



NBS TECHNICAL NOTE 874

U.S. DEPARTMENT OF COMMERCE / National Bureau of Standards

Software Testing for Network Services

QC
100
.U5753
no. 874
1975
C.2

The National Bureau of Standards¹ was established by an act of Congress March 3, 1901. The Bureau's overall goal is to strengthen and advance the Nation's science and technology and facilitate their effective application for public benefit. To this end, the Bureau conducts research and provides: (1) a basis for the Nation's physical measurement system, (2) scientific and technological services for industry and government, (3) a technical basis for equity in trade, and (4) technical services to promote public safety. The Bureau consists of the Institute for Basic Standards, the Institute for Materials Research, the Institute for Applied Technology, the Institute for Computer Sciences and Technology, and the Office for Information Programs.

THE INSTITUTE FOR BASIC STANDARDS provides the central basis within the United States of a complete and consistent system of physical measurement; coordinates that system with measurement systems of other nations; and furnishes essential services leading to accurate and uniform physical measurements throughout the Nation's scientific community, industry, and commerce. The Institute consists of a Center for Radiation Research, an Office of Measurement Services and the following divisions:

Applied Mathematics — Electricity — Mechanics — Heat — Optical Physics — Nuclear Sciences² — Applied Radiation² — Quantum Electronics³ — Electromagnetics³ — Time and Frequency³ — Laboratory Astrophysics³ — Cryogenics³.

THE INSTITUTE FOR MATERIALS RESEARCH conducts materials research leading to improved methods of measurement, standards, and data on the properties of well-characterized materials needed by industry, commerce, educational institutions, and Government; provides advisory and research services to other Government agencies; and develops, produces, and distributes standard reference materials. The Institute consists of the Office of Standard Reference Materials and the following divisions:

Analytical Chemistry — Polymers — Metallurgy — Inorganic Materials — Reactor Radiation — Physical Chemistry.

THE INSTITUTE FOR APPLIED TECHNOLOGY provides technical services to promote the use of available technology and to facilitate technological innovation in industry and Government; cooperates with public and private organizations leading to the development of technological standards (including mandatory safety standards), codes and methods of test; and provides technical advice and services to Government agencies upon request. The Institute consists of a Center for Building Technology and the following divisions and offices:

Engineering and Product Standards — Weights and Measures — Invention and Innovation — Product Evaluation Technology — Electronic Technology — Technical Analysis — Measurement Engineering — Structures, Materials, and Life Safety⁴ — Building Environment⁴ — Technical Evaluation and Application⁴ — Fire Technology.

THE INSTITUTE FOR COMPUTER SCIENCES AND TECHNOLOGY conducts research and provides technical services designed to aid Government agencies in improving cost effectiveness in the conduct of their programs through the selection, acquisition, and effective utilization of automatic data processing equipment; and serves as the principal focus within the executive branch for the development of Federal standards for automatic data processing equipment, techniques, and computer languages. The Institute consists of the following divisions:

Computer Services — Systems and Software — Computer Systems Engineering — Information Technology.

THE OFFICE FOR INFORMATION PROGRAMS promotes optimum dissemination and accessibility of scientific information generated within NBS and other agencies of the Federal Government; promotes the development of the National Standard Reference Data System and a system of information analysis centers dealing with the broader aspects of the National Measurement System; provides appropriate services to ensure that the NBS staff has optimum accessibility to the scientific information of the world. The Office consists of the following organizational units:

Office of Standard Reference Data — Office of Information Activities — Office of Technical Publications — Library — Office of International Relations.

¹ Headquarters and Laboratories at Gaithersburg, Maryland, unless otherwise noted; mailing address Washington, D.C. 20234.

² Part of the Center for Radiation Research.

³ Located at Boulder, Colorado 80302.

⁴ Part of the Center for Building Technology.

27 1975

acc,

100

153

874

15

2

Software Testing for Network Services

Rona B. Stillman and Belkis Leong-Hong

Institute for Computer Sciences and Technology
U.S. National Bureau of Standards
Washington, D.C. 20234

Technical note no. 874

Sponsored by

The National Science Foundation
1800 G Street, N.W.
Washington, D.C. 20550



U.S. DEPARTMENT OF COMMERCE, Rogers C. B. Morton, *Secretary*
NATIONAL BUREAU OF STANDARDS, Richard W. Roberts, *Director*

Issued July 1975

Library of Congress Catalog Card Number: 75-600046

National Bureau of Standards Technical Note 874

Nat. Bur. Stand. (U.S.), Tech. Note 874, 40 pages (July 1975)

CODEN: NBTNAE

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1975

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
(Order by SD Catalog No. C13.46:874). Price \$1.00 (Add 25 percent additional for other than U.S. mailing).

TABLE OF CONTENTS

	Page
Abstract	1
1. Purpose	1
1.1 Software As A Deterrent to Networking	1
1.2 Limitations of Software Production Techniques and Need for Tools	1
1.3 Specific Objectives	2
2. Methodology for Reliable Software	3
2.1 Categorization of Available Testing Tools	3
2.2 General Criteria for Evaluating Tools and Their Benefits	5
2.3 Systematic Testing and Testing Objectives	7
3. Compiler Validation	8
3.1 Importance of Validation to Network Usage	8
3.2 Summary Description of the NBS FORTRAN Test Routines	9
3.3 Experimental Application of Validation on the ARPANET	11
3.4 Conclusions and Recommendations for Network Validation	14
4. Dynamic Analyzers	15
4.1 Description of Various Analyzer Designs	15
4.2 The NBS Analyzer	17
4.3 Results From Analysis: An Example	20
4.4 The Experiment	26
4.4.1 The Programs Analyzed	26
4.4.2 The Results	27
4.4.3 Conclusions on Benefits	31
5. Application Guidelines in a Networking Environment	32
6. Future Considerations for Networking	33
6.1 Recommended Research in Program Analysis	33
6.2 Exploratory Research Possibilities: A Network Software Production Facility	35
References	36

SOFTWARE TESTING FOR NETWORK SERVICES

by

Rona B. Stillman, Ph.D.
Belkis Leong-Hong

This report is a first step toward identifying effective software test and measurement tools, and developing a guide for their usage network-wide. The utility of two tools, the NBS FORTRAN Test Routines and the NBS Analyzer, is studied experimentally, and indications of their role in systematic testing in a networking environment are given.

Keywords: Dynamic analysis; NBS Analyzer; NBS FORTRAN Test Routines; networking; systematic testing; testing tools.

1. PURPOSE

1.1 Software As A Deterrent to Networking

Although resource-sharing computer networks offer new opportunities for extending computing capabilities, their continued development is impeded by low service quality and dependability, system incompatibilities, and inadequate support to remote users. In single-system installations, the high incidence of software errors and user misunderstandings of unexpected and unforgiving application limitations have necessitated a close working relationship between software developers and users. In networking, the geographical separation, institutional barriers, and information interchange costs become additional aggravations in the software quality problem, and are compelling reasons to develop a network-wide approach to software testing and related documentation.

1.2 Limitations of Software Production Techniques and Need for Tools

"... as a slow witted human being, I have a very small head and had better learn to live with it and to respect my limitations and give them full credit, rather than try to ignore them, for the latter vain effort will be punished by failure."

———— Edsger W. Dijkstra

The quality of today's software systems is uneven at best. The high error content of software released for operational use is traceable to the inability of the programmer to deal effectively with the logical complexity of the program. Whereas software products have become increasingly larger and more sophisticated, there has been no corresponding improvement in the tools and techniques for software production and development. Current practice is to design and implement a system, and then to test it for some arbitrary subset of possible input values and environmental conditions. (In general, it is impossible to test a program under all conceivable running conditions, since even simple software packages may have an infinite input domain and an extraordinarily large number of execution paths.) The program is accepted when it executes these test cases correctly. However, despite the fact that testing often accounts for over half the total cost of software development, there are usually a significant number of residual errors. The user uncovers these errors in the course of operation, when the software fails to run for certain inputs, when the computer results are clearly incorrect, or when the software reacts with its environment in unexpected and undesirable ways. The cost to users is substantial.

To improve software quality, a set of tools is required which augments the programmer's ability to define, construct, and evaluate the effectiveness of his test cases.

1.3 Specific Objectives

This report is a first step toward identifying effective software test and measurement tools, and developing a guide for their usage network-wide. The guide should assist network host centers toward a uniform approach to testing which increases the dependability of their services. It should be valuable to users, as well, in testing their own programs.

The approach adopted was to identify available testing tools and techniques, categorize them functionally, and then select individual tools for further consideration and experimentation. Three criteria were used in choosing test tools for study and in structuring the experiments:

1. the tool should be general purpose and of interest to the mainstream of network hosts and users
2. the tool must be available at NBS
3. the experiment should shed some light on the utility of the tool (what help can it provide, how badly is it needed), its transportability, and the desirability/undesirability of particular features of the tool.

Important issues include potential benefit for network users and hosts, acceptability of disseminating test results and adopting testing standards among cooperating centers, and future development essential to establishing a comprehensive test methodology spanning a major segment of network services from initial design to continuing operation.

2. METHODOLOGY FOR RELIABLE SOFTWARE

2.1 Categorization of Available Testing Tools

A testing tool is any program, data base, or hardware/software probe which facilitates effective and systematic testing of computer service. Typically, testing tools are software products that assist in generating, accumulating, and evaluating software testing information. Testing tools available today can be categorized as follows:

- A. Audit Routines are collections of test data and expected results which serve to define some (minimum) standard of acceptability for a generic class of programs. Audit routines describe the correlation between requirements (what the software is supposed to do as defined by the expected results) and software capabilities (what the software does, as defined by the actual results). For example, the NBS FORTRAN Test Programs are a set of FORTRAN programs (and expected results) which test the ANSI FORTRAN features of a FORTRAN compiler ([3], and Section 4.2). Similar routines exist for COBOL and JOVIAL compilers.
- B. Test Data Generators are programs which generate test data files according to user specifications, e.g., number of (fixed or variable length) records desired, with "random", controlled increment/decrement, or user-specified values, for tape, disc, or on cards, etc. They reduce the time and effort required to create comprehensive data files, and facilitate test data documentation. Typical test data generators are Synergetics' PRO/TEST, Hoskyns Systems Research's TESTMASTER, and Management and Computer Services' DATAMACS*.
- C. Static Analyzers examine source code and perform one or more of the following functions:

*Reference to specific products in this report does not convey an endorsement or recommendation by NBS, and is intended only for amplification and clarification of technical discussion.

- check for compliance with programming standards, identify violations and correct when possible (e.g., the Bell Laboratories' Verifier).
- display the program structure and logic flow, describe the use of global data, produce cross-reference tables, etc. (e.g., NASA's COMGEN, Westinghouse's COBOL Programmer's Tool).
- report the frequency of occurrence of each syntactic statement type in the program (e.g., NBS's STATIC Analyzer).

Static analyzers are useful in identifying portions of code which are blocked, disconnected, or not readily transportable.

D. Dynamic Analyzers collect data on a program while it is in execution. This data can include:

- the frequency of execution of each statement, code segment, or subprogram
- the range of values of specified variables
- a trace of the path of execution (e.g., the last fifty instructions executed)
- the time spent in each subprogram.

Some typical dynamic analyzers are Life Service Company's UTLPATH, Republic Software Products' COBOL-DEBUGGER, Capex's FORTUNE and COTUNE, Digital Solutions' FUS and NBS's Static/Dynamic Analyzer. Frequency of execution information permits a programmer to assess how thoroughly a program has been tested, and identifies those portions of the program that must be exercised by subsequent tests.

E. Test Control Packages allow user interaction while testing. Capabilities provided may include:

- breakpointing (i.e., suspension of execution at specified points in the code, examination and alteration of variable values, and resumption of execution, e.g., Binary System's BREAKPOINT, Worldwide Computer Services' AUTODEBUGGER, SofTech's FORTRAN DEBUGGING SYSTEM)
- simulation of program modules to permit the testing of incomplete programs
- error recovery (i.e., intercept certain aborts, substitute new values for offending ones, and produce selective or complete diagnostics).

The availability of commercial testing tools is depicted in TABLE 1, with tools characterized by major function, language processed, and machine environment. Because this is a fluid field, some of the packages included in TABLE 1 may no longer be offered, and others which are available may have been overlooked. Moreover, the numbers which appear do not always tally: some packages are implemented on more than one machine, some have versions for several languages, and some information could not be obtained for some of the packages. TABLE 1, therefore, should be taken in the spirit in which it is offered, that is, solely an indicator of broad characteristics of the testing tool market.

2.2 General Criteria for Evaluating Tools and Their Benefits

The following questions illuminate the utility of a testing tool in a networking environment:

- what are the costs to build and maintain the tool, to set up a run, to use the tool?
- is the tool portable across different languages, different operating systems, different machines?
- is it general-purpose, e.g., useful for large programs as well as small, systems software as well as applications programs, for programs with real time constraints, in batch as well as interactive mode, with compilers as well as interpreters?
- are the output files transportable, are they conveniently formatted, to what degree can the user control the quantity and data output?
- can the tool be used to design better testing procedures and, ultimately, better programs? Will it facilitate regression testing after program modification, as well as initial testing?
- are the results reproducible (as contrasted with statistical)? Can they be used in a legally binding contract of requirements or specifications?

MAJOR FUNCTION

	# PACKAGES	LANGUAGE						MACHINE					
		LANG. INDEPENDENT	ALC	BAL	COBOL	FORTRAN	PL/1	CDC 6600	DEC PDP-10	HONEYWELL 6000	IBM 360/370	RCA SPECTRA 70	UNIVAC 1108
TEST DATA GENERATION	14	1	1	4	7					1	13	2	4
STATIC ANALYSIS [LANGUAGE STANDARDS AUDIT	3				2	1			1		2	1	1
DYNAMIC ANALYSIS [FREQUENCY/TIME MONITOR	16			1	10	6	2	2	1	1	14	3	2
TRACE	7			2	2	1		2			5		
BREAKPOINT	5			1	2	2			2		4		
TEST CONTROL [ERROR RECOVERY	8	1	1	4		7					8	1	
SUBROUTINE SIMULATORS	2		1		1	2	1			2			

TABLE 1. A Survey of Commercially Available Tools.

2.3 Systematic Testing and Testing Objectives

In systematic testing, it is essential to formulate well-defined goals, and to measure progress toward those goals (in particular, to know when they have been reached). Responsible test objectives include testing both program function and program structure.

Functional testing regards the program as a black box, and focuses on requirements. It demonstrates that some designated level of performance has been achieved. For example, the NBS FORTRAN Test Routines indicate that a FORTRAN compiler processes legal ANSI FORTRAN in accordance with the Standard. Adequate functional testing requires:

- complete and unambiguous definition of each function, including expected input and output characteristics;
- a test case (cases) for each function, constructed to reflect requirements. Where functional dependencies exist, the test cases should be structured as a hierarchy, with basic (lower) functions tested first, and only tested functions usable in other (higher) functional tests.
- extrema testing, where appropriate. For example, if operation x is required to accept from four to sixteen operands, then test cases should include (at least) the two extreme possibilities.
- testing of singular values, where appropriate.
- testing of out-of-range and illegal values, to determine if the modes of failure are rational and non-destructive.

Structural tests, by contrast, are constructed in response to the internal structure of the program. Experience shows that the error rate of newly installed programs is high, and falls off as the program is used. Structural testing is based upon the premise that many of the early errors occur in (or were instigated by) portions of the program which were not exercised during testing. Some of these errors, then, could be eliminated by exercising a program thoroughly before

releasing it. In effect, structural testing is an attempt to systematize and compress the "break-in" period, and to incorporate it into the program's testing phase. The goal of structural testing, then is to thoroughly exercise the program. Depending upon available resources (time, money, personnel) this means exercising each (specified)

- subprogram
- subprogram entry, exit, and halt
- critical path
- statement
- branch

The process of systematic testing is central to this effort. Therefore, the tools which were selected for further study were those used in performing and/or constructing systematic tests: the NBS FORTRAN Test Routines are a carefully built set of functional tests for FORTRAN compilers; the NBS Analyzer, a FORTRAN execution monitor, facilitates structured testing.

3. COMPILER VALIDATION

3.1 Importance of Validation to Network Usage

Computer networks provide opportunities and facilities for sharing information and other resources which were hitherto unavailable. Networks have made it easier for the users to access and manipulate desired resources at a relatively small cost. However, in order for sharing to be effective, it is essential that there be at least a minimum degree of software compatibility among the nodes in the network, whether the computer hardware at the different nodes be homogeneous (e.g. the WWMCCS system*) or heterogeneous (e.g. the ARPANET**). The necessity for compatibility is evident when an attempt is made to use a program on a system different from the one for which it was originally created. The new system may process the program differently than intended, yielding results other than the

*WWMCCS is the World-Wide Military Command and Control Systems Network, serving the military community.

**Advanced Research Projects Agency

expected, or no results at all.

Unfortunately, this is not an uncommon problem. Where software compatibility does not exist, the sharing of data, programs and other computer resources is an arduous task, due to the substantial amount of "incompatibility debugging" that must be performed on the "working" program before it becomes useful to another user at another installation. Successful usage of a network, therefore, requires a certain measure of compatibility among the nodes.

In addition to serving as a basic set of functional tests, then, compiler validation test routines are a means of ensuring some measure of compatibility among compilers on a network.

This study investigated the feasibility of using validation tests in a networking environment, by experimenting with the NBS FORTRAN compiler validation test routines. This set of test routines was designed to validate that an individual compiler conforms with the ANSI FORTRAN document, X3.9-1966[9]. (There are several ongoing validation efforts: the U. S. Navy's ADPESO, under an agreement with the National Bureau of Standards, is providing a COBOL Compiler Validation Service to the Federal Government; NBS, in conjunction with the ANSI X3J2 BASIC standardization, is developing a set of minimal BASIC validation routines; ALCOA, in cooperation with Purdue University, is working on tests for standard FORTRAN 2).

3.2 Summary Description of the NBS FORTRAN Test Routines

Each of the NBS FORTRAN Test Routines [3] tests a particular feature or capability as described in the ANSI FORTRAN standard document. These tests were not designed to debug a FORTRAN compiler, but rather to test the compiler's adherence and acceptance of form and interpretation of the standard document, X3.9-1966. These tests cannot test every possible interaction of every FORTRAN statement, but those that they do test are representative of the complex set of specifications as set forth in the standard reference document.

The complete set of test routines consists of 116 main program segments and 63 subprograms. The test units are generally small, self-contained main programs, with straight line logic. Test units can be run separately or linked end-to-end. When the FORTRAN processor fails to accept a particular segment of the test routines, this failure may be manifested in one of the following ways:

A. At Compile Time.

- Compilation is not completed, with no indication of the cause.
- Compilation is completed and diagnostic messages are produced (Note: diagnostics are outside of the FORTRAN standard specification).

B. At Link Edit and Load Time.

- The executable program fails to link or to load. The cause may or may not be indicated.

C. At Execution Time.

- The test program is aborted before completion, producing no results.
- Unexpected results are produced. This can occur if:
 - Some well defined element of the Standard FORTRAN language was implemented in the compiler in a variant way.
 - Some ill defined part of the language was interpreted by the compiler writer different from the test program writers.
 - The test program writer interpreted the standard improperly.
 - There is a bug in the test programs.
 - There is a bug in the compiler.

To test the feasibility of using the FORTRAN routines to validate compilers across a network, a small subset of the FORTRAN compiler validation test routines was arbitrarily chosen. This subset is briefly discussed below:

AFRMT: Tests for the correct interpretation and usage of FORMAT statements. FORMAT statements contain specifications which define the structure of a record and the form of the data fields comprising the record. It also tests for input/output statements especially as related to the formatted transfer of alphanumeric data, i.e., the A-conversion.

BLKDT and BLOKD: These two segments should be run together. BLKDT tests the correct usage of BLOCK DATA subprograms which are used to initialize data to be stored in a common area. BLOKD is a BLOCK DATA subprogram which uses all the permissible statements in a BLOCK DATA subprogram.

BSFTS: Tests for the correct usage of Statement Functions (i.e., internal subprograms defined in a single statement) previously defined within the program. Intrinsic Functions (e.g., ABS, FLOAT, MOD, etc.) and External Functions (e.g., EXP, SQRT, SIN, etc.) are assumed to be working.

DATA 2: Tests initialization of variables with all allowable types of constants, and tests the contents of variables set by DATA statements.

DOTRM: DO-LOOPS are tested with all allowable types of terminal statements, e.g., CONTINUE, ASSIGN, logical IF, etc.

IFBMS: Tests that the Intrinsic Functions, supplied by the FORTRAN compiler, behave as prescribed in the standard reference document.

SBRTN, AAQ, ABQ, ACQ: These four segments consist of one main (calling) program SBRTN, and three subroutine subprograms AAQ, ABQ, ACQ. Together, these segments test for proper behavior of the main program and of the subroutines which may or may not return to the main program.

3.3 Experimental Application of Validation on the ARPANET

Because of its generality and availability, the ARPA Network was chosen as the site for the experiment. The (subset of) test routines were installed and executed at various nodes. Installation-dependent changes that were required were documented to aid ingauging the portability of the test routines.

"Installation" involves physically copying the test routines from a batch source (for example, magnetic tapes, punched cards, etc.) to an on-line directory (for example, disc, etc.) such that these routines are easily accessible for on-line manipulation. Installation was performed only once (at an arbitrarily chosen ARPANET host). Thereafter, the routines were transferred to other nodes by using the File Transfer Protocol (FTP) facilities which are available at each ARPANET node.

Once installed, the routines were made operational by making such changes as logical unit assignments, altering file nomenclature conventions, etc. This was all that was required to render the programs compilable and executable. Resulting output was compared with predetermined results. If the two sets of results were in agreement, it was concluded that the compiler adhered to and accepted the form and interpretation of the FORTRAN standard document X3.9-1966. If not, attempts were made to determine the cause of the failure.

The major problem during installation arises from character incompatibilities between batch source (magnetic tape recording-mode) and the on-line directory media. For example, if the files to be installed are recorded in Honeywell's H-200 recording-mode (which is incompatible with any of the copying facilities at BBN), then the files cannot be installed at BBN without copying them onto other batch sources such as punched card, or paper tape.

During the transfer stage, there were relatively few problems, because the user is safeguarded against destruction of his original set of files. Noise on the communication lines may damage the integrity of the files (that is why the files should be checked after the transfer is completed). Other than that, the major problem during the transfer stage could be attributed to the human factor, and not to the network.

When the programs were ascertained to be executing properly, they were moved to other nodes on the ARPANET using the File Transfer Protocol (FTP). The FTP used is described as:

A protocol for file transfer between HOSTs (including terminal IMPs), on the ARPA Computer Network (ARPANET). The primary function of FTP is to transfer files efficiently and reliably among HOSTs and to allow the convenient use of remote file storage capabilities.

The objectives of FTP are 1) to promote sharing of files (computer programs and/or data), 2) to encourage indirect or implicit (via programs) use of remote computers, 3) to shield a user from variations in file storage systems among HOSTs, and 4) to transfer data reliably and efficiently. FTP, though usable directly by a user at a terminal, is designed mainly for use by programs.

Transfer by FTP followed the procedure described below:

1. Learn the FTP procedures for each host, and then perform the FTP of the files. Although the FTP was designed with the concept of uniformity throughout the ARPANET, each "user" host has its own FTP implementation idiosyncracies, as is amply illustrated by the FTP sign-on procedures, and the allowable commands.
2. Check the directory at the receiving hosts (hereafter, known as the "user" host).
3. Identify and correct, if possible, any problems caused by the transfer. If the problem cannot be corrected, delete the new file and do the transfer again.
4. Identify installation-dependent modifications and make changes which conform to the installation requirements.
5. Compile and execute programs to verify the correctness of the transfer and to maintain the integrity of the files transferred. Check outputs against expected results.
6. Create, whenever possible, a control file to allow for multiple execution, and a multiple output file.

The Experiment

A small subset of the test routines--rather than the full set--was chosen for our feasibility study. At this time, our considerations were storage space, time, cost and manageability of the experimental subset. The nature of these validation routines is such that an isolated routine can test the compliance or non-compliance of a given compiler with respect to the acceptance of interpretation and form of a particular FORTRAN statement. Thus, the results derived from this study are meaningful as a first step in future validation efforts in a networking environment.

In performing our experiment, the subset of compiler test routines was copied onto a 7-track magnetic tape in BCD and sent to the ARPANET host BBN, where the tape was installed, i.e. copied onto our disc directory. After necessary changes were made to the test programs using the on-line editor, TECO, the programs were compiled and executed without difficulty. All the tests, except one, yielded expected results. The test which failed was the BSFTS (segment 110) which checks for statement functions previously defined within the same program. Further debugging and consultation with the BBN system analyst led us to the conclusion that the problem was due to:

a) a "compatibility package" problem resulting from using the TENEX system

or b) a compiler implementation problem

or c) a DEC-10 compiler code generation problem.

After further investigation, it was concluded that the failure was due to a DEC-10 compiler code generation bug, since the problem was eliminated by patching the assembly code. To reaffirm this, the program was transferred, via FTP, to a regular DEC-10 compiler (the one at Harvard) in an attempt to compile and execute BSFTS. The same problem was found as with the BBN TENEX (DEC-10) compiler, even though the code generated was somewhat different. A similar patch to the assembly code was made and again the program executed correctly. BSFTS was then transferred to a third (completely different) system, Multics. The segment in question compiled and executed without any problem.

In the next phase of the experiment, the subset of routines were transferred to three different ARPANET hosts. The receiving hosts were:

- USC-ISI, having an identical system, TENEX.
- Harvard, with a regular DEC-10 system.
- MIT-Multics, with a completely different system (H6160).

The transfer of the files from BBN to each of the other three hosts was accomplished with relatively little difficulty. Changes made to the routines at the user hosts, ISI and Harvard, were minimal, concerned mostly with file naming conventions. The subset of test routines tested was executed without incident (with the exception of the segment l10, BSFTS, which behaved the same way as when executed at BBN).

At Multics, we found that there were some departures from the usual FORTRAN language conventions, such as having all executable statements begin in column 11 as opposed to column 7, and requiring that all executable FORTRAN statements be in lower case type. To make its FORTRAN compatible, the Multics compiler has an option to convert all upper case executable statements to lower case and to make the necessary indentations so that these statements would be recognizable to the Multics compiler. (This option can be invoked at the time a FORTRAN program is compiled.) After these and other minor changes were made, the subset of programs were executed without problems, yielding expected results--including segment l10.

In summary, the processes of installing the files on an ARPANET host, and transferring them to different nodes, were relatively simple, once the "poring-through-the-documentation" stage was concluded. The greatest impediment to this experiment was the documentation provided for each system. The documentation was poorly organized, inadequate, and inaccurate. Critical information was scattered throughout, and the method of multi-pointers employed often led to cryptic information. Several features described in detail in the documentation were not fully implemented, and when implemented, did not always correspond to the description. For example, the DEC-10 system command ASSIGN(lu) cannot be used because it was not implemented properly, and the system command, RUNFIL on the TENEX system does not behave as described.

3.4 Conclusions and Recommendations for Network Validation

The purpose of the experiment was to determine the feasibility of validating the compilers in a network environment using a subset of the FORTRAN Compiler Validation routines. The results of the experiments proved to be encouraging: transferring the validation routines over the ARPA network was not difficult (despite the idiosyncracies at each node), and the relative ease of compilation and execution at the few selected nodes proved that the routines were highly portable. This was due in large measure to the fact that the routines are written in standard FORTRAN.

The DEC-10 compiler bug that was identified emphasizes the importance of validating compilers across the network. In particular, it points up the utility of test routines such as the NBS FORTRAN Compiler Validation Test Routines.

To the casual user, however, the validation routines may seem cumbersome. Thorough familiarity with [3] is required to invoke the routines (each routine is stored under a name given in [3]), to understand what feature was being tested, and to interpret the results. Actual and expected results must be compared manually. Moreover, running all of the validation routines could be tedious, since the user is required to call each routine by name. Desirable changes to the validation routines which should be incorporated as a part of future efforts in network compiler validation are:

- clearer, more user-oriented names, or methods of calling, specific routines
- identification of the feature tested as part of the output
- automatic comparison of actual vs expected output values, indicating whether or not they matched
- effective diagnostics imbedded in the test routines, to assist in discovering the cause of a test failure.

4. DYNAMIC ANALYZERS

4.1 Description of Various Analyzer Designs

A recent revival of interest in measuring program execution behavior has led to a number of distinct approaches. To some extent, of course, form follows function, that is, certain elements of the analyzer's design are determined by the characteristics of the data it gathers. A package which measures elapsed time must necessarily make use of a clock; one which monitors source statement activity can be expected to insert "probes" into the source text. Table 2 identifies four typical design philosophies (the columns) and, for each, assesses factors of cost and convenience (the rows). For a good discussion of problems in elapsed-time measurements, see [1].

CLOCK INTERRUPTS VIA OPERATING SYSTEM	COUNTERS PLACED INSIDE A SEGMENTED PROGRAM	INSERTED CALLS TO A SYSTEM CLOCK	EVENT-DRIVEN HARDWARE PROBES
COST TO SET UP A PROGRAM	None to individual program. System assumes overhead.	Depends upon the cost of segmenting a program's source text.	Must segment source code.
COLLECTION OVERHEAD TO PROGRAM DURING RUN	Generally low.	For counters: low. Moderate for tallying by calls to a subroutine.	In some cases, quite high.
USEABILITY ACROSS VARIOUS LANGUAGE COMPILERS WITHIN A SYSTEM	Excellent.	Limited to one language.	At best, none.
PORTABILITY ACROSS DIFFERENT SYSTEMS FOR A GIVEN LANGUAGE	Sometimes poor for interpretive systems.	Excellent.	Excellent.
ACCURACY AND PRECISION OF INDICATIONS OF WHERE PROGRAMS SPEND TIME	Good with a fast clock. Clock irregularities are not very important.	One must know approximate costs for each statement type. Formatted i/o and hidden conversions are often difficult to handle.	Varies with machine address- ing schemes and memory management. Depends upon where probes are set, and how many. Setting up can be tedious.
USEFUL IN VALIDATION TESTING	Varies from poor to good.	Excellent. Unambiguous indications are given if a path is traversed.	Problem here is getting a complete covering of the program's segments
APPLICABILITY TO PROGRAMS WITH REAL- TIME CONSTRAINTS	Good if clock doesn't interrupt too often.	Tally by calls to sub- routines is generally too slow. Inserted counters and global var- iables will work if latitude exists.	Excellent.
DEPENDENCE ON CLOCK CONSISTENCY	Clock irregularities are not critical.	Clock is not used.	Can supply own clock.
CONVENIENCE FOR MONITORING HIGH LEVEL CODE	Poor. Monitors activity of memory locations, not source statements.	Excellent.	Very poor. Blocks of memory monitored.

TABLE 2. Four Approaches to Analyzer Design

4.2 The NBS Analyzer

There are many packages available which monitor program executions [2, 4, 8]. Some are quite elaborate and expensive. The design criteria for the NBS analyzer were:

1. It must monitor (American National Standards) FORTRAN programs on the branch and statement level.
2. It must be suitable for use in a networking environment. Therefore, it should itself be, and should produce files which are portable across different systems.
3. It should be as simple as possible.
4. It should be convenient to use.
5. It should be a suitable basis for developing a more sophisticated and powerful tool.

The NBS analyzer is of the type described as "counters placed inside a segmented program" (Column 2) in Table 2. Use of hardware or systems clocks was avoided. Like others of this type, the NBS analyzer consists essentially of two phases. The original source code is first accepted as input to the analyzer, and an "instrumented" or augmented version of the program is produced as output. Instrumentation consists of inserting calls to a tallying function into the original source code. A second phase occurs when the augmented version is actually run, the tallying function causing the program to compute and report execution frequency statistics in addition to performing its normal function. The entire process is represented in Figure 1.

Execution activity of a FORTRAN program is treated as a series of invocations of code segments. Each code segment is defined as a non-empty sequence $(S_k, S_{k+1}, \dots, S_{k+n})$ of statements where S_k is a unique entry statement in the sequence, and S_{k+n} the sole exit. Control flows from S_{k+j} to S_{k+j+1} for $0 \leq j < n$, i.e., it is strictly sequential. A segment continues until a rule of ending or starting a new segment can apply. Certain statements always begin a new segment, e.g., labelled statements. Other statements always end a segment, e.g., unconditional GOTO statements, RETURNS and STOPS. Some statements are themselves one or more segments. Among these are IF statements, computed GOTOs, and DO-loop terminators. The analyzer assigns segment numbers sequentially throughout the program text.

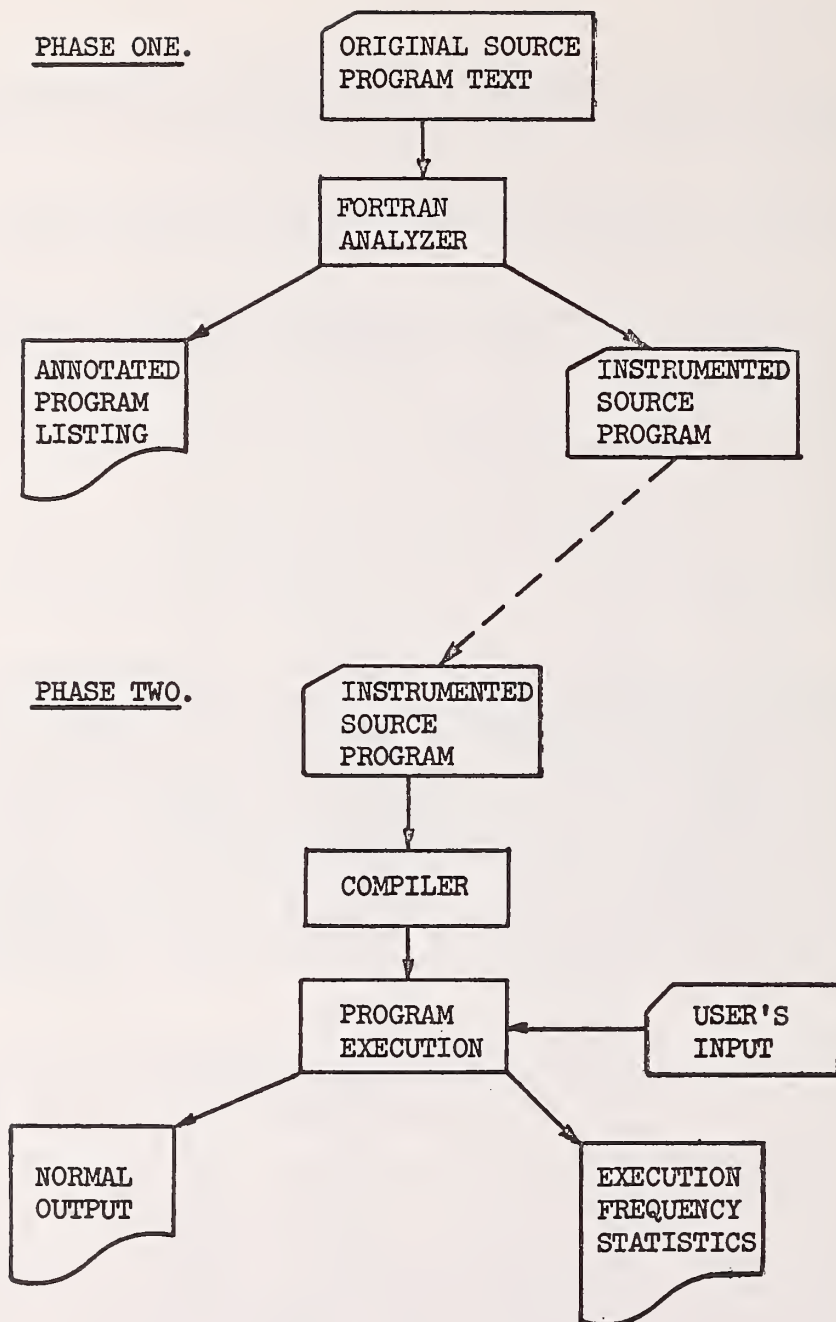


FIGURE 1. Dynamic Analysis

The analysis provides for both computation and control activity monitoring. In computation activity analysis, execution frequencies of each code segment are monitored independent of other program segments. Control activity analysis records flow from segment to segment as well.

FORTRAN's lack of any compound-statement facility (such as BEGIN-END brackets in ALGOL) often has forced FORTRAN analyzers to be inelegant and unwieldy. Many analyzers introduce new variables and labels, and require a symbol table and (sometimes) multiple passes to avoid naming collisions. The NBS analyzer aims at simplicity, with some sacrifice in performance and analysis power (these weaknesses are detailed later). Instrumentation is a single pass operation. No new labels or variables are introduced into the source code; only one function name and a few additional statements are added.

The analyzer does transform many statements into a "statistics collection" form which calls an integer function with the reserved name IY2L9T (IY2L9T was chosen to avoid collision with usual FORTRAN names). The function IY2L9T, which is included as part of the instrumented program, maintains and prints execution frequency statistics. To avoid complicated interactions through COMMON and BLOCK DATA, statistics are kept in an array (SEGFRQ) which is local to IY2L9T. The dimension of SEGFRQ is fixed after a program has been analyzed, and is, of course, equal to the number of segments in the program. This approach implicitly assumes that local variables are invariant across calls to a subprogram, i.e., they have properties of ALGOL own variables or PL/1 static variables [6].

A detailed description of the NBS analyzer is given in [7]. The analyzer has certain shortcomings, however, generally owing to its simplicity. Control activity analysis for assigned GOTOs is somewhat incomplete. Flow out of the GOTO is not recorded. In our opinion assigned GOTOs contribute little to the quality of a FORTRAN program and like ALTER in COBOL, are best avoided.

The transforms assume an American National Standard FORTRAN with mixed mode arithmetic [9]. Non-standard FORTRAN constructs which seemed inconvenient or difficult to handle were ignored. For example, the analyzer assumes that all calls to subprograms have standard returns; READ(...,END=...) and READ(...,ERR=...) are treated as simple READs; IF statements are not expected to contain Hollerith fields.

Because no symbol table is maintained, the analyzer cannot distinguish an arithmetic statement function definition

$$X(I, J, K) = I + J + K \qquad (i)$$

from an array element assignment. Since statement function definitions (in American National Standards FORTRAN) should precede the first executable statement of a FORTRAN program unit, a special comment card

```
C+ *BEGIN EXECUTABLES*
```

could be used to trigger instrumentation of executable code. Some FORTRAN variants allow a more explicit statement function definition:

```
DEFINE X(I, J, K) = I + J + K (ii)
```

Definitions of form (ii) are handled easily.

To keep from complicating the analysis, no attempt has been made to minimize segmentation [5]. END statements are always monitored, even when preceded by a STOP or RETURN; decision portions of IFs and computed GOTOs are monitored as segments even when the statements are unlabelled; a DO loop terminator is always instrumented as a separate segment, even when there are no explicit transfers to it inside the loop. Whether reachable or not, STOPS are modified to call for a printout. Until more experience is gained with the analyzer, the significance of these points is uncertain. It should not be difficult to modify or extend the analyzer whenever appropriate.

4.3 Results From Analysis: An Example

Figure 2-a is a listing of a FORTRAN program to be analyzed. The annotated source, identifying code segments, is presented in 2-b. Notice that the arithmetic IF in the middle of the figure 2-b is monitored as four segments, segment 7 for the expression evaluation, and 8 through 10 labelling the branches. Similarly, each logical IF is treated as two segments, a predicate and a consequent segment.

A listing of the instrumented program, including function IY2L9T, is displayed in figure 3. Execution of this program produced the matrix of segment execution frequencies shown in figure 4. Each matrix row appears under heading "X" and specifies ten segments. The execution statistics of each segment within a decade appear in columns 0-9. (Code constituting each segment is defined via annotated listing, as in figure 2-b.) Figure 4 shows that segment 11 --the DO-loop entry-- was used once. Note also that segment 12, a decision portion of a logical IF, has frequency counts which equal the sum of those for segments 13 and 14, the potential branches from the IF.

```

      INTEGER LENGTH, TABLE(100), ITEM, INDEX, TEMP
      READ(5,10) LENGTH, (TABLE(I), I=1, LENGTH), ITEM
10  FORMAT(12I5)
      TEMP = LENGTH
      CALL FIND7(ITEM, INDEX, LENGTH, TABLE)
      IF(TEMP .NE. LENGTH) GOTO 100
      WRITE(6,11)(ITEM, INDEX)
11  FORMAT(1H ,15,8H IS THE ,15,17HTH ENTRY IN TABLE)
      GOTO 200
100 WRITE(6,101)ITEM
101 FORMAT(1H ,15,19H NOT FOUND IN TABLE)
200 STOP
      END

      SUBROUTINE FIND7(ITEM, INDEX, LENGTH, TABLE)
C  FIND7 ATTEMPTS TO LOCATE ITEM IN TABLE, RETURNING ITS INDEX IN INDEX
C  IF SUCCESSFUL.  OTHERWISE, IT PLACES ITEM AT THE END OF TABLE,
C  INCREMENTS LENGTH, AND RETURNS THIS NEW VALUE ALSO AS INDEX.
      IMPLICIT INTEGER (A-Z)
      INTEGER TABLE(1)
      IF(LENGTH)300,200,100
100 DO 150 I=1,LENGTH
      IF(TABLE(I) .NE. ITEM) GOTO 150
      INDEX = I
      RETURN
150 CONTINUE
200 LENGTH = LENGTH + 1
      IF(LENGTH .GT. 100) GOTO 400
      TABLE(LENGTH) = ITEM
      INDEX = LENGTH
      RETURN
300 WRITE(6,301)
301 FORMAT(1H ,32HNEGATIVE VALUE OF LENGTH ILLEGAL)
      RETURN
400 WRITE(6,401)
401 FORMAT(1H ,24HALLOCATED SPACE EXCEEDED)
      RETURN
      END

```

FIGURE 2-a. Sample Problem


```

      INTEGER LENGTH, TABLE(100), ITEM, INDEX, TEMP
C***** SEGMENT      1 *****
      READ(5,10) LENGTH, (TABLE(I), I=1, LENGTH), ITEM
      10 FORMAT(12I5)
      TEMP = LENGTH
      CALL FIND7(ITEM, INDEX, LENGTH, TABLE)
C***** SEGMENTS      2 THRU      3 *****
      IF(TEMP .NE. LENGTH) GOTO 150
C***** SEGMENT      4 *****
      WRITE(6,11)(ITEM, INDEX)
      11 FORMAT(1H ,15,8H IS THE ,15,17H TH ENTRY IN TABLE)
      GOTO 200
C***** SEGMENT      5 *****
      100 WRITE(6,101) ITEM
      101 FORMAT(1H ,15,19H NOT FOUND IN TABLE)
C***** SEGMENT      6 *****
      200 STOP
      END

END PASS1, VALUE=      13
      SUBROUTINE FIND7(ITEM, INDEX, LENGTH, TABLE)
C FIND7 ATTEMPTS TO LOCATE ITEM IN TABLE, RETURNING ITS INDEX IN INDEX
C IF SUCCESSFUL. OTHERWISE, IT PLACES ITEM AT THE END OF TABLE,
C INCREMENTS LENGTH, AND RETURNS THIS NEW VALUE ALSO AS INDEX.
      IMPLICIT INTEGER (A-Z)
      INTEGER TABLE(1)
C***** SEGMENTS      7 THRU      10 *****
      IF(LENGTH) 300, 200, 100
C***** SEGMENT      11 *****
      100 DO 150 I=1, LENGTH
C***** SEGMENTS      12 THRU      13 *****
      IF(TABLE(I) .NE. ITEM) GOTO 150
C***** SEGMENT      14 *****
      INDEX = I
      RETURN
C***** SEGMENT      15 *****
      150 CONTINUE
C***** SEGMENT      16 *****
      200 LENGTH = LENGTH + 1
C***** SEGMENTS      17 THRU      18 *****
      IF(LENGTH .GT. 100) GOTO 400
C***** SEGMENT      19 *****
      TABLE(LENGTH) = ITEM
      INDEX = LENGTH
      RETURN
C***** SEGMENT      20 *****
      300 WRITE(6,301)
      301 FORMAT(1H ,32HNEGATIVE VALUE OF LENGTH ILLEGAL)
      RETURN
C***** SEGMENT      21 *****
      400 WRITE(6,401)
      401 FORMAT(1H ,24HALLOCATED SPACE EXCEEDED)
      RETURN
      END

```

FIGURE 2-b. Annotated Listing


```

1*      INTEGER LENGTH, TABLE(100), ITEM, INDEX, TEMP
2*      INTEGER IY2L9T
3*      IF(IY2L9T( 1, .TRUE., 0.00, 6).EQ.0) READ(5, 10) LENGTH, (TABLE(I), I=
4*      *1, LENGTH), ITEM
5*      10 FORMAT(12I5)
6*      TEMP = LENGTH
7*      CALL FIND7(ITEM, INDEX, LENGTH, TABLE)
8*      IF(IY2L9T( 2, TEMP.NE.LENGTH, 0.00, 1).EQ.0) GOTO 100
9*      IF(IY2L9T( 4, .TRUE., 0.00, 6).EQ.0) WRITE(6, 11) (ITEM, INDEX)
10*     11 FORMAT(1H ,15,8H IS THE ,15,17H TH ENTRY IN TABLE)
11*     GOTO 200
12*     100 IF(IY2L9T( 5, .TRUE., 0.00, 6).EQ.0) WRITE(6, 101) ITEM
13*     101 FORMAT(1H ,15,19H NOT FOUND IN TABLE)
14*     200 IF(IY2L9T( 6, .TRUE., 0.00, 8).EQ.0) STOP
15*     IF(IY2L9T( 6, .TRUE., 0.00, 9).EQ.0) STOP
16*     END

```

```

1*      SUBROUTINE FIND7(ITEM, INDEX, LENGTH, TABLE)
2*      INTEGER IY2L9T
3*      IMPLICIT INTEGER (A-Z)
4*      INTEGER TABLE(1)
5*      IF(IY2L9T( 7, .TRUE., 0.00+(LENGTH), 2)) 300, 200, 100
6*      100 DO 150 I=1, LENGTH
7*      IF(IY2L9T( 11, I.EQ.1, 0.00, 10).EQ.0) CONTINUE
8*      IF(IY2L9T( 12, TABLE(I).NE.ITEM, 0.00, 1).EQ.0) GOTO 150
9*      IF(IY2L9T( 14, .TRUE., 0.00, 6).EQ.0) INDEX=I
10*     RETURN
11*     150 IF(IY2L9T( 15, .TRUE., 0.00, 6).EQ.0) CONTINUE
12*     200 IF(IY2L9T( 16, .TRUE., 0.00, 6).EQ.0) LENGTH=LENGTH+1
13*     IF(IY2L9T( 17, LENGTH.GT.100, 0.00, 1).EQ.0) GOTO 400
14*     IF(IY2L9T( 19, .TRUE., 0.00, 6).EQ.0) TABLE(LENGTH)=ITEM
15*     INDEX = LENGTH
16*     RETURN
17*     300 IF(IY2L9T( 20, .TRUE., 0.00, 6).EQ.0) WRITE(6, 301)
18*     301 FORMAT(1H ,32HNEGATIVE VALUE OF LENGTH ILLEGAL)
19*     RETURN
20*     400 IF(IY2L9T( 21, .TRUE., 0.00, 6).EQ.0) WRITE(6, 401)
21*     401 FORMAT(1H ,24HALLOCATED SPACE EXCEEDED)
22*     RETURN
23*     IF(IY2L9T( 21, .TRUE., 0.00, 9).EQ.0) STOP
24*     END

```

FIGURE 3. Instrumented Program Including IY2L9T.

```

S640*Y.ELTIY2L9T(0)
1      INTEGER FUNCTION IY2L9T(SEGNO,RE,AE,ENT)
2      INTEGER SEGNO,ENT,IAE,SEGFRQ( 21)
3      DOUBLE PRECISION AE
4      LOGICAL RE
5      DATA SEGFRQ/ 21*0/
6      INTEGER NLINES,NRAGES,PTRAGE,LRRERG,MULT,ENOLIN
7      DATA LRRERG/S0/
8      SEGFRQ(SEGNO)=SEGFRQ(SEGNO)+1
9      GOTO(10,20,30,40,50,60,70,80,90,100),ENT
10     IF(RE)GOTO 11
11     IY2L9T=1
12     RETURN
13     11 SEGFRQ(SEGNO+1)=SEGFRQ(SEGNO+1)+1
14     IY2L9T=0
15     RETURN
16     20 IF(AE)21,22,23
17     21 SEGFRQ(SEGNO+1)=SEGFRQ(SEGNO+1)+1
18     IY2L9T=-1
19     RETURN
20     22 SEGFRQ(SEGNO+2)=SEGFRQ(SEGNO+2)+1
21     IY2L9T=0
22     RETURN
23     23 SEGFRQ(SEGNO+3)=SEGFRQ(SEGNO+3)+1
24     IY2L9T=1
25     RETURN
26     30 IAE=AE+DSIGN(.200,AE)
27     SFGFRQ(SEGNO+IAE)=SEGFRQ(SEGNO+IAE)+1
28     IY2L9T=0
29     RETURN
30     40 IF(RE)GOTO 41
31     IY2L9T=1
32     RETURN
33     41 IY2L9T=0
34     RETURN
35     50 IF(RE)GOTO 51
36     IY2L9T=1
37     RETURN
38     51 SFGFRQ(SEGNO+1)=SEGFRQ(SEGNO+1)+1
39     IAE=AE+DSIGN(.200,AE)
40     SFGFRQ(SEGNO+1+IAE)=SEGFRQ(SEGNO+1+IAE)+1
41     IY2L9T=0
42     RETURN
43     60 IY2L9T=0
44     RETURN
45     70 IF(RE)GOTO 71
46     IY2L9T=1
47     RETURN
48     71 SEGFRQ(SEGNO+1)=SEGFRQ(SEGNO+1)+1
49     IY2L9T=0
50     NLINES= 21/10+1
51     PTRAGE=NLINES-(NLINES/LRRERG)*LRRERG
52     IF(PTRAGE.NE.0)PTRAGE=1
53     NPAGES=(NLINES/LRRERG)+PTRAGE
54     MULT=0
55     DO 88 IAGE=1,NPAGES
56     WRITE( 6,81)
57     81 FORMAT(1H1,4JX,29HSEGMENT EXECUTION FREQUENCIES,/,/)
58     WRITE( 6,82)(1,1=1,9)
59     82 FORMAT(1H ,4X,1HX,5X,7H ,9(2X,17),/)
60     DO 87 ILINE=1,LRRERG
61     IF(MULT.GT. 21)GOTO 89
62     ENOLIN=MULT+9
63     IF(ENDLIN.GT. 21)ENOLIN= 21
64     IF(MULT.NE.0)GOTO 84
65     WRITE( 6,83)(MULT,(SEGFRQ(1),1=1,ENOLIN))
66     83 FORMAT(1H ,15,5X,7H-----,9(2X,17))
67     GOTO 86
68     84 WRITE( 6,85)(MULT,(SEGFRQ(1),1=MULT,ENOLIN))
69     85 FORMAT(1H ,15,3X,10(2X,17))
70     MULT=MULT+10
71     CONTINUE
72     CONTINUE
73     RETURN
74     90 SEGFRQ(SEGNO)=SEGFRQ(SEGNO)-1
75     GOTO 80
76     100 IF(.NOT.RE)SEGFRQ(SEGNO)=SEGFRQ(SEGNO)-1
77     IY2L9T=0
78     RETURN
79     END
END RRT

```

FIGURE 3. (Continued).

SEGMENT EXECUTION FREQUENCIES

X	SEGMENT EXECUTION FREQUENCIES									
	0	1	2	3	4	5	6	7	8	9
0	-----	1	1	1	0	1	1	1	0	0
10	1	1	10	10	0	10	1	1	0	0
20	9	9						1	9	1

FIGURE 4. Results of Example.

4.4 The Experiment

Five "real" programs and test data sets were analyzed, and empirical evidence was collected concerning the utility of execution monitors. The programs studied were selected primarily because they, and their developers were readily available. Because the sample was small and important factors were ignored (e.g., programmer expertise), no attempts were made to draw sweeping conclusions. Instead, we sought insight into the following questions:

- Can convincing empirical evidence of benefit be offered to motivate use of software monitoring techniques on a network-wide basis?
- Are these techniques sufficiently simple and inexpensive to be used by applications programmers?
- What additional capabilities or enhancements are desirable in software monitors?

4.4.1 The Programs Analyzed

Five programs were analyzed:

- NBS Static/Dynamic Analyzer, written by D. Orser, G. Lyon and R. Stillman of NBS. The test cases consisted of the analyzer itself, and the four programs described below. Excepting those portions of code devoted to diagnostic and error reporting (subroutine XERROR), to processing unusual statements (e.g., ENCODE, DECODE) and to future functions (subroutine ENCOD), the test cases were thought to exercise the analyzer "fairly thoroughly".
- CLAY, a structural engineering package written by Robert Ewald of the University of Colorado, and currently in use there. The test cases were constructed using the University's own dynamic analyzer. It is somewhat coarser than the NBS analyzer, monitoring execution at the statement but not the branch level. Based upon his results, Dr. Ewald thought the test cases exercised all of his programs.
- INTEGRATION, written by I. Stegun of NBS to compute certain integrals (e.g., hyperbolic trigonometric functions) according to various parameters. Dr. Stegun carefully tested this small program before releasing it for use at NBS, and believed it had been thoroughly exercised.

- STAPLE, written by S. Stewart of NBS. STAPLE is a preprocessor which translates from a FORTRAN-based structured programming language (STAPLE) into standard FORTRAN. It is used as a vehicle to investigate the concept involved in and benefits to be derived from structured programming and was itself written in STAPLE (and then hand translated). Except for the error-handling portions, Dr. Stewart thought the test cases exercised the program "fairly thoroughly".
- STEM and LEAF, written by D. Hogben and S. Peavy of NBS, to compute frequency distributions from tabular data, and to present them pictorially (e.g., draw histograms). The program has been widely used at NBS. Dr. Hogben intuitively felt that his test cases exercised the program "fairly thoroughly".

To the best of our knowledge, CLAY, INTEGRATION, and STAPLE are single-author programs. STEM and LEAF was written by a closely coordinated team of two. The basic analyzer routines were written by Don Orser modified and rearranged to produce static reports by Gordon Lyon, and then expanded into a dynamic analyzer by Rona Stillman. Each stage of expansion was separate and independent from the preceding one.

4.4.2 The Results

The results of both static and dynamic analysis are summarized in table 3. The number of branches was computed as follows: (for each)

CALL	- 1 branch
RETURN	- 1 branch
arithmetic IF	- 3 branches
logical IF	- 2 branches
unconditional GOTO	- 1 branch
computed GOTO	- n branches

	ANALYZER	CLAY	INTEGRATE	STAPLE	STEM & LEAF
SUBPROGRAMS	16	19	2	3	14
NOT EXERCISED	2	0	0	0	0
STATEMENTS	3188	2304	417	683	2461
COMMENTS	1178	1660	8	4	743
% OF STATEMENTS	37.0	72.0	1.9	.6	30.2
EXECUTABLES	1754	515	387	601	1627
% OF STATEMENTS	55.0	22.3	92.8	88.0	66.1
NOT EXERCISED	839	52	13	121	181
% NOT EXERCISED	47.8	10.1	3.5	20.1	11.2
BRANCHES	1455	238	206	360	1076
BRANCHES/EXECUTABLES	.8	.5	.5	.6	.7
NOT EXERCISED	692	54	4	76	189
% NOT EXERCISED	47.6	22.7	1.9	21.1	17.6
DO LOOPS	52	15	2	0	68
SEGMENTS	1504	385	210	402	1232
EXECUTABLES/SEGMENT	1.2	1.3	1.8	1.5	1.2
BEHAVIOR: % OF ACTIVITY	81.7	61.8	52.6	73.2	60.9
IN	IN	IN	IN	IN	IN
% OF SEGMENTS	8.1	5.7	9.1	11.7	5.8

TABLE 3. Analysis Results

The smallest program, INTEGRATE, was the most thoroughly exercised, missing only 3.5% of its executable statements and 1.9% of its branches. The largest program, the ANALYZER, was the least thoroughly tested. It is also the program whose development was most spontaneous and uncontrolled (i.e., its purpose was modified and personnel changed three times). Size did not always correlate with degree of testedness. STEM and LEAF, with 1627 executable statements, was more thoroughly tested than CLAY and STAPLE, both substantially shorter.

Some of the unexercised code was known to be either unreachable (subprograms to be integrated into future versions of the program, but currently dead), or unexercised by the test cases. For example, one of the unexecuted analyzer subprograms is a symbol table encoding routine (the analyzer does not now use a symbol table); the other handles unrecognized (syntactically incorrect) FORTRAN statements (the test cases were known to be legal FORTRAN programs).

Arithmetic IF's and computed GOTO's accounted for most of the missed branches. Some of the missed branches were unexpected. Using a statement level analyzer, Dr. Ewald thought CLAY was "nearly 100%" exercised when, in fact, 22.7% of the branches had not been executed. Some of the missed branches were logically impossible, and might suggest reprogramming. Some, however, were the result of programming for clarity and flexibility, and are instances of good programming style. For example, in the course of recognizing various FORTRAN statements, the ANALYZER uses computed GOTO's with 26 branches, one for each letter in the alphabet. The test cases used did not -- and were not expected to -- exercise all of these.

The pattern of execution behavior was calculated on the basis of execution counts. If f_i is the number of times that segment i was executed, then segment i accounts for

$$\left[100 \times \frac{f_i}{\sum_{\text{all segments}} f_i} \right]$$

percent of the program's activity. Table 4 summarizes the considerable increase in storage incurred by instrumenting the code.

	ANALYZER	CLAY	INTEGRATE	STAPLE	STEM & LEAF
ORIGINAL WORDS CODE	11328	6913	4517	7427	10066
ORIGINAL WORDS DATA	8937	6163	3492	5524	7460
INCREASE IN CODE WORDS	8518	2731	1672	3131	8893
% INCREASE	75.2	39.5	37.0	42.2	88.3
INCREASE IN DATA WORDS	2574	848	419	783	2261
% INCREASE	28.8	13.8	12.0	14.2	30.3
TOTAL STORAGE INCREASE	11092	3579	2091	3914	11154
% INCREASE	54.7	27.4	26.1	30.2	63.6

TABLE 4. Storage Costs for Instrumentation

4.4.3 Conclusions on Benefits

Programmer reaction to the analyzer was favorable: it was felt to be a much needed tool. Intuition did not prove to be a reliable predictor of testing thoroughness. CLAY's test cases, which were thought to exercise 100% of the program, actually missed 10.1% of the statements and 22.7% of the branches. In testing STEM and LEAF, 11.2% of the statements and 17.6% of the branches were missed. The analyzer's tests missed 47.8% of the statements and 47.6% of the branches. While it had been recognized that the test cases did not exercise the analyzer completely, the very poor coverage did come as a surprise. Small programs like INTEGRATE, written by a single programmer and very carefully tested over a period of time, fared best. Nonetheless, the author was grateful for data supporting her "feeling" that the program had been thoroughly tested.

A great deal of interest was expressed in using the analyzer to identify portions of the program that could be profitably optimized, i.e., those portions exercised most often.

Since every one of the programs analyzed exhibited a non-uniform pattern of segment activity wherein the bulk of the activity was concentrated in a small fraction of the segments, using the analyzer to direct optimization efforts seems promising.

Several suggestions were made to increase the analyzer's utility:

- streamline the process of calling the analyzer and setting up a run
- improve the output format by separating or otherwise highlighting those segments which either were not executed at all or were most frequently executed
- provide a capability to instrument partial programs. This serves two functions: it avoids wasting resources and provides more pertinent output when only specific portions of the program are of interest; it permits programs whose instrumented versions are too large or too slow to be monitored piecewise.

Functional test sets, like the NBS FORTRAN Test Routines, and analytical tools, like the NBS analyzer, can be used to advantage by service providers and users throughout program development.

- Procurement and Acceptance Testing

When "standard" test sets are available, their successful execution should be included in the specifications and incorporated into the acceptance tests. Furthermore, some level of testing thoroughness, e.g., exercising 90% of the program's branches, should be specified and (using a dynamic analyzer) demonstrated.

- Program Development

Effective testing can be achieved by beginning with a functional test set, and continuing to test until the program has been thoroughly exercised. If functional test sets and dynamic analyzers are made available over a network, subscribers as well as service providers could use them to improve their programs.

- Program Management

In managing a program production project, some of the chief difficulties are to ascertain how much of the program is done, how much more remains to be done, and what is the quality of the work. "Standard" functional tests and dynamic analyzer output assist management in determining the status of a developing program quantitatively, i.e., in determining how many functions have been successfully implemented, how many are left to be done, how much testing has been done, etc. Analyzers can be built to instrument partially complete programs (programs with stubs) to complement top-down program design and testing.

- Program Optimization

The effort to tune a program should not be invested until the program's execution pattern has been determined, i.e., until dynamic analysis has been performed.

Underlying all of this is the basic position that the tests performed are a legitimate and important part of program documentation. Tests are, in fact, "machine-readable" documentation which can be effectively disseminated and used in a networking environment.

6.1 Recommended Research in Program Analysis

The NBS FORTRAN dynamic analyzer represents a somewhat fresh approach to monitoring FORTRAN programs. Using a set of transforms [7] the analyzer modified FORTRAN source code to collect run-time data. The design is distinctive in its simplicity. No symbol table is necessary. Instrumenting is a single pass operation. No new labels or variables are introduced into the source code; only one function name and a few statements are added. Under this NSF Grant (AG-350), the feasibility and utility of this dynamic analyzer were demonstrated. Using it as a test bed, several important aspects of program instrumentation and execution behavior analysis can now be investigated.

The quantity and character of the output is of central importance in any program analysis tool. A tool which inundates the programmer with volumes of unwanted data will be avoided. To maximize an analyzer's utility, then, we must be concerned with what is reported, how it is presented, and means by which the user can tailor these to fit his needs.

Currently, the NBS analyzer reports the number of times each of 118 FORTRAN statement types appears in the program text (i.e., static analysis), provides an annotated source listing which blocks the code into segments, and, when the instrumented code is executed, reports the frequency with which each segment is executed. The entire program is always monitored. If the user is concerned only with specific subsets of his code, full execution frequency reports can be excessive. Moreover, since instrumentation substantially increases time and space demands, it is also wasteful. For very large programs, in fact, it might be impossible to fit the instrumented code into available memory. The problem of selecting, specifying, and instrumenting only particular portions of code, then, is of considerable practical importance, as well as being of technical interest, and should be thoroughly investigated.

Arbitrary subsets of programs can be monitored using the current version of the analyzer. The subset to be monitored is input to the analyzer, and instrumented subprograms replace the original subset at compilation. The code is compiled, linked, loaded, and run. Only the subset is monitored during execution and, when the program terminates, the results are printed. It is important to note that since the instrumentation causes the results to be printed, the program must halt in a monitored subprogram. It is the user's responsibility to ensure that this is the case.

For greater flexibility, special comments could be used to switch analysis off and on throughout a program:

```
..... text (instrumented)
C+ *END INSTRUMENTING*
..... more text (not instrumented)
C+ *BEGIN INSTRUMENTING*
```

Some care would have to be exercised. For example, beginning instrumenting inside a BLOCK DATA should not cause the BLOCK's END to call the tallying program. Similarly, initiating segmenting inside a DO loop should not cause the analyzer to miss the loop terminator. Both programmer and analyzer would have to be more alert, and again, the program would have to halt in a monitored segment.

Rather than making the user responsible for assuring that all possible halt points are monitored, the analyzer could do so. In other words, the analyzer would modify user-specified portions of code plus all (unspecified) termination points.

Additional user convenience could be attained by programming certain useful monitoring choices into the analyzer as options. For example, by requesting "MONITOR AT SUBPROGRAM LEVEL", only subroutine entries (and, of course, halt statements) would be monitored.

The next generation of program analyzers will provide a capability to express and process optionally compilable assertions. These are special statements describing conditions which hold at particular points or throughout a subprogram. At the user's option, the analyzer would translate the assertions into run time tests or treat them as documentation, i.e., ignore them. Because assertions can formalize assumptions made across subroutine boundaries, they are especially valuable in managing the development and checkout of large, multi-authored programs. The problems of defining an assertion capability, designing an assertion language, and incorporating assertion processing into an analyzer should be addressed.

Finally, by suitably modifying the static analysis portion of the analyzer, it could serve as a standards enforcer as well.

6.2 Exploratory Research Possibilities: A Network Software Production Facility

A Network Software Production Facility is a sharable environment conducive to the production of reliable software systems. It is an integrated collection of tools and techniques for creating, debugging, testing, documenting, maintaining and transporting reliable software. It would satisfy needs which arise in the generation of both network and user programs, and would serve to improve the quality of both.

The technology already exists to construct a Software Production Facility. Its components, e.g., advanced programming languages, structured programming precompilers, dynamic analyzers, intelligent text editors and sophisticated file management systems, are within the state of the art. What has not been done is to identify and develop an effective set of tools, integrate them into a single system, and make that system available to a broad spectrum of users.

As a first step, overall system structure should be described, focusing on:

- types of tools to be included
- where they fit in the life cycle of the software, and how they contribute to its reliability
- how to ensure consistency, identifying potential areas of difficulty and concentrating on user/tool and tool/tool interfaces
- features to make the tools convenient and effective in remote interactive operation.

To provide practical experience with the concepts and to lend concreteness to the study, a prototype "Reliable FORTRAN Production Facility" could be assembled. Potential components available at NBS include a structured programming language-to-FORTRAN precompiler, the FORTRAN analyzer, and the full set of FORTRAN Compiler Test Routines. These could be installed for experimental usage on the ARPANET and on the NBS PDP-10. Other tools could be considered as well, should they become available.


REFERENCES

- [1] Gentleman, W. M., and Wichmann, B. A., "Timing on Computers," SIGARCH 2, 20-23, October 1973.
- [2] Hoffmann, R. H., Automated Verification System User's Guide, TRW Note #72-FMT-8 Project Apollo, Task MSC/TRW A-527, 1972.
- [3] Holberton, F. E., and Parker, E. G., "NBS FORTRAN Test Programs," NBS Special Publication #399, Vols. 1-3, October 1974.
- [4] Ingalls, D., "The Execution Time Profile as a Programming Tool," In R. Rustin (ed.), Compiler Optimization, 2nd Courant Computer Science Symposium (1970), 107-128, Prentice-Hall, 1972
- [5] Knuth, D. E., and Stevenson, F. R., "Optimal Measurement Points for Program Frequency Counts," B.I.T. 13, 313-322, 1973.
- [6] Larmouth, J., "Serious FORTRAN", Software-Practice and Experience 3, 87-108, 1973.
- [7] Lyon, G. E., and Stillman, R. B., "Simple Transforms for Instrumenting FORTRAN Decks," to appear in Software-Practice and Experience.
- [8] Stucki, L. G., "A Prototype Automatic Program Testing Tool," Proc., Fall Joint Computer Conference, Anaheim, California, Dec 5-7 (1972).
- [9] American National Standards Institute, FORTRAN X3.9-1966. (Available in the U. K from the British Standards Institute.)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. NBS TN-874	2. Gov't Accession No.	3. Recipient's Accession No.
4. TITLE AND SUBTITLE <i>Software Testing for Network Services</i>		5. Publication Date July 1975	
		6. Performing Organization Code	
7. AUTHOR(S) <i>Rona B. Stillman, Ph.D. and Belkis Leong-Hong</i>		8. Performing Organ. Report No.	
9. PERFORMING ORGANIZATION NAME AND ADDRESS NATIONAL BUREAU OF STANDARDS DEPARTMENT OF COMMERCE WASHINGTON, D.C. 20234		10. Project/Task/Work Unit No. 640.2415	
		11. Contract/Grant No. NSF AG-350	
12. Sponsoring Organization Name and Complete Address (Street, City, State, ZIP) National Science Foundation 1800 G Street, NW Room 648 Washington, DC 20550		13. Type of Report & Period Covered FINAL	
		14. Sponsoring Agency Code	
15. SUPPLEMENTARY NOTES Library of Congress Catalog Card Number: 75-600046			
16. ABSTRACT (A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here.) <i>This report is a first step toward identifying effective software test and measurement tools, and developing a guide for their usage network-wide. The utility of two tools, the NBS FORTRAN Test Routines and the NBS Analyzer, is studied experimentally, and indications of their role in systematic testing in a networking environment are given.</i>			
17. KEY WORDS (six to twelve entries; alphabetical order; capitalize only the first letter of the first key word unless a proper name; separated by semicolons) <i>Dynamic analysis; NBS Analyzer; NBS FORTRAN Test Routines; networking; systematic testing; testing tools.</i>			
18. AVAILABILITY <input type="checkbox"/> For Official Distribution. Do Not Release to NTIS <input checked="" type="checkbox"/> Order From Sup. of Doc., U.S. Government Printing Office Washington, D.C. 20402, SD Cat. No. C13, 46:874 <input type="checkbox"/> Order From National Technical Information Service (NTIS) Springfield, Virginia 22151	19. SECURITY CLASS (THIS REPORT) UNCLASSIFIED	21. NO. OF PAGES 40	
	20. SECURITY CLASS (THIS PAGE) UNCLASSIFIED	22. Price \$1.00	

get
a line on
science and
technology.
subscribe to

DIMENSIONS



Whether you're in business, or a teacher, scientist, or consumer, you'll want to keep up with the latest developments in science and technology. DIMENSIONS/NBS, the monthly magazine from the Commerce Department's National Bureau of Standards, can help keep you informed. Every day at NBS, one of the nation's largest research laboratories, scientists seek new answers to a host of national problems, including energy conservation, product safety, metric conversion, and pollution abatement. Their findings, reported each month in DIMENSIONS/NBS, have a direct impact on our daily lives.

Subscription price: \$9.45 per year.
Order prepaid from the
Superintendent of Documents,
U.S. Government Printing Office,
Washington, D.C. 20402
SD Catalog No. C13.13

PERIODICALS

JOURNAL OF RESEARCH reports National Bureau of Standards research and development in physics, mathematics, and chemistry. It is published in two sections, available separately:

- **Physics and Chemistry (Section A)**

Papers of interest primarily to scientists working in these fields. This section covers a broad range of physical and chemical research, with major emphasis on standards of physical measurement, fundamental constants, and properties of matter. Issued six times a year. Annual subscription: Domestic, \$17.00; Foreign, \$21.25.

- **Mathematical Sciences (Section B)**

Studies and compilations designed mainly for the mathematician and theoretical physicist. Topics in mathematical statistics, theory of experiment design, numerical analysis, theoretical physics and chemistry, logical design and programming of computers and computer systems. Short numerical tables. Issued quarterly. Annual subscription: Domestic, \$9.00; Foreign, \$11.25.

DIMENSIONS/NBS (formerly *Technical News Bulletin*)—This monthly magazine is published to inform scientists, engineers, businessmen, industry, teachers, students, and consumers of the latest advances in science and technology, with primary emphasis on the work at NBS. The magazine highlights and reviews such issues as energy research, fire protection, building technology, metric conversion, pollution abatement, health and safety, and consumer product performance. In addition, it reports the results of Bureau programs in measurement standards and techniques, properties of matter and materials, engineering standards and services, instrumentation, and automatic data processing.

Annual subscription: Domestic, \$9.45; Foreign, \$11.85.

NONPERIODICALS

Monographs—Major contributions to the technical literature on various subjects related to the Bureau's scientific and technical activities.

Handbooks—Recommended codes of engineering and industrial practice (including safety codes) developed in cooperation with interested industries, professional organizations, and regulatory bodies.

Special Publications—Include proceedings of conferences sponsored by NBS, NBS annual reports, and other special publications appropriate to this grouping such as wall charts, pocket cards, and bibliographies.

Applied Mathematics Series—Mathematical tables, manuals, and studies of special interest to physicists, engineers, chemists, biologists, mathematicians, computer programmers, and others engaged in scientific and technical work.

National Standard Reference Data Series—Provides quantitative data on the physical and chemical properties of materials, compiled from the world's literature and critically evaluated. Developed under a world-wide

program coordinated by NBS. Program under authority of National Standard Data Act (Public Law 90-396).

NOTE: At present the principal publication outlet for these data is the *Journal of Physical and Chemical Reference Data* (JPCRD) published quarterly for NBS by the American Chemical Society (ACS) and the American Institute of Physics (AIP). Subscriptions, reprints, and supplements available from ACS, 1155 Sixteenth St. N. W., Wash. D. C. 20056.

Building Science Series—Disseminates technical information developed at the Bureau on building materials, components, systems, and whole structures. The series presents research results, test methods, and performance criteria related to the structural and environmental functions and the durability and safety characteristics of building elements and systems.

Technical Notes—Studies or reports which are complete in themselves but restrictive in their treatment of a subject. Analogous to monographs but not so comprehensive in scope or definitive in treatment of the subject area. Often serve as a vehicle for final reports of work performed at NBS under the sponsorship of other government agencies.

Voluntary Product Standards—Developed under procedures published by the Department of Commerce in Part 10, Title 15, of the Code of Federal Regulations. The purpose of the standards is to establish nationally recognized requirements for products, and to provide all concerned interests with a basis for common understanding of the characteristics of the products. NBS administers this program as a supplement to the activities of the private sector standardizing organizations.

Federal Information Processing Standards Publications (FIPS PUBS)—Publications in this series collectively constitute the Federal Information Processing Standards Register. Register serves as the official source of information in the Federal Government regarding standards issued by NBS pursuant to the Federal Property and Administrative Services Act of 1949 as amended, Public Law 89-306 (79 Stat. 1127), and as implemented by Executive Order 11717 (38 FR 12315, dated May 11, 1973) and Part 6 of Title 15 CFR (Code of Federal Regulations).

Consumer Information Series—Practical information, based on NBS research and experience, covering areas of interest to the consumer. Easily understandable language and illustrations provide useful background knowledge for shopping in today's technological marketplace.

NBS Interagency Reports (NBSIR)—A special series of interim or final reports on work performed by NBS for outside sponsors (both government and non-government). In general, initial distribution is handled by the sponsor; public distribution is by the National Technical Information Service (Springfield, Va. 22161) in paper copy or microfiche form.

Order NBS publications (except NBSIR's and Bibliographic Subscription Services) from: Superintendent of Documents, Government Printing Office, Washington, D.C. 20402.

BIBLIOGRAPHIC SUBSCRIPTION SERVICES

The following current-awareness and literature-survey bibliographies are issued periodically by the Bureau: Cryogenic Data Center Current Awareness Service

A literature survey issued biweekly. Annual subscription: Domestic, \$20.00; foreign, \$25.00.

Liquefied Natural Gas. A literature survey issued quarterly. Annual subscription: \$20.00.

Superconducting Devices and Materials. A literature

survey issued quarterly. Annual subscription: \$20.00. Send subscription orders and remittances for the preceding bibliographic services to National Technical Information Service, Springfield, Va. 22161.

Electromagnetic Metrology Current Awareness Service Issued monthly. Annual subscription: \$100.00 (Special rates for multi-subscriptions). Send subscription order and remittance to Electromagnetics Division, National Bureau of Standards, Boulder, Colo. 80302.

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Washington, D.C. 20234

OFFICIAL BUSINESS

Penalty for Private Use, \$300

POSTAGE AND FEES PAID
U.S. DEPARTMENT OF COMMERCE
COM-215

SPECIAL FOURTH-CLASS RATE
BOOK

