AD-A062 194 NAVAL POSTGRADUATE SCHOOL MONTEREY CALIF F/G 9/2
EXPERIMENTS IN DOMONSTRATING THE CORRECTNESS OF SOFTWARE.(U)
SEP 78 C W MONK

UNCLASSIFIED NL


OF
2194

(2) LEVEL II

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

Experiments in Demonstrating the Correctness of Software

by

Carl Warren Monk, Jr.

September 1978

Thesis Advisor: Norman Schneidewind

Approved for public release; distribution unlimited.

11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, CA 93940	12. REPORT DATE (11) September 1978
	13. NUMBER OF PAGES 159
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, CA 93940	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

(12) 168 p.

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

current literature concerning software testing and formal proofs of correctness, select a well-documented program of intermediate size for experimentation, apply selected verification methods to that program, and finally to compare the results of the several experimental demonstrations of correctness. The experiments conducted included a proof of correctness and dynamic testing with test data cases selected by a condition table method, by path analysis, and by structural decomposition of the program.

Approved for public release; distribution unlimited

EXPERIMENTS IN DEMONSTRATING THE CORRECTNESS OF SOFTWARE

by

Carl Warren Monk, Jr.
Lieutenant Commander, United States Navy

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
September 1978

Author Carl W. Monk Jr.

Approved by: Norman Lehrfeld
Thesis Advisor

Second Reader

Chairman, Department of Computer Science

Dean of Information and Policy Sciences

ABSTRACT

This thesis was undertaken to examine the post-development process of verifying the correctness of software programs, specifically to evaluate the effectiveness and practicality of several proposed methods of verification. Of interest were the degree to which utilization of a given method can be said to demonstrate correctness and the feasibility for general use of that method. The method of research was to study current literature concerning software testing and formal proofs of correctness, select a well-documented program of intermediate size for experimentation, apply selected verification methods to that program, and finally to compare the results of the several experimental demonstrations of correctness. The experiments conducted included a proof of correctness and dynamic testing with test data cases selected by a condition table method, by path analysis, and by structural decomposition of the program.

TABLE OF CONTENTS

TABLE OF SYMBOLS

$\wedge$	Logical conjunction ("and")
$\vee$	Logical disjunction ("or")
$\neg$	Logical negation ("not")
$\Rightarrow$	Conditional connective ("implies")
$\Leftrightarrow$	Biconditional connective ("if and only if")
$\forall x$	Universal quantifier ("for all x")
$\exists x$	Existential quantifier ("there exists an x such that")
ϵ	Set membership connective ("belongs to")
$\subseteq$	Proper subset connective ("is included in")

ACKNOWLEDGEMENTS

The continued support and critical help provided by thesis advisor Professor N. F. Schneidewind is gratefully acknowledged.

The preliminary text of this thesis was prepared using the excellent software of the UNIX operating system, operating on a PDP-11/50 of the Computer Science Laboratory. Software verification experiments were conducted using the IBM System 360/67 of the Naval Postgraduate School's W. R. Church Computer Center.

I. INTRODUCTION

This first chapter reviews the environment in which software verification methods are employed and presents generally accepted definitions and descriptions of software testing.

A. THE SOFTWARE PREDICAMENT

1. Scope of Software Development

Ware Myers has characterized the current state of expanding programming applications as a serious software predicament (28). The cost of computer hardware has declined significantly over the last decade, making more and more applications feasible for automation. In 1973 it was estimated that between $15 and $25 billion were being spent annually on software development and maintenance (3). The Department of Defense spent about $3 billion in 1976 on computer software (34), and has been doubling the number of functions performed by software every few years, primarily in converting weapons systems components from analog to digital.

2. Problems in Software Development

Unfortunately for the growing user communities, and despite the expectations raised with the advent of modern programming practices (MPP), all is not well with software development.

a. Our ability to estimate time and resources required for the design and development of software has not appreciably increased.

b. Most major software projects have required that

special support tools be developed, that new automated testing aids be generated, or that a new language capability be acquired; in short, there is inadequate transfer of technology between projects within single organizations, and even less among organizations.

c. An unreasonable share of software effort has been expended on maintenance of existing programs. Estimates of 75% or more of a company's programming effort being devoted to maintenance have not been uncommon.

d. The rate of increase in programmer productivity has not kept pace with the introduction of improved management and programming methodologies, let alone with the rapidly increasing hardware capabilities; in fact, we are still struggling to learn to measure this productivity.

e. The quality of software has been less than desired. While there is no agreement on how to quantify the quality of software, many shortcomings are apparent (4): software is still difficult to read, understand and modify; programs are frequently hard to use properly and easy to misuse; they are often lacking in sufficient generality to be used in several applications or transported to different machines; and program reliability has been disappointing.

The applications planned for automation require bigger and more complex systems than ever before. Dijkstra pointed out that complex systems which are perhaps one thousand times larger than existing system cannot be constructed with the same techniques used to compose the smaller systems; order-of-magnitude

leaps in scale cannot be treated as gradual increases that can be handled by simple induction (10). A systematic and scientific method to accomplish functions of such magnitude is required.

Improvements in the reliability of software (the extent to which programs can be expected to satisfactorily perform their intended functions) are desired without incurring the staggering costs of totally exhaustive testing. That reliability needs improvement seems obvious. The relative level of programming effort devoted to maintenance of existing programs bears testament to the existence of errors in programs presumably tested and certified correct prior to their release for use. Even diligent application of the modern programming practices by talented programmers has not necessarily produced reliable software. Gerhart and Yelowitz (15) identified errors in programs that were published to demonstrate these MPP, errors in specifications, in construction, and in programs formally "proven" correct!

B. SOFTWARE ENGINEERING

1. Current Trends

A discipline has arisen, referred to as software engineering, which draws from established principles of science and engineering in attempting to formally define a systematic approach to software development. While the goals of this discipline are broad in scope, the application of software engineering toward attainment of correct software is of particular interest.

Recent trends in software engineering have emphasized the role of design and implementation in preventing errors over the role of testing to detect errors for correction. These techniques include:

a. More thorough analysis and definition of requirements: The lack of adequate system specifications has been a major cause of software shortcomings (12).

b. Modular design of programs.

c. Top-down design and implementation techniques.

d. Management practices such as the use of the team concept (possibly including the chief programmer team concept), project workbooks, formal documentation requirements, formal and peer review, and structured walkthrough of code.

e. Structured programming: This concept is at the core of modern programming practices. While many managers complain that they encounter resistance in implementing structured programming in their organization (5), there is growing data to support the proposition that structured programming techniques can produce more reliable and more cost effective software systems (28).

2. Need for Post-Development Testing

The experience of the research reported in later chapters of this thesis confirmed the need for greater relative emphasis on design and implementation techniques as compared to verification techniques. Much of the process of constructing a formal proof of correctness of the sample program selected for experimentation was clearly a duplication of the design and implementation

effort; it was necessary to analyze the completeness and consistency of the program specifications, verify the particular modular design selected, justify the control flow of each procedure, and determine the reasoning behind the choice of particular stopping criteria for program loops. To a somewhat lesser extent the same was true of the most rigorous of the dynamic testing strategies employed (Chapter III, Section C). The duplication of effort is simply not cost effective.

The successful and widespread use of as yet undeveloped techniques that will ensure development of correct software programs lies many years ahead. For the foreseeable future, programs under development will need to be subjected to an effective and practical ex post facto verification process. Diligent application of existing program testing techniques has been shown to enhance the software development process (1,2,6). Significant efforts are required in applying the discipline of software engineering toward refinement or replacement of the verification methods now in use.

C. DEFINITIONS IN SOFTWARE TESTING

Methods of testing or proving the correctness of software have been developed until quite recently on an ad hoc basis, to fill particular needs in the field rather than to build a complete scientific model for the verification process. The terms used to describe the activities involved in demonstrating correctness have likewise evolved sporadically. The following definitions are widely but not universally accepted.

1. Correctness of Software

The central issue in software testing, verification, validation, or similar terms is a demonstration that the software at hand is correct; i.e., that the given software system produces the intended results. The assertion that a program is correct is in effect a statement that it performs precisely those functions, and only those functions, called for in its specifications, and furthermore that its specifications are an accurate representation of a design suitable to satisfy the intended requirement. The term "correctness" has a connotation inclusive of and stronger than the notion of reliability of software, which was defined earlier as the extent to which programs can be expected to satisfactorily perform their intended functions. A thorough demonstration of correctness involves more than showing that a program satisfies some written specifications; it involves an analysis of the completeness, clarity, and consistency of the specifications as well.

2. Debugging

This term has frequently been used synonymously with testing, but should be distinguished in the following sense: testing is a process to uncover errors, while debugging begins when an error has been detected and is the process of isolating and correcting these known errors. A succinct statement of this distinction is:

> When debugging is completed the program definitely solves some problem. Testing seeks to guarantee that it is the problem that was intended (18).

3. Testing

Testing has just been distinguished from debugging. In Reference 24 five major activities in testing were defined, using the terms most commonly accepted in practice:

a. Verification is the process of establishing logical correctness of a program (i.e., correspondence with formal requirements) in a test environment. Verification is typically accomplished by actual program execution using selected test data. This process of dynamic execution is the single activity most often intended by use of the general term testing.

b. Validation is the process of establishing logical correctness in some given external environment, although not necessarily in the operational environment.

c. Proving is the mathematical establishing of logical correctness without regard to the environment; it does not involve program execution.

d. Certification is an organizational endorsement that a program meets certain standards of correctness and effectiveness in a useful environment.

e. Performance testing is the demonstration of non-logical properties, usually execution timing and throughput capability.

4. Verification and Validation

Verification and validation is a currently popular term used to describe the processes involved in testing software prior to user acceptance; used as such the term actually encompasses to some degree all five of the above testing activities

(with the frequent exception of proving).

5. Scales of Testing

During the development and operation of a software system, differing scales of testing must be performed. Traditional scales of testing are unit testing, integration testing, and regression testing.

a. Unit Testing

Unit testing is the testing of the independent modules comprising the functional decomposition of a large system. Testing at the unit level involves examination of the internal logic of the module to ensure that the module's effects on the larger program containing it are consistent with those effects required by the specification, and verification of the assumptions made within the module about the larger program. Because specifications are frequently ambiguous, unit testing often results in reexamination of the unit specification.

The two methods employed in unit testing are functional (black-box) tests, which are based on no knowledge of the internal structure of the program, and logical tests which are based on program structure. Selection of test data which are appropriate for an ideal test (as described in Chapter II, Section A) is difficult or impossible for black-box testing because it is impossible to distinguish data that are treated similarly or differently internal to the black-box. Therefore program verification tests by the developer are nearly always based on program structure. Acceptance testing by the user is generally functional in nature of necessity.

b. Integration Testing

Integration testing is conducted to determine the extent to which a system meets its specifications in an environment similar to its working environment. The focus of integration testing is on module interaction as opposed to internal module operations. When unit testing has been conducted thoroughly beforehand, integration testing is primarily a verification that modules do not modify those relationships in the environment that the specification states must be preserved (e.g., that protected portions of memory have not been affected and that global variables have not been modified in an undesirable or unanticipated manner), and a thorough test of the consistency of the specification itself. Were the techniques of formal definition of requirements and automated verification of specifications more developed, integration testing would be less than the crucial and expensive effort it so often has become.

The relationship between unit and integration tests is formalized by the following theorem which is the basis for demonstrating the correctness of software by testing or proving:

> If it can be demonstrated that each module in a system meets its specifications assuming only that all submodules meet their specifications, then the entire system is correct (16).

It should be noted that the demonstration required by the above statement is essentially a verification that the interactions of modules are consistent with the specifications.

There is controversy over the best strategy for sequencing unit and integration tests. The classical strategy of bottom-up testing proceeds from unit to integration tests as lower-level

modules are completed, and requires test program drivers to run lower-level module tests until the calling modules are ready to be incorporated. Common criticisms of this strategy are the duplication of effort in writing driver programs, the cost involved since execution of higher-level modules involves reexecution of the lower modules, and most severely the fact that integration errors if present are discovered at a late stage in development, thus inflating the cost of their correction.

The top-down testing strategy is much heralded as an integral part of the top-down design technique, but it is little practiced. Its benefits are purported to be the early detection of specification errors relating to interfacing, since the high-level program skeleton is coded and executed first, with simple dummy programs required as sub-program stubs. These stubs are to be replaced by the actual lower-level modules as they are written, thus facilitating the check-out of new modules one at a time while continually verifying the correctness of module interfaces. In introducing the concept of a built-in package of "test procedures" deliverable with a software product, Panzl contends that neither of the above goals are well served by the top-down strategy (30). He states that top-down testing discourages thorough testing of lower-level modules because they are never executed directly; in fact, it is often difficult to find an input data combination that will force execution of a desired submodule.

In practice a mix of top-down and bottom-up strategies has been used. Unfortunately, a common choice of strategies is to

defer all testing until the entire system can be tested. This all-at-once strategy is costly in execution time because all modules must always be present, and costly in effectiveness because the intellectual task of isolating errors is greatly magnified when a systematic strategy is missing. However, it is the most prevalent testing strategy today (30). In fact, the experiments in testing and proving which are reported in the following chapters of this thesis were performed in an all-at-once fashion by virtue of the fact that a program which had completed development was the subject of the experiments. Even so, elements of bottom-up testing were apparent in several of the strategies used.

c. Regression Testing

Regression testing is the reverification and revalidation of software after adding new capabilities or after performing maintenance to correct errors discovered in operation. Its purpose is to verify that the desired modification and none other has been made. Regression testing has until recently received very little formal attention (12), which is puzzling in light of the previously mentioned estimates of the percentage of programming effort devoted to maintenance. A simple management technique to enhance regression testing is to ensure that the test cases produced during development testing are collected and documented as a package, and maintained to be reexecuted after maintenance. If extensive maintenance is performed, additional analysis of the thoroughness of test achieved with the saved data should be conducted. While the test procedures suggested by Panzl (30) add to the effort of development, their

value at regression test time merits careful consideration of the idea.

D. APPROACHES TO DEMONSTRATING CORRECTNESS

As suggested earlier, testing is a discipline that has been learned through application, with little formal basis until very recently. The first major collection of testing concepts resulted from a 1972 conference at the University of North Carolina, the proceedings of which were published in book form by W. C. Hetzel (18). The first significant attempt to establish a mathematically-based theory of program testing was a paper by John B. Goodenough and Susan L. Gerhart (17); this theory is discussed in Chapter II.

In the development of testing methods, two complementary approaches arose, static analysis and dynamic testing. Methods that have been employed in an attempt to show correctness have ranged from purely static (e.g., formal proof of correctness) to purely dynamic (e.g., execution of the programs with randomly selected test data), although usually a combination of the two approaches has been used.

1. Static Analysis

a. Capabilities

Static analysis refers to a wide range of activities that can be performed without program execution (although more and more such activities involve automated analysis of source programs by software tools). Quite often static analysis is perfomred prior to live testing to help in test planning and test data preparation. The technique can in itself detect errors

in logic (such as uninitialized variables) and questionable software practices (such as initialized but never referenced variables). It can also be used as a means of enforcing programming standards.

The most familiar form of static analysis is program checking, or desk checking. While automated static analysis techniques of significant capability are being developed, thorough desk checking is still an efficient method for insuring software correctness. When formalized by management in effective peer review or structured walkthrough policies, program checking is "very effective" in reducing errors (1). Program checking can detect syntax errors, undeclared variables, unreachable program segments, etc.

Directed graphs of program control flow and data flow are common tools of static analysis, and are the basis for the path analysis techniques of dynamic testing. Graphs of data flow lend themselves to detection of errors in initialization and referencing of data. Control flow graphs provide visual evidence of program adherence to structured programming practices and offer several measures of program complexity. Considerable study of the relationship between program structure and complexity and resultant error characteristics has been conducted at the Naval Postgraduate School (most recently reported in Ref. 33). A strong relationship has been found to exist between complexity and errors, suggesting that complexity measures may be used to establish programming standards (note that constructs of structured programming have lower complexity measures than

non-structured constructs) and to indicate how to allocate scarce testing resources.

b. Automated Aids

Particularly when dynamic testing is to be performed with the number of paths tested as a criterion, static analysis techniques are used to aid in developing test cases, frequently by automation. Fairley (12) suggests that static analysis algorithms (including cross-reference tables, numbers of occurrence of statement types, number of subprogram calls, graph analysis, etc.) can and should be included in compilers. Ramamoorthy et al. (32) discuss the techniques and problems involved in automated generation of test data inputs selected to satisfy varying requirements for coverage of the branches of a program graph, and describe a prototype generator included in their Fortran Automated Code Evaluation System (FACES).

c. Limitations

Whether test data is generated manually or by automated means, there is unfortunately no reliable way of automating the computing of the correct output. Needless to say, dynamic testing presupposes a known (or at least bounded) output, and specifications must be available for each program tested. A limitation in the degree of testing which is feasible is frequently imposed by the difficulty of determining the desired output.

A serious limitation of static analysis, and particularly of test case generators, is the decidability problem (12). Algorithms may be easily written to identify all syntactic

paths of control flow in a program, but it is not possible to algorithmically determine the semantic paths (those syntactic paths which can in fact be executed for some set of input data). Therefore it is not possible for all programs to determine whether some statements (including termination statements) can be executed for any input data. In these cases dynamic testing or the more difficult techniques of symbolic execution or proofs of correctness are often used to decide the question.

2. Dynamic Testing

Dynamic testing is the process of actual program execution to provide evidence upon which some conclusion may be reached as to the correctness of the software. Applications of theoretically-based testing methods (as described in Chapter II) have not yet countered Dijkstra's pronouncement that "Program testing can be used to show the presence of bugs, but never to show their absence!" (10). Nonetheless, dynamic testing has been and will remain the most common evidence proffered to show program correctness or reliability.

a. Selecting Test Cases

The critical activity during dynamic testing is the selection of test cases. Intuitively it is desirable to select a set of test cases which are representative of the actual input domain the program will have to contend with during operation. The principle guide in selecting test cases has been to test for the likely kinds of errors in the program, particularly in the program modules deemed most critical to proper program operation. Since it is not possible to anticipate all the

possible errors, it is unlikely that this principle alone can be relied on to select a suitable set of test data. Selection of a data set truly representative of the input domain is particularly difficult without knowledge of the internal structure of the program (i.e., when doing black-box testing), because it is then impossible to distinguish data that are treated similarly internally. Knowledge of the program structure can be used to identify the complex portions of the program which should be subjected to the most thorough testing, and to help identify the groups of data that are handled similarly and thus aid in selecting those cases representative of certain subsets of the input domain. However, use of this knowledge is of limited value in that there is no certainty that the program structure is a correct representation of the conditions required for correct operation of the program (16).

b. Thoroughness of Tests

The thoroughness of test, or degree of test coverage, is intended to provide a measure of the reliability of the testing process. The more thorough the test, the less probable that undetected errors remain. Unfortunately, no perfect quantitative measure of test thoroughness has been recognized, although it is clear that the criteria used to select test cases will determine the thoroughness of test.

Typically, estimates of test thoroughness have been based on a count of the source statements executed or the program control paths traversed. While a test which causes the execution of all program source statements may appear thorough, there are

numerous counterexamples which demonstrate the actual lack of thoroughness (for example, in Ref. 22). Since there are typically an infinite number of possible sequences of statement execution, any finite tests which execute each program statement at least once cannot be said to have tested all possible sequences, and thus may fail to reveal all program errors. The path analysis strategy of dynamic testing involves execution of selected control flow paths in the program under test. Because the number of control flow paths may be very large or infinite (due to the presence of loops), practical path analysis strategies are limited to execution of some subset of the total paths. Huang defined a "minimally thorough" test as one which caused at least one traversal of every branch or arc in the program's control flow graph (flowchart) (22); however, such a test cannot assure detection of all errors. In fact there is no agreement as to what an adequate measure of thoroughness may be (traverse each arc twice, traverse all possible arc pairs, etc.). Nonetheless, it has been shown that for many programs (65% of a small survey of eleven programs conducted by Howden), path analysis criteria are "almost reliable" (21). Given the alternatives, testing based on path analysis is today a sound choice, particularly when accompanied by careful program checking and a structured walk-through of the design itself.

The discussion in Chapter II of a theory of testing further examines the critical matter of thoroughness of tests.

c. Automated Aids

Automated aids to support dynamic testing include

the experimental test case generators discussed in the section on static analysis, and programs which compute the degree of coverage of the program graph according to the given path analysis criterion. Pimont and Rault (31) describe an implementation of such a technique, with a more ambitious coverage criterion than most of the path analysis techniques.

The insertion of software counters in the target branches, a relatively simple form of program instrumentation for testing, assists in test data selection by providing feedback as to the coverage obtained from each set of input data (22, 23). A common form of program instrumentation is the use of assertion statements expressing the relationships among data which should hold at various nodes in the program. During execution the assertions are evaluated to check their validity. Program instrumentation can also be used to perform data flow analysis by setting state flags as variables are defined, referenced, and undefined, and noting any illegal state transitions. Program instrumentation requires much of the information normally available in a compiler; therefore it is becoming a feature of experimental test facilities that program instrumentation be performed as a compiler option.

More extensive instrumentation of the source program is involved in execution analysis or execution histogram systems. Such systems have as a goal the creation of a data base of program execution information that can be output in batch fashion or remain available for interactive query. These systems facilitate source language debugging, can provide control and data flow information, environmental information, assertion checking,

tracebacks, etc. The information can aid static analysis and dynamic testing (path analysis or other). There are drawbacks to these systems: very high cost, both in development and use, lack of generality (machine and language dependent usually), and the attendant problems of handling the large data bases that can be created. Fairley describes an Interactive Semantic Modeling System (ISMS) implemented experimentally for application to Algol 60 prgrams (11). The Naval Sea Systems Command has successfully used a software processor AUDIT to aid program verification and to monitor adherence to structured programming standards (71).

Program instrumentation in most cases involves modification of the source code, and generally incurs an unacceptable performance penalty (as does the evaluation of assertions). Therefore it is common to remove such instrumentation from production programs, and to repeat dynamic testing with the optimal set of test data to ensure that program performance remains correct.

There are two simpler automated aids to dynamic testing that should be mentioned. Generators of random test data are not uncommon; although random data do not generally provide a thorough test, they are easy to obtain. While automated computation of the expected results of tests is not feasible (because such computation amounts to automation of the function of the program under test and is the object of verification), automated comparison of actual results with the expected results is practical, and useful as a labor-saving aid. The expected results are typically computed by hand or determined from historical data,

or bounds defining reasonable results are established if computation is overly difficult.

d. Limitations

Limitations to the effectiveness of dynamic testing at ensuring the correctness of software were evident in the discussion of the difficulty of determining a reliable measure of test thoroughness. An additional drawback to any form of dynamic testing is the cost both in time and resources.

Testing or verification techniques include several independent or even contradictory methods, due to the infancy of software engineering and program testing theory. The rationalization of this apparent inconsistency lies in the realization that, given the present understanding of software, nearly every software development is a unique and individual design. Cerification of such programs requires the testing team to be familiar with a variety of testing methods and tools, and to judiciously apply those which seem best suited to the task at hand.

II. NATURE OF THE PROBLEM

A. A THEORY OF TESTING

As alluded to in the first chapter, a tentative theoretical basis for the testing of software has been formulated by Goodenough and Gerhart (17). That theory is capsulized in this section.

1. Types of Errors

Testing is a process of collecting and analyzing evidence relating to the presence of errors. To reach a meaningful conclusion as to the presence of errors, the nature of errors must be clear. On a logical basis, errors can result from failure in implementing the specification (construction errors), failure of the specification to accurately reflect the design, failure of the design to adequately solve the requirements that are understood, or failure to identify the real requirements. Each of these logic errors will ultimately appear as an inappropriate effect produced by the implementation, namely as:

a. missing control flow paths,

b. inappropriate path selection, or

c. inappropriate or missing action on a path.

Recognizing the types of inappropriate effects that may be caused by errors, the problem in testing is to select test cases that can show that these errors do not arise. As mentioned earlier, a common criterion for selecting test data is to choose data which will exercise each arc or branch in the directed graph representing the program at least once. Because logic errors,

particularly specification, design, and requirements errors, may be manifested by missing paths, it should be obvious that this criterion alone cannot select test data that will thoroughly test a program.

2. Criteria for Test Case Selection

Goodenough and Gerhart defined an ideal test as one which succeeds only when a program contains no errors. They defined two predicates about a criterion C for selecting test cases that if satisfied are sufficient to establish that any complete test is an ideal test (a complete test is one using the criterion C to select a set of test data T which are subsequently used to dynamically test the program). These predicates, RELIABLE(C) and VALID(C), are defined as follows. A criterion is reliable only if all complete tests yield the same (not necessarily accurate) result; that is, if one complete test is successful (no program errors are revealed), then all complete tests must be successful, and if one complete test reveals an error, all must reveal that error. Reliability of criteria refers to consistency; using a reliable criterion, a second complete test is redundant as it can provide no new information. A criterion is valid only if for every error that can exist in a program there is (at least) one complete test that can show the presence of the error.

The concept of reliability of a criterion for selecting test cases is not to be confused with the earlier definition of software reliability as a measure of the extent to which programs satisfactorily perform their intended functions.

The fundamental theorem of testing that Goodenough and Gerhart have suggested is simply that there exist some criterion C for selecting test data and some subset T of a program's input domain D such that when it is shown that a test using test data T is a complete test and that the criterion C is both reliable and valid, then success of the test implies that the program is correct.

Stated formally, the theorem is:

$$(\exists T \subseteq D)(\exists C)[(\text{COMPLETE}(T,C) \wedge \text{RELIABLE}(C) \wedge \text{VALID}(C) \wedge \text{SUCCESSFUL}(T)) \Rightarrow (\forall d \in D)\text{OK}(d)],$$

where COMPLETE (T,C) is a predicate indicating that the test T is complete according to the criterion C, and OK(d) is a predicate indicating that program execution with the element d from the input domain D produces the results required by the program specification.

The theorem is not profound; its proof is simple and is assured by the convenient definitions of reliable and valid criteria. If there is some complete test capable of revealing any error (valid criterion) and if all complete tests yield the same result (reliable criterion), then clearly any complete test based on a valid and reliable criterion must correctly demonstrate the presence or absence of errors.

For the skeptic, a proof of the theorem may be written as follows:

- As to the existence of such a $T \subseteq D$ and criterion C, either the program is correct or it is not. If it is correct, then a criterion C such that a complete test T is {d}, where d is any

element of D, will satisfy the theorem. If the program is not correct, there is some element $d \in D$ such that $\neg OK(d)$. Then a criterion C such that $T=\{d\}$ is a complete test. In either case, the required conditions are satisfied, and the criterion and test set exist (the challenge for the program tester is to discover them).

- As to the theorem's implication, assume the truth of the hypotheses and assume $(\exists d \in D) \neg OK(d)$; i.e., assume the theorem is false.

- Then $VALID(C) \Rightarrow (\exists T \subseteq D)[COMPLETE(T,C) \wedge \neg SUCCESSFUL(T)]$.

- Then $RELIABLE(C) \Rightarrow$ "all complete tests fail."

- But this contradicts an hypothesis of the theorem, namely $(\exists T \subseteq D) SUCCESSFUL(T)$.

- End of proof.

Use of the theorem is not an easy matter. A criterion for selecting test data must be chosen and that criterion must be proven reliable and valid. Techniques using dynamic testing to "prove" software correctness will be practical only if the proofs of criterion reliability and validity are simpler to construct and at least as convincing as proofs of program correctness. The experiments reported in this thesis examined some aspects of applying the above theorem.

B. SATISFYING THE PREMISES OF THE THEORY

1. Formal Proofs of Correctness

A degenerate application of the above fundamental theorem of testing is selection of a criterion C such that a complete test is complete only if T is the null set; in this

case, no testing is done. The criterion C is clearly reliable since there can be no tests. To show that C is valid (any possible program error will be revealed by at least one complete test), it must be shown that the program has no errors at all. Application of the theorem with such a criterion is therefore equivalent to a proof of program correctness. Such a proof was constructed for the sample program of this thesis (Chapter IV, Section B).

2. Symbolic Execution

Symbolic execution is a technique whereby symbols are used as input values rather than real data elements and the program is symbolically executed by replacing all data operations with symbolic operations. Intermediate results then are computational expressions of the input symbols rather than data objects. In the case of conditional branching in the program, logical statements on the input symbols, called path conditions, describe the conditions under which a given control path may be traversed. Program correctness is shown by proving that at termination the constraints of path conditions imply that the computational expressions of input symbols are equivalent to those required by the program output specification. That proof and the required proof of similar intermediate theorems constitute a general theorem-proving problem; automated theorem-proving capabilities are currently quite limited, and proving the theorems by hand is quite tedious. This drawback restricts the practical use of formal proofs of correctness as well. A system for symbolic program execution is described in Ref. 8.

Symbolic execution is related to the theorem of testing in that the criterion C is to choose input symbols satisfying the program's input specification; to show the reliability and validity of C, it must be shown that the output specification of the program can be expressed as a computational expression of input symbols and that the intermediate expressions are valid over the entire domain of values for input symbols.

The technique of sumbolic execution was not directly applied in these experiments; however, the experiment using test data execution and the principle of distributed correctness (Chapter IV, Section C) relied on some of the ideas of symbolic execution.

3. Test Data Execution

Clearly one ideal test is execution of the program for every member of the program's input domain. Since most input domains are infinite, this test is usually impossible and nearly always prohibitively costly, and can therefore hardly be called ideal in any practical sense. Goodenough and Gerhart used in Reference 17 a "condition table method" to select test data for prøgram execution. While they were not able to conclusively prove the reliability or validity of this method as a selection criterion, they attempted to show that they did identify equivalence classes covering the input domain of an example program and choose a representative of each class which by induction tested each member of that equivalence class. The condition table method was incorporated in an experiment of this thesis in conjunction with the distributed correctness principle.

Test data selection techniques such as path analysis and independent sub-structure analysis are also attempts to identify equivalence classes in the input domain, while again only informally trying to establish criterion reliability and validity. These techniques were used in the experiment reported in Chapter IV, Sections D and E; in each case, however, it was not possible to determine whether the equivalence classes identified actually covered the input domain until comparison with the results of other experiments.

III. EXPERIMENTAL PROCEDURES

An example program was selected for experimentation, and several verification methods were applied to demonstrate the correctness of the example program The hours of effort required for each method were recorded and qualitative assessments were made about the degree of difficulty of using each method. The example program and the methods employed are described in this chapter; the actual results and a comparison of the methods is presented in the next chapter.

A. THE PROGRAM AND INTUITIVE TEST DATA

1. Origin and Description

Reference 20 is a report of an experiment in software error occurrence and detection conducted at the Naval Postgraduate School. Four programming projects were undertaken and data were recorded on the man-hours expended in each development phase, time of detection and occurrence of errors, and man-hours expended correcting errors. Errors were classified according to the development phase in which they occurred and by descriptive types, and were analyzed with respect to development phase, correction time, and program complexity.

Project number one of Reference 20 was chosen as a program for experimental verification. The subject program reads and processes a variable length string of text characters, recording all occurrences of palindromes (sub-strings which read the same forward or backward), including overlapping palindromes and omitting palindromes entirely included with another. The program

was written in Algol-W to run under OS/MVT. It contains 141 source statements and consists of the main program and ten procedures, five of which are significant to the palindrome-finding algorithm; the remaining five are called to print the results. The original program development required 5.0 man-hours in the design phase, 7.0 in the coding phase, 4.0 in debugging, and 5.8 hours in the original testing phase.

2. Program Listing

Figure 1 is a listing of the palindrome program.

```
begin
comment  This program finds palindromes within a character string
         of maximal length = 256.
         Minimum string length is 2.
         All input cards will be listed.
         The program will produce a list of only those palindromes
         which are not entirely included in a larger palindrome;


comment  data declarations;

string(1) array text(1::256);              comment  contains character
                                                    string;
string(80) cardbuffer;                     comment  i/o buffer for cards;
integer array begin_of_palindrome, end_of_palindrome(1::256);
integer cardlimit, length_of_text, bufferposition, card_counter,
        palindrome_counter;
integer ix, jx;                            comment  index variables;


comment  initialization;

    procedure initialize;
    comment  initialize all variables, read length_of_text, write text1;
    begin
    text1;
    jx:=1;
    palindrome_counter:=1;
    cardlimit:=80;
    intfieldsize:=5;
    read(length_of_text);
    if ((length_of_text < 2) or (length_of_text > 256)) then
         begin
         write("Illegal input:",
               "  length of input string is: ",length_of_text);
         assert(false);
         end;
    cardbuffer:=" ";
    end initialize;


comment  utilities;

    procedure blank_lines(integer value n);
    comment  write n blank lines;
    begin
    integer i;                  comment  local counter;
    assert(n>0);                comment  safety check;
    for i:=1 step 1 until n do write(" ");
    end blank_lines;

    procedure text1;
    begin
    write("Find all palindromes within the following string:");
    blank_lines(2);
    write("Card                  Text");
    write("Number");
    blank_lines(1);
    end text1;
```

FIGURE 1

Page 1 of 4

PROGRAM LISTING

```
procedure text2;
begin
blank_lines(2);
write("The following palindromes have been detected:");
blank_lines(1);
write("Palindrome     Begin                End");
write("Number      Card  Character    Card  Character");
write("           Number  Position   Number  Position");
write("-------------------------------------------------");
writeon("-------------------------------------------------");
end text2;

procedure text3;
begin
write("No palindromes found.  End of run.");
end text3;

procedure read_and_write_input_cards;
comment  read input cards according to given length_of_text;
begin
integer number_of_input_cards;
number_of_input_cards:=(length_of_text - 1) div cardlimit + 1;
ix:=1;                    comment  reset text index;
for card_counter:=1 step 1 until number_of_input_cards do
     begin
     write(card_counter);
     writeon(" ");
     readcard(cardbuffer);
     writeon(cardbuffer);
     bufferposition:=0;        comment  reset index;
     while ((ix<=length_of_text) and (bufferposition<cardlimit)) do
          begin
          text(ix):=cardbuffer(bufferposition|1);
          ix:=ix+1;
          bufferposition:=bufferposition+1;
          end;       comment  done for all characters on a card;
     end;            comment  done for all cards;
end read_and_write_input_cards;

procedure write_all_palindromes;
comment  list all palindromes being found;
begin
integer i,j;              comment  local counters;
text2;
for i:=1 step 1 until jx-1 do
     begin
     if end_of_palindrome(i) ~=0 then
          begin
          write(palindrome_counter);
          writeon(" ");
          writeon(((begin_of_palindrome(i)-1) div cardlimit) + 1);
          writeon(" ");
          if (begin_of_palindrome(i) rem cardlimit = 0) then
               writeon(cardlimit)
          else
               writeon((begin_of_palindrome(i) rem cardlimit));
          writeon("     ");
          writeon(((end_of_palindrome(i)-1) div cardlimit) + 1);
          writeon("  ");
```

FIGURE 1

Page 2 of 4

PROGRAM LISTING

```
                if (end_of_palindrome(i) rem cardlimit =0) then
                     writeon(cardlimit)
                else
                     writeon((end_of_palindrome(i) rem cardlimit));
                write("     ");
                for j:=begin_of_palindrome(i) step 1
                until end_of_palindrome(i) do writeon(text(j));
                write("-----------------------------------------------");
                writeon("-----------------------------------------------");
                blank_lines(1);
                palindrome_counter:=palindrome_counter+1;
                end;                comment  end if;
          end;                      comment  done for all palindromes;
     end write_all_palindromes;


comment  subroutines;

     procedure palindrome_check;
     comment  find all palindromes within given text string;
     begin
          comment  scan text from left to right;
     for ix:=2 step 1 until length_of_text do
          begin
          if text(ix-1) = text(ix) then continue_checking((ix-1),ix);
          if ix ~= 2 then
               if text(ix-2)=text(ix) then continue_checking((ix-2),ix);
          end;
     end palindrome_check;


     procedure continue_checking (integer value first,last);
     comment  Given first and last as pointers to a palindrome
               of size 2 or 3, this procedure checks whether or not this
               palindrome is included in a larger palindrome;
     begin
     logical palindrome;
     palindrome:=true;
     while ((first>1) and (last<length_of_text) and (palindrome=true)) do
          begin
          if text(first-1) = text(last+1) then
               begin
               comment  larger palindrome found;
               first:=first-1;
               last:=last+1;
               end
          else
               begin
               palindrome:=false;       comment  largest palindrome found;
               end;
          end;
     record_palindrome(first,last);
     end continue_checking;
```

FIGURE 1

Page 3 of 4

PROGRAM LISTING

```
    procedure record_palindrome (integer value first,last);
    comment  Record only max length palindromes.  Flag previously
             recorded palindromes if they are included in the palindrome
             specified by first and last.
             jx was initialized to 1.  After completion jx points to the
             next entry in begin_of_palindrome and end_of_palindrome;
    begin
    integer i;                  comment  local counter;
    logical entry;
    entry:=true;
    for i:=1 step 1 until jx-1 do
         begin
         if ((first>=begin_of_palindrome(i))
              and (last<=end_of_palindrome(i))) then
              begin
              comment  Palindrome is entirely included in a previously
                       recorded palindrome.  No entry required;
              entry:=false;
              end
         else
              begin
              if ((begin_of_palindrome(i) >= first)
                   and (end_of_palindrome(i) <= last)) then
                   begin
                   end_of_palindrome(i):=0;
                   comment  flag smaller palindrome;
                   end;
              end;
         end;       comment  All previously recorded palindromes
                             compared with last input;
    if entry = true then
         begin
         comment  larger than all previous or overlapping or disjoint;
         begin_of_palindrome(jx):=first;
         end_of_palindrome(jx):=last;
         jx:=jx+1;
         end;
    end record_palindrome;


comment  main;

initialize;
read_and_write_input_cards;
palindrome_check;
if jx=1 then text3
else write_all_palindromes;
end.
```

FIGURE 1

Page 4 of 4

PROGRAM LISTING

3. Program Graphs

Figure 2 presents the control flow directed graph representations of the significant procedures of the example program. The graphs, originally presented in Reference 20, are annotated with key-words indicating the structured programming constructs comprising the decision nodes, lower-case letters which label the arcs, upper-case letters representing the counters placed on the individual decision-to-decision paths for path analysis, and with complexity measures of the procedures. The complexity measures shown are defined as follows:

a. The number of statements is a count of the source code statements in the procedure.

b. The number of nodes is a count of the nodes in the control flow graph. Nodes are points of starting, stopping, branching, or merging of control flow; i.e., decision points.

c. The number of arcs is a count of the arcs in the graph. Arcs are concatenations of zero or more program statements with no decisions except at the nodes.

d. The cyclomatic number of a strongly connected graph is equal to the maximum number of linearly independent circuits (27). A program control flow graph with one entry and one exit such that each node can be reached from the entry and such that the exit can be reached from every node can be considered as a strongly connected graph with an imaginary arc from the exit node back to the entry node. For such a control flow graph the cyclomatic number can be variously interpreted as the maximum number of independent paths, one more than the number of predicate

nodes (nodes with more than one path leaving), the number of regions in the graph (plane graph with no arcs crossing), or the number of arcs minus the number of nodes plus two (27). The experimental method described in Section E of this chapter makes use of the cyclomatic number.

Procedure:
INITIALIZE
Number of statements: 14
Number of nodes: 5
Number of arcs: 5
Cyclomatic number: 2

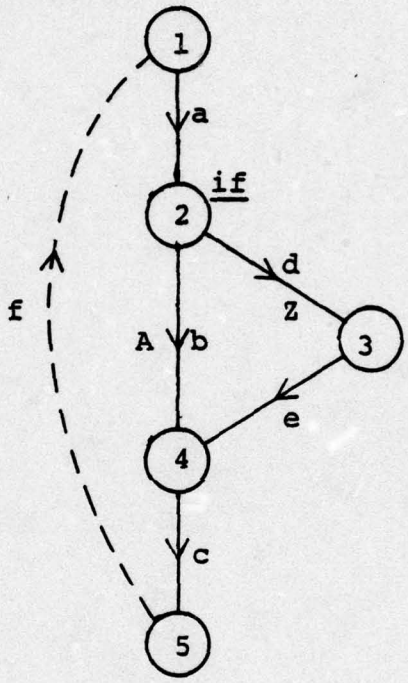


FIGURE 2

Page 1 of 6

PROGRAM GRAPHS

Procedure:
READ_AND_WRITE_
INPUT_CARDS
Number of statements: 18
Number of nodes: 6
Number of arcs: 7
Cyclomatic Number: 3

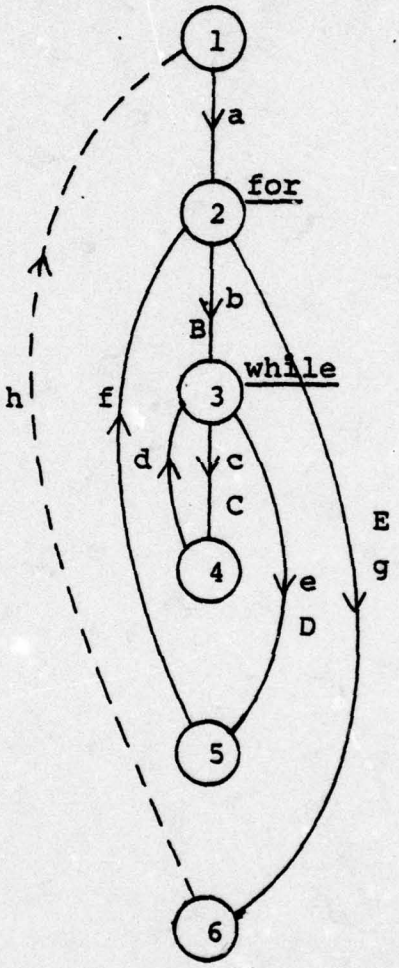


FIGURE 2

Page 2 of 6

PROGRAM GRAPHS

Procedure:
PALINDROME_CHECK
Number of statements: 7
Number of nodes: 10
Number of arcs: 13
Cyclomatic number: 5

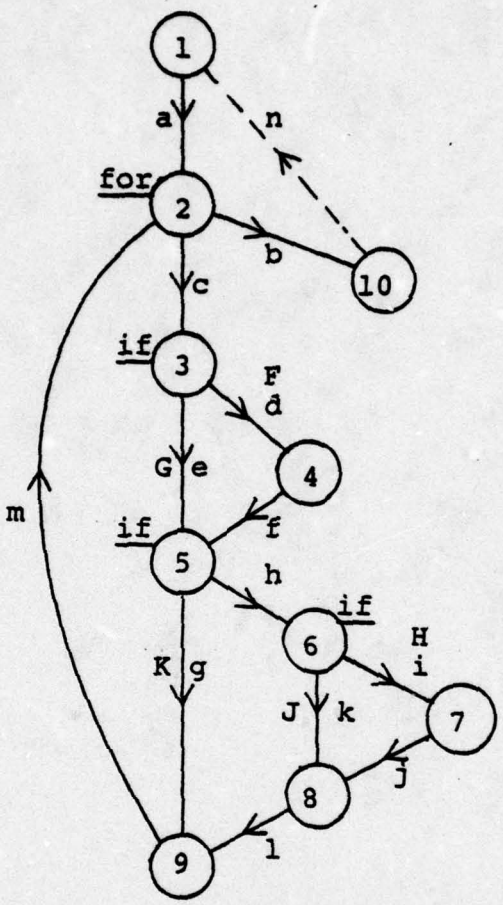


FIGURE 2

PROGRAM GRAPHS

```
Procedure:
   CONTINUE←CHECKING
Number of statements: 15
Number of nodes:       8
Number of arcs:        9
Cyclomatic number:     3
```

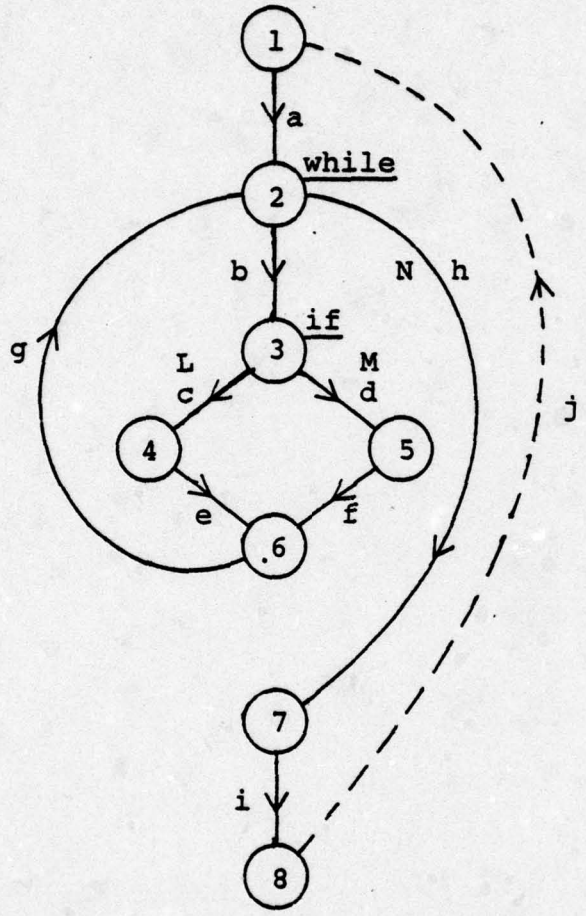


FIGURE 2

PROGRAM GRAPHS

Procedure:
RECORD←PALINDROME
Number of statements: 21
Number of nodes: 10
Number of arcs: 13
Cyclomatic number: 5

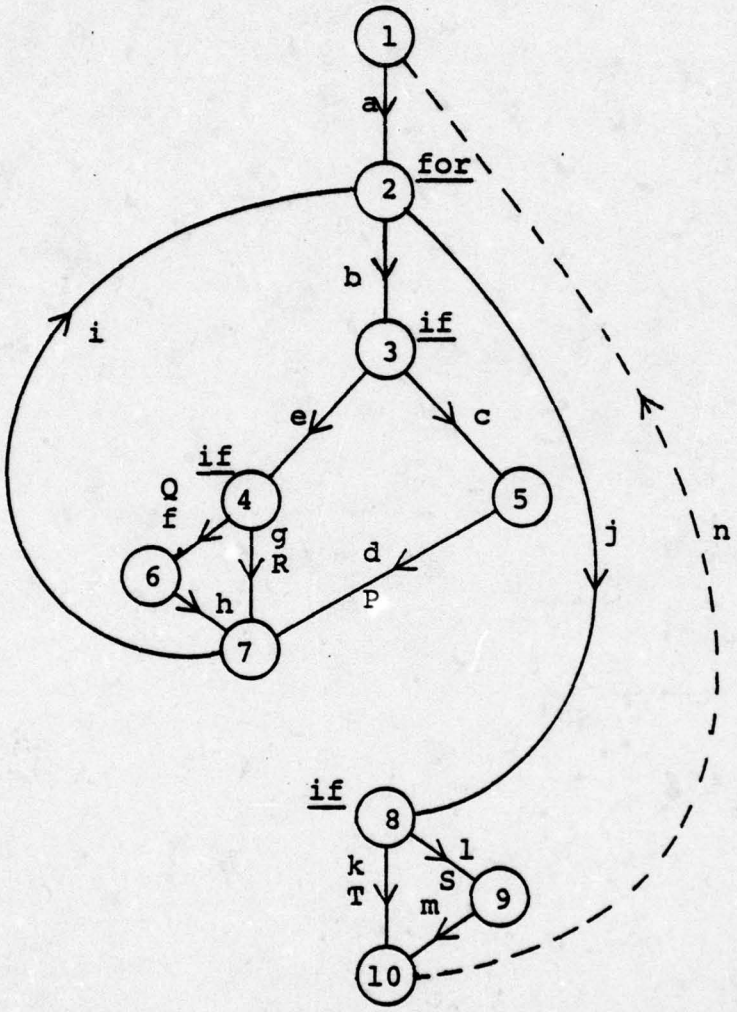


FIGURE 2

PROGRAM GRAPHS

Procedure:
MAIN
Number of statements: 6
Number of nodes: 6
Number of arcs: 6
Cyclomatic number: 2

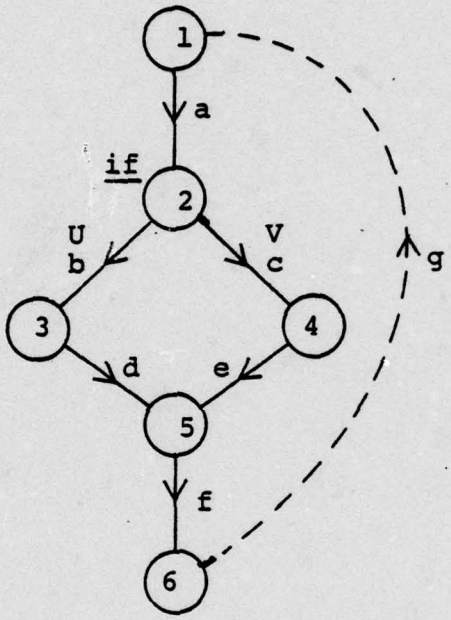


FIGURE 2

Page 6 of 6

PROGRAM GRAPHS

4. Error Data

Forty-four errors were detected during original program development. Error descriptions by type are recorded in Reference 20. Table I presents the errors (using Hoffman's original error numbering) which could be directly related to a particular program fragment.

TABLE I

SELECTED ERRORS

Error Number	Procedure	Description
1	read←and←write...	Need for 256-character variable "text" in addition to an 80-character buffer noted.
3	initialize	Syntax error.
4	initialize	Syntax error.
35	palindrome←check	Error in limits to counter of for statement.
36	palindrome←check	Missing "if ix¬=2" resulted in indexing error at run.
37	initialize	"jx" not initialized; resulted in indexing error.
42	initialize	"jx" initialized to 0 vice 1.

5. Intuitive Test Cases

After a first examination of the example program and before commencing any of the verification experiments, a set of test cases was selected by intuition which appeared to test the program's handling of all conditions significant to proper program operation. Those test cases are listed in Table II. The intuitive test cases were used for conducting additional dynamic testing after completion of the experiments using the methods described in the remainder of this chapter.

TABLE II

INTUITIVE TEST CASES

Test Number	String Length	String or String Description
1	2	"xx" - length 2 palindrome
2	3	"xyz" - length 3 palindrome
3	1	Invalid string length
4	257	Invalid string length
5	256	A maximum length string containing palindromes of length 10, 9, 6, and 12, of which the first included length 3 palindromes at both ends, the second and third overlapped, and the fourth was written across an input card boundary.
6	15	String with no palindromes
7	100	Entire string one palindrome
8	256	One maximum length palindrome

B. PROOF OF CORRECTNESS

A method of formal proof of correctness was used to verify the logical correctness of the algorithm of the example program. The method required the assumption that the environment in which the program operates is also logically correct, most particularly that the compiler and operating system ensure performance of the expected semantic actions for the program statements.

References 9,13,19,25 and 26 were consulted to develop the methodology for constructing the proof. The work of Floyd (13) is considered the basis for mathematical program proofs; the paper by Manna and Waldinger (26) was particularly helpful in an operational sense, and Hoare's paper (19) was applicable to the treatment of procedure calls.

The first step in constructing the proof was to formalize the required result of the program in a logical statement called the output assertion, and to describe the restrictions on input

data as an input assertion. Translating the rather loosely defined purpose of finding palindromes to an output assertion required significant analysis of the requirements.

The proof itself was constructed using the method of invariant assertions to show partial correctness; i.e., that when the input data satisfied the input assertion, the output assertion was satisfied if the program terminated. Termination of the program was proven separately by finding for each loop a set and an ordering relation on that set such that the set can have no infinite decreasing sequences (well-founded ordering), and defining a termination expression which is a member of that set and which is decreased each time control passes through the loop. The proofs of partial correctness and termination together establish the "total correctness" (26) of the program.

The method of invariant assertion involved inserting appropriate intermediate assertions (also called Floyd assertions) at selected labels in the program such that they defined a condition which would be logically true each time control passed through that label (hence the name invariant). At least one intermediate assertion was inserted on every loop. Corresponding to each path between assertions a verification condition was written. A verification condition is a theorem of the form <assertion 1> and <semantic meaning of program statements on path> implies <assertion 2>. Proving all verification conditions completed the proof of partial correctness.

The construction of appropriate verification conditions was aided by using the invariant deductive system described in

Reference 26. The notation {P} F {Q}, where P and Q are logical statements (assertions) and F is a program fragment, is used to mean that if P holds before executing F and if F terminates, then Q holds after executing F. Thus the proof of program correctness is a proof of the invariant statement:

{input assertion} program {output assertion}.

Rules of inference were supplied in Reference 26 to provide subgoals for proving certain invariant statements; in particular there is one rule of inference for each statement type in the programming language. For instance, the rule for conditional statement "if R then F else G" is written as:

$$\frac{\{P \text{ and } R\}\ F\ \{Q\},\ \{P \text{ and } \neg R\}\ G\ \{Q\}}{\{P\}\ \underline{\text{if}}\ R\ \underline{\text{then}}\ F\ \underline{\text{else}}\ G\ \{Q\}}$$

The notation signifies that proof of the two invariant statements in the nominator of the "fraction" is sufficient to infer the invariant statement in the denominator.

The reference provided rules of inference for assignment statements, conditional statements, while statements, and for the concatenation of statements. Additional rules were formulated for iterative for statements and procedure calls. The rules are introduced as required in the presentation of the proof (Appendix A).

C. DISTRIBUTED CORRECTNESS

Test cases for dynamic testing were selected using a criterion based on the condition table method of Goodenough and Gerhart (17) and the principle of distributed correctness described by

Geller (14). The criterion was not proven reliable and valid, but an effort was made to show that the data selected were representative of equivalence classes covering the input domain of the program such that correct operation for all data in the domain could be induced from correct operation for the test data.

As in the proof of correctness, input and output assertions were stated. At selected labels in the program (fewer than in the correctness proof), "synthesized assertions" (14) were inserted, similar to Floyd assertions but more general. The synthesized assertions expressed invariant conditions of the program. A condition table was constructed (if necessary) to analyze all conditions affecting program operation from the previous assertion to the one under consideration. Test data were selected for each class identified in the table (similar to the decision rules of a limited-entry decision table), and a "test data assertion" was stated, namely that execution of the program fragment with the test data would satisfy the synthesized assertion. The test data assertion was verified by dynamic testing. At the same label, a generalization assertion was made attempting to state the conditions for correct operation of the program fragment, and where possible the synthesized assertion was proven from the test data and generalization assertions. In several cases, however, a basis for verifying the generalization assertion could not be found and hence the verification method could not be claimed to have proved program correctness through testing. Certainly, however, an intuitive feeling of "high" reliability and validity of the test case selection criterion

was established.

The principle of distributed correctness was called upon to infer program correctness from the correct operation of the distributed program fragments verified by the synthesized assertions.

D. PATH ANALYSIS

1. Basic Technique

Path analysis techniques are described in References 21, 22, 23, 31 and 32. For the example program, test data were selected to force program traversal of each arc of the program control flow graph. Execution of the arcs labeled with upper-case letters on the graphs of Figure 2 is sufficient to ensure traversal of all arcs. Data were selected to satisfy path predicates for each such arc, predicates which describe constraints on the inputs to ensure execution of the arc. The program was instrumented with a counter on each labeled arc to count the number of arc traversals, thus ensuring after testing that no arcs had been missed.

More stringent criteria could have been applied, but execution of all possible control paths was not possible since the program has an infinite number of paths.

2. Extended Technique

The method of selecting test data as described above was repeated with the additional criterion that each conditional branching statement with more than one predicate be executed with each possible combination of truth values of the predicates,

thus expanding the class of errors that might be detected by the tests. Additional path predicates were considered and a larger set of test data was selected.

E. INDEPENDENT SUB-STRUCTURES.

A method for selecting test cases which is similar to path analysis has been suggested in References 27 and 33. As applied herein, the technique was to decompose the control flow graph of each procedure of a program into independent circuits corresponding to language constructs and to use these sub-structures as an aid in identifying control flow paths for testing. The cyclomatic number of a single-entry single-exit procedure is the maximum number of such sub-structures; these independent circuits can be identified by inspection for simple graphs or more generally as follows:

A spanning tree (33) is a connected sub-graph consisting of all nodes of a procedure's graph but containing no circuits. Its arcs are called branches. There is one independent circuit corresponding to each arc of the parent graph not included in the spanning tree; these arcs, including an imaginary arc from the exit node back to the entry node, are called chords. As each chord is added to the spanning tree, the corresponding independent circuit can be identified.

A matrix representation of the circuits was useful in generating control paths for dynamic testing from the sub-structures. A fundamental circuit matrix (33) was constructed with rows corresponding to arcs (chords and branches); its entries were

0 for arcs not contained in a circuit, 1 for arcs oriented in the direction of an assumed circuit flow, and -1 for arcs oriented against the assumed flow. Chords were listed on the left of each row, forming a unit matrix because there is a one-to-one correspondence between chords and circuits.

The usefulness of the fundamental circuit matrix was that linear combinations of circuits (the rows) having at least one arc in common generated control paths useful for testing. Selection of paths in this manner satisfied a criterion more stringent than traversal of each arc at least once, while considerably less stringent than traversal of all possible paths.

In Section E of Chapter IV, the results of the application of this technique are discussed. Following the generation of the paths to be tested, test data were selected to satisfy the path predicates and dynamic testing was conducted.

IV. PRESENTATION OF RESULTS

A. STATIC ANALYSIS

Prior to the commencement of the experimental methods, 1.0 man-hours of effort were expended in static analysis of the example program. Code was read to check syntax and reachability of program fragments, and control flow graphs were examined to check for adherence to structured programming constructs and to learn the general flow of the program. No exceptions were noted in these areas.

All global and local variables were examined to ensure proper declarations and to check the transitions among the states of being undefined, defined and not referenced, defined and referenced, and an anomalous state (23). No illegal state transitions were found; however, two instances of questionable programming practice were noted. First, the string array variable "cardbuffer" is initialized to contain 80 blank characters in the procedure "initialize" (state = defined and not referenced). In a data flow sense, the next action performed on that variable is to re-define it in the procedure "read←and←write←input←cards," thus transitioning to the anomalous state. Since the variable is re-defined before being referenced, the initializing of "cardbuffer" in the "initialize" procedure is superfluous. Second, all iterative counters in for statements in the example program are explicitly declared. Because the Algol-W compiler implicitly declares such counters, the effect in the example program is to explicitly declare several variables that are

subsequently not defined or referenced. Although this is a shortcoming in style, the superfluous declarations cause no ill effects other than to waste a small amount of storage.

The verifications performed during static analysis were required to ensure satisfaction of several assumptions for the experiments in correctness demonstration, particularly that the program has no significant data flow anomalies and that the program is well structured and procedures do not contain statements leading to unexpected side effects. The two instances of style noted above were not corrected prior to experimentation.

B. PROOF OF CORRECTNESS

The detailed proof of correctness for the example program is presented in Appendix A. The first step in constructing the proof was to formalize the output specification of the program, a task requiring 0.8 man-hours of effort. It was desired to prove that at program termination the arrays "begin←of←palindrome" and "end←of←palindrome" would contain, corresponding to indices starting from 1, integers representing the beginning and ending characters in the string "text" for all palindromes in the string, subject to the constraint that palindromes fully contained within a larger palindrome would not be recorded. A palindrome initially recorded in the arrays and subsequently found to be included in a larger one would be "deleted" by setting the "end←of←palindrome" entry to zero. Overlapping palindromes would be recorded.

The formal statements made for the specification are ("ℓ" is an abbreviation for "length-of-text"; "bop" and "eop" are abbreviations for the beginning and ending entry arrays; "jx" is a counter which is equal to one more than the number of entries made in those arrays):

Q: $\forall x[(2<=x<=\ell \wedge text(x-1)=text(x)) \Rightarrow$
$\exists y(y<jx \wedge 1<=bop(y)<=x-1 \wedge x<=eop(y)<=\ell)]$

R: $\forall x[(3<=x<=\ell \wedge text(x-2)=text(x)) \Rightarrow$
$\exists y(y<jx \wedge 1<=bop(y)<=x-2 \wedge x<=eop(y)<=\ell)]$

S: $\forall y[(0<y<jx \wedge bop(y)>1 \wedge 0<eop(y)<\ell) \Rightarrow$
$(text(bop(y)-1)=text(eop(y)+1))]$

T: $\forall y[(0<y<jx \wedge \neg(eop(y)=0)) \Rightarrow$
$[string(bop(y),eop(y))=ok \wedge bop(y)>=1 \wedge eop(y)<=\ell$
$\wedge \forall z((0<z<jx \wedge \neg(z=y)) \Rightarrow$
$(\neg(bop(z)=bop(y))$
$\wedge (bop(z)<bop(y) \Rightarrow eop(z)<eop(y))))]$

The assertions Q and R require, respectively, that all palindromes of length 2 and length 3 are included within some entry in the arrays "bop" and "eop." Due to the symmetry of palindromes, all must contrain a palindrome of length 2 or 3 at the center; therefore, when conditions Q and R are satisfied, all palindromes have at least been detected by the workings of the algorithm, even if their total extent has not been recognized. Condition S requires that any valid entry in the arrays "bop" and "eop" represents as long a palindrome as can be recognized starting from the length 2 or 3 palindrome at the center; symmetry is again relied upon. Finally, condition T

requires that all strings represented by valid entries (those with non-zero "eop" entry) are in fact palindromes (the notation "string(1,2)=ok" is used to indicate that the sub-string starting at position 1 in "text" and ending at position 2 is a palindrome), and that no entry in the arrays "bop" and "eop" is included with another entry.

Together the four assertions provided a formal statement that could be proved from the semantics of program statements and the assumption that the input constraints (see Appendix A) have been satisfied.

Given the output specification to be proven, five procedures were determined during static analysis to have no bearing on the program's performance of the desired process. Procedures "text1," "text2," and "text3," merely print output labels; "blank←lines" inserts blank lines in the output. Procedure "write←all←palindromes," while complex, serves only to print the strings corresponding to the entries in the previously mentioned arrays. No correctness proof other than for termination was supplied for these procedures.

For the remaining significant procedures, Table III presents the man-hours of effort expended in constructing their proofs. As is discussed in Chapter V, there was a relationship between procedure complexity and the time to construct a proof for the procedure. The time required to prove procedure "read←and←write←input←cards," the first somewhat complex procedure proved, was distorted by the presence of a significant learning curve.

TABLE III

TIMES TO CONSTRUCT PROOFS

Task Accomplished or Procedure Proved	Man-Hours
Formalize output specification	0.8
"initialize"	0.8
Show termination of 5 utilities	0.6
"read←and←write←input←cards"	4.2
"palindrome←check"	1.6
"continue←checking"	1.8
"record←palindrome"	3.5
"main"	0.2
Total	13.5

C. DISTRIBUTED CORRECTNESS

The detailed demonstration of correctness of the example program using the condition table method and the principle of distributed correctness (described in Chapter III) is presented in Appendix B. As in the formal proof of correctness, this method required formalization of the output specification as a first step; the same output assertions were established as in the formal proof. The six significant procedures were tested by choosing synthetic assertions, test data assertions, and generalization assertions, and then executing test data to verify the assertions. Table IV presents the man-hours of effort expended in demonstrating correctness by this method.

TABLE IV

TIMES TO DEMONSTRATE CORRECTNESS

Task Accomplished or Procedure Shown	Man-Hours
Formalize output specification	0.8
"initialize"	0.2
"read←and←write←input←cards	0.7
"palindrome←check"	0.9
"continue←checking"	1.3
"record←palindrome"	3.2
"main"	0.4
Total	7.5

The test data elements used to verify the test data assertions in the correctness demonstration are consolidated in Table V into one comprehensive set of test cases; the palindromes which should be recorded for the given texts are indicated by underscoring. The degree to which these test cases satisfied the requirements for an ideal test is discussed in a later section.

TABLE V

TEST CASES IDENTIFIED BY CONDITION TABLE METHOD

Test Number	Length of text	Text	Test Number	Length of text	Text
1	2	ab	13	4	aabc
2	2	aa	14	4	abba
3	80	Note 1	15	4	abbc
4	81	Note 1	16	5	abbad
5	160	Note 1	17	5	abbcd
6	240	Note 1	18	5	abccb
7	256	Note 1	19	5	abccd
8	3	aba	20	6	abccba
9	3	aab	21	6	abccbd
10	3	abb	22	6	abccde
11	4	abcc	23	9	baaabaaab
12	4	aaaa			

Note 1 - any text which includes overlapping palindromes

For purposes of comparison with later sets of test cases, it is noted that execution of the above tests required 0.88 seconds of CPU time.

D. PATH ANALYSIS

1. Basic Technique

Static analysis of the program identified 21 individual decision-to-decision paths in the program control flow graph. The program was instrumented with integer counters placed in added assignment statements to record the number of traversals of each of these arcs (which are identified by upper-case letters in Figure 2, Chapter III). The analysis and instrumentation required 0.8 man-hours of effort.

Path predicates were established and test cases selected to force traversal of each arc at least once (0.3 man-hours required). For the example program the path predicates were quite simple to satisfy. Finally, the test data (shown in Table VI) were used in dynamic testing of the program (0.2 man-hours). Table VII lists the individual arcs and the number of traversals of each which were recorded. No program errors were detected. Each arc was traversed at least once, and thus the selected data provided at least a "minimally thorough" test of the program (22). The total effort involved in the application of this method was 1.3 man-hours. The tests required 0.05 seconds of CPU time.

TABLE VI

TEST CASES IDENTIFIED

BY PATH ANALYSIS

Test Number	Length of text	Text
1	1	--
2	2	xy
3	7	baaaaca

TABLE VII

ARC TRAVERSALS USING

PATH ANALYSIS DATA

Arc:	Z	A	B	C	D	E	F	G	H	J	K	L	M	N	P	Q	R	S	T	U	V
Test 1	1																				
Test 2		1	1	2	1	1		1			1									1	
Test 3		1	1	7	1	1	3	3	3	2	1	1	5	6	3	2	4	3	3		1
Total	1	2	2	9	2	2	3	4	3	2	2	1	5	6	3	2	4	3	3	1	1

2. Extended Technique

Preceding the arcs labeled D, N, P and Q in Figure 2 are decision statements involving two or more predicates (e.g., "if (A and B) then"). The basic path analysis technique provided simply for traversal of each arc following such decisions (i.e., "A and B" true, and "A and B" false). A more thorough test of a bi-conditional decision would execute the decision statement under the four truth combinations for the two predicates (A true and B false, A true and B true, A false and B true, A false and B false); similarly a tri-conditional decision could be executed under the eight truth combinations for the

three predicates.

Analysis of the appropriate path predicates identified four additional test cases (Table VIII) which, when added to the three cases selected by the basic technique, ensure execution of multi-condition decision statements under all (possible) combinations of truth values for the decision predicates. Table IX presents the arc traversals recorded for the additional tests.

No program errors were detected. The total additional effort to select the additional cases was 0.9 man-hours; therefore, the total time to apply the extended path analysis method was 2.2 hours. The additional tests required 0.18 seconds of CPU time, for a total CPU time of 0.23 seconds for the extended method.

TABLE VIII

TEST CASES IDENTIFIED BY

EXTENDED PATH ANALYSIS

Test Number	Length of Text	Text
4	257	--
5	3	aaa
6	5	baaab
7	160	*

*Any string of 160 characters

TABLE IX

ARC TRAVERSALS USING

EXTENDED PATH ANALYSIS DATA

Arc:	Z	A	B	C	D	E	F	G	H	J
Test 4	1									
Test 5		1	1	3	1	1	2		1	
Test 6		1	1	5	1	1	2	2	1	2
Test 7		1	2	160	2	1	1	158		158
Total, Tests 1-7	2	5	6	177	6	5	8	164	5	162

Arc:	K	L	M	N	P	Q	R	S	T	U	V
Test 4											
Test 5	1			3		2	1	3			1
Test 6	1	1	2	3		2	1	3			1
Test 7	1	3	1	1				1			1
Total, Tests 1-7:	5	5	8	13	3	6	6	10	3	1	4

E. INDEPENDENT SUB-STRUCTURES

The techniques described in Chapter III, Section E, were applied to the six significant procedures of the example program to identify control paths for testing based on the independent language constructs of the procedures. A spanning tree (consisting of all nodes but containing no circuits) was constructed corresponding to each procedure control flow graph presented in Figure 2 (Chapter III); the arcs of the spanning tree were considered as branches. The remaining arcs of the parent graphs were considered to be chords, with a single independent circuit or language construct corresponding to each. The fundamental

matrices for the six procedures are presented below; the rows are labeled with the language construct names of the fundamental circuits; the columns are headed with the arc (branches and chords) labels. If a fundamental matrix has n rows, the first n column labels from the left are the chords and the remaining column labels are the branches. The matrix entries have the meanings described earlier.

Procedure "initialize":

f	e	a	b	c	d	
1	0	1	1	1	0	main-line
0	1	0	-1	0	1	if-then

Procedure "read←and←write←input←cards":

h	d	f	a	b	c	e	g	
1	0	0	0	0	0	0	1	main-line
0	1	0	0	0	1	0	0	while-do
0	0	1	0	1	0	1	0	for

Procedure "palindrome←check":

n	f	l	j	m	a	b	c	d	e	g	h	i	k	
1	0	0	0	0	1	1	0	0	0	0	0	0	0	main-line
0	1	0	0	0	0	0	0	1	-1	0	0	0	0	if-then
0	0	1	0	0	0	0	0	0	0	-1	1	0	1	if-then
0	0	0	1	0	0	0	0	0	0	0	0	1	-1	if-then
0	0	0	0	1	0	0	1	0	1	1	0	0	0	for

Procedure "continue←checking":

j	f	g	a	b	c	d	e	h	i	
1	0	0	1	0	0	0	0	1	1	main-line
0	1	0	0	0	-1	1	-1	0	0	if-then-else
0	0	1	0	1	1	0	1	0	0	while-do

Procedure "record←palindrome":

n	d	h	i	m	a	b	c	e	f	g	j	k	l	
1	0	0	0	0	1	0	0	0	0	0	1	1	0	main-line
0	1	0	0	0	0	0	1	-1	0	-1	0	0	0	if-then-else
0	0	1	0	0	0	0	0	0	1	-1	0	0	0	if-then
0	0	0	1	0	0	1	0	1	0	1	0	0	0	for
0	0	0	0	1	0	0	0	0	0	0	0	-1	1	if-then

Procedure "main":

g	e	a	b	c	d	f	
1	0	1	1	0	1	1	main-line
0	1	0	-1	1	-1	0	if-then-else

The control paths for testing were selected by using the rows (fundamental circuits) and all linear combinations of rows having one or more branches in common to identify sequences of arcs which should be tested. The paths for each procedure that were selected in this manner are presented below; paths are identified here in a node-to-node format (node numbers correspond to those in Figure 2).

Procedure "initialize":

1-2-3
1-2-4-5

Procedure "read←and←write←input←cards":

1-2-3-5-2-6 *
1-2-3-4-3-5-2-6

Procedure "palindrome←check":

1-2-10 *
1-2-3-4-5-9-2-10
1-2-3-5-2-9-10
1-2-3-5-6-7-8-9-2-10
1-2-3-5-6-8-9-2-10

Procedure "continue←checking":

1-2-7-8
1-2-3-4-6-2-7-8
1-2-3-5-6-2-7-8

Procedure "record←palindrome":

1-2-8-9-10 *
1-2-8-10
1-2-3-5-7-2-8-10
1-2-3-4-6-7-2-8-9-10
1-2-3-4-7-2-8-9-10

Procedure "main":

1-2-3-5-6
1-2-4-5-6

Path predicates were established for the control paths listed above and test cases were selected to force their traversal. The paths above followed by "*" have path predicates which cannot be satisfied by any input data; the given execution sequence is impossible for any allowable input. The minimal set of test cases selected to satisfy the path predicates is presented in Table X. Dynamic testing with these test cases revealed no program errors; 0.12 seconds of CPU time were required for the tests. Table XI describes the allocation of the 2.2 man-hours of effort expended in the application of this method.

TABLE X

TEST CASES IDENTIFIED BY

INDEPENDENT SUB-STRUCTURES METHOD

Test Number	Length of text	Text
1	1	--
2	2	xy
3	2	aa
4	3	aba
5	3	abc
6	7	aabbbba

TABLE XI

TIMES TO APPLY METHOD OF

INDEPENDENT SUBSTRUCTURES

Task Accomplished or Procedure Shown	Man-Hours
"initialize"	0.1
"read←and←write←input←cards"	0.2
"palindrome←check"	0.6
"continue←checking"	0.3
"record←palindrome"	0.7
"main"	0.1
Conduct dynamic tests	0.2
Total	2.2

For purposes of comparison of the degree of arc coverage of tests conducted by this method with that of the path analysis tests, the instrumented version of the example program was executed with the test cases from Table X; the results are given in Table XII.

TABLE XII

ARC TRAVERSALS USING

INDEPENDENT SUB-STRUCTURES DATA

	Arc:	Z	A	B	C	D	E	F	G	H	J	K	L	M	N	P	Q	R	S	T	U	V
Test 1		1																				
Test 2			1	1	2	1	1		1			1									1	
Test 3			1	1	2	1	1	1				1			1				1			1
Test 4			1	1	3	1	1		2	1		1			1				1			1
Test 5			1	1	3	1	1		2		1	1									1	
Test 6			1	1	7	1	1	4	2	2	3	1	2	4	6	3	2	7	3	3		1
Total		1	5	5	17	5	5	5	7	3	4	5	2	4	8	3	2	7	5	3	2	3

F. INTUITIVE TESTS

The intuitive test data selected prior to conduct of the previous experimental demonstrations of correctness were used for dynamic testing. No program errors were revealed. For purposes of comparison of the degree of coverage with other methods, Table XIII presents the results of execution of the instrumented program with the intuitive test data listed in Table II (Chapter III, Section A). The selection and testing of these data required 0.7 man-hours of effort; the tests consumed 0.44 seconds of CPU time.

TABLE XIII

ARC TRAVERSALS USING

INTUITIVE DATA

Arc:	Z	A	B	C	D	E	F	G	H	J
Test 1		1	1	2	1	1	1			
Test 2		1	1	3	1	1		2	1	
Test 3	1									
Test 4	1									
Test 5		1	4	256	4	1	3	252	3	251
Test 6		1	1	15	1	1		14		13
Test 7		1	2	100	2	1	1	98		98
Test 8		1	4	256	4	1	1	254		254
Total, Tests 1-8	2	6	13	632	13	6	6	620	4	616

Arc:	K	L	M	N	P	Q	R	S	T	U	V
Test 1	1			1				1			1
Test 2	1			1				1			1
Test 3											
Test 4											
Test 5	1	14	6	6	1	1	10	5	1		1
Test 6	1									1	
Test 7	1	49		1				1			1
Test 8	1	127		1				1			1
Total, Tests 1-8	6	190	6	10	1	1	10	9	1	1	5

V. CONCLUSIONS AND RECOMMENDATIONS

A. COMPARISON OF METHODS

1. Level of Effort

Table XIV is a summary of the man-hours of effort required for each method of demonstrating the correctness of the example program, broken down by procedure for those methods where procedures were examined individually. The cyclomatic number and number of statements for each procedure are included in the table as measures of procedure complexity. The times to apply the verification techniques are to be compared with one another and with original program development times of 5.0 hours to design, 7.0 hours to code, 4.0 hours to debug, and 5.8 hours to test (21.8 hours total).

TABLE XIV

SUMMARY OF TIMES

TO DEMONSTRATE CORRECTNESS

Task or Procedure	Cyclomatic Number and No. of Stmts	Time (Man-Hours) Per Method				
		A	B	C	D	E
Formalize output Specification	--	0.8	0.8	--	-	-
"initialize"	2 ; 14	0.8	0.2	--	-	0.1
Show termination of utilities	-	0.6	-	-	-	-
"read←and←write..."	3 ; 18	4.2	0.7	-	-	0.2
"palindrome←check"	5 ; 7	1.6	0.9	-	-	0.6
"continue←checking"	3 ; 15	1.8	1.3	-	-	0.3
"record←palindrome"	5 ; 21	3.5	3.2	-	-	0.7
"main"	2 ; 6	0.2	0.4	--	-	0.1
Conduct dynamic testing	-	-	*	-	-	0.2
Total		13.5	7.5	1.3	2.2	2.2

Methods:

A - Formal proof
B - Condition table/distributed correctness
C - Basic path analysis
D - Extended path analysis
E - Independent sub-structures
* - Test time included in times for procedures.

Quite expectedly the two more formalized verification techniques (proof and condition table method) required considerably more effort than any of the other methods, and in fact required more hours of effort than were involved in program design (time to construct the proof exceeded even the time to design and code). As the subsequent discussion of the thoroughness of each method indicates, these two methods provide higher confidence in program

correctness than the other simpler and more mechanical methods. The question becomes whether the increased return from formal methodology is worth the cost. The complexity and importance of the software, budget constraints, and several other factors come to bear on the decision, and the decision should be made separately for each software project.

From a philosophical standpoint at least, ex post facto proof of correctness is inefficient because of a great deal of duplication of effort. The logical process of constructing the proof, in the case of this experiment and in general, requires at least as thorough an understanding of the application and of the program control structure and data flow as does the design and implementation effort. The logical techniques of proof can give excellent evidence as to the correctness of programs (but not conclusive; see Ref. 15 and others) and are clearly desirable for use, but a higher cost effectiveness than that suggested from this experiment is required. Possibilities include attempts to automate ex post facto proving of correctness or to introduce the lgocal techniques in the design and implementation of software. These alternatives are major areas of research; brief mention is made here.

Structured programming concepts, advanced language design, and formal definition of requirements are examples of software engineering efforts to emphasize the use of logical methods in the design and implementation of provable programs. Mann has discussed the feasibility of using logic techniques to generate programs guaranteed to satisfy the output specifications, thus

obviating the need for ex post facto verification (26). It is in this area that the greatest promise for correct software is to be found.

Several aspects of the formal proof process may be subject to automation. Manna discussed briefly in Reference 26 (pp. 203-204) progress that has been made in systems to automatically generate invariant assertions and verification conditions and systems (theorem provers) to prove the verification conditions. Both tasks are formidable and it is unlikely that full automation can be achieved; yet, partial solutions would be extremely helpful in reducing the tedium involved in the process of formal proof. The limitations on automation of the process are succinctly stated by Dijkstra (9):

> The distorting spell of speed still seems to take its victims. We see automatic theorem provers proving toy theorems, we see automatic program verifiers verifying toy programs and one observes the honest expectation that with faster machines with lots of concurrent processing, the life-size problems will come within reach as well. But, honest as these expectations may be, are they justified? I sometimes wonder...

The level of effort required to apply the condition table method of selecting test data in a fashion as nearly reliable and valid as possible (according to the theory of testing discussed in Chapter II) compares more favorably with the original program design effort. If similar relationships exist for large-scale applications, the method is likely to be effective at reducing the life cycle cost of software as the method appeared in these experiments to offer a greater ability to locate errors than less formal methods, thus reducing maintenance costs.

Evidence of the cost effectiveness of this or similar formalized test case selection criteria is required from large-scale applications in a commercial environment to be conclusive.

It is interesting to relate the effort required to apply to individual procedures the methods A, B, and E (Table XIV) with the complexity measures of these procedures. While the sample size of the data collected here is too small for statistical significance and the data are distorted by the presence of learning curves (e.g., procedure "read←and←write←input←cards" was the first procedure with loops proven correct, and several false starts were included in the 4.2 hours of effort), there did appear to be a relationship between complexity and effort to demonstrate correctness. Although the level of effort did not linearly relate to the magnitude of either of the measures of complexity shown in Table XIV, effort appeared to increase with increasing complexity, and the cyclomatic number appeared to be a better predictor of effort than did the number of statements.

2. Thoroughness of Verification

As discussed in Section B of Chapter II, the requirement for an ideal test according to the theory of testing presented in that chapter may be satisfied in several ways. Selecting no test data and proving that the program is correct (i.e., contains no errors) clearly satisfies the criteria for an ideal test. Accordingly, on the basis of the proof of correctness constructed as part of this experiment it can be stated that the example program is correct as written, provided the proof contains no

errors and no unwarranted assumptions. That proviso is not easily ignored; Reference 15 is only one source of examples of programs "proven" correct which in fact contain errors. Nonetheless, the proof provides a high degree of confidence in the correctness of the example program.

The several path analysis strategies, including the identification of paths for testing by decomposition into independent sub-structures, did not include any attempt to show the reliability and validity of the test cases selected. Consideration of possible errors in the program, particularly in the statements controlling control flow, revealed several potential errors that would be detected using the test cases of the extended path analysis technique (multi-conditions emphasized) but not by one or both of the other two path analysis methods.

For example, if the statement

```
while ((ix<=length←of←text) and
  (bufferposition<card limit)) do
```

in the procedure "read←and←write←input←cards" had the incorrect relational operator "bufferposition<=" vice "<", none of the test cases selected by the basic path analysis technique or by sub-structure analysis would reveal the error (no string sufficiently long), but test 7 (length 160) of the extended path analysis method would reveal the error through a run-time indexing error on assigning the 81st character to "text." Similarly, if in procedure "record←palindrome" the statement

```
if ((first>=begin←of←palindrome(i)) and
   (last<=end←of←palindrome(i))) then
```

omitted the "or equal" test from the ">=" and "<=" operators, the basic and extended path analysis test case 3 ("baaaaca") would reveal the error by recording three palindromes (baaaaca, baaaaca, and baaaaca) which are included in the larger palindrome (baaaaca) and should be ignored. The error would not be revealed by the test data selected from independent circuit considerations.

Because there exist potential errors that would not be revealed by the path analysis test sets (including the extended method set, as will be shown in the next paragraph), the criteria as applied in these experiments were not valid and reliable in the meaning of testing theory. While the particular errors revealed or not revealed in these experiments are peculiar to the program under test, selection of test data by any means of path analysis other than exhaustive path testing cannot be expected to detect all program errors.

The test cases selected by condition tables were capable of revealing all of the errors considered above, including the two specific errors mentioned (test cases 4, 5, 6 and 7 for the first, test 23 for the second). Additionally, there were potential errors which could be detected through dynamic testing only by the test cases selected by this method. These results are peculiar to the specific program under test but are considered representative of the capabilities of the several methods. (It is not suggested that the condition table method is in general capable of revealing all errors.)

For example, if in procedure "continue←checking" the

statement

```
while ((first>1) and (last<length←of←text)
      and (palindrome=true)) do
```

mistakenly contained "last<=" vice "last<," test case 3 ("baaaaca') of the path analysis techniques and test 6 ("aabbbba") of the independent circuit method would both erroneously cause traversal of the arc "M" (see Figure 2; arc "M" sets palindrome" equal to the value "false") during the calls to the procedure with string (first,last) being the underscored palindrome, but there would be no external effect noticeable to the tester and the error (which has potential to cause an indexing error) would go undetected. However, the condition table for assertion B17-18 (paragraph E.1. of Appendix B) would cause the error to be noticed when examining the predicates of the assertion as required by the method during execution of test element "abb."

This localization of test effort afforded by using the principle of distributed correctness is one of the most powerful aspects of the method as used in this experiment. While the localization added to the effort required (full knowledge of the program's internal structure was required), it was the localization of analysis that enabled some positive statements to be made regarding the validity and reliability of the selected test data. While in several cases it was not possible to prove the generalization assertion for the selected test data assertion, in each case a high degree of confidence was established that the test data did in fact partition the input domain for the program fragment under consideration into equivalence classes with respect to program operation.

From this confidence and the fact that no input data were excluded from selection, it was concluded that the criterion for selecting test data as applied to this program was valid and "almost reliable." Additional work in identifying theorems which can be used in generalizing from specific test data to the entire input domain is needed and offers an opportunity for a highly reliable testing methodology.

The error data provided in Table I, from the original program development process, was not useful for discriminating among the methods applied in verifying the program. Each method would detect the presence of the errors described in Table I.

It is interesting that the set of intuitive test data (Table II) selected prior to thorough analysis of the example program is capable of revealing all of the errors considered, including the "last<=" vice "last<" error in "continue←checking" (a run-time indexing error would occur for "text(last+1)" for test case 8 as a result of that error). However, the method of casual or intuitive selection of test data is not in any way desirable; the error detection capability in this case is due only to luck and the relative simplicity of the program function, and the cost-effectiveness penalty in terms of CPU time expended on test cases which are in fact not distinguishable can be severe.

B. SUMMARY OF CONCLUSIONS AND RECOMMENDATIONS

This section presents a summary of the conclusions drawn in the main text of this thesis.

1. The need exists for greater relative emphasis on design and implementation techniques as compared to verification techniques (pages 13 and 77-78).

2. Significant efforts are required in applying the discipline of software engineering toward refinement or replacement of the verification methods now in use (page 14).

3. The formal proof of correctness and the application of the condition table method based on distributed correctness required significantly more effort than the path analysis strategies (page 76).

4. There are serious limitations on the feasibility of automating the process of formally proving program correctness (page 78).

5. There was some positive correlation of the level of effort required to demonstrate correctness with the complexity of procedures as represented by the cyclomatic number (page 79).

6. The proof of correctness of the example program provides a high degree of confidence in the correctness of the program (page 80).

7. The path analysis strategies as applied in these experiments did not provide reliable and valid criteria for selecting test cases. Selection of test data by any means of path analysis other than exhaustive path testing cannot be expected to detect all program errors (page 81).

8. The condition table method using the principle of distributed correctness afforded a valid and "almost reliable" criterion for test case selection (page 83).

9. Additional work in identifying theorems which can be used in generalizing from specific test data to the entire input domain is needed and offers an opportunity for a highly reliable testing methodology (page 83).

10. Casual or intuitive selection of test data is not in any way desirable (page 83).

APPENDIX A

FORMAL PROOF

A. ASSUMPTIONS, ABBREVIATIONS, AND NOTATION

Several assumptions about the example program were made in addition to those verified by static analysis and those mentioned in Section B of Chapter IV. It was assumed that the host environment of the program (compiler and operating system) was error-free. All input data read by the program were assumed type compatible with variables - integers for integer variables, valid printable characters (including blanks) for string variables. It was assumed that the number of characters following the integer stating the length of the input character string was equal to that integer. The domain of the numerical values in all predicate formulae in the proof was assumed to be the integers only, and only interger division was intended; the operators div and rem were used to represent the integer quotient and remainder, respectively.

Because several of the program variable names are verbose, the abbreviations listed below were used in the assertions and formulae of the proof:

-ℓ	length of text
-ℓt	cardlimit
-cb	cardbuffer
-n	number←of←input←cards
-c	card←counter
-bp	bufferposition
-bop	begin←of←palindrome
-eop	end←of←palindrome
-p	palindrome

Figures 3 through 8, shown in the following pages, are listings of six program procedures with labeled invariant assertions inserted for purposes of the proof. Assertions A0 and A58 are the program input and output assertions. Assertions are contained within braces "{ }," and in Figures 3 through 8 wherever successive labeled assertions follow a program statement, the intermediate assertion to be proved at that point is a conjunction of those assertions. Frequently assertions contain within the braces the names (labels) of earlier assertions; the meaning implied is a literal replacement of the label with its earlier expansion. In the terminology of Section B of Chapter III, the proof presented in this appendix is a proof of the invarient statement:

{A0} program {A58}

This terminology and that for rules of inference (see the same section) are used throughout this appendix.

B. ADDITIONAL RULES OF INFERENCE

As mentioned in Chapter III, rules of inference similar to those in Reference 26 were formulated for iterative for statements and procedure calls. Those rules are presented here:

1. Iterative Rule

The statement for C:=E step 1 until L do F is logically equivalent to the program fragment:

```
        {P}          // an assertion
        C:=E         //C a counter, E an expression
        LIMIT:=L     //L an expression
        {I}          //an assertion
more:   if C>LIMIT then goto fini
        F            //loop body
```

```
            C:=C+1
            goto more
fini:
            {q}          //an assertion
```

Corresponding to this form of the statement, the rule of inference is:

$$\frac{P \Rightarrow I,\ \{I \wedge C<=LIMIT\}\ F\ \{I\},\ I \wedge C>LIMIT \Rightarrow Q}{\{P\}\ \text{for statement}\ \{Q\}}$$

2. Call Rule

All procedures in the example program pass parameters by value, so that operations on the formal parameters within the procedure body do not affect the actual parameters. Global variables may however be modified in the procedures. The notation p(f,g) represents a procedure p with some formal parameters f which operates on some global parameters g; the procedure has a body F and input and output assertions Q and R. A call to the procedure with actual parameters a is denoted by call p(a). S(a:=f) and T(a:=f) are the assertions in the calling program located before and after the call, with formal parameter names substituted for actual. The rule of inference is:

$$\frac{S(a:=f) \Rightarrow Q,\ R \Rightarrow T(a:=f),\ \{Q\}\ F\ \{R\}}{\{S\}\ \text{call}\ p(a)\ \ \{T\}}$$

The rule is essentially a statement that a procedure call is proved when an in-line substitution of the procedure body is shown to be valid. In showing that $R \Rightarrow T(a:=f)$, the prover must verify that global variables referenced in T but not in R are not modified by execution of the procedure body.

C. PROCEDURES TEXT1,2,3

Because these procedures do not contribute to the essential program performance (as was determined during static analysis), only a superficial proof of correctness was performed. It was, however, necessary to show termination in order to verify that program execution would not endlessly loop in one of these non-essential procedures.

1. Input Assertion: {true} i.e., no restriction.
2. Output Assertion: {true}; i.e., no restriction.
3. Verification Condition:

{true} procedure {true}, or true $\wedge$ null $\Rightarrow$ true, where null is a notation for program statements having no significant effect. Proof of the verification condition is immediate.

4. Termination

The procedure has only one entry and one exit and contains no loops; therefore, it terminates.

D. PROCEDURE BLANK←LINES; TERMINATION OF FOR STATEMENTS

Similarly, only a superficial proof of this non-essential procedure was performed.

1. Input Assertion: {n>0} .
2. Output Assertion: {true} ; i.e., no restriction.
3. Verification Condition: {n>0} procedure {true}.
4. Proof

Regardless of the value of the antecedent, the consequent remains the logical value true; therefore, the implication holds.

5. Termination

Termination is assured if the error exit at the statement "assert(n>0)" is not taken, and if the for loop following terminates. The error exit is taken when n<=0. The input assertion guarantees n 0; therefore, the error exit is not taken.

The for statement loop terminates whenever the value of the loop counter exceeds some pre-defined limit. Informally it is clear that, given a finite starting value and finite limit for the counter and given that the loop body itself terminates (as it does in this case - no nested loops) either the loop body is not executed at all (starting value exceeds limit) or eventually the counter must exceed the limit (since it is incremently by 1 following each loop execution, by virtue of the "step 1" portion of the statement, and no other assignments are made to the counter in the loop body), and the loop will terminate.

More formally, let EXP=LIMIT+1-COUNTER be a termination expression, let N be the set of natural numbers, and let > be the usual greater-than relation. Note that N is well-ordered by >. In all implementations of a for statement of the type described in the iterative rule of inference, if the initial value of EXP is zero or not contained in N (i.e., negative), then the loop body is not executed and termination is assured. Likewise, if the initial value of EXP>=1, the loop body is executed, COUNTER is incremented, and the subsequent value of EXP is (EXP-1)$\in$ N. Since an infinite decreasing sequence of values of EXP$\in$ N is not possible (N is well ordered), the loop must terminate (EXP=0).

An important conclusion can be reached from the above proof of termination of the for statement in procedure "blank lines": every occurrence of a statement of the type for C:=E step 1 until L do F terminates provided the program fragment F terminates.

E. PROCEDURE WRITE←ALL←PALINDROMES

As in the preceding cases, only a superficial proof was required for this non-essential procedure (non-essential in terms of the palindrome search problem).

1. Input Assertion: {true}.
2. Output Assertion: {true}.
3. Verification Condition:

   {true} procedure {true}. The proof is immediate.

4. Termination

   The procedure has only one entry and one exit. It contains no loops other than for statements (which have been shown to terminate); therefore the procedure terminates.

F. PROCEDURE INITIALIZE

Figure 3 is a listing of procedure "initialize" with the necessary assertions included. The notation "input(ℓ)" refers to the data value in the input stream which will be assigned to the program variable "ℓ".

1. Input Assertion: Assertion A1.
2. Output Assertion: Assertion A2.
3. Verification Condition: {A1} procedure {A2}.
4. Proof.

   The proof of this verification condition follows directly

```
     procedure initialize;
     comment  initialize all variables, read length_of_text, write text1;
     begin
A1:  { 2<=input(1)<=256 ∧ ca=0 }
     text1;
     jx:=1;
     palindrome_counter:=1;
     cardlimit:=80;
     intfieldsize:=5;
     read(length_of_text);
     if ((length_of_text < 2) or (length_of_text > 256)) then
          begin
          write('Illegal input:',
                '  length of input string is: ',length_of_text);
          assert(false);
          end;
     cardbuffer:=' ';
A2:  { 2<=1<=256 ∧ ca=0 ∧ jx=1 ∧ lt=80 ∧ cb=blank }
     end initialize;
```

FIGURE 3

PROCEDURE INITIALIZE

from the semantic meaning of the assignment statements and read statement. Each predicate of the output assertion is just an expression of an assignment made in the procedure body, except the term "ca=0", which is just a restatement of an input assertion predicate. The proof variable "ca" is discussed in the proof of the next procedure. A series of intermediate assertions could have been made, one following each assignment statement, to more formally indicate the method of proof. In particular, the if statement has been ignored in the preceding simple proof; it is discussed in the termination proof.

Note that in Algol-W the meaning of an assignment of a single character value to a string of length greater than one is to pad out the string variable on the right with blanks; thus the predicate "cb=blank", following the assignment statement "cardbuffer:=" "," means that "cb" initially contains 80 blanks ("cb" is an abbreviation for "cardbuffer", a string array of 80 characters).

5. Termination

The procedure is a concatenation of assignment statements, a read statement, a call to procedure "textl", and an if statement containing a potential error exit (the assert statement). Because "textl" terminates, "initialize" will terminate at the output assertion provided the error exit is not taken. Since assertion A1 ensures that $2 <= \ell <= 256$, the compound statement forming the consequent of the if statement cannot be executed; thus the error exit cannot be taken, and the procedure does terminate.

G. PROCEDURE←READ←AND←WRITE←INPUT←CARDS

Figure 4 contains the assertions for this procedure; several of the assertions use abbreviations listed earlier in this appendix. A proof variable "ca", a variable that is not an actual program variable, was used in this proof to represent the characters assigned, or the number of characters that had been read from the input stream and transferred into the string array "text".

1. Input Assertion: A4

2. Output Assertion: A15

3. Intermediate Assertions:

A5 through A14. Verification conditions are provided for all possible assertion-to-assertion paths in the following paragraphs.

4. Path A4 to A5

The verification condition is:

$2<=\ell<=256 \wedge ca=0 \wedge \ell t=80 \wedge cb=blank$

$\wedge\ n=((\ell-1)\underline{div}\ \ell t)+1 \wedge ix=1 \Longrightarrow$

$2<=\ell<=256 \wedge ca=0 \wedge \ell t=80 \wedge ix=1$

$\wedge\ n>=1 \wedge n=((\ell-1)\underline{div}80)+1$

Given the truth of the antecedents, the consequents are shown true as follows:

a. $2<=\ell<=256 \wedge ca=0 \wedge \ell t=80 \wedge ix=1$:

these predicates are just a restatement of antecedents which have not been affected by the intervening program statements.

```
      procedure read_and_write_input_cards;
      comment  read input cards according to given length_of_text;
      begin

 A4:  { 2<=i<=256 ∧ ca=0 ∧ lt=80 ∧ cb=blank }

      integer number_of_input_cards;
      number_of_input_cards:=(length_of_text - 1) div cardlimit + 1;
      ix:=1;                        comment  reset text index;

 A5:  { 2<=i<=256 ∧ ca=0 ∧ lt=80 ∧ ix=1 ∧ n>=1 ∧ n=((i-1)div80)+1 }

      for card_counter:=1 step 1

 A6:  { (ix-1)rem80=0 ⟹ c=((ix-1)div80)+1 }
 A7:  { 0<=ix-1<=1 ∧ (ix-1<1 ⟹ c<=n) }
 A8:  { ~((ix-1)rem80=0) ⟹ c=((ix-1)div80)+2 }
 A9:  { (ix-1=1 ⟹ c>n) ∧ ((ix-1)rem80=0 ∨ ix-1=1) }
A10:  { 2<=i<=256 ∧ ca=ix-1 ∧ lt=80 ∧ n>=1 ∧ n=((i-1)div80)+1 }

           until number_of_input_cards do
           begin
           write(card_counter);
           writeon(" ");
           readcard(cardbuffer);
           writeon(cardbuffer);
           bufferposition:=0;          comment  reset index;
           while

A11:  { 0<=bp<=lt   0<=ix-i<=1   (ix-1-bp)rem80=0   A10
        cb=string of next 80 or fewer characters }

                 ((ix<=length_of_text) and (bufferposition<cardlimit)) do
                 begin
                 text(ix):=cardbuffer(bufferposition|1);

A12:  { 2<=i<=256 ∧ lt=80 ∧ n>=1 ∧ n=((i-1)div80)+1
      ∧ cb=string of 80 characters }
A13:  { ca=ix ∧ 0<=bp<lt ∧ 1<=ix<=i ∧ (ix-1-bp)rem80=0 }

                 ix:=ix+1;
                 bufferposition:=bufferposition+1;

A14:  { A12 ∧ 0<=ix-1<=1 ∧ ca=ix-1 ∧ [(bp=80 ∧ (ix-1)rem80=0) ∨ ix-1=1] }

                 end;          comment  done for all characters on a card;
           end;                comment  done for all cards;

A15:  { ix-1=1 ∧ ca=1 ∧ 2<=i<=256 ∧ lt=80 }

      end read_and_write_input_cards;
```

FIGURE 4

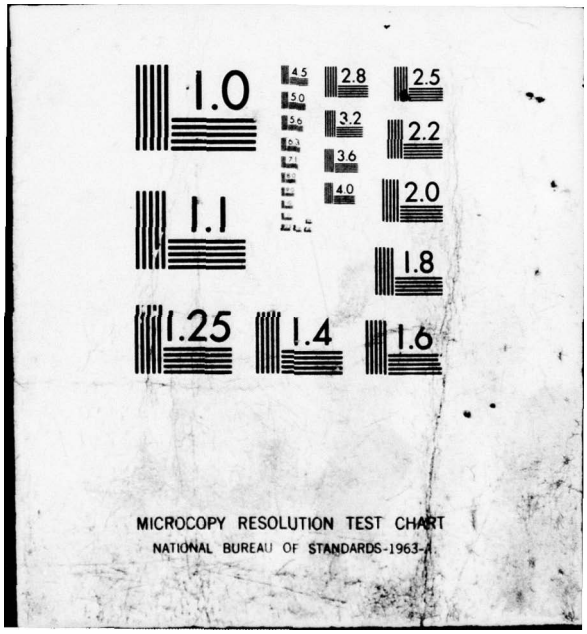
PROCEDURE READ←AND←WRITE←INPUT←CARDS

AD-A062 194 NAVAL POSTGRADUATE SCHOOL MONTEREY CALIF F/G 9/2
EXPERIMENTS IN DOMONSTRATING THE CORRECTNESS OF SOFTWARE.(U)
SEP 78 C W MONK

UNCLASSIFIED NL

2 OF 2
AD A062194

END
DATE FILMED
3 -79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

b. $n=((\ell-1)\underline{div}80)+1$:

follows from $n=((\ell-1)\underline{div}\ \ell t)+1 \wedge \ell t=80$.

c. $n>=1$:

$n=((\ell-1)\underline{div}80)+1$;

$\ell>=2 \Rightarrow \ell-1>=1$;

$\ell-1=1 \Rightarrow (\ell-1)\underline{div}80>=0$;

therefore $n>=1$.

5. Path A5 to A6-10

The verification condition is:

$$A5 \wedge c=1 \Rightarrow A6 \wedge A7 \wedge A8 \wedge A9 \wedge A10$$

The verification condition is proved by considering the consequents one at a time:

a. $(ix-1)\underline{rem}80=0 \Rightarrow c=((ix-1)\underline{div}80)+1$ {A6}:

from A5, $ix=1$;

$ix=1 \Rightarrow ((ix-1)\underline{div}80)+1=0+1=1$;

because c has been set equal to 1, the consequent of the above conditional is true, and the conditional is true regardless of the truth of the antecedent.

b. $0<=ix-1<=\ell$ {from A7} :

$ix=1 \Rightarrow ix-1=0$, and $\ell>=2$.

c. $ix-1<\ell \Rightarrow c<=n$ {from A7}:

$c=1$ and $n>=1$, therefore $c<=n$, and the conditional must hold.

d. $\neg((ix-1)\underline{rem}80=0) \Rightarrow c=((ix-1)\underline{div}80)+2$ {A8}:

$ix=1 \Rightarrow ix-1=0$; thus $(ix-1)\underline{rem}80=0$ is true;

so $\neg((ix-1)\underline{rem}80=0)$ is false,

and the conditional is true.

e. $(ix\text{-}1=\ell \Rightarrow c>n) \wedge ((ix\text{-}1)\underline{rem}80=0 \vee ix\text{-}1=\ell)$ {A9}

ix-1=0 and $\ell>=2$, so ix-1=ℓ is false, and the conditional is true, and (ix-1)rem80=0 is true, and therefore the disjunction is true; thus, A9 must hold.

f. ca=ix-1 {from A10}:

ca=0 and ix=1 is sufficient for ca=ix-1.

g. the remainder of assertion A10:

these predicates are just a restatement of antecedents which have not been affected by the intervening statement.

6. Path A6-10 to A11

The verification condition is:

A6 $\wedge$ A7 $\wedge$ A8 $\wedge$ A9 $\wedge$ A10 $\wedge$ c<=n $\wedge$ bp=0

$\wedge$ cb=string of next 80 or fewer characters $\Rightarrow$ A11

Again the consequents (predicates of A11) are considered one at a time.

a. 0<=bp<=ℓt:

this follows from bp=0 and ℓt=80.

b. (ix-1-bp)rem80=0:

$c<=n \Rightarrow \neg(c>n)$;

$\neg(c>n) \wedge (ix\text{-}1=\ell \Rightarrow c>n) \Rightarrow \neg(ix\text{-}1=\ell)$;

$\neg(ix\text{-}1=\ell) \wedge ((ix\text{-}1)\underline{rem}80=0 \vee ix\text{-}1=\ell) \Rightarrow (ix\text{-}1)\underline{rem}80=0$;

finally, $(ix\text{-}1)\underline{rem}80=0 \wedge bp=0 \Rightarrow (ix\text{-}1\text{-}bp)\underline{rem}80=0$,

which is that which was to be proved.

c. The remainder of the predicates:

these predicates are just a restatement of antecedents which have not been affected by the interveining statements.

7. Path A11 to A12-13

The verification condition is:

A11 $\wedge$ ix<=ℓ $\wedge$ bp<ℓt $\wedge$ text(ix)=cb(bp|1) $\Rightarrow$ A12 $\wedge$ A13

The consequents of the verification condition are considered one at a time.

a. ca=ix {from A13}:

ca=ix-1 $\wedge$ text(ix)=cb(bp|1) is sufficient for ca=ix to be true. text(ix) is the ix-th character, and it is assigned a value in this program fragment (that it is the proper value is shown shortly). If ix-1 characters have been previously correctly assigned to text, and one more character is assigned, then ix characters have been assigned when control reaches assertion A13. In Algol-W, the string indexed by bp|1 is a string of length 1 (i.e., a character) at position bp in the larger string (cb); the first position is 0, and the 80th is 79. 0<=bp<ℓt guarantees that bp is in range; "cb=string of next 80 or fewer characters" ensures that the proper characters are in the buffer, and (ix-1-bp)rem80=0 means that ix and bp+1 (the next character to be assigned in text and the next character in the buffer available for assignment) always differ by a multiple of 80, which is correct when an 80-character buffer is used. So the proper character is being assigned on this control flow path.

b. 0<=bp<ℓt {from A13}:

0<=bp<=ℓt $\wedge$ bp<ℓt $\Rightarrow$ 0<=bp<ℓt.

c. 1<=ix<=ℓ {from A13}:

0<=ix-1<=ℓ $\wedge$ ix<=ℓ $\Rightarrow$ 1<=ix<=ℓ.

c. $1 <= ix <= \ell$ {from A13}:

$0 <= ix-1 <= \ell \wedge ix <= \ell \Rightarrow 1 <= ix <= \ell$.

d. A12 and remaining predicate of A13:

these predicates are just a restatement of antecedents.

8. Path A12-13 to A11 (return to start of loop)

The verification condition is:

$$A12 \wedge A13 \quad ix:=ix+1;\ bp:=bp+1 \quad \{A11\}$$

The verification condition as expressed above is rewritten as follows, replacing ix and bp by ix+1 and bp+1 in assertion A11 (as the assignment rule of inference requires):

$$A12 \wedge A13 \Rightarrow$$

$$[0 <= bp+1 <= \ell t \wedge 0 <= ix <= \ell \wedge (ix-bp-1)\ \underline{rem}\ 80 = 0 \wedge 2 <= \ell <= 256$$

$$\wedge\ ca = ix \wedge \ell t = 80 \wedge n >= 1 \wedge n = ((\ell-1)\ \underline{div}\ 80)+1$$

$$\wedge\ cb = \text{string of next 80 or fewer characters}]$$

The verification condition is once more proved by considering the consequents one at a time.

a. $0 <= bp+1 <= \ell t$:

$0 <= bp \Rightarrow 0 <= bp+1$; and since only integer arithmetic is permitted, $bp < \ell t \Rightarrow bp+1 <= \ell t$.

b. the remainder of the consequents:

restatements of predicates of A12 and A13.

9. Path A11 to A14

The verification condition is:

$$A11 \wedge \neg(ix <= \ell \wedge bp < \ell t) \Rightarrow A14$$

The proof is constructed by showing the consequents of the verification condition one at a time.

a. [(bp=80 ∧ (ix-1)rem80=0) ∨ ix-1=ℓ]:

¬(ix<=ℓ ∧ bp<ℓt) means that either ix>ℓ or bp>=ℓt, or both.

Consider ix>ℓ as case 1;

ix>ℓ ∧ 0<=ix-1<=ℓ ⇒ ℓ<ix<=ℓ+1,

so ix=ℓ+1 and ix-1=ℓ , in which case the consequent is true.

Consider bp>=ℓt as case 2;

bp>=ℓt ∧ 0<=bp<=ℓt ⇒ bp=ℓt;

ℓt=80 ∧ bp=ℓt ⇒ bp=80;

bp=80 ⇒ (ix-1-bp)rem80=(ix-1)rem80;

since (ix-1-bp)rem80=0, then (ix-1)rem80=80,

and the consequent is true.

So in either case the disjunctive consequent holds.

b. the remainder of the predicates:

these predicates are just a restatement of antecedents.

10. Path A6-10 to A14

The verification condition is:

{A6 ∧ A7 ∧ A8∧A9∧A10} F; while C do P {A14} ,

where F represents the statements between A6-10 and the while statement, C represents the predicate of the while statement and P is the while loop body. The previous proofs of the verification conditions for paths A6-10 to A11, A11 to A12-13, A12-13 to A11, and A11 to A14 satisfy the requirements of the rule of inference for while statements, and therefore the verification condition for path A6-10 to A14 is proven.

11. Path A14 to A6-10

The verification condition after substituting c+1 for

c in assertions A6-A9 (as required by assignment rule applicable to "for c:=1 step 1" which increments c) is:

$$A14 \Rightarrow [((ix-1)\underline{rem}80=0 \Rightarrow c+1=((ix-1)\underline{div}80)+1)$$
$$\wedge\ 0<=ix-1<=\ell \wedge (ix-1<\ell \Rightarrow c+1<=n)$$
$$\wedge \neg((ix-1)\underline{rem}80=0) \Rightarrow c+1=((ix-1)\underline{div}80)+2$$
$$\wedge (ix-1=\ell \Rightarrow c+1>n) \wedge ((ix-1)\underline{rem}80=0 \vee ix-1=\ell) \wedge A10]$$

The proof of the verification condition is shown for each consequent.

a. A10 $\wedge$ 0<=ix-1<=ℓ $\wedge$ ((ix-1)rem80=0 $\vee$ ix-1=ℓ):
these predicates are just a restatement of antecedents.

b. (ix-1)rem80=0 $\Rightarrow$ c+1=((ix-1)div80)+1:
In addition to being a counter for the for loop, "c" is a count of the number of data cards that have been read, because there is exactly one readcard statement in the for loop body. ca=ix-1 is the number of characters that have been assigned from the buffer into the array "text"; each time the loop is executed, 80 characters are assigned into "text", except that the last time the loop is executed, from 1 to 80 characters may be assigned.
If (ix-1)rem80=0, an even multiple (namely (ix-1)div80) of 80 characters have been assigned, and loop has been executed that number of times as control returns to A6-10; thus, c+1 (the new value after the step) is then one more than that number, or c+1=((ix-1)div80)+1. Therefore the consequent holds.

c. $\neg$((ix-1)rem80=0) $\Rightarrow$ c+1=((ix-1)div80)+2:
The proof for this consequent is similar to the preceding, except that because (ix-1)rem80 is not equal to zero, less than

80 characters have been read when control returns to A6-10, and the loop has been executed one time more than in the case above. Thus, c+1=((ix-1)div80)+2; q.e.d.

d. ix-1<ℓ ⇒ c+1<=n:

If ix-1<ℓ, then since

(ix-1=ℓ ∨ (ix-1)rem80=0) (proved above),

it must be that (ix-1)rem80=0.

Therefore, from the above proved conditional,

c+1=((ix-1)div80+1 is true,

and ix-1<ℓ ⇒ ix-1<=ℓ-1,

so c+1<=((ℓ-1)div80)+1.

Then because ((ℓ-1)div80)+1=n, c+1<=n.

e. ix-1=ℓ ⇒ c+1>n:

There are two possibilities.

First, assume ¬((ix-1)rem80=0.

Then c+1=((ix-1)div80)+2,

and c+1>((ix-1)div80)+1.

If ix-1=ℓ, then ix-1>ℓ-1 and

(ℓ-1)div80 =(ix-1)div80;

therefore c+1>((ℓ-1)div80)+1, thus c+1>n.

Second, assume (ix-1)rem80=0.

Then c+1=((ix-1)div80)+1,

and since ℓ=ix-1, ℓ rem 80=0

and (ℓ-1)div80<ℓ div 80.

Therefore c+1>((ℓ-1)div80)+1, thus c+1>n.

In either case, c+1>n and the consequent is proved.

12. Path A6-10 to A15

The verification condition is:

A6 $\wedge$ A7 $\wedge$ A8 $\wedge$ A9 $\wedge$ A10 $\wedge$ c>n $\Rightarrow$

ix-1=ℓ $\wedge$ ca=ℓ $\wedge$ 2<=ℓ<=256 $\wedge$ ℓt=80

2<=ℓ<=256 and ℓt=80 are restatements of predicates contained in the antecedent of the verification condition; the remaining two predicates are verified as follows:

a. ix-1=ℓ:

ix-1<ℓ $\Rightarrow$ c<=n is true (an antecedent), so the contrapositive is also true:

c>n $\Rightarrow$ (ix-1)>=ℓ.

Since c>n, ix-1>=ℓ.

Also, 0<=ix-1<=ℓ (an antecedent), therefore ix-1=ℓ

b. ca=ℓ:

ca=ix-1 $\wedge$ ix-1=ℓ $\Rightarrow$ ca=ℓ.

13. Path A5 to A15

The verification condition is:

A5 for statement {A15}.

The previous proofs of the verification conditions for paths A5 to A6-10, A6-10 to A14, A14 to A6-10, and A6-10 to A15 satisfy the requirements of the for rule; therefore the verification condition for this path is proved.

14. Input Assertion to Output Assertion

The proof of partial correctness for this procedure is completed by concatenation of paths A4 to A5 and A5 to A15.

15. Termination

During the preceding proof of partial correctness, the reader should have been convinced that the procedure terminates by the relations among ix-1, c, and n, and by those among ix, ℓ, bp, and ℓt. If not, termination is assured because the program has one entry and one exit and is a concatenation of assignment statements and a for statement. The for statement terminates if its loop body terminates; in this case, the body terminates provided the while loop eventually terminates.

The formality of well-ordering could be applied to show the termination of the while statement; however, termination is evident since "ℓt" is fixed at 80 and "bp" starts from 0 and is incremented by one on each execution of the loop body (which terminates as it has no loops). Thus "bp" must eventually exceed "ℓt" and the while statement must terminate (it may terminate earlier if ix=ℓ).

H. PROCEDURE PALINDROME←CHECK

Figure 5 contains the assertions for this procedure.

1. Input Assertion: A17.

2. Output Assertion: A27-29.

3. Intermediate Assertions:

A18 through A26. Verification conditions are provided for all possible assertion-to-assertion paths in the following paragraphs.

4. Path A17 to A18-22

The verification condition is:

```
      procedure palindrome_check;
      comment  find all palindromes within given text string;
      begin

A17:  { 2<=1<=256 ∧ ca=1 ∧ jx=1 }

            comment  scan text from left to right;
      for ix:=2 step 1

A18:  { 2<=1<=256 ∧ ca=1 ∧ jx>0 ∧ 2<=ix<=1+1 }
A19:  { ∀x[(2<=x<=ix-1 ∧ text(x-1)=text(x)) ⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-1 ∧ x<=eop(y)<=1)] }
A20:  { ∀x[(3<=x<=ix-1 ∧ text(x-2)=text(x)) ⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-2 ∧ x<=eop(y)<=1)] }
A21:  { ∀y[(0<y<jx ∧ bop(y)>1 ∧ 0<eop(y)<1) ⇒
         ~(text(bop(y)-1)=text(eop(y)+1))] }
A22:  { ∀y[ (0<y<jx ∧ ~(eop(y)=0)) ⇒
         [string(bop(y),eop(y))=ok ∧ bop(y)>=1 ∧ eop(y)<=1
          ∧ ∀z( (0<z<jx ∧ ~(z=y)) ⇒
               (~(bop(z)=bop(y)) ∧ (bop(z)<bop(y) ⇒ eop(z)<eop(y))) )] ] }

            until length_of_text do
            begin
            if text(ix-1) = text(ix) then continue_checking((ix-1),ix);

A23:  { 2<=1<=256 ∧ ca=1 ∧ jx>0 ∧ 2<=ix<=1 ∧ A20 ∧ A21 ∧ A22 }
A24:  { ∀x[(2<=x<=ix ∧ text(x-1)=text(x))⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-1 ∧ x<=eop(y)<=1)] }

            if ix ~= 2 then
                 if text(ix-2)=text(ix) then continue_checking((ix-2),ix);

A25:  { A23 ∧ A24 ∧ A21 ∧ A22 }
A26:  { ∀x[(3<=x<=ix ∧ text(x-2)=text(x))⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-2 ∧ x<=eop(y)<=1)] }

            end;

A27:  { 2<=1<=256 ∧ ca=1 ∧ jx>0 ∧ A21 ∧ A22 }
A28:  { ∀x[(2<=x<=1 ∧ text(x-1)=text(x)) ⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-1 ∧ x<=eop(y)<=1)] }
A29:  { ∀x[(3<=x<=1 ∧ text(x-2)=text(x))⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-2 ∧ x<=eop(y)<=1)] }

      end palindrome_check;
```

FIGURE 5

PROCEDURE PALINDROME←CHECK

A17 ∧ ix=2 ⇒ A18 ∧ A19 ∧ A20 ∧ A21 ∧ A22

Proof is shown by considering the consequents one at a time.

a. 2<=ℓ<=256 ∧ ca=ℓ {from A18}:

These two predicates are a restatement of part of the input assertion. Their truth is not modified in this procedure, and they are repeated in all intermediate assertions. They will not be discussed in the discussion of the remaining verification conditions for this procedure.

b. jx>0 {from A18}:

jx=1 ⇒ jx>0.

c. 2<=ix<=ℓ+1 {from A18}:

(ix=2 ∧ 2<=ℓ) ⇒ 2<=ix<=ℓ+1.

d. A19:

No x can satisfy the antecedent 2<=x<=ix-1 because ix-1=1; therefore the conditional is true.

e. A20:

This consequent is similarly true.

f. A21:

No integer y can satisfy the antecedent 0<y<jx because jx=1; therefore the conditional is true.

g. A22:

Similarly.

5. Path A18-22 to A27-29

The verification condition is:

A18 ∧ A19 ∧ A20 ∧ A21 ∧ A22 ∧ ix>ℓ ⇒A27 ∧ A28 ∧ A29

The consequents are all restatements of the antecedents except A28 and A29; proofs of their validity follow:

a. A28:

$ix <= \ell + 1 \wedge ix > \ell \Rightarrow ix - 1 = \ell$.

Therefore assertion A28 is a restatement of A19 with ℓ replacing ix-1 in the first predicate of the antecedent, and the assertion holds.

b. A29:

This assertion holds similarly.

6. Path A18-22 to A23-24, Case 1

Case 1 for this path is arrival of control at assertions A23-24 after execution of the if statement with true predicates. In this case, the verification condition is:

$$A18 \wedge A19 \wedge A20 \wedge A21 \wedge A22 \wedge$$
$$ix <= \ell \wedge text(ix-1) = text(ix) \Rightarrow A23 \wedge A24$$

The following condition, deducible from the antecedents of the verification condition, becomes the input assertion to procedure "continue←checking" whenever the actual parameters "ix-1" and "ix" are replaced by the formal parameters "first" and "last":

$$2 <= \ell <= 256 \wedge ca = \ell \wedge jx > 0 \wedge 1 <= ix-1 <= \ell - 1 \wedge 2 <= ix <= \ell$$
$$\wedge ix-1 < ix \wedge text(ix-1) = text(ix) \wedge A21 \wedge A22$$

The output specification of the procedure "continue←checking," after replacing "current" by the value "current" was assigned at the procedure call in the proof of correctness of that procedure, namely the value "ix", is:

A21 ∧ A22 ∧ 2<=ℓ <=256 ∧ ca=ℓ ∧ jx>0 ∧

∃y(y<jx ∧ 1<=bop(y)<=ix-1 ∧ ix<=eop(y)<=ℓ)

Given that the procedure "continue←checking" has been proven correct (presented in the next section of this appendix), all requirements for the rule of inference for procedure calls have been satisfied, and the output assertion of "continue←checking," as rewritten above, can be used to show the truth of assertions A23-24. In fact, assertion A23 is entirely a restatement of either this output assertion or the antecedents of the verification condition. Assertion 19 ensures the validity of assertion A24 over the range of x from 2 to ix-1;

(text(ix-1)=text(ix) (an antecedent)

and ∃y(y<jx ∧ 1<=bop(y)<=ix-1 ∧ ix<=eop(y)<=ℓ)

(from the output assertion of "continue checking") together extend the range of x to ix, and therefore assertion A24 holds. Thus the verification condition for this case has been proved.

7. Path A18-22 to A23-24, Case 2

Case 2 is the case when the if statement preceding A23-24 is executed with the predicates false; in this case, the verification condition is:

A18 ∧ A19 ∧ A20 ∧ A21 ∧ A22 ∧

ix<=ℓ ∧ ¬(text(ix-1)=text(ix)) ⇒A23 ∧ A24

A11 of the consequents but A24 are a restatement of antecedents; A24 is a restatement of A19 with the range of x increased to include x=ix, the ix-th value having been checked for compliance with the assertion in the current iteration of the for loop.

Since the antecedent of the conditional contained in assertion A24 is false in this case for x=ix, the conditional is true for x=ix, and assertion A24 holds. Thus the verification condition is proved.

8. Path A23-24 to A25-26, Case 1

This first case occurs whenever ix=2, or when $\neg$(ix=2) and also $\neg$(text(ix-2)=text(ix)); in this case, no call is made to procedure "continue←checking", and the verification condition is:

$$A23 \wedge A24 \Rightarrow A25 \wedge A26$$

Assertion A25 is a restatement of antecedents, and assertion A26 is just assertion A20 with the range of x extended to include x=ix. The conditions for this case ensure that for x=ix the antecedent of the conditional contained in A26 is false; either ix=2 and 3<=x is false or $\neg$(text(ix-2)=text(ix)). Therefore the conditional is true for x=ix and thus assertion A26 holds; the verification condition is proved.

9. Path A23-24 to A25-26, Case 2

In this case the call to "continue←checking" is executed, and the verification condition is:

$$A23 \wedge A24 \wedge \neg(ix=2) \wedge text(ix-2)=text(ix) \Rightarrow A25 \wedge A26$$

Similarly to the proof for the previous path containing a call to procedure "continue←checking", it may be shown that the antecedents of the verification condition satisfy the procedure's

input assertion and that the output assertion, together with the antecedents, satisfy assertions A25-26. In assertion A26 the range for the universal quantifier x is extended to include the case x=ix as before.

10. Path A25-26 to A18-22

The verification condition is (substituting ix+1 for ix in A18-20 because of the assignment):

$$A25 \wedge A26 \Rightarrow A21 \wedge A22 \wedge$$
$$2 \leq \ell \leq 256 \wedge ca = \ell \wedge jx > 0 \wedge 2 \leq ix+1 \leq \ell+1$$
$$\wedge \forall x[(2 \leq x \leq ix \wedge text(x-1)=text(x)) \Rightarrow$$
$$\exists y(y < jx \wedge 1 \leq bop(y) \leq x-1 \wedge x \leq eop(y) \leq \ell)]$$
$$\wedge \forall x[(3 \leq x \leq ix \wedge text(x-2)=text(x)) \Rightarrow$$
$$\exists y(y < jx \wedge 1 \leq bop(y) \leq x-2 \wedge x \leq eop(y) \leq \ell)]$$

A11 of the consequents but $2 \leq ix \leq \ell+1$ are restatements of antecedents;

and $2 \leq ix \leq \ell \Rightarrow 3 \leq ix+1 \leq \ell+1$, thus $2 \leq ix+1 \leq \ell+1$ is true and the verification condition is proved.

11. Path A17 to A27-29

The verification condition is:

{A17} for statement {A27-29}

The previous proofs of the verification conditions for paths A17 to A18-22, A18-22 to A23-24 to A25-26, A25-26 to A18-22, and A18-22 to A27-29 satisfy the requirements of the for rule; therefore the verification condition for this path is proved, thus completing the proof of partial correctness for this procedure.

12. Termination

The procedure has one entry and one exit, and the only loop is a for statement, which have been shown to terminate. Therefore the procedure terminates.

I. PROCEDURE CONTINUE←CHECKING

Figure 6 contains the assertions for this procedure. A constant "current" is introduced in the proof and the assertions; this constant is given the value of "last" at the time of the procedure call.

1. Input Assertion: A30-32.

2. Output Assertion: A38.

3. Intermediate Assertions:

A33 through A37. Verification conditions are provided for all possible assertion-to-assertion paths in the following paragraphs.

4. Path A30-32 to A33

The verification condition is:

$A30 \wedge A31 \wedge A32 \wedge p=true \Rightarrow A33$

The consequent is a restatement of the antecedents with the addition of the predicate "current=last," as mentioned above. Thus proof of the verification condition is immediate.

5. Path A33 to A34

The terminology "string(first,last)=ok" used in these assertions indicates that the substring from text(first) to text(last) is a valid palindrome. The verification condition for this path is:

```
      procedure continue_checking (integer value first,last);
      comment  Given first and last as pointers to a palindrome
               of size 2 or 3, this procedure checks whether or not this
               palindrome is included in a larger palindrome;
      begin

A30:  { 2<=1<=256 ∧ ca=1 ∧ jx>0 ∧ 1<=first<=1-1 ∧ 2<=last<=1 ∧ first<last
      ∧ text(first)=text(last) }
A31:  { ∀y[(0<y<jx ∧ bop(y)>1 ∧ 0<eop(y)<1) ⇒
        ~(text(bop(y)-1)=text(eop(y)+1))] }
A32:  { ∀y[ (0<y<jx ∧ ~(eop(y)=0)) ⇒
         [string(bop(y),eop(y))=ok ∧ bop(y)>=1 ∧ eop(y)<=1
          ∧ ∀z( (0<z<jx ∧ ~(z=y)) ⇒
               (~(bop(z)=bop(y)) ∧ (bop(z)<bop(y) ⇒ eop(z)<eop(y))) )] ] }

      logical palindrome;
      palindrome:=true;

A33:  { A30 ∧ A31 ∧ A32 ∧ p=true ∧ current=last }

      while

A34:  { A30 ∧ A31 ∧ A32 ∧ current<=last ∧ string(first,last)=ok
      ∧ (p=false ⇒ ~(text(first-1)=text(last+1))) }

           ((first>1) and (last<length_of_text) and (palindrome=true)) do
           begin
           if text(first-1) = text(last+1) then
                begin
                comment  larger palindrome found;
                first:=first-1;
                last:=last+1;

A35:  { A34 }

                end
           else
                begin
                palindrome:=false;       comment  largest palindrome found;

A36:  { A34 }

                end;
           end;

A37:  { A30 ∧ A31 ∧ A32 ∧ current<=last ∧ string(first,last)=ok
      ∧ (first=1 ∨ last=1 ∨ ~(text(first-1)=text(last+1))) }

      record_palindrome(first,last);

A38:  { A31 ∧ A32 ∧ 2<=1<=256 ∧ ca=1 ∧ jx>0 }
      ∧ ∃y(y<jx ∧ 1<=bop(y)<=current-1 ∧ current<=eop(y)<=1) }

      end continue_checking;
```

FIGURE 6

PROCEDURE CONTINUE-CHECKING

A33 $\Rightarrow$ A34

The consequents are considered one at a time.

a. A30 $\wedge$ A31 $\wedge$ A32:

these predicates are just a restatement of antecedents.

b. current<=last:

follows directly from current=last.

c. string(first,last)=ok:

static analysis of the program showed that, for all calls to this procedure, either first=last-1 or first=last-2 (this could have been a predicate of the input assertion), and this fact and text(first)=text(last) ensures that string(first,last) is a palindrome when control reaches assertion A34 from A33.

d. p=false $\Rightarrow$ $\neg$(text(first-1)=text(last+1):

since p=true, the conditional is true regardless of the truth of the consequent.

6. Path A34 to A35

If program control reaches assertion A35, then the predicates of the while and if statements are true and the verification condition for this path is (substituting first-1 and last+1 for first and last in the consequent, due to the assignment statements):

A34 $\wedge$ first>1 $\wedge$ last<ℓ $\wedge$ p=true

$\wedge$ text(first-1)=text(last+1) $\Rightarrow$

2<=ℓ<=256 $\wedge$ ca=ℓ $\wedge$ jx>0 $\wedge$ 1<=first-1<=ℓ-1 $\wedge$ 2<=last+1<=ℓ

$\wedge$ first-1<last+1 $\wedge$ text(first-1)=text(last+1) $\wedge$ A31 $\wedge$ A32

$\wedge$ current<=last+1 $\wedge$ string(first-1,last+1)=ok

The consequents of this verification condition are considered one at a time.

a. $2 \le \ell \le 256 \wedge ca = \ell \wedge jx > 0$:

these predicates are just a restatement of antecedents; they remain valid throughout this procedure and will not be discussed during the proof of further verification conditions.

b. $1 \le first-1 \le \ell-1$:

$1 \le first \le \ell-1 \wedge first > 1 \Rightarrow 1 \le first-1 \le \ell-1$.

c. $2 \le last+1 \le \ell$:

$2 \le last \le \ell \wedge last < \ell \Rightarrow 2 \le last+1 \le \ell$.

d. $first-1 < last+1$:

follows directly from $first < last$.

e. $text(first-1) = text(last+1) \wedge A31 \wedge A32$:

these predicates are just a restatement of antecedents.

f. $current \le last+1$:

follows directly from $current \le last$.

g. $string(first-1, last+1) = ok$:

$string(first, last) = ok \wedge text(first-1) = text(last+1)$

$\Rightarrow string(first-1, last+1) = ok$.

This concludes the proof of verification condition.

7. Path A34 to A36

If control reaches assertion A36 from A34, then the predicate of the while statement is true, that of the if statement false, and the verification condition is:

$$A34 \wedge first > 1 \wedge last < \ell \wedge \neg(text(first-1) = text(last+1)) \wedge p = false \Rightarrow A34$$

The consequents are shown one at a time.

a. p=false $\Rightarrow$ $\neg$(text(first-1)=text(last+1)):

the antecedent and consequent of this conditional are both true (antecedents of the verification condition); therefore, the conditional is true.

b. the remainder of the consequents:

these predicates are just a restatement of antecedents.

8. Paths A35 to A34 and A36 to A34

Since A35 and A36 are each identical to A34 and since there are no program statements on these paths, the verification condition is A34 $\Rightarrow$ A34, which must be true.

9. Path A34 to A37

The verification condition is:

A34 $\wedge$ $\neg$(first>1 $\wedge$ last<ℓ $\wedge$ p=true) $\Rightarrow$ A37

The consequents are considered one at a time.

a. [first=1 $\vee$ last=ℓ

$\vee$ $\neg$(text(first-1)=text(last+1))]:

from the antecedent $\neg$(first>1 $\wedge$ last<ℓ $\wedge$ p=true),

DeMorgan' Rule gives:

first<=1 $\vee$ last>=ℓ $\vee$ p=false.

Since also first>=ℓ and last<=ℓ , and

since p=false $\Rightarrow$ (text(first-1)=text(last+1),

the above is equivalent to:

first=1 $\vee$ last=ℓ $\vee$ $\neg$(text(first-1)=text(last+1));

thus the consequent is shown.

b. the remainder of the consequents:

these predicates are just a restatement of antecedents.

10. Path A33 to A37

The verification condition is:

{A33} while statement {A37}

The previous proofs of the verification conditions for paths A33 to A34, A34 to A35, A34 to A36, A35 to A34, A36 to A34, and A34 to A37 are sufficient to show this verification condition.

11. Path A37 to A38

Assertion A37 satisfies the input assertion to procedure "record←palindrome" (in this case the names first, last and current retain the same connotations). Assertion A38 is precisely the output assertion of "record←palindrome." Therefore the verification condition for path A37 to A38 is verified by the proof of correctness for the called procedure (in the next section).

12. Input Assertion to Output Assertion

The proof of partial correctness for this procedure is completed by concatenation of paths A30-32 to A33, A33 to A37, and A37 to A38.

13. Termination

Procedure "continue←checking" has one entry, one exit, and but one loop, the while statement. Assuming "record←palindrome" terminates (proven elsewhere), this procedure terminates if the loop terminates. Clearly the loop body terminates, so loop execution will terminate if one of the three conditions:

first>1, last<ℓ , or p=true

ever takes on a value of false. Since initially first>=1 and "first" is decremented by 1 on each loop iteration for which "p" is not set equal to false (in which case termination would be assured), then the well-ordering principle of the natural numbers requires eventually first<=1 (unless the loop terminates sooner). So the procedure terminates.

J. PROCEDURE RECORD←PALINDROME

Figure 7 contains the assertions for this procedure.

1. Input Assertion: A39-41.

2. Output Assertion: A51-52.

3. Intermediate Assertions:

A43 through A48. Verification conditions are provided for all possible assertion-to-assertion paths in the following paragraphs.

4. Path A39-41 to A42-44

The verification condition is (substituting 1 for i in assertions A42 and A44 because of the assignment to the for loop counter):

$$A39 \wedge A40 \wedge A41 \wedge entry=true \Rightarrow$$

$$A39 \wedge A40 \wedge A41 \wedge 1<=jx \wedge A43 \wedge$$

$$[\ (entry=true \wedge \neg(eop(0)=0)) \Rightarrow \forall z[0<z<j \Rightarrow$$

$$(\ \neg(bop(z)=first) \wedge (bop(z)<first \Rightarrow eop(z)<last))]\ \ ]$$

The consequents are considered one at a time.

a. A39 ∧ A40 ∧ A41:

these predicates are just a restatement of antecedents.

b. 1<=jx:

follows directly from jx>0.

c. A43:

since entry=true, the antecedent of the conditional is false and the conditional is true.

d. the remaining complex predicate:

since there is no integar z satisfying 0<z<i, the consequent of the antecedent of this predicate is true, and the conditional is shown; thus, the final consequent of the verification condition is proved.

5. Path A42-44 to A45-46

In the event control passes to assertion A45-46 from A42-44, the predicate of the intervening if statement is true and the verification condition is:

A42 ∧ A43 ∧ A44 ∧ i<=jx-1 ∧ first>=bop(i)

∧ last<=eop(i) ∧ entry =false ⇒A45 ∧ A46

Proof is by considering the consequents one at a time.

a. A39 ∧ A40 ∧ A41 {from A45}:

these predicates are just a restatement of antecedents.

b. i<=jx-1 {from A45}:

this predicate is a restatement of the antecedent resulting from the test on the loop counter.

c. entry=false ⇒ ∃y(y<=jx ∧ 1<=bop(y)<=current-1

∧ current<=eop(y)<=ℓ) {A43}:

entry=false is true; thus it must be shown that there exists a value for y such that the predicates following the "existential"

```
      procedure record_palindrome (integer value first,last);
      comment  Record only max length palindromes.  Flag previously
               recorded palindromes if they are included in the palindrome
               specified by first and last.
               jx was initialized to 1.  After completion jx points to the
               next entry in begin_of_palindrome and end_of_palindrome;
      begin

A39:  { 2<=i<=256 ∧ ca=1 ∧ jx>0 ∧ 1<=first<=i-1 ∧ 2<=last<=i ∧ first<last
      ∧ current<=last ∧ string(first,last)=ok
      ∧ (first=1 ∨ last=1 ∨ ~(text(first-1)=text(last+1))) }
A40:  { ∀y[(0<y<jx ∧ bop(y)>1 ∧ 0<eop(y)<i)⇒
         ~(text(bop(y)-1)=text(eop(y)+1))] }
A41:  { ∀y[ (0<y<jx ∧ ~(eop(y)=0))⇒
         [string(bop(y),eop(y))=ok ∧ bop(y)>=1 ∧ eop(y)<=i
          ∧ ∀z( (0<z<jx ∧ ~(z=y))⇒
               (~(bop(z)=bop(y)) ∧ (bop(z)<bop(y) ⇒ eop(z)<eop(y))) )] ] }

      integer i;                    comment  local counter;
      logical entry;
      entry:=true;
      for i:=1 step 1

A42:  { A39 ∧ A40 ∧ A41 ∧ i<=jx }
A43:  { entry=false ⇒∃y(y<=jx ∧ 1<=bop(y)<=current-1 ∧ current<=eop(y)<=i) }
A44:  { (entry=true ∧ ~(eop(i-1)=0)) ⇒
         ∀z[0<z<i ⇒ (~(bop(z)=first) ∧ (bop(z)<first ⇒ eop(z)<last))] }

           until jx-1 do
           begin
           if ((first>=begin_of_palindrome(i))
                and (last<=end_of_palindrome(i))) then
                begin
                comment  Palindrome is entirely included in a previously
                         recorded palindrome.  No entry required·
                entry:=false;

A45:  { A39 ∧ A40 ∧ A41 ∧ i<=jx-1 ∧ A43 }
A46:  { (entry=true ∧ ~(eop(i)=0)) ⇒
         ∀z[0<z<=i ⇒ (~(bop(z)=first) ∧ (bop(z)<first ⇒ eop(z)<last))] }

                end
           else
                begin
                if ((begin_of_palindrome(i) >= first)
                     and (end_of_palindrome(i) <= last)) then
                     begin
                     end_of_palindrome(i):=0;
                     comment  flag smaller palindrome;
                     end;

A47:  { A45 ∧ A46 }

                end;

A48:  { A45 ∧ A46 }

           end;       comment  All previously recorded palindromes
                               compared with last input;
```

FIGURE 7

Page 1 of 2

PROCEDURE RECORD-PALINDROME

```
A49:  { A39 ∧ A40 ∧ A41 ∧ A43 }
A50:  { (entry=true ∧ ~(eop(jx)=0)) ⇒
          ∀z[0<z<jx ⇒(~(bop(z)=first) ∧ (bop(z)<first ⇒ eop(z)<last))] }

      if entry = true then
           begin
           comment  larger than all previous or overlapping or disjoint;
           begin_of_palindrome(jx):=first;
           end_of_palindrome(jx):=last;
           jx:=jx+1;
           end;

A51:  { A40 ∧ A41 ∧ 2<=1<=256 ∧ ca=1 ∧ jx>0 }
A52:  { ∃y(y<jx ∧ 1<=bop(y)<=current-1 ∧ current<=eop(y)<=1) }

      end record_palindrome;
```

FIGURE 7

Page 2 of 2

PROCEDURE RECORD←PALINDROME

quantifier hold.

It is proposed that y=i will satisfy those conditions.

Since i<=jx-1, i<jx holds.

i<jx ∧ A41 ⇒ bop(i)>=1;

current<=last ∧ first<last ⇒ current-1<=first;

current-1<=first ∧ bop(i)<=first ⇒ bop(i)<=current-1;

therefore 1<=bop(i)<=current-1 holds.

current<=last ∧ last<=eop(i) ⇒ current<=eop(i);

i<jx ∧ A41 ⇒ eop(i)<=ℓ;

so current<=eop(i)<=ℓ holds.

Therefore, the necessary predicates are all true when y is chosen equal to i; this consequent of the verification condition is proved.

d. A46:

since entry=false is an antecedent of the verification condition, entry=true is false and the conditional which is assertion A46 is true.

6. Path A42-44 to A47, Case 1

Control can pass to assertion A47 from A42-44 either by executing the compound statement with the comment "flag smaller palindrome" or by failing to execute that compound statement when the predicate of the preceding if is false. For case 1, the case where the predicate is false, the verification condition is:

$$A42 \wedge A43 \wedge A44 \wedge i \leq jx-1 \wedge$$
$$\neg(first \geq bop(i) \wedge last \leq eop(i)) \wedge$$
$$\neg(bop(i) \geq first \wedge eop(i) \leq last)$$

$\Rightarrow$ A47

All of the consequents contained in A47= A45 $\wedge$ {A46} except A46 are restatements of the antecedents.

a. (entry=true $\wedge$ $\neg$(eop(i)=0)) $\Rightarrow$

$\forall z[0<z<=i \Rightarrow (\neg(bop(z)=first) \wedge$

(bop(z)<first $\Rightarrow$ eop(z)<last))] {A46} :

If entry=false, the conditional is true without further proof.

If entry=true $\wedge$ $\neg$(eop(i)=0), the antecedent of the verification condition provides that the generalization on z is true for 0<z<i; if it is shown to hold for z=i, then it is true for 0<z<=i and this consequent of the verification condition is proved.

Either bop(i)=first or bop(i)<first.

Suppose bop(i)=first;

then either eop(i)<=last or last<=eop(i),

and then one of first>=bop(i) $\wedge$ last<=eop(i),

or bop(i)>=first $\wedge$ eop(i)<=last, must be true.

But the antecedent of this verification

condition indicates both are false;

therefore $\neg$ (bop(i)=first.

Now suppose bop(i)<first;

then first>=bop(i) is true, and

from $\neg$(first>=bop(i) $\wedge$ last<=eop(i)),

it is shown that last<=eop(i) must be false.

Thus bop(i)<first $\Rightarrow$ eop(i)<last, and

the generalization on z holds for 0<z<=i;

therefore this consequent is true.

7. Path A42-44 to A47, Case 2

For case 2, the case where the predicate of the if statement immediately preceding A47 is true, the verification condition is:

$$A42 \wedge A43 \wedge A44 \wedge i<=jx-1 \wedge$$
$$bop(i)>=first \wedge eop(i)=0 \Rightarrow A47$$

All of the consequents contained in A47= {A45 ∧ A46} except A46 are restatements of the antecedents.

a. $(entry=true \wedge \neg(eop(i)=0)) \Rightarrow$
$\forall z[0<z<=i \Rightarrow (\neg(bop(z)=first) \wedge$
$(bop(z)<first \Rightarrow eop(z)<last))]$ {A46}:

since eop(i)=0, the antecedent of this conditional is false, and the conditional is true; this completes the proof of the verification condition for this path.

8. Path A45-46 to A48 and Path A47 to A48

There are no program fragments on these paths, so the verification conditions are trivially true; they are:

$$A45 \wedge A46 \Rightarrow A45 \wedge A46$$

9. Path A48 to A42-44

The verification condition is:

$$A48 \Rightarrow A39 \wedge A40 \wedge A41 \wedge i+1<=jx$$
$$\wedge A43 \wedge (entry=true \wedge \neg(eop(i)=0)) \Rightarrow$$
$$\forall z[0<z<=i+1 \Rightarrow (\neg(bop(z)=first)$$
$$\wedge (bop(z)<first \Rightarrow eop(z)<last))]$$

Proof is shown by considering the consequents one at a time.

a. A39 ∧ A40 ∧ A41:

these predicates are just a restatement of antecedents.

b. i+1<=jx:

i<=jx-1 ⇒ i+1<=jx.

c. A43:

this predicate is just a restatement of an antecedent.

d. (entry=true ∧ ¬(eop(i)=0)) ⇒

∀z[0<z<=i+1 ⇒ (¬(bop(z)=first)

∧ (bop(z)<first ⇒ eop(z)<last))] :

from A46 it is known that the generalization on z is valid over the range 0<z<=i, which is equivalent to the range in this consequent, namely 0<z<i+1; therefore this consequent holds. Thus, the verification condition for this path is proved.

10. Path A42-44 to A-49-50

The verification condition is:

A42 ∧ A43 ∧ A44 ∧ i>jx-1 ⇒ A49 ∧ A50

The consequents are considered one at a time.

a. A49:

This predicate is a restatement of antecedents of the verification condition and thus is true.

b. (entry=true ∧ ¬(eop(jx)=0)) ⇒

∀z[0<z<=jx ⇒ (¬(bop(z)=first) ∧

(bop(z)<first ⇒ eop(z)<last))] {A50} :

i>jx-1 ∧ i<=jx ⇒ i=jx.

Therefore, this consequent is just a restatement of an antecedent with jx=i replacing i.

11. Path A39-41 to A49-50

The verification condition is:

{A39-41} for statement {A49-50}

The previous proofs of the verification conditions for paths A39-40 to A42-44, A42-44 to A45-46 to A48, A42-44 to A47 to A48, A48 to A42-44, and A42-44 to A48 satisfy the requirements of the for rule; therefore, the verification condition for this path is proved.

12. Path A49-50 to A51-52, Case 1

Case 1 is the case where entry=false and the compound statement intervening is not executed. The verification condition in this case is:

$$A49 \wedge A50 \wedge entry=false \Rightarrow A51 \wedge A52$$

Consider the consequents one at a time.

a. A51:

The predicates of this consequent are restatements of given predicates.

b. A52:

$entry=false \wedge A43 \Rightarrow A52$ (A43 is one of the predicates contained in assertion A49).

13. Path A49-50 to A51-52, Case 2

In this case, entry=true and the compound statement intervening is executed. The verification condition (with jx+1 replacing jx in A51-52) is:

$$A49 \wedge A50 \wedge entry=true \wedge bop(jx)=first$$
$$\wedge\ eop(jx)=last \Rightarrow$$
$$\{\ 2<=\ell<=256 \wedge ca=\ell \wedge jx+1>0 \wedge$$
$$\forall y[(0<y<jx+1 \wedge bop(y)>1 \wedge 0<eop(y)<\ell) \Rightarrow$$
$$\neg(text(bop(y)-1)=text(eop(y)+1))] \wedge$$
$$\forall y[\ (0<y<jx+1 \wedge \neg(eop(y)=0)) \Rightarrow$$
$$[string(bop(y), eop(y))=ok \wedge bop(y)>=1 \wedge eop(y)<=\ell$$
$$\wedge \forall z(\ (0<z<jx+1 \wedge \neg(z=y))$$
$$\Rightarrow (\ \neg(bop(z)=bop(y)) \wedge$$
$$(bop(z)<bop(y) \Rightarrow eop(z)<eop(y)))\ )]\ ] \wedge$$
$$\exists y(y<jx+1 \wedge 1<=bop(y)<=current-1 \wedge current<=eop(y)<=\ell\ )\ \}$$

The consequents of this verification condition are considered one at a time.

a. $2<=\ell<=256 \wedge ca=\ell \wedge jx+1>0$:

these predicates are just a restatement of antecedents.

b. $\forall y[(0<y<jx+1 \wedge bop(y)>1 \wedge 0<eop(y)<\ell) \Rightarrow \neg(text(bop(y)-1)=text(eop(y)+1))]$:

the antecedent A40 (contained within assertion A49) establishes the generalization on y for $0<y<jx$. If the statement is true for $y=jx$ as well, then this consequent is proved. The antecedents of the verification condition allow the generalization statement for $y=jx$ to be written as:

$(first>1 \wedge 0<last<\ell) \Rightarrow \neg(text(first-1)=text(last+1))$;

from the input assertion of this procedure it is known that

$(first=1 \vee last=\ell \vee \neg(text(first-1)=text(last+1)))$;

the generalization statement for $y=jx$ has an antecedent which

negates the first two predicates of this disjunction; therefore the third predicate, which is also the consequent of the generalization statement for y=jx, must be true. Thus this consequent of the verification condition is proved.

c. $\forall y[\ (0<y<jx+1 \wedge \neg(eop(y)=0)) \Rightarrow$

$[string(bop(y),eop(y))=ok \wedge\ bop(y)>=1$

$\wedge\ eop(y)<=\ell \wedge\ \forall z(\ (0<z<jx+1 \wedge\ \neg(z=y))$

$\Rightarrow (\neg(bop(z)=bop(y))$

$\wedge\ (bop(z)<bop(y) \Rightarrow eop(z)<eop(y)))\ )]\ ]$:

The antecedent A41 (contained within A49) establishes the generalization on y for 0<y<jx. If the generalization statement is demonstrated true for y=jx, then this consequent holds. For the case y=jx, the conditional which must be proved is:

$\neg(last=0) \Rightarrow [string(first,last)=ok \wedge\ first>=1$

$\wedge\ last<=\ \ell \wedge \forall z(\ (0<z<jx+1 \wedge \neg(z=jx)) \Rightarrow (\neg(bop(z)=first)$

$\wedge\ (bop(z)<first \Rightarrow eop(z)<last))\ )]$

The antecedent of the above conditional is clearly true; if the several consequential predicates are true, then the verification condition consequent in question is proved. The first 3 predicates follow from the input assertion. The fourth predicate, the generalization on z, has already been shown to be true for the range 0<z<jx

(from entry=true $\wedge \neg(eop(jx)=0) \wedge$ A50);

further, if z=jx then $\neg(z=jx)$ is false and the conditional which is the fourth predicate is true.

This completes the proof for this consequent of the verification condition.

d. $\exists y(y<jx+1 \wedge 1<=bop(y)<=current-1$

$\wedge$ current<=eop(y)<=ℓ):

It is proposed that y=jx will satisfy this existential statement.

jx<jx+1;

bop(jx)=first $\wedge$ 1<=first<=ℓ-1 $\wedge$ first<last

$\wedge$ current<=last $\wedge$ last<=ℓ $\Rightarrow$

1<=bop(jx)<=current-1;

eop(jx)=last $\wedge$ current<=last $\wedge$ last<=ℓ $\Rightarrow$

current<=eop(jx)<=ℓ;

so y=jx satisfies the existential statement, and this last consequent of the verification condition is shown. The verification conditions for both cases on path A49-50 to A51-52 have been proved.

14. Input Assertion to Output Assertion

The proof of partial correctness for this procedure is completed by concatenation of paths A39-41 to A49-50 and A49-50 to A51-52.

15. Termination

The procedure has one entry and one exit, the only loop is a for statement; therefore, the procedure terminates.

K. PROCEDURE MAIN

Figure 8 contains the assertions for the main body of the example program.

1. Input Assertion: A0.

```
      comment  main;

A0:   { 2<=input(1)<=256 ∧ ca=0 }

      initialize;

A3:   { 2<=1<=256 ∧ ca=0 ∧ jx=1 ∧ lt=80 ∧ cb=blank }

      read_and_write_input_cards;

A16:  { 2<=1<=256 ∧ ca=1 ∧ jx=1 }

      palindrome_check;

A53:  { 2<=1<=256 ∧ ca=1 ∧ jx>0 }
A54:  { ∀x[(2<=x<=1 ∧ text(x-1)=text(x)) ⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-1 ∧ x<=eop(y)<=1)] }
A55:  { ∀x[(3<=x<=1 ∧ text(x-2)=text(x)) ⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-2 ∧ x<=eop(y)<=1)] }
A56:  { ∀y[(0<y<jx ∧ bop(y)>1 ∧ 0<eop(y)<1) ⇒
         ~(text(bop(y)-1)=text(eop(y)+1))] }
A57:  { ∀y[ (0<y<jx ∧ ~(eop(y)=0)) ⇒
         [string(bop(y),eop(y))=ok ∧ bop(y)>=1 ∧ eop(y)<=1
          ∧ ∀z( (0<z<jx ∧ ~(z=y)) ⇒
               (~(bop(z)=bop(y)) ∧ (bop(z)<bop(y) ⇒ eop(z)<eop(y))) )] ] }

      if jx=1 then text3
      else write_all_palindromes;

A58:  { A53 ∧ A54 ∧ A55 ∧ A56 ∧ A57 }

      end.
```

FIGURE 8

PROCEDURE MAIN

2. Output Assertion: A58.

3. Intermediate Assertions:

The intermediate assertions are assertions A3, A16, and A53-57. They are precisely the input and/or output assertions of the procedure calls they precede and/or follow. The verification condition path for assertion A0 to A53-57 is proved by repeated application of the rule of inference for procedure calls, and then by concatenation. The verification condition for path A53-57 to A58 is simply:

{A53-57} non-significant statement {A53-57} ,

because the intervening if statement merely prints the results which have already been proven correct; its proof is immediate.

4. Termination

All of the procedures called from this main body have been shown to terminate; this program has one entry, one exit and no loops; therefore, it terminates. This completes the proof of total correctness of the example program.

APPENDIX B

APPLICATION OF DISTRIBUTED CORRECTNESS TECHNIQUES

A. ASSUMPTIONS, ABBREVIATIONS, AND NOTATION

In addition to the assumptions about the example program verified by static analysis (Chapter IV, Section A), it was further assumed that all input data read by the program were type compatible with variables and that the correct number of input characters were present in the input data stream. Integer arithmetic was also assumed. Because actual dynamic testing was involved, assumptions that the operating system and compiler operated correctly were at least partially verified during testing.

Because several of the program variable names are verbose, the abbreviations listed below were used in presenting the assertions and their verification:

-ℓ	length←of←text
-ℓt	cardlimit
-cb	cardbuffer
-n	number←of←input←cards
-c	card←counter
-bp	bufferposition
-bop	begin←of←palindrome
-eop	end←of←palindrome
-p	palindrome

Figures 9 through 12 are listings of four program procedures with labeled synthetic assertions inserted to aid the discussion of the correctness demonstration. Assertions B0 and B29-33 are the input and output specifications, respectively. Assertions are contained within braces " { }," and in Figures 9 through 12

wherever successive labeled assertions follow a program statement, the intended synthetic assertion for that point is a conjunction of those assertions. Frequently assertions contain within the braces the names (labels) of other assertions; the meaning implied is a literal replacement of the label with its expansion.

Condition tables in the following sections list on their left the several predicates which were considered to partition the input domain of the given program fragments. The columns to the right of the predicates list the conceivable combinations of truth values for the several predicates. Corresponding to each column, a test data element was selected to verify program operation for each composite predicate (conjunction of the truth value entries in each column). The following entries were used in the columns:

y	:	Yes, or true.
n	:	No, or false.
-	:	Don't care; either true or false.
(y)	:	Required to be true by the value for another entry in the same column.
(n)	:	Similar to (y), except false.

B. UTILITY PROCEDURES

The procedures "text1," "text2," "text3," "blank←lines," and "write←all←palindromes" do not affect program performance of the output specification. (They effect the neat printing of the results obtained in the significant procedures.) As in the presentation of the formal proof for the example program, these procedures will not be examined here. However, it should be noted that because the method reported in this appendix

involved actual program execution, a side effect of the tests performed was to verify the performance of these non-essential procedures.

C. PROCEDURE MAIN

The methodology using the principle of distributed correctness and the condition table method for selecting test data (where needed) was first applied to the main body of the example program; procedure calls were treated either as an in-line expansion of code or as program statements whose semantic meaning was defined by the input and output assertions of the called procedure. It was assumed that the input assertion B0 is satisfied when program execution begins. Figure 9 contains the synthesized assertions for this procedure.

1. Synthesized Assertion B1:

$2 <= \ell <= 256 \wedge jx=1 \wedge \ell t=80 \wedge cb=blank$

a. Test Data Assertion and Verification

The test data assertion is that for $\ell=2$ and for any corresponding character string (of length 2), the synthesized assertion is valid. Verification was obtained by executing the program statement "initialize;" preceeding assertion B1 with input data $\ell=2$.

b. Generalization Assertion and Verification

The generalization assertion is that for any input data satisfying the input assertion B0, the same result as above will be obtained. Verification is made by static analysis of procedure "initialize" - the assignments satisfying B1 are

```
      comment  main;

B0:   { 2<=1<=256 }

      initialize;

B1:   { 2<=1<=256 ∧ jx=1 ∧ lt=80 ∧ cb=blank }

      read_and_write_input_cards;

B5:   { 2<=1<=256 ∧ ca=1 ∧ jx=1 }

      palindrome_check;
      if jx=1 then text3
      else write_all_palindromes;

B29:  { 2<=1<=256 ∧ ca=1 ∧ jx>0 }
B30:  { ∀x[(2<=x<=1 ∧ text(x-1)=text(x)) ⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-1 ∧ x<=eop(y)<=1)] }
B31:  { ∀x[(3<=x<=1 ∧ text(x-2)=text(x)) ⇒
         ∃y(y<jx ∧ 1<=bop(y)<=x-2 ∧ x<=eop(y)<=1)] }
B32:  { ∀y[(0<y<jx ∧ bop(y)>1 ∧ 0<eop(y)<1) ⇒
         ~(text(bop(y)-1)=text(eop(y)+1))] }
B33:  { ∀y[ (0<y<jx ∧ ~(eop(y)=0)) ⇒
         [string(bop(y),eop(y))=ok ∧ bop(y)>=1 ∧ eop(y)<=1
          ∧∀z( (0<z<jx ∧ ~(z=y)) ⇒
               (~(bop(z)=bop(y)) ∧ (bop(z)<bop(y) ⇒ eop(z)<eop(y))) )] ] }

      end.
```

FIGURE 9

PROCEDURE MAIN

executed unconditionally.

c. Proof of Synthesized Assertion

The synthesized assertion follows directly from the test data and generalization assertions. Note that the procedure call "initialize;" was treated as an in-line substitution of code. The proof of B1 amounted to a demonstration of correctness of the procedure "initialize."

2. Synthesized Assertion B5:

$$2<=\ell<=256 \wedge ca=\ell \wedge jx=1$$

The control path from assertion B1 to B5 contains only a procedure call to "read←and←write←input←cards"; assertion B5 is proved by showing that it is equivalent to the output specification of the procedure.

Let assertion B2 be identical to B1, and let it be the input assertion for procedure "read←and←write←input←cards" (see Figure 4 in Appendix A); clearly B2 holds since B1 precede the procedure call and has been shown true. Examination of the procedure reveals that the predicates 2<=ℓ<=256 and jx=1 are not modified in its execution; only the predicate ca=ℓ (which as before means that "ℓ" characters have been properly read from the input stream and assigned to the string variable "text" in the proper position) remains to be shown. This will be done by verification of the output assertion for the procedure called.

a. Synthesized Assertion B3:

Let B3 be an assertion inserted following the first statement in "read←and←write←input←cards"

(n:=((ℓ-1)div ℓt)+1;). The assertion is that:

"n" is the correct number of
input cards for the characters
in a string of length "ℓ."

The test data assertion for B3 was determined using the following condition table to divide the input domain into equivalence classes:

Predicate				
2<=ℓ<= 80	y	(n)	(n)	(n)
81<=ℓ<=160	(n)	y	(n)	(n)
161<=ℓ<=240	(n)	(n)	y	(n)
241<=ℓ<=256	(n)	(n)	(n)	y
Test data (ℓ)	2	81	240	256
Correct value (n)	1	2	3	4

The test data assertion (i.e., that the program will execute properly for the test data elements identified in the preceding table) was verified by execution of the program to assertion B3 with the four test data values of "ℓ" and checking for the assignment of the correct value to "n".

The generalization assertion at B3 is that n is totally determined by the four predicates on "ℓ" given in the condition table; from the program statement "n:=((ℓ-1)div ℓt)+1" and the predicate ℓt=80 it is apparent that this is so.

The proof of the synthesized assertion B3 follows directly from the test data and generalization assertions and Theorem 2.1 of Reference 14 (the theorem states that if two functions on the same domain D are totally determined by the same predicates,

then those predicates partition D into equivalence classes for testing purposes). Theorem 2.1 applies in this case as the program performance (first function) and the assignment algorithm (second function) are both totally determined by the four predicates on "ℓ."

b. Synthesized Assertion B4:

Let B4 be inserted following the last statement in procedure "read←and←write←input←cards" (i.e., the output assertion for the procedure). The assertion is:

$$ix=\ell+1 \wedge ca=\ell$$

The test data assertion for B4 was determined using the following condition table to divide the input domain into equivalence classes:

Predicate				
$2<=\ell<80$	y	n	n	n
$2<=\ell<=256 \wedge \ell$ rem 80=0	(n)	y	y	n
$80<\ell<=256$	(n)	n	y	(y)
n=1	(y)	(y)	(n)	(n)
n>1	(n)	(n)	(y)	(y)
Test data (ℓ)*	2	80	160	81
Correct value (ix)	3	81	161	82

*An input string of "ℓ" characters must also be provided.

The predicates listed above are those which were presumed to have all possible bearing on program operation; note that the two predicates on "n" were actually unnecessary since "n" is totally determined by "ℓ."

The test data assertion is that the correct results will be obtained for the four test data elements identified in the condition table; verification was successfully performed by program execution.

The generalization assertion at B4 is that ix is totally determined by the three predicates on "ℓ" listed in the condition table (and "ca" is one less than ix). This was verified by inspection of the program statements between assertions B3 and B4.

The proof of the synthesized assertion B4 follows directly from test data and generalization assertions and Theorem 2.1 (14).

c. Proof of Synthesized Assertion B5

It has been shown that assertion B1 preceding the call to procedure "read←and←write←input←cards" satisfies the input assertion for the procedure, and that the procedure correctly assures the validity of its output assertion (B4). Since B4 requires that ca=ℓ, synthesized assertion B5 is shown by the distributed correctness of the called procedure.

3. Synthesized Assertion B29-33

The assertions B29-33 are the output specification for the example program. The only significant program statement intervening between assertion B5 and B29-33 in the main program is a call to procedure "palindrome←check" (Figure 10 is a listing of the procedure). Note that assertion B5 satisfies the input assertion (B6-8) to the procedure (because B6 is a restatement of B5 and the conditionals which constitute assertions B7 and B8 have antecedents which are necessarily false when jx=ℓ; therefore

the conditionals are true) and that the output assertion of the procedure (B12) is identical to assertion B29-33. Therefore, if the distributed correctness of procedure "palindrome←check" is shown separately and if the procedure call statement is executed for the test cases identified in the verification of the procedure, then the synthesized assertion B29-33 is demonstrated to be true, and the verification of the main program is complete. The correctness of the called procedure is demonstrated in the next section.

D. PROCEDURE PALINDROME CHECK

Figure 10 contains the synthesized assertions for this procedure. Similar to the way correctness of the main program was verified by relying on the distributed correctness of this procedure, this procedure will be verified by relying on the distributed correctness of the procedure which it calls, namely "continue checking."

1. Synthesized Assertion B6-8:

$$B6: \quad 2 \le \ell \le 256 \wedge jx=1 \wedge ca=\ell$$

$$B7: \quad \forall y[(0<y<jx \wedge bop(y)>1 \wedge 0<eop(y)<\ell) \Rightarrow \neg(text(bop(y)-1)=text(eop(y)+1))]$$

$$B8: \quad \forall y[\ (0<y<jx \wedge \neg(eop(y)=0)) \Rightarrow [string(bop(y),eop(y))=ok \wedge bop(y)>=1 \wedge eop(y)<=\ell \wedge \forall z(\ (0<z<jx \wedge \neg(z=y)) \Rightarrow (\neg(bop(z)=bop(y)) \wedge (bop(z)<bop(y) \Rightarrow eop(z)<eop(y)))\ )]\ \}$$

Synthesized assertion B6-8 is the input assertion for the procedure. Static analysis of the program reveals that the

```
      procedure palindrome_check;
      comment  find all palindromes within given text string;
      begin
            comment  scan text from left to right;

 B6:  { 2<=ix<=256 ∧ jx=1 ∧ ca=1 }
 B7:  { ∀y[(0<y<jx ∧ bop(y)>1 ∧ 0<eop(y)<ix)⇒
         ~(text(bop(y)-1)=text(eop(y)+1))] }
 B8:  { ∀y[ (0<y<jx ∧ ~(eop(y)=0))⇒
         [string(bop(y),eop(y))=ok ∧ bop(y)>=1 ∧ eop(y)<=ix
          ∧ ∀z( (0<z<jx ∧ ~(z=y)) ⇒
               (~(bop(z)=bop(y)) ∧ (bop(z)<bop(y) ⟹ eop(z)<eop(y))) )] ] }

      for ix:=2 step 1 until length_of_text do
           begin
           if text(ix-1) = text(ix) then

 B9:  { 2<=ix<=256 ∧ jx>0 ∧ ca=1 ∧ B7 ∧ B8 }
B10:  { text(ix-1)=text(ix) ∧ start=ix-1 ∧ finish=ix
        ∧ string(ix-1,ix)=ok ∧ 1<=ix-1 ∧ ix<=ix }

                continue_checking((ix-1),ix);
           if ix ~= 2 then
                if text(ix-2)=text(ix) then

B11:  { B9 ∧ text(ix-2)=text(ix) ∧ start=ix-2 ∧ finish=ix
        ∧ string(ix-2,ix)=ok ∧ 1<=ix-2 ∧ ix<=ix }

                      continue_checking((ix-2),ix);
           end;

B12:  { 2<=ix<=256 ∧ ca=1 ∧ jx>0 ∧ B30 ∧ B31 ∧ B32 ∧ B33 }

      end palindrome_check;
```

FIGURE 10

PROCEDURE PALINDROME←CHECK

only call to procedure "palindrome←check" is the call in "main" following assertion B5; assertion B5 is identical to B6, therefore B6 holds at the time of the procedure call. B5 ensures that jx=1, and because there is no integer y such that 0<y<1, the antecedents of the conditionals which constitute B7 and B8 are necessarily false at the time of the procedure call. Therefore the conditionals must be true at this point, and the input assertion to the procedure is satisfied whenever it is called.

2. Synthesized Assertion B9-10:

B9: $2 <= \ell <= 256 \wedge jx > 0 \wedge ca = \ell \wedge B7 \wedge B8$

B10: $text(ix-1) = text(ix) \wedge start = ix-1 \wedge finish = ix \wedge string(ix-1, ix) = ok \wedge 1 <= ix-1 \wedge ix <= \ell$

The assertion B9-10 is inserted to state that the input specification is satisfied for the procedure "continue←checking," which is called immediately following the assertion. All pre-predicates of B9-10 are a restatement of the input assertion B6-10 (and have not been modified by the intervening program statements) except:

a. text(ix-1)=text(ix):

this predicate is assured since control reaches B9-10 only if it is satisfied (preceding if statement).

b. start=ix-1 ∧ finish=ix:

these predicates are true by definition; "start" and "finish" are constants, initialized to the values with which "continue←checking" will be called, which are used in the proofs of synthesized assertions.

c. string(ix-1,ix)=ok:

follows immediately from text(ix-1)=text(ix).

d. jx>0 ∧ B7 ∧ B8:

these predicates hold on the first loop iteration because of the input assertion; they hold on subsequent iterations due to the distributed correctness of "continue checking" (they are contained within the procedure's output assertion).

The preceding discussion reveals that the synthesized assertion B9-10 is always valid; thus no test data and generalization assertions are required.

3. Synthesized Assertion B11:

B11: B9 ∧ text(ix-2)=text(ix) ∧ start=ix-2 ∧ finish=ix ∧ string(ix-2,ix)=ok ∧ 1<=ix-2 ∧ ix<=ℓ

The assertion B11 is inserted to state that the input specification is satisfied for the procedure "continue←checking," which is called immediately following the assertion. In a fashion similar to that discussed above, it may be verified that assertion B11 always holds, and no test data and generalization assertions are required.

4. Synthesized Assertion B12:

2<=ℓ<=256 ∧ ca=ℓ ∧ jx>0 ∧ B30 ∧ B31 ∧ B32 ∧ B33

Assertion B12 is the output assertion for procedure "palindrome←check"; the expansions for assertions B30 through B33, which are predicates of assertion B12, are given below:

$$B30: \quad \forall x[(2<=x<=\ell \wedge text(x-1)=text(x)) \Rightarrow$$
$$\exists y(y<jx \wedge 1<=bop(y)<=x-1 \wedge x<=eop(y)<=\ell)]$$
$$B31: \quad \forall x[(3<=x<=\ell \wedge text(x-2)=text(x)) \Rightarrow$$
$$\exists y(y<jx \wedge 1<=bop(y)<=x-2 \wedge x<=eop(y)<=\ell)]$$
$$B32: \quad \forall y[(0<y<xj \wedge bop(y)>1 \wedge 0<eop(y)<\ell) \Rightarrow$$
$$\neg(text(bop(y)-1)=text(eop(y)+1))]$$
$$B33: \quad \forall y[\ (0<y<jx \wedge \neg(eop(y)=0)) \Rightarrow$$
$$[string(bop(y),eop(y))=ok \wedge bop(y)>=1 \wedge eop(y)<=\ell$$
$$\wedge \forall z(\ (0<z<jx \wedge \neg(z=y)) \Rightarrow$$
$$(\neg(bop(z)=bop(y)) \wedge (bop(z)<bop(y) \Rightarrow eop(z)<eop(y)))\ )]\ ]$$

The truth of assertion B12 may be verified partially through logical techniques and partially through dynamic testing. By static analysis it is noted that if during execution of the procedure no statements which call "continue←checking" are actually executed, then all the predicates of assertion B12 are merely restatements of the input assertion B6-8 and are not modified by program execution (no positive processing takes place).

If calls to "continue←checking" are executed, then from the output assertion of that procedure and the principle of distributed correctness (the demonstration of correctness and a listing, Figure 11, of that procedure are presented subsequently), the following predicates remain unchanged by execution of the procedure:

$$2<=\ell<=256 \wedge ca=\ell \wedge jx>0 \wedge B32 \wedge B33$$

The predicates B30 and B31 will be demonstrated through testing in the following manner. If "continue←checking" is

called with actual parameters "start" and "finish", then its output assertion verifies that:

$$\exists y(0<y<jx \wedge 1<=bop(y)<=start \wedge finish<=eop(y)<=\ell)$$

Predicates B30 and B31 are verified, and thus so is the synthesized assertion B12, if it is shown that:

$$\forall x(2<=x<=\ell \wedge text(x-1)=text(x) \Longrightarrow$$

"continue←checking" is called,

with start=x-1 and finish=x; and

$$\forall x(3<=x<=\ell \wedge text(x-2)=text(x) \Rightarrow$$

"continue←checking" is called,

with start=x-2 and finish=x.

a. Test Data Assertion and Verification

It was presumed from static analysis of this procedure that if procedure calls are correctly mode to "continue←checking" for the first three characters of a text string, they will be correctly made for all characters. (Only the character patterns over a sub-string of length three are examined by the statements which determine whether and when to call "continue←checking".) Thus test data were selected to consider all possible conditions arising in the first three characters. A condition table was prepared as follows:

Predicate											
ℓ=2 ∧ ix=1	y	n	n	n	n	n	n	n	n	n	n
ℓ=2 ∧ ix=2	(n)	y	y	n	n	n	n	n	n	n	n
ℓ=3 ∧ ix=1	(n)	(n)	(n)	y	n	n	n	n	n	n	n
ℓ=3 ∧ ix=2	(n)	(n)	(n)	(n)	y	y	n	n	n	n	n
ℓ=3 ∧ ix=3	(n)	(n)	(n)	(n)	(n)	(n)	y	y	y	y	y
text(1)=text(2)	--	y	n	-	y	n	y	y	n	n	n
text(1)=text(3)	-	-	-	-	-	-	y	n	y	n	n
text(2)=text(3)	-	-	-	-	-	-	(y)	(n)	(n)	n	y
Test Data: (ℓ)	*	2	2	*	3	3	3	3	3	3	3
(text)		bb	ab		bbb	aba	bbb	bba	aba	abc	abb

*These compound predicates cannot be satisfied for any input data values.

The preceding condition table identifies seven unique test data elements which were input to the program for dynamic testing. Correct results were obtained for all elements; the correct results were defined as being the recording in the arrays "bop" and "eop" of entries which included those character positions corresponding to all truth values of "y" in the three rows of predicates on "text".

b. Generalization Assertion and Verification

The generalization assertion is that the preceding predicates totally determine the procedure calls made to "continue←checking", and thus the results recorded in the arrays "bop" and "eop". Verification of this assertion was not formally stated; verification relies on the thoroughness with which the applicable condition table was prepared.

c. Proof of Synthesized Assertion

The synthesized assertion follows from the discussion preceding the presentation of the applicable

condition table and from the test data and generalization assertions. Sufficiently general theorems to formally state a proof of the assertion were not available or forthcoming from this effort; however, the careful analysis of predicates built a high confidence that the program fragment is correct.

E. PROCEDURE CONTINUE←CHECKING

Figure 11 contains the synthesized assertions for this procedure. The input assertion is B13-16; since it was verified in the preceeding section that this assertion was satisfied for all calls to this procedure, it will be assumed that this assertion is satisfied at the time of invocation of this procedure.

1. Synthesized Assertion B17-18:

B17: B13 ∧ B14 ∧ B15

B18: text(first)=text(last) ∧ 1<=first<=start
∧ finish<=last<=ℓ ∧ string(first, last)=ok
∧ (finish=start+1 ∨ finish=start+2) ∧
(first=1 ∨ last=ℓ ∨ ¬(text(first-1)=text(last+1)))

a. Test Data Assertion and Verification

Test data were selected using the condition table method to consider all predicates which were considered to have a bearing on program processing with respect to assertion B17-18. The applicable condition table is presented below, in two sections.

```
      procedure continue_checking (integer value first,last);
      comment  Given first and last as pointers to a palindrome
                of size 2 or 3, this procedure checks whether or not this
                palindrome is included in a larger palindrome;
      begin

B13:  { 2<=1<=256 ∧ ca=1 ∧ jx>0 }
B14:  { ∀y[(0<y<jx ∧ bop(y)>1 ∧ 0<eop(y)<1) ⇒
        ~(text(bop(y)-1)=text(eop(y)+1))] }
B15:  { ∀y[ (0<y<jx ∧ ~(eop(y)=0)) ⇒
          [string(bop(y),eop(y))=ok ∧ bop(y)>=1 ∧ eop(y)<=1
           ∧ ∀z( (0<z<jx ∧ ~(z=y)) ⇒
                 (~(bop(z)=bop(y)) ∧ (bop(z)<bop(y) ⇒ eop(z)<eop(y))) )] ] }
B16:  { text(first)=text(last) ∧ start=first ∧ finish=last
        ∧ string(first,last)=ok ∧ 1<=start ∧ finish<=1
        ∧ (finish=start+1 ∨ finish=start+2) }

      logical palindrome;
      palindrome:=true;
      while ((first>1) and (last<length_of_text) and (palindrome=true)) do
           begin
           if text(first-1) = text(last+1) then
                begin
                comment  larger palindrome found;
                first:=first-1;
                last:=last+1;
                end
           else
                begin
                palindrome:=false;       comment  largest palindrome found;
                end;
           end;

B17:  { B13 ∧ B14 ∧ B15 }
B18:  { text(first)=text(last) ∧ 1<=first<=start ∧ finish<=last<=1
        ∧ string(first,last)=ok ∧ (finish=start+1 ∨ finish=start+2)
        ∧ (first=1 ∨ last=1 ∨ ~(text(first-1)=text(last+1))) }

      record_palindrome(first,last);

B19:  { B13 ∧ B14 ∧ B15 }
B20:  { ∃y(0<y<jx ∧ 1<=bop(y)<=start ∧ finish<=eop(y)<=1) }

      end continue_checking;
```

FIGURE 11

PROCEDURE CONTINUE←CHECKING

Predicate								
start+1=finish	n	y	y	y	--	y	y	y
start+2=finish	n	(n)	(n)	(n)	-	(n)	(n)	(n)
start=1	-	y	y	y	-	n	n	n
start=2	-	(n)	(n)	(n)	-	y	y	y
start>2	-	(n)	(n)	(n)	-	(n)	(n)	(n)
finish=ℓ	-	y	n	n	n	y	n	n
finish=ℓ-1	-	(n)	y	n	n	(n)	y	y
finish<ℓ-1	-	(n)	(n)	y	n	(n)	(n)	(n)
text(start-1)= text(finish+1)	-	-	-	-	-	-	y	n
text(start-2)= text(finish+2)	-	-	-	-	-	-	-	-
Test Data: (ℓ)	*	2	3	4	*	3	4	4
(text)		aa	aab	aabc		abb	abba	abbc

*These compound predicates cannot be satisfied for any input data values.

Predicate								
start+1=finish	y	y	y	y	y	y	y	y
start+2=finish	(n)	(n)	(n)	(n)	(n)	(n)	(n)	(n)
start=1	n	n	n	n	n	n	n	n
start=2	y	y	n	n	n	n	n	n
start>2	(n)	(n)	y	y	y	y	y	y
finish=ℓ	n	n	y	n	n	n	n	n
finish=ℓ-1	n	n	(n)	y	y	n	n	n
finish<ℓ-1	y	y	(n)	(n)	(n)	y	y	y
text(start-1)= text(finish+1)	y	n	-	y	n	y	y	n
text(start-2)= text(finish+2)	-	-	-	-	-	y	n	-
Test Data: (ℓ)	5	5	4	5	5	6	6	6
(text)	A	B	C	D	E	F	G	H

A: abbad B: abbcd C: abcc D: abccb
E: abccd F: abccba G: abccbd H: abccde

The preceding condition table identifies fourteen test data elements; these were used as input to the program for

dynamic testing. Correct results were verified for all elements by printing variable values following assertion B17-18 and verifying that the predicates of the assertion were satisfied. During presentation of the preceding condition table, no columns with an "n" entry for the first predicate and a "y" entry for the second were added because no new insights to the procedure's operation would have been gained.

b. Generalization Assertion and Verification

The generalization assertion is that the satisfaction of the synthetic assertion B17-18 is totally determined by the predicates of the condition table; the only verification was the analysis which served as a basis for the preparation of the table.

c. Proof of Synthesized Assertion

The synthesized assertion follows from the test data and generalization assertions. No formal proof could be offered.

2. Synthesized Assertion B19-20:

B19: B13 $\wedge$ B14 $\wedge$ B15

B20: $\exists y(0<y<jx \wedge 1<=bop(y)<=start \wedge finish<=eop(y)<=\ell)$

Assertion B19-20 is the output assertion for this procedure. Inspection of Figure 12, the listing and assertions for procedure "record←palindrome" reveals that assertion B17-18, which precedes the only call to that procedure, which call in turn precedes assertion B19-20, satisfies the input assertion to "record←palindrome", and further that the output assertion of

"record←palindrome" satisfies assertion B19-20. Therefore synthesized assertion B19-20 is verified by the correctness of "record←palindrome" (which is shown in the next section), and test data and generalization assertions are not required here.

F. PROCEDURE RECORD←PALINDROME

Figure 12 contains the synthesized assertions for this procedure. The input assertion is B21-24; since it was verified in the preceding section that this assertion is satisfied whenever the procedure is called, the input assertion is assumed to hold at procedure invocation.

1. Synthesized Assertion B25:

$$B21 \wedge B22 \wedge B23 \wedge B24 \wedge entry=true$$

a. Proof of Synthesized Assertion

The test data and generalization assertions are simply the observation that any and all input data will cause the execution of the statement assigning "entry" equal to true; the rest of the assertion is a restatement of the input assertion, none of which has been modified. Verification was postponed until the verification of the test data assertion for synthesized assertion B26. The proof of synthesized assertion B25 follows directly from this observation.

2. Synthesized Assertion B26:

$$(entry=false \wedge B27 \wedge B28) \vee$$
$$[entry=true \wedge B21 \wedge B22 \wedge B23 \wedge B24$$
$$\wedge \forall z(0<z<jx \Rightarrow (\neg(bop(z)=first$$
$$\wedge (bop(z)<first \Rightarrow eop(z)<last))]$$

```
      procedure record_palindrome (integer value first,last);
      comment  Record only max length palindromes.  Flag previously
               recorded palindromes if they are included in the palindrome
               specified by first and last.
               jx was initialized to 1.  After completion jx points to the
               next entry in begin_of_palindrome and end_of_palindrome;
      begin

B21:  { 2<=i<=256 ∧ ca=1 ∧ jx>0 }
B22:  { ∀y[(0<y<jx ∧ bop(y)>1 ∧ 0<eop(y)<1) ⇒
         ~(text(bop(y)-1)=text(eop(y)+1))] }
B23:  { ∀y[ (0<y<jx ∧ ~(eop(y)=0)) ⇒
         [string(bop(y),eop(y))=ok ∧ bop(y)>=1 ∧ eop(y)<=i
          ∧ ∀z( (0<z<jx ∧ ~(z=y)) ⇒
                (~(bop(z)=bop(y)) ∧ (bop(z)<bop(y) ⇒ eop(z)<eop(y))) )] i }
B24:  { text(first)=text(last) ∧ 1<=first<=start ∧ finish<=last<=i
        ∧ string(first,last)=ok ∧ start<finish
        ∧ (first=1 ∨ last=i ∨ ~(text(first-1)=text(last+1))) }

      integer i;                  comment  local counter;
      logical entry;
      entry:=true;

B25:  { B21 ∧ B22 ∧ B23 ∧ B24 ∧ entry=true }

      for i:=1 step 1 until jx-1 do
           begin
           if ((first)=begin_of_palindrome(i))
                and (last<=end_of_palindrome(i))) then
                begin
                comment  Palindrome is entirely included in a previously
                         recorded palindrome.  No entry required;
                entry:=false;
                end
           else
                begin
                if ((begin_of_palindrome(i) >= first)
                     and (end_of_palindrome(i) <= last)) then
                     begin
                     end_of_palindrome(i):=0;
                     comment  flag smaller palindrome;
                     end;
                end;
           end;       comment  All previously recorded palindromes
                               compared with last input;

B26:  { (entry=false ∧ B27 ∧B28) ∨
        [entry=true ∧ B21 ∧ B22 ∧ B23 ∧ B24
        ∧ ∀z(0<z<jx ⇒ (~(bop(z)=first)
        ∧ bop(z)<first ⇒ eop(z)<last)] }

      if entry = true then
           begin
           comment  larger than all previous or overlapping or disjoint;
           begin_of_palindrome(jx):=first;
           end_of_palindrome(jx):=last;
           jx:=jx+1;
           end;

B27:  { 2<=i<=256 ∧ ca=1 ∧ jx>0 ∧ B22 ∧ B23 }
B28:  { ∃y(0<y<jx ∧ 1<=bop(y)<=start ∧ finish<=eop(y)<=i) }

      end record_palindrome;
```

FIGURE 12

PROCEDURE RECORD-PALINDROME

This assertion is a statement that either entry=false and one set of predicates apply, or that entry=true and another set apply. If entry=false, no further action will be taken in the procedure, and the output assertion is valid at this point. If entry=true, the input assertion is still valid, and the entry for the current palindrome, which will be the jx-th entry in "bop" and "eop", will only be disjoint or overlapping to all previous entries in those arrays for which the "eop" entry is not zero.

a. Test Data Assertion and Verfication

Test data were selected using the condition table method to consider all predicates which were considered to bear on the program processing with respect to assertion B26. The applicable condition table is presented below.

Predicate												
eop(i)=0	n	n	n	n	n	n	n	n	n	y	y	y
first>bop(i)	y	y	y	n	n	n	n	n	n	y	n	n
first=bop(i)	(n)	(n)	(n)	y	y	y	n	n	n	(n)	y	n
last<eop(i)	y	n	n	y	n	n	y	n	n	(n)	(n)	(n)
last=eop(i)	(n)	y	n	(n)	y	n	(n)	y	n	(n)	(n)	(n)
test data:												
first	6	5	3	1	2	1	1	1	1	6	1	1
last	7	9	4	3	3	9	2	3	5	7	3	9
i	4	4	1	2	-	3	-	-	1	1	1	1
bop(i)	1	1	2	1	2	1	2	2	2	2	1	2
eop(i)	9	9	3	4	3	5	3	3	3	0	0	0
text	B	B	B	A	*	B	*	*	B	B	A	B
jx	5	5	2	3	-	4	-	-	3	5	3	4
correct action:	X	X	-	X	X	Y	-	Y	Y	Z	Z	-

*These compound predicates cannot be satisfied for any input data values.

A: aaaa B: baaabaaab
X: Set entry = false
Y: Set eop(i) = zero.
Z: None required, but program resets eop(i) = zero, which is permissable (eop(i) is already zero).
- As an action, means no action performed; entry remains true and a new entry will be made.

The preceding condition table identifies two input strings which were used as input data to the program; intermediate values of program values were inspected on each iteration of the for loop in procedure "record←palindrome" to determine when the compound predicates from the condition table were satisfied so that verification of the correct action as specified in the table could be made. The correct action was observed for each test data element.

b. Generalization Assertion and Verification

The generalization assertion is that the actions performed by the procedure are totally determined by the five predicates in the condition table; if execution of the for loop body performs properly for the identified test data elements, it performs properly for all data satisfying the same conjunctions of the five predicates. The verification offered is the analysis forming the basis for the condition table entries.

c. Proof of Synthesized Assertion

Since predicate B23 of the procedure input specification is valid at the entry to the for loop, it cannot happen that the current palindrome (string (first, last)) both includes a previous entry and is included by a different previous entry. Since program action (either entry:=false or eop(i):=0) is taken only when one of these conditions exists, repeated execution from "i" equal 1 to jx-1 of the loop body cannot cause an undesirable result such as setting "entry" to false and also setting eop(i) to zero for some "i". Thus the synthesized assertion follows from the test data and generalization assertions, although a formal proof cannot be offered.

3. Synthesized Assertion B27-28:

B27: $2 \leq \ell \leq 256 \wedge ca = \ell \wedge jx > 0 \wedge B22 \wedge B23$

B28: $\exists y(0 < y < jx \wedge 1 \leq bop(y) \leq start \wedge finish \leq eop(y) \leq \ell)$

Assertion B27-28 is the output assertion for this procedure.

a. Test Assertion and Verification

The test data assertion is that if the test data

elements identified in the first and sixth columns of the condition table used for assertion B26 are executed to the procedure's termination, the output specification will hold. The correct results were observed, namely that for the column-one element no new entries were made to the arrays "bop" and "eop", and that for the column-six element the proper (fourth) entry was made in the arrrays; in both cases the output specification was observed to hold.

b. Generalization Assertion and Verification

The generalization assertion is that the variable "entry" divides input to the procedure into two equivalence classes and that proper execution of one element of each class (as observed in the verification of the test data assertion) ensures proper execution for the entire class.

c. Proof of Synthesized Assertion

The synthesized assertion follows from the test data and generalization assertions; however, no theorem is available to formally prove the sets of input data identified are in fact equivalence classes.

This verification of synthesized assertion B27-28 completes the demonstration of correctness of procedure "record←palindrome". The principle of distributed correctness has been applied to show the correctness of the main program from the correctness of the called procedures.

BIBLIOGRAPHY

1. Belford, P.C., J. D. Donahoo, and W. J. Heard, "An Evaluation of the Effectiveness of Software Engineering Techniques," IEEE Compcon, Fall 1977, pp 259-67.

2. Black, Rachel, "Effects of Modern Programming Practices on Software Development Costs," IEEE Compcon, Fall 1977, pp 250-53.

3. Boehm, Barry W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973, pp 48-59.

4. Boehm, B.W., J. R. Brown, and M. Lipow, "Quantitative Evaluation of Software Quality," Proceedings of the 2nd International Conference on Software Engineering, October 1976, pp 592-605.

5. Brooks, Frederick P., The Mythical Man-Month, Addison-Wesley, 1975.

6. Brown, John R., "Modern Programming Practices in Large Scale Software Development," IEEE Compcon, Fall 1977, pp 254-58.

7. Culpepper, L. M., "A System for Reliable Software," IEEE Transactions on Software Engineering, June 1975, pp 174-78.

8. Darringer, John A., and James C. King, "Applications of Symbolic Execution," Computer, April 1978, pp 51-60.

9. Dijkstra, Edsger W., "Correctness Concerns and, Among Other Things, Why They are Resented," Proceedings of the International Conference on Reliable Software, ACM SIGPLAN, April 1975, pp 546-50.

10. Dijkstra, Edsger W., "Notes on Structured Programming," in Structured Programming (O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare), Academic Press, 1972.

11. Fairley, Richard E.,"An Experimental Program-Testing Facility," IEEE Transactions on Software Engineering, December 1975, pp 350-57.

12. Fairley, Richard E., "Tutorial: Static Analysis and Dynamic Testing of Computer Software," Computer, April 1978, pp 14-23.

13. Floyd, Robert W., "Assigning Meaning to Programs," Proceedings of Symposia of Applied Mathematics, American Mathematical Society, 1967, pp 19-32.

14. Galler, Matthew, "Test Data as an Aid in Proving Program Correctness," Communications of the ACM, May 1978, pp 368-75.

15. Gerhart, Susan L., and L. Yelowitz, "Observations of Fallibility in Applications of Modern Programming Methodologies," IEEE Transactions on Software Engineering, September 1976, pp 195-207.

16. Goodenough, John B., et al., "MAIDS Information Dynamics Technology Requirements Study," Final Report, Software Technology Company, April 1973.

17. Goodenough, John B., and Susan L. Gerhart, "Toward a Theory of Test Data Selection," IEEE Transactions on Software Engineering, June 1975, pp 156-73.

18. Hetzel, W.C., ed., Program Test Methods, Prentice-Hall, 1973.

19. Hoare, C.A.R., "Procedures and Parameters: An Axiomatic Approach," in "Symposium on Semantics of Algorithmic Languages," Lecture Notes in Mathematics, Vol. 188, Springer-Verlag, 1971, pp 102-16.

20. Hoffman, Heinz-Michael, "An Experiment in Software Error Occurrence and Detection," Master's Thesis, Naval Postgraduate School, June 1977.

21. Howden, W.E., "Reliability of the Path Analysis Testing Strategy," IEEE Transactions on Software Engineering, September 1976, pp 208-15.

22. Huang, J.C., "An Approach to Program Testing," Computing Surveys, September 1975, pp 113-28.

23. Huang, J.C., "Program Instrumentation and Software Testing," Computer, April 1978, pp 25-32.

24. IBM Corporation, "Validation and Verification Study," Structured Programming Series, Vol. XV, Gaithersburg, Maryland, July 1974 (Contract F30602-74-C-0186).

25. London, R.L., "A View of Program Verification," Proceedings of the International Conference on Reliable Software, ACM SIGNPLAN, April 1975, pp 534-45.

26. Manna, Z., and R. Waldinger, "The Logic of Computer Programming," IEEE Transactions on Software Engineering, May 1978, pp 199-229.

27. McCabe, Thomas J., "A Complexity Measure," IEEE Transactions on Software Engineering, December 1976, pp 308-20.

28. Myers, Ware, "The Need for Software Engineering," Computer, February 1978, pp 12-26.

29. Panzl, David J., "Automatic Software Test Drivers," Computer, April 1978, pp 44-50.

30. Panzl, David J., "Test Procedures: A New Approach to Software Verification," Proceeding of the 2nd International Conference on Software Engineering, October 1976, pp 477-85.

31. Pimont, Simone, and Jean-Claude Rault, "A Software Reliability Assessment Based on Structural and Behavorial Analysis of Programs," Proceedings of the 2nd International Conference on Software Engineering, October 1976, pp 486-91.

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

33. Schneidewind, N.F. and H. M. Hoffman, "An Experiment in Software Error Data Collection and Analysis," Proceedings of the 6th Texas Conference on Computing Systems, November 1977, pp 4A1-4A12.

34. Whitaker, W.A., "A Defense View of Software Engineering," Proceedings of the 2nd International Conference on Software Engineering, October 1976, pp 358-62.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4.	Professor Norman F. Schneidewind, Code 52 Ss Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5.	Professor Donald P. Gaver, Code 55 Gv Department of Operations Research Naval Postgraduate School Monterey, California 93940	1
6.	LCDR Carl W. Monk, Jr., USN 5355 Gainsborough Drive Fairfax, Virginia 22030	1