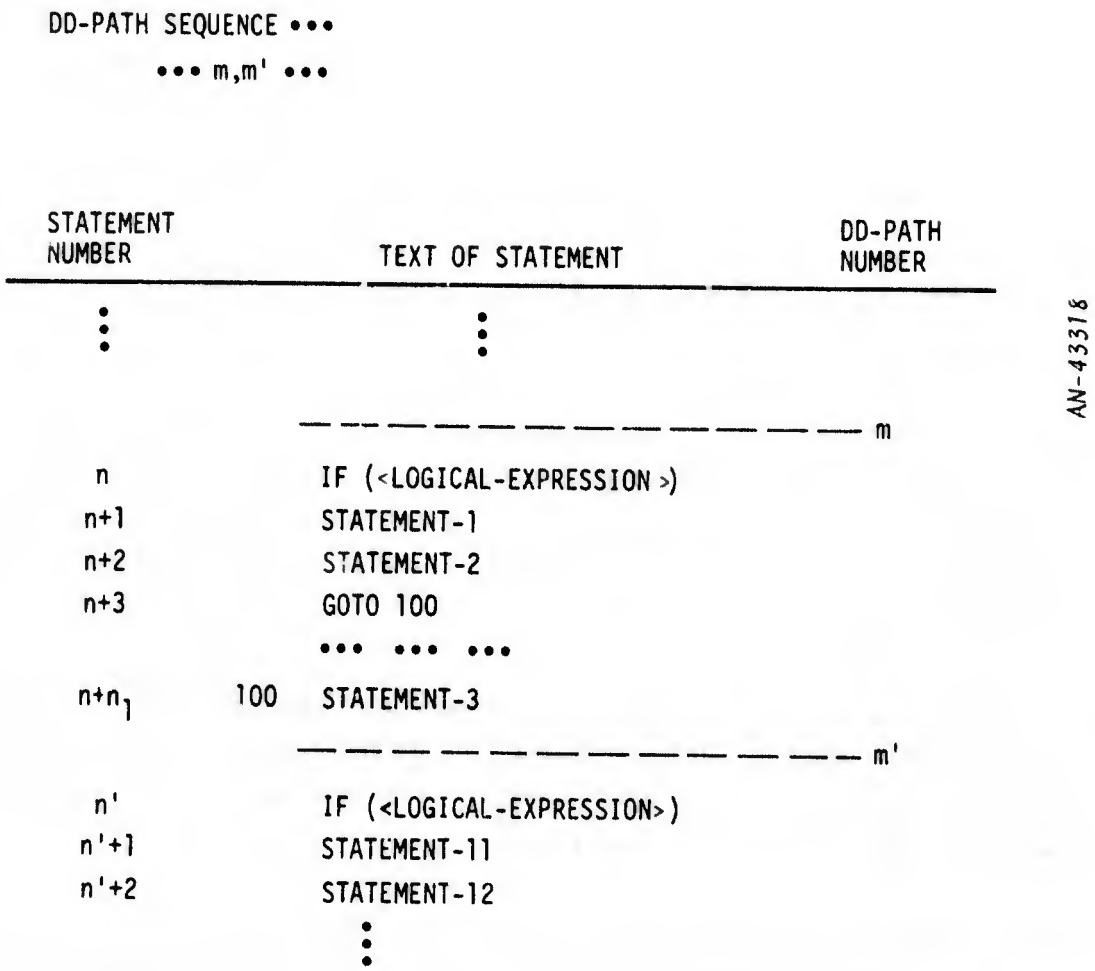


Figure 4.5. Form of DD-Path Sequence Output

4.4.1 System Instrumentation

During system testing, the program tester always has the option of selecting one or more single modules for detailed testing coverage analysis; this is accomplished with the techniques described in Sec. 4.3. Another form of instrumentation which is valuable to obtain is so-called "invocation checking" for all of the modules of a system, or all of the modules of a system element (subsystem, component, etc.). This is accomplished very simply by imposing the following convention on the assignment of DD-path numbers to the DD-paths of individual modules: DD-path number 1, for any module, is always the "invocation DD-path" for that module. Such a DD-path must be an entry DD-path, as defined in Sec. 2.2.2.

Conventional instrumentation and data collection can now be applied, taking full advantage of the implication of this simple convention. If a single module is to be instrumented, then each DD-path receives treatment; if only invocation instrumentation is needed, then only the entry DD-paths for the module are probed.

Testing data collection for system-wide testing is similar to that for single-module testing, except that only the names of the modules executed are recorded and analyzed; this name is provided normally as part of a standard DD-path probe (see Fig. 4.4, for example).

4.4.2 System Testing Measurement

The number of times individual modules are invoked can be easily analyzed, using techniques similar to those described for single-module testing coverage analysis. System testing reports must have somewhat different interpretations:

- The subsystem (or component) testing coverage report indicates the number of times each module has been invoked.
- The data reduction routines report which single modules, within the subsystem (or component) being considered, have not been executed.

This information, combined with knowledge of the software system's invocation tree (dynamic organization) provides the information basic to automating the testing target selection process.

The most direct form of selection process is to allocate testing effort to the module or modules which have been invoked the fewest times. Design and implementation of more sophisticated measures of software system complexity is beyond the scope of this report.

4.5 SOFTWARE RETESTING GUIDANCE FACILITY

As already mentioned, the function of the Retesting Guidance capability of the AVS is to assess the impact of modifications of the source text of individual modules on the level of testedness already achieved by the existing sets of testcases. Two avenues for assessing this impact are to:

1. Re-execute the existing set of testcases, and analyze the single-module and system element coverages obtained. All untested DD-paths (within single modules) and all uninvoked modules (within the software system) are obvious subjects for retesting analysis.

2. Analyze the data structure dependencies up and down the module invocation hierarchy. This analysis would identify the modules which invoke the changed module and the modules which the changed module invokes. Assuming that the change made in the system is localized to a particular module, this analysis will identify those modules for which testing must be repeated.

The facilities provided by the AVS to support retesting guidance are simple ones:

1. The AVS must, upon command, provide a report which identifies the modules which invoke, and the modules invoked by, the single module which has been altered.

2. The AVS must, through its normal instrumentation facilities, be able to identify DD-paths and single modules which remain untested upon re-execution of the old testcase set against the software system with the changed module replacing its older copy.

These two sets of modules should be analyzed for the overall impact they individually induce in the testing effort, and for the decrease in testing coverage which occurs as the result of the change. It should be clear that, if the effect of a change is widespread enough, the software testing process must begin anew.

SECTION 5

IMPACT OF TESTING AUTOMATION

The methodologies described in the preceding sections have indicated a rational, systematic route to achieving a first level of program testedness. These methodologies are characterized by their universality and automatability. This section addresses the question: "What will the impact of widespread Air Force use of this methodology be?"

Some caveats are necessary here:

- The technical basis of the methodology, although a strong one, has undergone only a few years of trial and experimentation.
- There is no data to compare the effect of use of the new methodology in quantitative terms against "the old way of doing things." The lack of detailed information is less the fault of any individual than it is a natural circumstance: software development has always been a highly intellectual, human-driven, activity. System developers who attempt to collect detailed statistics on their costs run the risk of being accused of misallocation of their very limited resources. Researchers, as a result, find it difficult to "instrument" the software production/verification process.

Clearly, we can only speculate on the general impact of this methodology. Even though the expected benefits (described below) are not yet realized, there is every reason to believe that many of these predictions will be highly accurate. Foremost among the justifications are the methodology designers' intuitions about software and its systematic verification; the authors take full responsibility for the veracity of these intuitions.

5.1 THE COST OF SOFTWARE

The widespread use of an AVS will probably not affect the total cost of producing software very much in the next few years. Instead, the mechanism by which software costs decrease as the result of AVS use will be indirect.

For example, suppose there is some fairly common program construct which is found to cause extreme difficulties for a program tester. This construct might be a strange multi-exit looping structure for which the identification of testcase data is very difficult. The word will get around because program testers will say something like "Please avoid the use of this construct because it complicates our job immeasurably." This plea, in and of itself, may have no effect; but amplified by the needs of responsible management, who will be able to discern the ultimate payoff in enhanced software, this kind of implicit restriction (guideline?) will be applied to future software development processes. The long-term result is decreased testing cost.

Furthermore, if the particular program construct is one which is troublesome in terms of the actual use of the software, then this same technological feedback loop would serve to improve the reliability of software systems.

5.2 THE COST OF TESTING SOFTWARE

Using an AVS will reduce the cost of testing software, or equivalently achieve a higher level of testedness at the same cost. The cost-effectiveness of the software testing process will increase in either instance.

Many of the processes involved in this methodology are very simple ones; consequently, the testing activity can be accomplished with a less expensive category of technical support personnel.

In Sec. 2 we indicated an expectation that a set of testcases for a single module would be no more numerous than 10% of the number of DD-paths present in the module. Although stated as a goal, this expectation is founded in the authors' experience of employing the single-module testing methodology. The implications are worth elaborating:

- Because the likely number of tests can be estimated, one can predict beforehand the degree to which testing has been accomplished (independently of the DD-path coverage), both for single modules and for whole systems.

- As long as this "figure of merit" for a testcase set is accurate, it can be used as a discriminant for the "degree of difficulty" of testing individual modules or systems. In other words, modules which require more than this rule-of-thumb number of testcases are "too complex" in some sense.

- The total number of tests needed to comprehensively exercise a software system composed of modules of approximately equal "degree of difficulty of testing" is predictable. This information will be of value to system production/verification managers.

In general terms, reduced numbers of required testcases will either (1) decrease the total testing effort, or (2) increase the level of coverage obtained with the same testing effort.

5.3 THE TIME NEEDED TO TEST SOFTWARE

The use of an AVS should greatly reduce the total time required to achieve a specified level of testedness for a large software system. The total time reduction is primarily the result of automation: instrumentation, data collection and reduction, and feedback into the testing process all are intrinsically automatable functions of the general methodology. The AVS will replace people-time by computer-time or, equivalently, will guarantee the capability of the same number of people to perform at previously unattainable levels.

5.4 THE RISKS OF SOFTWARE

As we mentioned in Secs. 2 and 3, our preliminary measures of testing coverage are far from exhaustive. Testing every DD-path in each program clearly will not rule out the possibility of "bugs" in the software. It may very soon be possible, however, to consider more sophisticated forms of "exhaustive exercise" of code. One possibility is to consider generating a set of testcases which, in aggregate, employs each level-i path class at least once in every possible legitimate combination.

Even with the present limitations, the DD-path-based testing goal of Sec. 2 is an attainable first step in the right direction. Programs subjected

to this level of testing exercise can be considered to harbor "no surprises". Comprehensive exercise of software to this level will locate the overwhelming majority of errors.

5.5 TESTING LARGE SOFTWARE SYSTEMS

Using AVS technology in testing large software systems is an area where the payoffs are immediate: without an AVS there would be no reasonable way to deal with the testing issue. Many of the functions described in Sec. 4 are a methodological overkill for small program texts which can be dealt with by hand. The advantage of the AVS methodology described here is that it will not suffer unacceptable performance degradation when confronted with large-scale software systems.

5.6 SUMMARY

The use of an AVS in the program verification stage of the software production cycle will, ultimately, reduce the cost, risks, and time for the application of computer software systems to real problems. The AVS provides automated versions of systematic computations that are easy to perform by hand on simple modules, but extremely difficult for large-scale software systems. The assured level of testedness achieved through AVS use will significantly increase the users' confidence in the analyzed software system. Moreover, AVS-based methodologies are an exciting environment for collecting valuable new information about the program testing process.

APPENDIX A

MODULE HIERARCHIES IN JOVIAL/J3

The JOVIAL/J3 language permits certain constructions which make it necessary to define the boundaries of an "invokable element of code" very carefully. In addition, it is necessary to describe the context which must be carried along with each invokable entity during the source-text instrumentation process, since this process assumes that each instrumented entry is separately invokable.

IN JOVIAL/J3 an executable text is one of the following three objects:

1. A main program, exclusive of any included closes or procedures
2. A close, exclusive of any included closes or procedures
3. A procedure, exclusive of any included closes

Note that JOVIAL/J3 does not permit a procedure to reside within another procedure.

The invocation points for all three of these types of modules are the defining "subprogram statement" of the appropriate kind. Each such module is invokable in the sense that control is passed to it by an invocation (an explicit reference to the module) and control is passed back to the invoking module.

A module is identifiable by the name used to invoke it, if that name is sufficiently global that there is no chance for ambiguity. For closes within closes, within a main program, or within a procedure, the JOVIAL/J3 naming rules leave some margin for ambiguity. Thus, to avoid all problems with the naming rules for invokable modules, we assume that all such modules are named according to the chain of containments within which they are defined. Thus, for example, if CLOSE A is defined within CLOSE B, which is defined within CLOSE C, which is defined within PROCEDURE D, which is defined within PROGRAM E, then the innermost close is named: A.B.C.D.E. This technique will assure uniqueness even when two separate procedures define a close named A.

During instrumentation and testing of a single module it may be necessary to operate with more than simply the instrumented text for that module alone. The JOVIAL/J3 language makes a single module meaningful only within a larger context, since that larger context may include statements which give meaning to operations within the instrumented module. This can occur, for example, when a Close is included in a Procedure or Main Program; if the Close is to be instrumented and tested, then the text of the Procedure or Main Program must be included also. Thus, instrumentation and testing of a single module is possible only if it is recognized that the module will be treated within the appropriate context.

APPENDIX B

DD-PATH DEFINITIONS FOR JOVIAL/J3

The JOVIAL/J3 language is relatively straightforward in its control structure, once the concept of an "invokable entity" is fully understood (see Appendix A). This Appendix presents detailed definitions of all of the possible forms of explicit DD-paths in JOVIAL/J3.

The recognition of DD-path sequences is based upon the recognition of key statement types which start these sequences in the programming language under consideration. In the JOVIAL/J3 language, the following statement types indicate the beginning of a DD-path sequence: PROC, CLOSE, the first executable statement of a main program, IF, IFEITH, ORIF, the GOTO which invokes a switch, and the END or terminating statement of a FOR loop. Examples of these DD-path sequences are shown below.

PROC STATEMENT

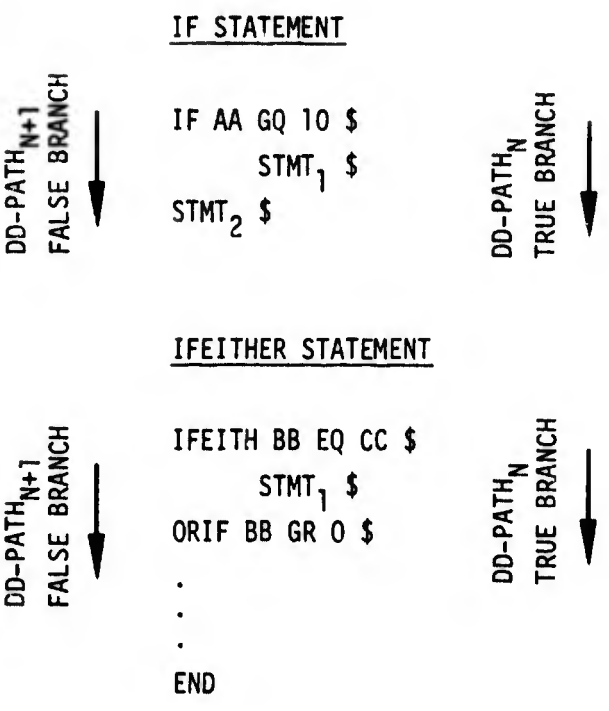

```
PROC P1 ( INPUT = OUTPUT ) $
BEGIN
      STMT1 $
      STMT2 $
      .
      .
      .
```

CLOSE STATEMENT

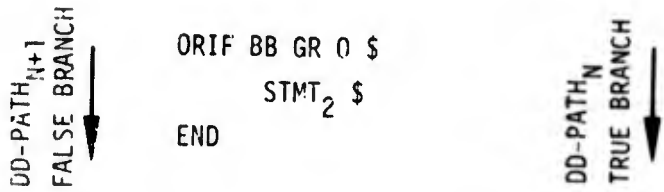

```
CLOSE C1 $
BEGIN
      STMT1 $
      STMT2 $
```

PROGRAM

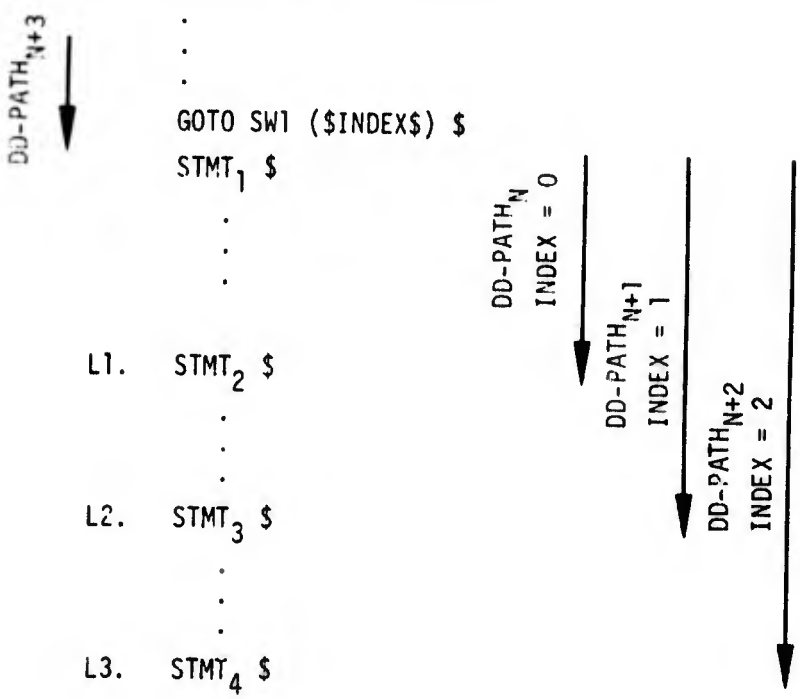

```
START $
      STMT1 $
      STMT2 $
```


IF STATEMENT
DD-PATH$_{N+1}$
FALSE BRANCH
IF AA GQ 10 $
STMT$_1$ $
STMT$_2$ $
DD-PATH$_N$
TRUE BRANCH
IFEITHER STATEMENT
DD-PATH$_{N+1}$
FALSE BRANCH
IFEITH BB EQ CC $
STMT$_1$ $
ORIF BB GR 0 $
.
.
.
END
DD-PATH$_N$
TRUE BRANCH

ORIF STATEMENT

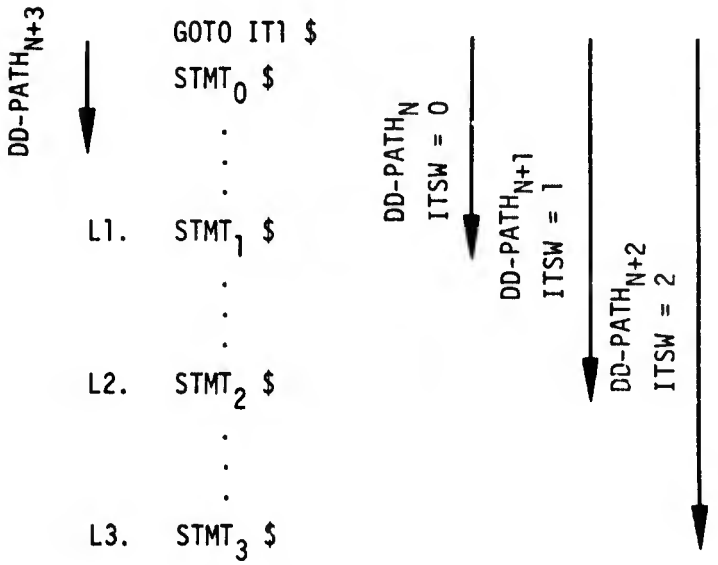


INDEX SWITCH GOTO

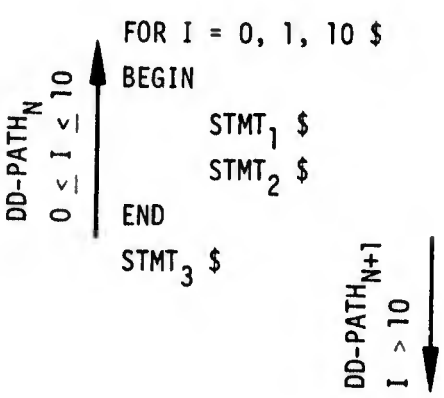


ITEM SWITCH GOTO

```
SWITCH IT1 (ITSW) = (0 = L1, 1 = L2, 2 = L3) $
        .
        .
        .
        GOTO IT1 $
        STMT0 $
        .
        .
        .
L1.     STMT1 $
        .
        .
        .
L2.     STMT2 $
        .
        .
        .
L3.     STMT3 $
        .
        .
        .
```



FOR STATEMENT



APPENDIX C

INSTRUMENTATION TEMPLATES FOR JOVIAL/J3

This appendix provides complete definitions of the instrumentation templates for JOVIAL/J3 program text, under the assumption that the instrumentation is to be performed by means of implanted subroutine calls. For each JOVIAL/J3 control-type statement, the revised text surrounding that statement which must be generated is presented in metalinguistic terms.

C.1 DD-PATH PROBE INSERTION

DD-path instrumentation is accomplished by inserting procedure invocations into the source module. When the module is executed, these invocations transfer control to a special auditing procedure which captures and records specific information describing the flow of control in the module. The placement of these procedure invocations in the module is done by identifying key JOVIAL statement types which indicate the beginning of DD-paths.

It should be noted that a unique procedure invocation is placed on each DD-path. This technique does not yield the minimum set of probes needed to identify the flow of control through the program, but it has the advantage of making the flow of control explicit in that a procedure invocation is executed every time a DD-path is executed. Note also that the insertion of procedure calls requires the reconstruction of some source module statements. This reconstruction is done in such a manner as to retain the semantics of the source module text.

The following information is recorded on the testing file by the auditing subroutine:

1. The name of the module being executed
2. The number of the DD-path being executed
3. The iteration level of the DD-path being executed
4. The system clock time at which the DD-path began execution

Examples of DD-path probe insertion for each key JOVIAL statement type are given below.

JOVIAL STATEMENT TYPE: PROCEDURE

STATEMENT KEY: PROC

STATEMENT SYNTAX:

```
PROC <NAME> [(<FORMAL PARAMETER LIST>)]$
<DECLARATIONS>$
BEGIN
      <PROCEDURE BODY>$
END
```


INSTRUMENTED SYNTAX:

```
PROC <NAME> [(<FORMAL PARAMETER LIST>)]$
<DECLARATIONS>
BEGIN
      PROBE(MODULE-NAME,DD-PATH1,LEVEL0, <TIME>)$
      <PROCEDURE BODY>$
END
```

JOVIAL STATEMENT TYPE: CLOSE

STATEMENT KEY: CLOS

STATEMENT SYNTAX:

```
CLOSE <NAME>$
BEGIN
      <CLOSE BODY>$
END
```

INSTRUMENTED SYNTAX:

```
CLOSE <NAME>$
BEGIN
      PROBE(MODULE-NAME,DD-PATH1,LEVEL0, <TIME>)$
      <CLOSE BODY>$
END
```

JOVIAL STATEMENT TYPE: IF

STATEMENT KEY: IF

STATEMENT SYNTAX:

```
IF <BOOLEAN FORMULA>$
      <STATEMENT BLOCK>$
```

INSTRUMENTED SYNTAX:

```
IFEITH <BOOLEAN FORMULA>$
BEGIN
      PROBE(MODULE-NAME,DD-PATH_N,LEVEL, <TIME>)$
      <STATEMENT BLOCK>$
END
ORIF 1 $
      PROBE(MODULE-NAME,DD-PATH_N+1,LEVEL, <TIME>)$
END
```

JOVIAL STATEMENT TYPE: IFEITH

STATEMENT KEY: IFEI

STATEMENT SYNTAX:

```
IFEITH <BOOLEAN FORMULA>$
      <STATEMENT BLOCK>$
```

INSTRUMENTED SYNTAX:

```
IFEITH <BOOLEAN FORMULA>$
BEGIN
      PROBE(MODULE-NAME,DD-PATH_N,LEVEL, <TIME>)$
      <STATEMENT BLOCK>$
END
```

JOVIAL STATEMENT TYPE: ORIF

KEY STATEMENT: ORIF

STATEMENT SYNTAX:

```
ORIF <BOOLEAN FORMULA>$
     <STATEMENT BLOCK>$
```

INSTRUMENTED SYNTAX:

```
ORIF <BOOLEAN FORMULA>$
BEGIN
     PROBE(MODULE-NAME,DD-PATH_{N+1},LEVEL, <TIME>)$
     :
     PROBE(MODULE-NAME,DDPATH_{N+2K-1},LEVEL, <TIME>)$
     PROBE(MODULE-NAME,DD-PATH_{N+2K},LEVEL, <TIME>)$
     <STATEMENT BLOCK>$
END
```

WHERE THIS ORIF IS THE KTH ORIF IN THE IFEITH-ORIF-END STRUCTURE.

JOVIAL STATEMENT TYPE: ORIF 1

STATEMENT KEY: ELSE

STATEMENT SYNTAX:

```
ORIF 1 $
     <STATEMENT BLOCK>$
END
```

INSTRUMENTED SYNTAX:

```
ORIF 1 $
BEGIN
     PROBE(MODULE-NAME,DD-PATH_N+1,LEVEL, <TIME>)$
     PROBE(MODULE-NAME,DD-PATH_N+3,LEVEL, <TIME>)$
     .
     .
     .
     PROBE(MODULE-NAME,DD-PATH_N+2K-1,LEVEL, <TIME>)$
     <STATEMENT BLOCK>$
END
END
```

WHERE K EQUALS THE NUMBER OF ORIF'S IN THE IFEITH-ORIF-END STRUCTURE.

JOVIAL STATEMENT TYPE: TWO-FACTOR FOR

STATEMENT KEY: FOR2

STATEMENT SYNTAX:

```
FOR <LOOP VARIABLE> = <INITIAL VALUE FORMULA>, <INCREMENT FORMULA>$
<STATEMENT BLOCK>$
```

INSTRUMENTED SYNTAX:

```
<TEMPORARY> = <INITIAL VALUE FORMULA>$
FOR <LOOP VARIABLE> = <TEMPORARY>, <INCREMENT FORMULA>$
BEGIN
      IF <LOOP VARIABLE> NQ <TEMPORARY>$
            PROBE(MODULE-NAME,DD-PATH_N,LEVEL, <TIME>)$
      <STATEMENT BLOCK>$
END
PROBE(MODULE-NAME,DD-PATH_N+1,LEVEL, <TIME>)$
```

JOVIAL STATEMENT TYPE: THREE-FACTOR FOR

STATEMENT KEY: FOR3

STATEMENT SYNTAX:

```
FOR <LOOP VARIABLE> = <INITIAL VALUE FORMULA>, <INCREMENT FORMULA>, <TERMINAL VALUE FORMULA>$
<STATEMENT BLOCK>$
```

INSTRUMENTED SYNTAX:

```
<TEMPORARY> = <INITIAL VALUE FORMULA>$
FOR <LOOP VARIABLE> = <TEMPORARY>, <INCREMENT FORMULA>, <TERMINAL VALUE FORMULA>$
BEGIN
      IF <LOOP VARIABLE> NQ <TEMPORARY>$
            PROBE(MODULE-NAME,DD-PATH_N,LEVEL, <TIME>)$
      <STATEMENT BLOCK>$
END
PROBE(MODULE-NAME,DD-PATH_N+1,LEVEL, <TIME>)$
```

JOVIAL STATEMENT TYPE: PARALLEL FOR

STATEMENT KEY: FORP

STATEMENT SYNTAX:

[FOR <LOOP VARIABLE$_I$> = <INITIAL VALUE FORMULA$_I$>$]*
FOR <LOOP VARIABLE$_J$> = <INITIAL VALUE FORMULA$_J$>, <INCREMENT FORMULA$_J$>, <TERMINAL VALUE FORMULA$_J$>$
[FOR <LOOP VARIABLE$_K$> = <INITIAL VALUE FORMULA$_K$>, <INCREMENT FORMULA$_K$>$]$^+$
<STATEMENT BLOCK>$

INSTRUMENTED SYNTAX:

[FOR <LOOP VARIABLE$_I$> = <INITIAL VALUE FORMULA$_I$>$]*
<TEMPORARY> = <INITIAL VALUE FORMULA$_J$>$
FOR <LOOP VARIABLE$_J$> = <TEMPORARY>, <INCREMENT FORMULA$_J$>, <TERMINAL VALUE FORMULA$_J$>$
[FOR <LOOP VARIABLE$_K$> = <INITIAL VALUE FORMULA$_K$>, <INCREMENT FORMULA$_K$>$]$^+$
BEGIN
    IF <LOOP VARIABLE$_J$> NQ <TEMPORARY>$
        PROBE(MODULE-NAME,DD-PATH$_N$,LEVEL, <TIME>)$
    <STATEMENT BLOCK>$
END
PROBE(MODULE-NAME,DD-PATH$_{N+1}$,LEVEL, <TIME>)$

JOVIAL STATEMENT TYPE: INDEX SWITCH GOTO

STATEMENT KEY: GTSW

STATEMENT SYNTAX:

```
GOTO <SWITCH NAME> ($<INDEX>$)$
```

INSTRUMENTED SYNTAX:

```
<TEMPORARY> = <INDEX>$
IFEITH <TEMPORARY> EQ. 0 $
      PROBE(MODULE-NAME,DD-PATH_N,LEVEL, <TIME>)$
ORIF <TEMPORARY> EQ 1 $
      PROBE(MODULE-NAME,DD-PATH_N+1,LEVEL, <TIME>)$
:
ORIF <TEMPORARY> EQ K $
      PROBE(MODULE-NAME,DD-PATH_N+K,LEVEL, <TIME>)$
END
GOTO <SWITCH NAME> ($<TEMPORARY>$)$
```

JOVIAL STATEMENT TYPE: ITEM SWITCH GOTO

STATEMENT KEY: GTSW

STATEMENT SYNTAX:

```
GOTO <SWITCH NAME> [($<OPTIONAL INDEX LIST>$)]$
```

INSTRUMENTED SYNTAX:

```
IFEITH <SWITCH ITEM> .EQ. <CONSTANT_0>$
      PROBE(MODULE-NAME.DD-PATH_N,LEVEL; <TIME>)$
ORIF <SWITCH ITEM> .EQ. <CONSTANT_1>$
      PROBE(MODULE-NAME,DD-PATH_N+1,LEVEL, <TIME>)$
:
ORIF <SWITCH ITEM< .EQ. <CONSTANT_K>$
      PROBE(MODULE-NAME,DD-PATH_N+K,LEVEL, <TIME>)$
END
GOTO <SWITCH NAME> [($<OPTIONAL INDEX LIST>$)]$
```

C.2 DIRECTIVE PROCESSING

The following directives are to be implemented for the JOVIAL Automated Verification System:

TRACE/OFFTRACE

ASSERT

EXPECT/TOLERANCE

The method of implementing directives is essentially the same as that of DD-path instrumentation; that is, the module source code is reconstructed to facilitate the capturing of the information required by the directive. One way in which to capture this information is to insert a procedure invocation in the source code which, when executed, implements the semantics of the directive. The use of this directives procedure allows the syntax and semantics of the directives to be changed easily. This method is illustrated in the examples which follow.

The general format for directives is:

".<directive keyword> <JOVIAL TEXT>"

The TRACE directive allows the user to observe the effect of computation upon the variables in the program at execution time. The TRACE directive can be implemented by declaring a trace buffer which contains the module being executed, the statement number of the statement currently being executed, the name of the variable being traced, and the current value of the variable. At execution time, when a variable being traced is encountered, the trace procedure is invoked and the trace buffer can be constructed and written to the output file.

The ASSERT directive provides the user with the means to monitor the truth value of a particular logical condition. It can be implemented by prefacing the JOVIAL statement block containing the logical expression with a procedure invocation which transfers control to an auditing procedure which prints the current truth value of the logical expression on the output file.

The EXPECT directive provides the user with a means to monitor the value of selected variables and detect when they exceed certain boundaries. The EXPECT directive can be implemented by comparing each occurrence of the selected variable with its declared range and, if the range is exceeded, invoking a special auditing procedure. The auditing procedure would then print on the output file the module being executed, the statement number of the statement being executed, the specific variable, and whether it is greater than or less than the specified tolerance.

The TOLERANCE directive specifies a range within which the value of the variable named in the EXPECT directive will be assumed to be correct. That is, in terms of the other EXPECT directive:

<LOW VALUE> = <VALUE> - <TOLERANCE>
<HIGH VALUE> = <VALUE> + <TOLERANCE>

Examples of the three directives follow.

DIRECTIVE TYPE: TRACE/OFFTRACE

DIRECTIVE FORMAT:

```
".TRACE [<NAME> [, <NAME>]]"
<STATEMENT BLOCK>$
".OFFTRACE"
```

INSTRUMENTED TEXT:

```
".TRACE AA"
AA = ... $
TRACE (<MODULE NAME>, <STATEMENT #>, 2H(AA), AA, <TYPE>)$
.
.
.
".OFFTRACE"
```

DIRECTIVE REPORTING FORMAT:

```
<MODULE NAME> <STATEMENT #> <NAME> = <VALUE>
```

DIRECTIVE TYPE: ASSERT

DIRECTIVE FORMAT:

```
".ASSERT <LOGICAL EXPRESSION>"
<STATEMENT BLOCK>$
```

INSTRUMENTED TEXT:

```
".ASSERT <logical expression>"
IFEITH <LOGICAL EXPRESSION>$
      ASSERT (<MODULE NAME>, <LOGICAL EXPRESSION>, 1, <STATEMENT #>)$
ORIF 1 $
      ASSERT (<MODULE NAME>, <LOGICAL EXPRESSION>, 0, <STATEMENT #>)$
END
<STATEMENT BLOCK>$
```

DIRECTIVE REPORTING FORMAT:

<MODULE NAME> <STATEMENT #> <LOGICAL EXPRESSION> = {TRUE / FALSE}

DIRECTIVE TYPE: EXPECT/TOLERANCE

DIRECTIVE FORMAT:

".EXPECT <NAME> = <VALUE>[,TOLERANCE = <VALUE>]"

OR

".EXPECT <NAME> = <LOW VALUE>, <HIGH VALUE>"

INSTRUMENTED TEXT:

```
".EXPECT <NAME> = <LOW VALUE>, <HIGH VALUE>"
<NAME> = ... $
IFEITH <NAME> LS <LOW VALUE>$
     EXPECT (<MODULE NAME>, <NAME>, <CURRENT VALUE>, <LOW VALUE>
          2H(LS), <STATEMENT #>)$
ORIF <NAME> GR <HIGH VALUE>$
     EXPECT (<MODULE NAME>, <NAME>, <CURRENT VALUE>,
          <HIGH VALUE>, 2H(GR), <STATEMENT #>)$
END
```

DIRECTIVE REPORTING FORMAT:

<MODULE NAME> <STATEMENT #> <NAME> = <CURRENT VALUE> {LS <LOW VALUE> / GT <HIGH VALUE>}

APPENDIX D

GLOSSARY

AVS. Automated Verification System.

AVS Database. In an AVS, the collection of information maintained internal to the AVS which contains all pertinent data about all modules known to the AVS.

Actual Parameter. In an invocation of a module, the set of variable names passed to the invoked module in the actual parameter list.

Ancestor. A level-i path class (at level i) which depends on some other level-i path class (at level i+1).

Arc. In a directed graph, the oriented connection between two nodes. Also, called an edge.

Automated Verification System. A system for the analysis of software systems oriented toward systematic, comprehensive testing (exercise) as a means to perform software validation.

Bottom-up Testing Strategy. A systematic testing philosophy which seeks to test those modules at the bottom of the invocation structure earliest.

Cardinality. This term is applied to a DD-path class, a level-i path class, or any sequence of DD-paths, to refer to the total number of flows represented by the class.

Collateral Testing. Testing coverage which is achieved indirectly, rather than as the direct object of a testcase generation activity.

Communication Space (of a module). 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.

Coverage Measure. See testing coverage measure.

DD-Path. Decision-to-decision path; the set of statements in a module which are executed as the result of the evaluation of some predicate (conditional) within the module. The DD-path should be thought of as including the sensing of the outcome of a conditional operation and the subsequent statement executions up to, and including, the computation of the next predicate value but not including its evaluation.

DD-Path Class. A set of parallel DD-paths, one of which is designated as the principal DD-path for the class.

DD-Path Instrumentation. The process of producing an altered version of a module which is logically equivalent to the unmodified module but which contains calls to a special data collection subroutine which accepts information as to the specific DD-path sequence incurred in an invocation of the module.

DD-Path Predicate. A logical formula involving variables and constants known to a module and, possibly, the values .TRUE. and .FALSE., which must be satisfied for the DD-path to be executed.

Decision Node. A node in the program graph which corresponds to a decision statement within the program.

Decision Statement. A statement in which an evaluation of some predicate is made which (potentially) affects the subsequent execution behavior of the module.

Decision-to-Decision Path. See DD-path.

Descendent. A level-i path class (at level i) which depends on some other level-i path class (at level i-1).

Digraph. Short for "directed graph."

Directed Graph. A set of nodes interconnected by oriented arcs. An arbitrary directed graph (digraph) may have many entry nodes and many exit nodes.

Edge. In a digraph, the oriented connection between two nodes. Also called an arc.

Entry DD-Path. An entry DD-path is one which has no predecessors, a situation which can occur only at the entrance (i.e., invocation point) of a module.

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.

Entry Structure. The entry structure of a program digraph is the set of DD-paths which can be reached only by invocation of the program and cannot be reached by flow within the internal structure of the program. There is always at least one DD-path in the entry structure of a module. DD-paths which reside in the entry structure of a module can be executed at most once per invocation.

Equivalence Class of Program Flow. Two program flows (sequences of DD-paths) belong to the same equivalence class of program flow if they involve sequences of DD-paths which touch the same sets of decision nodes.

Executable Statement. A statement in a module which is executable in the sense that it produces object code instructions.

Exit DD-Path. An exit DD-path is one for which there are no successor DD-paths. This occurs only when the consequence of the DD-path is an exit from the module.

Exit Node. In a digraph, a node which has more than one inway, but has zero outways. An exit node has an out-degree of zero, and non-zero in-degree.

Exit Structure. The exit structure of a program digraph is the set of DD-paths which, if executed, lead unalterably to termination of program flow without involving subsequent repetition of any DD-path.

Explicit Program Predicate. A program predicate whose formula is displayed explicitly in the program text. For example, a single conditional always involves an explicit program predicate.

Flow. A particular sequence of DD-paths, also called a level-i path.

Flow Class. A particular sequence of DD-path classes, also called a level-i path class.

Formal Parameter. For an invokable element of program text, the set of variable names which are assigned value outside of the program text.

FORTRAN. American National Standards Institute X3.9 FORTRAN.

Functional Specifications. A set of behavioral and performance requirements which, in aggregate, determine the functional properties of a software system.

Functional Testcases. A set of testcase datasets for software which are derived from structural testcases.

Global Variable. In a module, a global variable is one which receives a value as the result of actions outside the module.

Implicit Program Predicate. A program predicate whose formula is not displayed in the program text. For example, the outcome of a read-parity error, in terms of future program behavior, is controlled by an implicit program predicate.

In-Degree. In a digraph, the number of inways for a node.

Incompatible DD-Paths. An ordered pair of DD-paths are incompatible if some computation performed on the first DD-path makes it impossible to execute the second DD-path.

Independent DD-Path Pair. A pair of DD-paths is (sequentially) independent when there are no assignment actions along the first DD-path which change any of the variables which are used in the predicate of the second DD-path.

Input Space. The input space of a module consists of that subset of a module's communication space which (1) can be altered externally to the module, and (2) is (potentially) used within the module in a way that affects its execution.

Invocation Point. The invocation point of a module is the first statement in the module (in FORTRAN, a function, subroutine, or program), or, if the module has multiple entry points, an entry statement.

Invocation Structure. The hierarchy of invocations of one module by another within a software system.

Inway. In a digraph, an arc (edge) arriving at a node.

Iteration Level. The level of iteration relative to the invocation of a module. A zero-level iteration characterizes flows with no iteration. A one-level iteration characterizes program flow which involves repetition of a zero-level flow.

Iteration Structure. The tree of level-i path classes for a module. All possible patterns of program operation are represented by the set of all subtrees of that tree.

Iterative Flow. Iterative flow is represented by a sequence of DD-paths (path classes) with the property that some DD-path belonging to the sequence can be executed more than one time.

JOVIAL. Unless further specified, any of the dialects of the family of JOVIAL languages.

JOVIAL/J3. The JOVIAL programming language, J3 subset, as defined in Air Force Manual AFM-100-24.

Level-i Path Class. A level-i path class describes a collection of level-i paths which differ only in the selection, within the level-i path class, between alternative members of included DD-path classes. If all included DD-path classes have cardinality exactly 1, then the level-i path class is a singleton.

Level-i Path Class Cardinality. The cardinality of a level-i path class is the product of the cardinalities of DD-paths resident on the level-i path class.

Level-i Path Class Tree. The tree structure which arises from a module by linking to the root of the tree all level-0 path classes and, thereafter, all level-i, $i > 0$, path classes, in order of their ancestry.

Logically Possible DD-Path Sequence. A sequence of DD-paths is logically possible if there is a setting for the input space, relative to the first DD-path in the sequence, which permits the sequence to execute.

Logically Impossible DD-Path Sequence. A DD-path sequence is logically impossible if there is no collection of settings of the input space, relative to the first DD-path in the sequence, which permits the sequence to execute.

Memory. A module is said to have memory if there is some interior code condition which makes it possible to execute some DD-path only by making two or more invocations of the module.

Memory Space. Those cells known to a module which allow it to have memory (see memory).

Module. A separately invokable element of a software system.

Node. A number assigned to a place within a program text. Generally, nodes are assigned only to executable statements.

Node Number. A unique node number is assigned at various critical places within each module. The node number is used to describe potential and/or actual program flow.

Non-Executable Statement. A declaration or directive within a module which does not produce (during compilation) object code instructions directly.

One-Shot Module. A module which has an empty input space and an empty memory space. A one-shot module can be executed any number of times, but produces identically the same execution behavior in each case.

Out-degree. In a digraph, the number of outways of a node.

Output Space. The output space of a module consists of the collection of variables, including file actions, which are (or could be) modified by some invocation of the module.

Outway. In a digraph, an arc (edge) leaving a node.

Parallel DD-Path. A parallel DD-path is one which has the same starting and ending nodes as some other DD-path (with which it is parallel) and which results from evaluation of the same predicate. A parallel DD-path has at least one statement which is not in common with any DD-path with which it is parallel.

Predicate. A formula involving variables, constants, and relational operators, which can be evaluated to .TRUE. or .FALSE..

Program. See module.

Program Predicate. See predicate.

Program Text. The set of statements, executable and non-executable, which make up a module. Program text is expressed in a programming language.

Program Validation. The process of developing and verifying the correspondence between an implemented software system and the set of functional specifications which correspond to it.

Reach Sequence. Relative to an ordered sequence of level-i path classes, the DD-path sequence leading to the last DD-path present on the highest-level level-i path in the sequence, but not repeating any DD-path.

Return Variable. An actual or formal parameter for a module which is modified within the module.

Singleton Level-i Path Class. A singleton level-i path class is a level-i path class with cardinality of 1.

Software System. A collection of modules, possibly organized into components and subsystems, which solves some problem.

Software Validation. See program validation.

Structural Testcases. A set of testcase patterns, derived from the iteration structure of a module (or a collection of modules). The combination of a structural testcase and appropriate program input data results in a functional testcase.

Terminal-Branch Level-i Path (Class). A level-i path (class) which has no descendents in the level-i path (class) tree.

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.

Testcase. See test.

Testcase Dataset. A testcase dataset is a specific set of values for variables in the communication space of a module which are used in a test.

Testing Coverage Measure. A measure of the testing coverage achieved as the result of one unit test, usually expressed as a percentage of the number of DD-paths within a module which were traversed in the test.

Testing Stub. A testing stub is a module which simulates the operations of a module which is invoked within a test. The testing stub can replace the real module for testing purposes.

Testing Target. The current module (system testing) or the current DD-path (unit testing) upon which testing effort is focused.

Testing Verification. The process of verifying that a set of functional test-cases meets the structural testing goals for which it was designed.

Top Down Testing Strategy. A systematic testing philosophy which seeks to test those modules at the top of the invocation structure earliest.

Unit Test. See test.

Unreachability. A statement (or DD-path) is unreachable if there is no logically obtainable set of input-space settings which can cause the statement (or DD-path) to be traversed.

REFERENCES

1. Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s (CCIP-85), SAMSO/XRS-71-1, February 1972.

2. USAF Software Manual, AFM-100-24, "Standard Computer Programming Language for Air Force Command and Control Systems," 15 June 1967.

3. F. T. Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal, 1972.

4. H. Mills, "Top-Down Programming in Large Systems," Debugging Techniques in Large Systems, R. Rustin (Editor), Prentice-Hall, 1971.

5. R. L. London, "Bibliography on Proving the Correctness of Computer Programs," Machine Intelligence, 1970.

6. Elspas, Levitt, Waldinger, and Waksman, "An Assessment of Techniques for Proving Program Correctness," ACM Computing Surveys, June 1972.

7. Don Reifer, Computer Program Verification/Validation/Certification, Aerospace Corporation, 30 May 1974.

8. E. F. Miller, Jr. "Interactive Real-Time Testing and Performance Analysis," Workshop on System Performance Analysis, Argonne, Illinois, October 6-7, 1971.

9. C. W. Perkins, et al., A Discussion of Data Processing Systems Analysis for Ballistic Missile Defense, General Research Corporation IMR-1386, September 1970.

10. E. F. Miller, A Survey of Major Techniques of Program Validation, General Research Corporation RM-1731, October 1972.

11. M. R. Paige, Methodology for Generating Paths in a Directed Graph, General Research Corporation IM-1816, November 1973.

12. E. F. Miller, et al., Structurally Based Automatic Program Testing, presented at EASCON '74, Washington, D.C., 7-9 October 1974.

13. E. F. Miller, et al., "Automatic Generation of Testcase Data," 1975 International Conference on Reliable Software.

14. J. Sullivan, Measuring the Complexity of Computer Software (MCCS), Mitre Corporation, June 1973.

MISSION
of
Rome Air Development Center

RADC is the principal AFSC organization charged with planning and executing the USAF exploratory and advanced development programs for information sciences, intelligence, command, control and communications technology, products and services oriented to the needs of the USAF. Primary RADC mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, and electronic reliability, maintainability and compatibility. RADC has mission responsibility as assigned by AFSC for demonstration and acquisition of selected subsystems and systems in the intelligence, mapping, charting, command, control and communications areas.