A11103 089231
NAT'L INST OF STANDARDS & TECH R.I.C.
A11103089231
Hardy, I. Trotter/Software tools : a bui
QC100 .U57 NO.500-14, 1977 C.1 NBS-PUB-C

C CE & TECHNOLOGY:

SOFTWARE TOOLS: A BUILDING BLOCK APPROACH

0
7
.500-14
77
3

NBS Special Publication 500-14
U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards

NATIONAL BUREAU OF STANDARDS

The National Bureau of Standards[1] 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, the Office for Information Programs, and the Office of Experimental Technology Incentives Program.

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 the Office of Measurement Services, and the following center and divisions:

Applied Mathematics — Electricity — Mechanics — Heat — Optical Physics — Center for Radiation Research — Laboratory Astrophysics[2] — Cryogenics[2] — Electromagnetics[2] — Time and Frequency[2].

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, the Office of Air and Water Measurement, and the following divisions:

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

THE INSTITUTE FOR APPLIED TECHNOLOGY provides technical services developing and promoting the use of available technology; cooperates with public and private organizations in developing technological standards, codes, and test methods; and provides technical advice services, and information to Government agencies and the public. The Institute consists of the following divisions and centers:

Standards Application and Analysis — Electronic Technology — Center for Consumer Product Technology: Product Systems Analysis; Product Engineering — Center for Building Technology: Structures, Materials, and Safety; Building Environment; Technical Evaluation and Application — Center for Fire Research: Fire Science; Fire Safety Engineering.

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 wthin the executive branch for the development of Federal standards for automatic data processing equipment, techniques, and computer languages. The Institute consist of the following divisions:

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

THE OFFICE OF EXPERIMENTAL TECHNOLOGY INCENTIVES PROGRAM seeks to affect public policy and process to facilitate technological change in the private sector by examining and experimenting with Government policies and practices in order to identify and remove Government-related barriers and to correct inherent market imperfections that impede the innovation process.

THE OFFICE FOR INFORMATION PROGRAMS promotes optimum dissemination and accessibility of scientific information generated within NBS; 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 Standards — Office of International Relations.

---

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

[2] Located at Boulder, Colorado 80302.

JAN 26 1979
not Acc-Circ
QC100
.057
NO. 5701
1977
c.2

COMPUTER SCIENCE & TECHNOLOGY:

Software Tools: A Building Block Approach

I. Trotter Hardy
Belkis Leong-Hong, and
Dennis W. Fife

Systems and Software Division
Institute for Computer Sciences and Technology
National Bureau of Standards
Washington, D.C. 20234

Partially sponsored by

The National Science Foundation
18th and G Streets, N.W.
Washington, D.C. 20550

U.S. DEPARTMENT OF COMMERCE, Juanita M. Kreps, Secretary

Dr. Sidney Harman, Under Secretary

Jordan J. Baruch, Assistant Secretary for Science and Technology

NATIONAL BUREAU OF STANDARDS, Ernest Ambler, Acting Director

Issued August 1977

Reports on Computer Science and Technology

The National Bureau of Standards has a special responsibility within the Federal Government for computer science and technology activities. The programs of the NBS Institute for Computer Sciences and Technology are designed to provide ADP standards, guidelines, and technical advisory services to improve the effectiveness of computer utilization in the Federal sector, and to perform appropriate research and development efforts as foundation for such activities and programs. This publication series will report these NBS efforts to the Federal computer community as well as to interested specialists in the academic and private sectors. Those wishing to receive notices of publications in this series should complete and return the form at the end of this publication.

National Bureau of Standards Special Publication 500-14

Nat. Bur. Stand. (U.S.), Spec. Publ. 500-14, 66 pages (Aug. 1977)
CODEN: XNBSAV

Library of Congress Cataloging in Publication Data

Hardy, I. Trotter
Software Tools.

(Computer science & technology) (National Bureau of Standards special publication ; 500-14)

Supt. of Docs. no.: C13.10:500-14

1. Programming (Electronic computers) I. Leong-Hong, Belkis, joint author. II. Fife, Dennis W., joint author. III. Title. IV. Series. V. Series: United States. National Bureau of Standards. Special publication ; 500-14.

QC100.U57 no. 500-14 [QA76.6] 602'.1s [001.6'425] 77-608213

U.S. GOVERNMENT PRINTING OFFICE
WASHINGTON: 1977

For sale by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402
(Order by SD Catalog No. C13.10:500–14). Stock No. 003–003–01823–6. Price $2.10
(Add 25 percent additional for other than U.S. mailing).

TABLE OF CONTENTS

NOTE

Certain commercial products are identified in this paper in order to specify adequately the experimental procedure, or to cite relevant examples. In no case does such identification imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the products or equipment identified are necessarily the best available for the purpose.

Software Tools:
A Building Block Approach

I. Trotter Hardy
Belkis Leong-Hong
Dennis W. Fife

The present status of software tools is described; the need for special-purpose tools and for new techniques with which to construct such tools is emphasized. One such technique involving the creation of general-purpose "building blocks" of code is suggested; an initial application of the technique to the construction of a text editor and syntax analyzer tool is described. An annotated bibliography of current literature relevant to software tools is provided.

Key words: Building blocks; programming aids; software tools; syntax analysis; text editing.

1. INTRODUCTION

Surveys such as Reifer's [Reifer] of the state of the art in software tools emphasize the benefits that these automated aids can bring and point out clearly the need for greater emphasis in this area. Further, workers with experience in large systems development projects invariably cite good tools as a major contributor to project success. Corbato [Corbato], for example, states very strongly that the use of a high-level language as a systems implementation tool was well worth the additional start-up effort it necessitated on the MULTICS project. Brooks [Brooks] reiterates the case for high-level language tools, and goes on to state that every large software development project requires a "tool maker" to maintain commonly used tools and to develop special purpose ones as needed.

The utility of software tools like high-level language compilers, debuggers, and text editors is thus quite clear. Yet, for most programming researchers and practitioners, the need for better tools is obvious (see [Balzer] and [Grosse-Lindemann], for example). Brooks' argument for a tool-maker on every major software project indicates also that the state of the art has not reached a level where prefabricated tools can merely be selected and put to immediate use. More importantly, perhaps, Brooks' observations show

the importance of special-purpose tools for particular projects, and the decided need for project managers to allow resources to be expended in the creation of such tools, both by the project tool-maker and the individual project members. Ad hoc tool creation in this manner--necessary as it is--unfortunately falls heir to all the errors and reliability problems of the applications software it is designed to support. Major software projects can thus be plagued with major tool expenditures just to allow efficient subsequent expenditures on the project's main product. If this resource expenditure could be reduced through some more organized approach to software tool libraries in general and to the creation of special-purpose tools in particular, might not worthy benefits in time, cost, and tool reliability accrue? The present investigation of software tools was undertaken to explore precisely that question; this report summarizes the results of the investigation.

2. TOOLSMITHING

It is evident from prevailing experience and research that every software production project, regardless of complexity, must include a tool provisioning activity. The toolsmith faces several questions, to be answered in collaboration with the project manager or "chief programmer" (if a different person than the toolsmith).

1. Is there a commonly accepted set of standardized tools applicable to every project?

2. What set of special tools for a given project can be identified at the outset?

3. Are necessary tools already available as commercial packages with acceptable cost?

4. What are the economical approaches to creating special tools and modifying them as may be needed in the course of a project?

The general evidence for answering these questions shows there is inordinate difficulty in providing tools under the present state of the art and conditions of the software marketplace. Many tools are available commercially at reasonable cost; but there is essentially no standardization of tool capabilities, and the number of suppliers and the complexity of packages presents an arduous selection problem for the customer. But equally important is that proprietary packages cannot be specifically modified and tailored by the user since the source code is usually not delivered in the purchase. Although a basic set of tools is identifiable for any project, we believe also that special modifications are warranted in many cases. Furthermore, a general expansion and integration of available tool functions would be well-advised to cope with the widely-recognized problem of software quality control. The following analysis tends to support a recommendation for standardization of basic tools at the source code level, so that future software production can be conducted with a common set of tools amenable to user extension and specialization. Moreover, it sets the perspective for the remainder of this research on specialization of tools.

Types of Tools

The only standard tool for software production today is the high-level language compiler. This statement applies the traditional understanding that a standard is a formal specification produced by a recognized professional group for nearly universal application. Yet, national and international standardization of compilers has only addressed programming language definition, while not considering the language's essential features, such as the form and content of output listings, the content and scope of diagnostic messages, debugging features, and alternative operational modes.

Even so, common usage of software tools and the economics of tool design have led to commonly discernible types of tools. For each type, many competitors may be found in the market. These types cannot be called defacto standards, for they only reflect similar purpose and function, and not by any means a near equivalence of capability brought about by competitive commercial demand. The following definitions of such common types of tools have been determined after a survey of commercial packages as to similarity. The definitions therefore categorize the tools found in the marketplace. This listing omits (for brevity or because of limited relevance) those packages that may be classified as compilers, assemblers, data base management systems, utility routines, application programs or libraries (e.g. mathematical routines), or replacement packages for software normally offered by a hardware vendor such as operating systems and I/O access methods.

Abort diagnosis - Provides a full or selective dump of the contents of registers and memory whenever an abnormal termination of program execution occurs. May also provide diagnostic analysis of the abnormal end.

Breakpoint control - In an interactive environment, permits the user to specify program points where execution is to stop and control is to be returned to him. He may then examine or change program values for debugging purposes.

Cross reference generator - Produces a listing of variables used in a program and subroutines, indicating where each variable is being referenced.

Data auditor/catalog - Examines source data definitions and analyzes data relationships, data structures, formats and storage usage for consistency checks, validation, and storage utilization. May provide a data dictionary or catalog that contains definitions of the attributes and characteristics of the data elements.

Error analysis and recovery - Certain kinds of abnormal termination are intercepted, and new program or data values may be substituted, thus allowing the execution to resume; produces appropriate error messages.

File or library manager - Facilitates organized and economical storage of program texts, data sets, and object modules for centralized retrieval and update. May collect accounting and usage data to assist in storage allocation.

Flowchart generator - Generates a pictorial diagram of flow of control of a given higher level language program.

Program auditor - Checks a source program to determine conformance to specified standards or criteria on design or programming language use. May provide test data and exercise the program for predetermined results to obtain a minimum standard of acceptability for a generic class of programs. May also analyze the program for static characteristics, such as frequency of occurrence of syntactic units or statement types.

Program execution monitor - Instruments source programs to collect data on the program during its execution. Data collected can include:

- frequency of execution of each statement,
- range of values of specified variables,
- trace of the path of execution through labeled statements, or
- snapshot dumps of variables during the trace.

Program formatter/documentor - Rearranges and structures source program text for improved readability. May provide limited text additions for documentation purposes.

Project manager - Provides data collection, storage and reporting facilities aimed at personnel time and task accounting. May be coupled with PERT packages and other productivity and scheduling management aids.

Resource monitor - Provides job accounting information about utilization of system resources. May include costing algorithm for billing purposes.

Shorthand or macro expander - Produces full source text for programs from an abbreviated programming language or parameterized input forms, to ease the coding effort for application programs or job control commands. Also includes a number of packages for generating higher level source statements from decision tables.

Source level translator - Translates from one high level language (e.g. RPG) to another (e.g. COBOL). Significant case is the translation from one version of a given language to its standard version. Also includes structured programming languages as preprocessors.

Test data generator - Generates test data files according to user specification, to be used for testing application software.

Test simulator - Simulates the execution and flow of control of a test program by program analysis with actual or dummy variables and subroutines. Control may be passed to the user, who then may specify a course of action to be taken during the testing procedure.

Text or program editor - Facilitates selective modifications and corrections of program or document text.

Trends

This limited survey of commercial tools shows trends in the relative availability, price range, and flexibility among them. The table below (pages 8-9) illustrates that among 315 packages examined (from announcements in widely published software catalogs), the dominant types relate to documentation and source level translation (specifically, the types named "shorthand or macro expander," "cross reference generator," "source level translator," etc.). There are significantly fewer packages available for debugging and testing of application programs--probably because of traditionally heavy dependence on the compiler alone as a debugging tool, and the lack of a broadly effective methodology for debugging and testing on which to base a commercial tool.

Another trend shown is that tools tend to have a narrow function corresponding to the definitions above. We have indicated under "secondary function" the number of additional packages falling under each type definition based on the secondary capabilities that may be present, as opposed to the primary purpose of each tool. Considerable overlap is seen in documentation aids, namely "cross reference generator" and "program formatter/documentor," where the number with such secondary capability is nearly as many as those in the primary class. But most other types such as "breakpoint control" and "resources monitor" are seen to be narrowly specialized.

Purchase prices tend to fall in a limited range up to about $8000, except for a few types such as "data auditor/catalog" and "project manager," where prices are relatively higher. Tools of most types can be had at a very low cost covering reproduction, particularly if the producer is a user outside the EDP industry. But for most tool types there are also commercial producers in the EDP industry whose prices fall in the higher range. Also, the catalogued tool descriptions indicate that most are applicable only to programs in one programming language, e.g. COBOL, and that many are available only for the computers and operating systems of the foremost manufacturer.

It should be noted that a wide variety of producers are involved in the software tools marketplace, including hardware manufacturers, computer users (commercial and not-for-profit), and independent software houses. Roughly 20 user groups, 1000 software companies, 50 hardware vendors, and 40,000 user installations may be potential sources of usable and effective tools.

Because of these circumstances, the availability and use of tools follows a mixed pattern across the population of computer installations. A recent NBS survey of federal computer installations [Deutsch] revealed that 75 percent of them acquired programming tools along with their hardware procurements, over 50 percent also developed tools themselves or acquired them from non-commercial sources such as user groups, while as few as 36 percent acquired tools under separate procurements. The latter statistic particularly suggests a weak market for tools from the independent software companies. Only about half of the installations reported that certain tools, namely flowchart generators and debugging packages, were available, yet 65 percent said that programming was primarily done by individuals or small project teams. These statistics warrant more extensive investigation, but they tend to confirm that planning and investment for adequate software tools is not consistently a high priority effort for installation managers.

Type of Tool	Number of Packages		Typical Price Range (Purchase)
	Primary Function	Secondary Function	
Abort Diagnosis	10	7	$0.2k - 2.5k
Breakpoint Control	11	1	$4.0k - 7.5k
Cross Reference Generator	27	33	$0.1k - 0.5k
Data Auditor/Catalog	15	6	$10.0k -15.0k
Error Analysis and Recovery	16	6	$0.2k - 1.0k
File or Library Manager	33	2	$0.1k - 0.5k $2.0k - 5.0k
Flowchart Generator	10	9	$0.2k - 0.5k $0.2k - 0.5k
Program Auditor	8	2	$0.2k - 1.0k $5.0k -11.0k
Program Execution Monitor	17	8	$2.0k - 6.0k
Program Formatter/ Documentor	24	19	$0.1k - 0.5k $1.0k - 8.0k
Project Manager	11	4	$4.0k - 12.0k
Resources Monitor	27	1	$2.0k - 8.0k
Shorthand or Macro Expander	42	8	$0.2k - 1.0k $5.0k -15.0k

(continued next page)

(continued from previous page)

Source Level Translator	30	1	$2.0k - 8.4k
Test Data Generator	9	5	$4.8k - 6.5k
Test Simulator	12	2	$3.0k - 4.5k
Text or Program Editor	13	7	$0.3k - 0.5k $3.0k - 5.0k
TOTAL	315	111	

Minimum Essential Tools

Prevailing experience and professional consensus are sufficient grounds to recommend certain types of tools as essential for almost any software development project. Exceptions may arise if the computer involved has such unusual architecture or limited capabilities (e.g. no mass storage) that it cannot support even these modest program packages. Minicomputer systems generally would not be excluded, particularly since the UNIX system [Ritchie] has demonstrated that a highly effective, interactive programming support system is practical on a low-cost minicomputer configuration.

In general, program system development should be done with support of an interactive computer system. Interactive support increases productivity in the effort of continual change, debugging, and testing that characterizes most projects. It also facilitates the ultimate objective to have the computer provide a total support environment to the project group.

The primary tool for any project should be a high-level programming language compiler. Again, experience has amply proven the advantages for programming productivity of high-level languages. Only selected programs that are critical to system performance may need to be optimized for efficiency through machine language.

However, other essential tools are the focus of this report, and the following are recommended as a minimum complement for most projects.

Text editor - For entering, correcting, and modifying such texts as program specifications and design documentation. Requires a facility for online storage and recall of named text units for inspection, printing or editing.

Program editor - For entering, correcting, and modifying program texts. With free-form programming languages, one editor could serve both as text and program editor.

Program librarian - For storing all program texts, associated job control statements, common data definitions, and test data, and maintaining a chronological record of modifications between distinct versions.

Debugger - For analyzing program behavior during execution on test data input, and deriving execution statistics and traces to help correlate program output with the results of individual high-level language statements.

Project manager - For recording chronologically the activity of the individual project members on defined program modules and deliverables of the project.

Specialization of Tools

The above definitions are meant only to give a general notion of the capability involved in each type of tool. Standard specifications of functions for each tool type appear feasible and desirable, and would assist those who undertake toolsmithing without benefit of prior study and experience. Yet it is clear that individual projects often may need to create special features that would not be available in standardized tools. Various project requirements or circumstances may dictate such specializations. For instance, large projects with many personnel especially would benefit from extensions to automatically enforce unique design standards and practices that are difficult to ensure through personal communications and code inspections. A contrasting example would be a one-person project of converting a program from one source system to another target system, that may benefit considerably from specializations tailored to the peculiar programming language dialect on the source machine.

Desirable specializations may range in difficulty from minor extensions of extant tool functions to new composite tools formed by integrating and refining several distinct packages. Both of these cases require the original tool's source code--ideally in a high-level language--and thorough documentation of course. The latter case also requires that the building block tools be carefully thought out, with flexible interfaces and modular design, permitting extensive modifications with relative ease.

It is appropriate therefore to recommend significant new research and development on programming tools, with the following goals:

1. to make widely available a set of building block tools, with standard designs and source code in a high-level language;

2. to evaluate alternative techniques for interfacing and modular design that would facilitate major modifications of tools without loss of efficiency and performance; and

3. to develop guidelines for rapid and reliable specialization of tools from available building blocks, based upon the characteristics of projects that would yield significant benefits from special tools.

3. EDITOR AND SYNTAX ANALYZER LINKAGE

A review of the program production aids described in section 2 shows that a great many tools--such as static and dynamic program analyzers, keyword extractors, cross reference listers, macro expanders, and the like--depend for their operation on knowledge of the programming language for which they were designed. Such a "knowledge" of a language in turn depends typically on several high-level algorithms, chiefly the following:

-- lexical analyzer, or scanner, which reads the characters of the source text and translates them into a series of single integers representing the basic elements of the language, such as reserved words, identifiers, operators, etc. Comments are usually passed over so they are never of concern to the syntax analyzer;

-- keyword recognizer, which is a simplified form of lexical analyzer that recognizes and encodes in an internal form all reserved words used in a program (e.g., BEGIN, END, IF, THEN, etc.) and perhaps other language elements as well, such as identifiers, or array references;

-- symbol table handler, which builds and maintains a table of identifier names and can determine if a given identifier has been encountered before, if it has been previously declared as to type (integer variable, subroutine name, etc.), perhaps--if it is a variable--whether it has been previously assigned a value, and so on;

-- syntax analyzer, which parses the source text (usually it's integer representation from the lexical analyzer) and determines what language construct is currently being read and if it is correct.

Since a large number of diverse tools were found to depend on a small number of moderately sophisticated algorithms, it seemed possible to code those algorithms independently in some fashion that would enable a competent programmer to join them with one another and with other programs rapidly and reliably in ad hoc ways. If such a

capability were feasible, it might greatly simplify the problem of building special purpose tools that require a degree of language "fluency."

In such a scheme, the individual algorithms would become "building blocks" available in a library, to be called upon when necessary. Such a library would be much like a typical installation's library of subroutines, only the individual "subroutines" would be considerably more sophisticated, perhaps even independent programs. Further, the library of building blocks would lend itself naturally and perhaps most usefully to network installation at a particular host designed as a special-purpose software development facility, such as the "National Software Works" presently under development [Carlson], since the presumed sophistication of the building blocks could easily make it impractical for a given installation to develop and maintain all of its own.

To test the building block idea, a novel (for this installation) tool composed of several building blocks was envisioned: a program editor with a syntax analyzing capability. The decision to pick this combination was based on a perception that productive work in programming today centers increasingly on the development of a complete programming environment, where all tools are coordinated and geared to program development and checkout ([Balzer], [Bratman], [Donzeau-Gouge], [Grosse-Lindemann], [Scowen]). The editors described in [Donzeau-Gouge] and [Van Dam] were especially influential in this choice. The development of a fluent editor could provide useful secondary experience in establishing a first step toward such an integrated programming environment, in addition to primary experience in analyzing techniques for the establishment of tool building blocks. The editor would be tailored for a particular programming language and would permit syntax checking of source code at its entry, as well as checks of code that had been previously entered without checking. Further, the editor might permit displays in terms of language constructs, such as statements, in addition to the usual editor displays in terms of lines or characters. The editor would be constructed by joining separate algorithms not necessarily intended for joining, observing carefully what changes were required, and reaching conclusions regarding what interfaces would have been best suited to an automatic and error-free linkage.

The editor has now been built as a prototype, and as expected, its construction has highlighted many of the problems of joining large building blocks of code in arbitrary ways; serendipitously, it has opened up several promising avenues for exploration of new concepts in program creation and editing generally. Unfortunately, the whole area of

linking programs and subroutines is one of great variation across hardware-software configurations, and experience from this study is primarily applicable to one programming language and its implementation environment: ALGOL on a Digital Equipment Corporation PDP-10. Yet, some conclusions into what is essentially a general problem were reached.

Functional Description

The editor was written independently as a stand-alone program with the typical features of a line-oriented text editor: lines can be entered, deleted, and listed by line number keys; strings of characters can be replaced by other strings or deleted; all or selected groups of lines can be written out to the permanent file at any time, or discarded; and so on. A variant of ALGOL-60 is the source language of the editor itself; ALGOL was chosen as it is both a high-level and a well structured language (and an available language on the NBS experimental computing facility). The particular variant available also offered what appeared to be fairly straightforward handling of individual characters and strings of characters, and this was felt to be a decided advantage in a text manipulating program.

After a working version of the editor was prepared, the existence of an ALGOL-60 syntax analyzer--written in ALGOL--was learned of and obtained [Wichmann]. Obtaining an existing algorithm was an unexpected benefit, for it lent realism to the task of joining algorithms that had not been written specifically for each other. A lexical scanner was required and written for the analyzer, and modifications made to the algorithm to enable it to recognize the local facility's ALGOL variant. The scanner was initially programmed and debugged as a stand-alone program, and later incorporated as a procedure within the syntax analyzer. The syntax analyzer with scanner was then completed and also initially made operational as a stand-alone program.

This syntax analyzing program accepted a file of ALGOL source code as input, and read that file until either an end-of-file or a syntax error was encountered, at which point it terminated with an appropriate message. This scheme, although rather simple, was well suited to the analyzer's purpose of avoiding complete compilation of a program to identify essentially trivial syntax errors. It was also well-suited to combination with an editor, since immediate termination at the point of an error would also allow immediate correction and quick resumption of the check.

Once both the editor and the syntax analyzer were operational independently, they were joined. The resulting combination is an editor that allows text entry, modification and deletion as before, but with the optional extra capability of checking that text for correct ALGOL syntax. The checks can be made immediately as each line is entered or suppressed at entry and made after entry is complete. These modes can also be alternated or suppressed entirely. Checking is done for one of three ALGOL constructs: block, procedure declaration or statement. Additionally, if checking of a block or procedure is requested upon its entry, the editor prompts for the first line with the keywords "BEGIN" or "PROCEDURE," as appropriate.

A few examples will suffice to give the flavor of a user's interaction with the editor-syntax analyzer combination. In the following examples, the user-typed material is in lower case, the editor-typed material in upper case. The asterisk is the editor's prompting symbol, by which it indicates its readiness to accept the next command.

*insert 100,10	the user asks to insert text (of any kind) at line 100 in lines incremented by 10
100 <user typed text> 110 <more text> . . . NNN <end of text>	a special character signals the end of text entry
*	the editor signals its readiness for the next command

```
*insert procedure 100,10          user requests the insertion of
                                  an ALGOL procedure starting at
                                  line 100--note similarity with
                                  previous command

100 PROCEDURE  <user typed procedure name>
110 <user typed ALGOL text>
        .
        .
NNN <end of ALGOL procedure>      if a syntactically correct
                                  procedure is entered,
                                  the editor stops entry
                                  automatically

[NO ERRORS]                       the editor notes that the
                                  procedure was okay, and

*                                 prompts for the next command.
```

```
*insert 300,10                    text entry with no syntax
                                  checking is requested again

300 procedure alpha(x,y);         an incomplete procedure
310 integer x,y;                  is entered (the initial
320 begin                         BEGIN on this line is not...
330     if x < y then x := y;
340 <end of text>                 matched by an END here)

*check procedure 300              now a syntax check
                                  is requested of the
                                  procedure starting on
                                  line 300

TOKEN MISSING OR MISSPELLED       the error is noted,
... NEAR LINE 340                 the location of the error,
MISSING TOKEN IS "END"            and the nature of the error


*insert 350                       the user attempts to
                                  correct the omission
350 end;
360 <end of text>
*check procedure 300              check it again
[NO ERRORS]                       it's okay
*
```

Design Description

Conceptually, one might view the joining of the editor and syntax checker as the uniting of two algorithms to access a common data structure: the editing algorithm, whose primary role is the interpretation of a user's commands and the consequent manipulation of the data structure in accord with those commands, and the analyzer algorithm, whose role is the syntactic analysis of text obtained from the data structure. A significant task in bringing about that union was the creation of a common routine to permit that common access. The resulting "linking pin" between the editor and analyzer in the present case is a small procedure within the syntax analyzer called "getchar." This procedure is invoked by another procedure, "getsymbol", which calls getchar repeatedly to build up the next source program symbol ("getsymbol" is really the lexical scanner). Getsymbol is in turn repeatedly called by the syntax analyzer whenever it needs to look at the next source program symbol.

The link with the editor is simply that getchar has access to the editor's buffer of text and to a boolean variable that indicates whether text is being entered at that moment, or if a check is being made on text already in the file. If text is being entered, getchar picks up each character from the terminal as it is typed, enters it into the editing buffer, and then passes it back to its caller, getsymbol; if a check is being performed on text that is already in the buffer, getchar simply picks up the characters one at a time from the editing buffer and passes them back to getsymbol without further processing. Thus when a user enters a command to the editor, it is first interpreted and then--if syntax checking is called for--control is passed promptly to the appropriate syntax analysis procedure, which does not know where the source program symbols are coming from at all, only that they are obtained by calling getsymbol. Commands that do not call for syntax checking are handled directly by the editor. The user, of course, is unaware of the internal distinctions.

Results

Experience with joining the pieces of the "fluent" editor--the lexical analyzer, the syntax analyzer, and the editor--has pointed out a number of areas of concern for the general problem of such linkages. In particular, the natural tendency to classify a library primarily by its content must be resisted: one library of language analyzers, another of text manipulators, another of output formatters, etc., may seem the logical order of classification, but such an order must be subordinate to a classification by type of

interface.

For example, each of the three building blocks of the editor was first coded as a stand-alone program, but two were later modified to run as procedures (subroutines) of the third. The process of changing an independent program to a subroutine is conceptually trivial, and yet that phase of the project was fraught with almost as many minor clerical errors as the entire coding from scratch of the lexical analyzer or the editor (the syntax analyzer, as noted, was coded elsewhere). While bugs that showed up were generally much more quickly tracked down during the linking phase (but not always), their obvious origin only made it seem all the more needless that they should have appeared at all. Certainly for a production library of building blocks, such errors should be obviated if at all possible.

Particular problems of the linking phase included some that are embarrassingly obvious in retrospect: the syntax analyzer, as a stand-alone program was designed by its authors to be simple and consequently quite fast in operation; this speed was obtained at the cost of a rather unsophisticated method for handling an error once detected: control was passed to a failure handling procedure, which issued an error message and then branched unconditionally to a label at the end of the program, causing program termination. This mechanism was immediately seen as unsatisfactory for use with the editor, as the syntax analysis was to be callable at different "entry points" (one to recognize a block, one a procedure, and one a statement), and returns had to be normal procedure exits to avoid repeated calls resulting in unwanted recursion. Consequently, the calls to the failure handling procedure were modified so that returns were made normally to the caller. Unfortunately, what was overlooked was the recursive nature of the analysis procedures during their normal operation (top-down, recursive descent parsing): once the editor-analyzer found an error, it was often three or four levels of recursion deep, and as it unwound it repeatedly issued error messages that gave different reasons--for what was essentially the same error--at each level of recursion. (This problem was, incidently, irritatingly simple to correct, but it proved one more obstacle to joining two building blocks cleanly and simply.)

The result of the linking effort, then, showed that the most fundamental classification scheme for tool building blocks should be whether the building block is coded as a stand-alone program or as a subroutine. Even with this apparently clean division, some operating systems blur the difference by allowing programs to be called as subroutines from the operating system level [Barron], but for most installations such calls are non-recursive and more restricted

than program calls to subroutines, and the stated classification stands up reasonably well. Further, it is clear that developing as a stand-alone program what is known to be later used as a subroutine, is a poor idea: too many irksome, trivial bugs show up as a direct result of the conversion. The proper approach appears to be to develop a subroutine as a subroutine, using if necessary a dummy block of code as the calling program. Parenthetically, one notes that this scheme is the reverse of the common structured programming approach, where a developed block of code initially calls dummy subroutines. The reversal seems appropriate, since the emphasis in this study is on subroutines (and programs) that will be used by unknown calling programs in unforeseeableways, not on known calling programs themselves.

Developing as one unit a subroutine that may subsequently be used with multiple entry points is also clearly a poor idea, unless possibly the programming language being used supports such entries. In the case of the ALGOL procedures of the present study, for example, the desire to call the syntax analyzer at different points and the necessity of having a common procedure linking them, made it impossible to view the analyzer as a "black box" of code, and it thus was incorporated into the editor as a series of individual procedures declared at the same scope level as the primary editor procedures. This type of incorporation also necessitated a large pool of global variables, namely the sum of all global variables in the editor and syntax analyzer, and in practice this meant quite a bit of text transposition and re-ordering, and some time spent ensuring that global variable names did not conflict (they did initially). This experience suggests that, in this case at least, a "syntax analyzer" as such may be too large a building block for flexible use (although certainly appropriate for some applications). Preferable, perhaps, would be smaller units of syntactic analysis, such as "for-blocks," "procedures," and "statements," or--for less structured languages such as FORTRAN--perhaps just for individual statement types. Alternatively, if a library of building blocks is being assembled for production use rather than experimental conclusions, use of language-dependent features such as multiple entry points or procedure names as parameters may well avoid fragmentation of conceptual entities.

The uniqueness of these "language-dependent features," in fact, makes it clear that "source program language" will be another major building block classification, along with "stand-alone program vs. subroutine." Some systems do have the capability of linking relocatable object modules output from different language processors, but for obvious reasons constraints on parameters are severe: on the PDP-10, for example, ALGOL programs can call FORTRAN subroutines, but the

parameters can only be integer, real or boolean variables called by value, and no input or output can take place in the subroutine. Thus for practical purposes, a library of building blocks will of necessity keep different source language elements distinct.

Despite one's best efforts at treating a piece of code as a "black box" and consequently incorporating it unchanged as a subroutine into another program, it may be necessary to create a subroutine interface between the two elements (as, for example, the "getchar" procedure described above). In such a case, the wisest (and obvious) course is to confine all such overlap to the single interface routine. In some languages, more explicit techniques for expressing the union of two routines where one cannot properly be considered a "black box" subroutine of the other are provided, and would doubtless have been useful in joining the editor and syntax checker: the implementation of "coroutines" in SIMULA 67 is such a technique [Dahl]. For commonly used languages like ALGOL, FORTRAN and COBOL, however, such techniques are typically absent.

Absent also from the majority of languages, but of potentially great help in reducing minor errors in such circumstances, are language features that permit the declaration of variables to be shared among a few specified procedures, in addition to the usual declaration mechanisms of "local" to a procedure, or "global" to the entire program. The need for such language mechanisms has long been noted and that need was only emphasized in the present project. In addition, whenever the joining of two algorithms depends principally on their common access to one data structure, a language that permits the maximum possible independence of the accessing procedures and the physical data structure itself will prove of value. Although that sort of independence is now routinely provided in, for example, data base management systems, its provision in generally-used programming languages is considerably less in evidence (note, however, recent languages which have begun to provide such a facility; see especially [Brinch Hansen], [Tennent], [Wulf71] and [Wulf73]). Future efforts at composing building block libraries will in fact focus much more strongly on the issue of data structures and their accessing procedures, as that seems the source of much productive work in module specification and linkage today ([Parnas72a], [Parnas72b], [Parnas72c], [Wulf73]).

The coding of the lexical scanner and the editor, and their joining with the syntax analyzer were largely the work of one person. It was observed that this individual's temptation to take advantage of intimate knowledge of all three elements by various forms of code "trickery" to expedite

their union was all but irresistible. Thus followed the observation that in a production oriented library of building blocks, each building block should be coded by a different individual, or at least by the same individual at different times, according to pre-defined interface criteria. Further, and more to the point practically, building blocks should not be created for particular applications and then entered into a library for general-purpose use, without at least thorough testing for generality by someone who did not participate in the initial design and coding. This separation of responsibilities accords, not surprisingly, with auditing and accounting practices.

The observations regarding the necessity of clearly identifying the type of interface of building blocks in a library suggested that a look at the various interface possibilities might be useful. To that end, the following operating systems--familiar to the researchers--were informally reviewed: Digital Equipment Corporation's TOPS-10 for the DECSystem-10, UNIVAC's Exec-8 for the 1108, Bell Labs' UNIX for the PDP-11, and Computer Science Corporation's CSTS (INFONET) for the 1108. The review showed that, in fact, a variety of linking mechanisms are typically available on present time-shared systems. These mechanisms range from an essentially human link, where a user might run a program, record or remember the results, and then provide those results to another program subsequently executed; to a quite sophisticated and very easy to use mechanism (UNIX) where a series of programs can be strung together through "pipes," or internal buffers established and maintained by the operating system for interprogram communication. In the latter case, for example, the command

```
alpha datafile | beta | gamma | delta
```

means that program 'alpha' operates on on file 'datafile,' passing the resulting output to program 'beta' as the latter's input; 'beta' in turn produces output which becomes input to 'gamma,' and so on. In between this elegant interface and the manual one fall the more usual mechanisms supplied by most operating systems, which customarily include communication by intermediate files for separate programs, and libraries of installation-standard subroutines and functions that are scanned automatically by compilers and loaders for joining with a calling program. For the present purpose of constructing a library of fairly sophisticated building blocks to be joined in unforeseen ways, the following classification of linking mechanisms was developed. Although reference is made only to joining two such blocks, the same ideas extend to any number of such blocks.

1. Stand-alone programs, linked through

1.1 Files

1.1.1 manual interface: user runs one program, then the next on the file of output produced by the first

1.1.2 operating system job control language: user sets up a job stream that accomplishes without intervention the same as (1.1.1) above

1.1.3 direct program-to-program: one program writes a file, then "calls" another directly (e.g., BASIC "CHAIN" statement); an operating system call is required, but hidden from the user

1.2 Buffers

Job control language specifies which programs are to pass output to which other programs, with no files used (UNIX is the only system to date to the researchers' knowledge that permits such a link).

1.3 Human interface

The user remembers or writes down the output of a program, then enters it manually to the next executed program.

2. Subroutines, linked through

2.1 Textual incorporation

The user types, or uses an editor or macro facility to automatically copy, the subroutine into the text of his program. Both explicit parameters and global variables can form the actual communication link; also, a common subroutine that is "aware" of both building blocks and can be called by either is possible.

2.2 Execution-time calls

> The system linker or loader ensures that an object-code copy of the subroutine is accessible to the user's program at execution time. Communication is through explicit parameters, or (if the language permits, like FORTRAN) through global (COMMON) variables.

Naturally, this scheme does not include everything of interest for joining blocks of code; there are other dimensions to the problem that do not lend themselves to incorporation in a simple outline format. In particular for subroutines, one notes the necessity of considering the type of parameter (integer, real, boolean, etc.), how the parameter is to be passed (by value, by reference, or by name), and which parameters are purely input to the routine and which are returned with a newly-assigned value. The possibility of program termination being invoked by the subroutine must also be considered, as well as the subroutine's method for handling exception conditions. Once a production library of building blocks became established, it is likely that a formal means of interface specification would be required. Fortunately, a number of efforts at formal interface specification have been undertaken, some with apparent success. Parnas [Parnas72a], [Parnas72b] in particular has done significant work in this area.

Conclusion

Experience with joining the text editor and the ALGOL syntax checker has led to the basic conclusion that libraries of software tool building blocks can probably be established to permit rapid and simple assembly of new, perhaps special-purpose tools that require some degree of programming language fluency. The "probably" is a necessary caveat, for it is clear that a prototype library of several building blocks must now be assembled and tested in programming projects where special purpose tools like the editor-syntax checker combination are desired. When this step is taken, a more definitive statement can be made regarding the feasibility of establishing a tool building block library.

Avenues for Further Exploration

Experience with the editor-syntax checker as a tool in its own right has led to a number of very interesting ideas for future work.

When the editor makes a syntactic check, for example, revealing that an "END" statement is missing, it would be a fairly simple matter for it to insert the missing statement itself and merely notify the user that it had done so. Likewise, missing semicolons or other punctuation might be recognizable and correctable. The syntax analysis cannot consistently identify the exact nature of the user's error: a missing "END" might really be a superfluous "BEGIN", for example. Yet, with unfailing notification of all editor-initiated changes permitting the user to override them, such an error correcting editor could prove of some utility.

Beyond error detection and correction, the possibility of a fully prompting editor that does not permit syntactic errors to be entered in the first place arises. Indeed, such an editor already exists: EMILY, at the Argonne National Laboratory [Van Dam]. EMILY offers its users a "menu" of choices at any point in the editing cycle. The user can select a particular language construct (from the PL/I dialect EMILY is written for) such as "statement" or "block," at which point another level of detail is entered and the menu offers choices appropriate to that level (such as the various types of statements syntactically valid at that point).

If an EMILY-like editor could be table-driven from a table of syntax definitions, it could be made quite general-purpose. Such an effort would require a great degree of sophistication within the editor itself, to compensate for the minimal "intelligence" contained in a syntax table. Thus, despite the suggestion, a table-driven editor might in practice be rather complex and slow for an external grammar of any size. One can see quickly, however, that the appropriate degree of generality might more practically be obtained by this project's method of building block integration. In this method, a complex syntax such as for a programming language would not be embodied in a table of syntax definitions, requiring a complex editor to interpret it; rather the syntax interpretation would be embodied in a separate program or subroutine (which might itself be table-driven, of course) designed to analyze syntax, to prompt for particular syntactic entries, and to do little else. This type of program should be only moderately complex and comparatively easy to understand, and its construction correspondingly of moderate complexity. The editor with which it would be joined would also be of only moderate

complexity as demonstrated by the editor written for this project: it would only have to perform the basic editing functions and provide simple "hooks" for the syntax interpretation and prompting program.

This method would also allow different parsing algorithms for different syntaxes, thus optimizing to whatever extent desired the syntax recognition and prompting part of the editor. It would, of course, be somewhat less easy than a table-driven editor to use for the quick creation of a prompting editor for a newly-created syntax, but syntax analysis algorithms today lend themselves to automatic generation by "parser generators" ([Feldman], [Gries], [Johnson], [McKeeman]) and in this fashion, relatively quick creation is possible. It would only be necessary to modify the parser generator algorithm to ensure that its output was a subroutine easily joined with the editor.

The possibilities of a family of syntax recognizing editors for many varieties of programming languages leads to an evolution in one's thinking about source program creation generally. Although a "menu" as provided by EMILY offers a great deal to the novice in a language, protecting him almost entirely from syntactic errors, it can be a bit cumbersome for the more fluent user. In [Van Dam], for example, it is noted that EMILY's designer felt that the light pen and menu convention was awkward for such users. A better approach was suggested where a terminal keyboard is specially created or modified for the programming language in question. On such a terminal--as indeed is already seen on the IBM 5100 BASIC/APL terminal/computer--one key would be set aside for each syntactic construct in the language, at perhaps several levels of detail. Thus a user would, when desiring to enter a block of ALGOL code for example, push the "block" key, followed by keys for the appropriate block entries (e.g., "if-statement," "then-clause," "else-clause,"). The only actual typing would be for identifier names and expressions.

The tremendous potential of this scheme is not only the ease and speed of program entry and concomitant lack of syntactic errors (as EMILY apparently provides now), but also the possibility of an immediate translation of the source code into an internal format such as Polish notation or quadruples for subsequent entry directly to an interpreter or code generating program. The "if" key, for example, would transmit the internal code for "if" and the syntax analyzer's major (though non-trivial) function would be to construct and order the internal format reflecting the code necessary to accomplish an "if" statement. Thus, much of the time-consuming steps of compilation--namely the source program scan, syntactic analysis and internal format generation

(for two or more pass compilers)--would either be greatly reduced or absorbed relatively unnoticed at the time of program entry to the editor. The compiler itself would be left largely with the task of code generation.

4. REFERENCES

Balzer, Robert M., "A Language-independent Programmer's Interface," AFIPS Conference Proceedings, vol. 43 (1974), pp. 365-370.

Barron, D. W. and I. R. Jackson, "The Evolution of Job Control Languages," Software--Practice and Experience, vol. 2 (April-June 1972), pp. 144-163.

Bratman, Harvey and Terry Court, "The Software Factory," Computer, May 1975, pp. 28-37.

Brinch Hansen, P., "The Programming Language Concurrent Pascal," IEEE Transactions on Software Engineering, vol. 1, no. 2 (June 1975).

Brooks, Frederick P., Jr., The Mythical Man-Month, Addison-Wesley Publishing Company, Reading, Mass., 1975, p. 128.

Carlson, William, "The National Software Works," presentation at the Federal ADP Users' Group meeting, General Services Administration, Washington, D.C., January 21, 1976.

Corbato, F. J., "PL/I as a Tool for Systems Programming," Datamation, vol. 15, no. 5 (May 1969), pp. 68-76.

Dahl, O.-J., and C.A.R. Hoare, "Hierarchical Program Structures," in Dahl, Dijkstra, and Hoare, Structured Programming, Academic Press, New York, 1972.

Deutsch, Donald R., "Appraisal of Federal Government COBOL Standards and Software Management: Survey Results," NBS Internal Report 76-1100, June 1976.

Donzeau-Gouge, V., G. Huet, G. Kahn, B. Lang, and J.J. Levy, "A Structure Oriented Program Editor: A First Step Towards Computer Assisted Programming," Rapport de Recherche no. 114, Laboratoire de Recherche en Informatique et Automatique, Institut de Recherche d'Informatique et d'Automatique, Domaine de Voluceau -- Rocquencourt, 78150 Le Chesnay (France).

Feldman, Jerome, and David Gries, "Translator Writing Systems," Communications of the ACM, vol. 11, no. 2 (Feb 1968), pp. 81 ff.

Gries, David, Compiler Construction for Digital Computers, John Wiley & Sons, Inc., New York, 1971, pp. 436 ff.

Grosse-Lindemann, C. O., and H. H. Nagel, "Postlude to a PASCAL-Compiler Bootstrap on a DECSystem-10," Software--Practice and Experience, vol. 6 (Jan-Mar 1976), p. 38.

Johnson, Stephen C., "YACC--Yet Another Compiler-Compiler," Documents For Use with the UNIX Time-Sharing System, Sixth Edition, Bell Telephone Laboratories, Murray Hill, New Jersey.

McKeeman, William M., James J. Horning, and David B. Wortman, A Compiler Generator, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1970, pp. 117 ff.

Parnas72a: Parnas, D. L., "A Technique for Software Module Specification with Examples," Communications of the ACM, vol. 15, no. 5 (May 1972), pp. 330-336.

Parnas72b: Parnas, D. L., "Some Conclusions from an Experiment in Software Engineering," Proceedings of the 1972 FJCC.

Parnas72c: Parnas, D. L., "On the Criteria to be used in Decomposing Systems into Modules," Communications of the ACM, vol. 15, no. 12 (Dec. 1972), pp. 1053-58.

Reifer, Donald J., "Automated Aids for Reliable Software," Proceedings of the International Conference on Reliable Software, SIGPLAN Notices, vol. 10, no. 6 (June 1975), pp. 131-140.

Ritchie, Dennis M., and Ken Thompson, "The UNIX Time-Sharing System," Communications of the ACM, vol. 17, no. 7 (July 1974), pp. 365-375.

Scowen, R. S., "Babel and SOAP: Applications of Extensible Compilers," Software--Practice and Experience, vol. 3 (Jan-Mar 1973), pp. 15-27.

Tennent, R. D., "PASQUAL: A Proposed Generalization of PASCAL," Technical Report no. 75-32, February 1975, Department of Computing and Information Science, Queens University, Kingston, Ontario.

Van Dam, Andries and David E. Rice, "On-line Text Editing: A Survey," ACM Computing Surveys, vol. 3, no. 3 (Sept. 1971), pp. 103-105.

Wichmann, B. A., "A Syntax Checker for ALGOL 60," NPL Report NAC 53, August 1974, Division of Numerical Analysis and Computing, National Physical Laboratory, Teddington, Middlesex, England.

Wulf71: Wulf, W. A., D. B. Russell and A. N. Habermann, "BLISS: A Language for Systems Programming," Communications of the ACM, vol. 14, no. 12 (Dec. 1971), pp. 780-790, esp. p. 786.

Wulf73: Wulf, W. A., "ALPHARD: Toward a Language to Support Structured Programs," Carnegie-Mellon University, 1973.

APPENDIX A: SOFTWARE TOOLS LABORATORY

Work with the editor-syntax checker combination has led to growing recognition of the necessity of further, more detailed investigation of software tools and their construction generally. To meet this recognized need, ICST is studying the implementation of a "software tools laboratory" within its Experimental Computer Facility.

Although plans for the laboratory are only now being made (November 1976), a statement of policy has been written; this statement follows.

Software Tools Laboratory

Policy Statement

In keeping with the NBS mission of advancing science and technology and promoting their effective application, and the Computer Science Section's objectives and functions of developing advanced methods, automated aids and standards for improving the management of software, the Software Tools Laboratory is established

> to develop and disseminate tools, and techniques for the effective use of tools, for software development.

This statement of purpose includes, among other things:

* creating new tools and techniques as well as refining old ones;
* formalizing techniques for combining tools into useful aggregates;
* developing measures of effectiveness for tools;
* developing techniques for the management of small team programming projects; and
* identifying, if possible, a minimal set of tools essential to the development of high-quality software.

The STL is NOT

* primarily concerned with collecting or evaluating existing tools, although such evaluations may play a part in focussing research efforts;
* in competition with private sector software suppliers of software tools;
* a standards-setting activity, although research and development done in the lab may well support subsequent standardization efforts.

Definitions

Product -- a tool or technique developed in the STL.

Release (of a product) -- formal distribution or announcement outside of ICST of a tool or technique developed within the STL (informal distribution for testing and feedback does not constitute a formal "release").

Technique -- a technical method of accomplishing a desired aim, in this case the efficient production of high-quality software and its accompanying documentation. Techniques can encompass programming as well as the management of programming.

Tool -- a computer program that assists a programmer in the process of designing and developing software or documentation. Typical tools include analyzers, editors, pre-compilers, problem statement languages, compilers, debuggers, and document processors.

Rationale

Many software tools are presently available and being developed in the commercial environment. Yet there is no clear body of techniques for making effective use of such tools, nor, more importantly, are really high quality tools available for the mundane aspects of computer programming, such as program text entry and editing, debugging, and static analysis. Those tools that are available are typically written by hardware vendors in machine dependent code, wildly non-standard across hardware types, of low utility relative to their potential because of poor human factors design and inflexibility, and so disparate in control language and implementation as to render impossible their effective use in concert. Further, tools produced by independent software suppliers are commonly designed to be self-contained, and are consequently rather large and inflexible, such as elaborate compilers, library maintenance systems, and data base systems. Large software tools, or packages, like these also command higher selling and maintenance prices than smaller ones.

There appears at the present time, then, to be little motivation for the private sector to develop small software tools that can be used "as is," or easily joined with others to form larger, special-purpose tools. The NBS can thus step into this area, certainly with an eye toward research and experimentation, without undue fear of competition with industry, and with a reasonable expectation of providing significant benefit to the federal government.

Policies

As primarily a research effort, the STL will not be managed for the unique benefit of anyone. Rather, products resulting from the lab will be in the public domain and will be publicized and disseminated as widely as their value justifies. In general, research will be geared to helping individual programmers, and to a lesser extent, to helping those who use the service of programmers.

Research will concentrate on small tools that are used by individuals and small project teams, such as editors, keyword extractors, and compilers, rather than operating systems, data base systems, or the like. This concentration is a reflection of both the existing active participation of industry in the latter area, and the Bureau's limited resources and consequent inability to realistically develop or test tools and techniques for large-scale use.

Products of the lab will be suitable for use on a variety of computer hardware and software systems.

As the Computer Science Section has a charge to "emphasize advanced techniques... undertake state-of-the-art studies... [and] structured programming experimentation," programming languages used in the lab will not be confined to those customarily used by the majority of federal government programmers. Nevertheless, techniques developed will be as broadly applicable as possible across programming languages.

Any programming done will be done in accord with currently known and recommended principles of good style and sound engineering.

Unless considerably greater funding, resources and management direction are applied, the STL will not attempt rigorously (i.e., scientifically or statistically) to evaluate the benefits of its products. Rather, informal, reasonable judgements will be made.

Research will be directed primarily toward improving the process of software development and its documentation, where such processes can be distinguished from software maintenance, testing and optimization.

Inasmuch as a large proportion of the work of software development is concerned with text manipulation (program texts, documentation, plans, schedules, etc.), tools and techniques relevant to such processing shall be considered within the domain of the STL.

Program products will be written according to the dictates of (1) clarity and readability, (2) transferability, and (3) efficiency, in that order of priority.

Products will be selected for development according to (1) breadth of applicability, (2) level of benefit, and (3) ease of applicability, in that order of priority. Note that (1) and (2) mean that--all things being equal--a product of broad applicability and modest significance is prefered over one of greater significance but limited applicability.

Procedures

No product of the STL will be released without (1) at least two individuals not directly connected with the lab having used the product--not just reviewed it--and reported favorably on such use; (2) the product's having been successfully used with at least two different hardware/software configurations, preferably of different manufacture; and (3) the product's being uniformly and clearly documented. Documentation will include an easy means for product users to provide comments back to the lab. Program products must additionally pass a "code inspection" by at least two people looking for clarity, good structure, robustness and self-evidentness of technique, and reasonable isolation of machine dependencies.

Use of the NBS Univac-1108 and other NBS computers for product tryouts is encouraged as a reasonable first step after use in the ECF.

STL products will illustrate their author's cognizance of current work outside NBS in the area of software tools and development.

APPENDIX B: SOFTWARE TOOLS BIBLIOGRAPHY

This bibliography includes literature on software tools and techniques used during the software development cycle. Sources surveyed included journals and conference proceedings as follows.

Journals

ACM Computing Surveys	1971-1975
Communications of the ACM	1971-1975
Computer	1975
Computer Decisions	1971-1975
DATAMATION	1970-1975
EDP Performance Review	1973-1975
Software--Practice and Experience	1971-1975

Conference Proceedings

First National Conference on Software Engineering, Washington, D.C., September 11-12, 1975.

Third Texas Conference on Computing Systems, Austin, Texas, Nov. 7-8, 1974.

Fourth Texas Conference on Computing Systems, Austin, Texas, November 17-18, 1975.

1973 IEEE Symposium on Computer Software Reliability, New York City, April 30- May 2, 1973.

1975 International Conference on Reliable Software, Los Angeles, April 21-23, 1975.

Workshop on Currently Available Program Testing Tools -- Technology and Experience, Los Angeles, April 24-25, 1975.

Courant Computer Science Symposium 1, "Debugging Techniques in Large Systems," New York City, June 29 - July 1, 1970.

Computer Program Test Methods Symposium, University of North Carolina, Chapel Hill, June 21-23, 1972.

National Computer Conference, Chicago, May 6-10, 1974.

National Computer Conference, Anaheim, May 19-22, 1975.

ACM 1974 Annual Conference, San Diego, November 11-13, 1974.

ACM 1975 Annual Conference, Minneapolis, October 21-23, 1975.

Follow-up of references cited in the above sources led to the acquisition of technical reports published in other sources.

Criteria For Inclusion

Included in this bibliography are papers on software tools or techniques, and applications of software tools or techniques. Where more than one paper by the same author was found on the same subject, the most recent was selected. Not included in this bibliography are papers on (1) purely theoretical subjects, e.g., proof of correctness; (2) system software performance and measurement tools, e.g., operating system software monitors; (3) system software tools, e.g., compilers; (4) debugging or testing methods for application software, e.g., numerical software error analysis methods.

Bibliography

Baird, R., "APET--A Versatile Tool for Estimating Computer Application Performance," Software--Practice and Experience, October-December 1973, vol. 3, no. 4, pp. 385-395.

The Application Program Evaluator Tool, APET, is an application of a general concept: prototyping functional models of a project to be undertaken, and then using the prototypes to answer performance questions. APET is a functional modelling method that produces synthetic jobs, which can be used to control computer system activity to measure selected hardware or software functions as they interact within an application under real operating conditions. APET is controlled by a language that provides the means for synthesizing a real job, thereby producing synthetic benchmarks of a general class of problems. A synthetic program produced by this tool has measurement facilities built into it in the form of timing routines, software or hardware hooks, event traces, etc.

Basili, V. R. and M. V. Zelkowitz, "Compiler Generated Programming Tools," Workshop on Currently Available Testing Tools, April 1975, p. 45.

Two compilers, SIMPL and PLUM, are implemented with data collection aids that provide compilation, execution and post- execution statistics. Compilation time data give an insight into syntactical bugs--and help the programmer debug his program more efficiently; and execution-time and post-execution time data permit analysis of program efficiency and correctness.

Bergeron, R. Daniel and Henri R. Bulterman, "A Technique for Evaluation of User Systems on an IBM S/370," Software--Practice and Experience, January-March 1975, vol. 5, no. 1, pp. 83-92.

The most frequently used techniques for identifying critical regions of a program are described. These techniques are: Gallup, in which the programmer must be polled for the critical region in his program; Spy, whereby an independent subroutine interrupts the execution of a program to determine in which of the user-defined areas the program was executing;

machine-language interpretation, which interprets each instruction in the user system; and the "AED" technique, which uses a runtime statistics-gathering package for user systems. The System for System Development (SSD), which uses the "AED" technique, is described. SSD consists of a special compiler for the systems programming language LSD, augmented to provide various systems-oriented facilities, including an evaluation facility. The SSD intercepts user subroutine calls, compiles and analyzes the information on the subroutines, and returns without affecting user processing.

Blair, Jim, "Extendable Non-Interactive Debugging," Courant Computer Science Symposium 1, (June 29- July 1, 1970), in Debugging Techniques in Large Systems, edited by Randall Rustin, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971, pp. 93-115.

The Purdue Extendable Debugging System (PEBUG) is a general purpose and flexible debugging tool, designed for use in either an interactive or non-interactive environment. Another design criterion was that this debugging system should be extendable by the user in terms of his source language, so that he can add his own debugging aids without complicated system interfaces. In keeping with this requirement, a basic system was produced from primitives, and these primitives built up in levels; thus the final system has a system of primitives on which other debugging aids can be built. PEBUG structure on top of the primitives has three basic components: the breakpoint interpreter, which controls the dynamic execution of the program being debugged; the command scanner, which is actually the program that controls the commands entered as input; and the group of debugging subroutines, which accomplish the actual debug processing.

Boehm, B. W.,R. R. McClean, and D. B. Urfrig, "Some Experience with Automated Aids to the Design of Large Scale Reliable Software," Proc. International Conference on Reliable Software, April 1975, pp. 105-113.

Recent experiences in analyzing and eliminating sources of error in the design phase of a large software development project are summarized. A taxonomy of software error causes and an analysis of the design-error data are presented. Investigations into the cost- effectiveness of using automated aids to detect inconsistencies between assertion of the

nature of inputs and outputs of the various elements of the software design have led to the development of a prototype version of such an automated aid, the Design Assertion Consistency Checker (DACC). Results of an experiment using the DACC on a large scale software project show that there is value to such a facility, although cost considerations should be weighed before using (or developing) such a tool.

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

SELECT is an experimental system for assisting in the formal debugging of programs, by systematically handling the paths of programs written. For each execution path SELECT returns simplified conditions on input variables that cause a particular path to be executed; and it returns simplified symbolic values for program variables at the path output. Potentially useful input test data that cause a particular path to be executed are generated; and if such data cannot be generated, then the particular path is no longer pursued. This experimental system allows the user to supply interactively executable assertions, constraint conditions or specifications for the intent of the program from which it is possible to verify the correctness of the paths of the program.

Bratman, Harvey and Terry Court, "The Software Factory," Computer, May 1975, pp. 28-37.

The Software Factory concept is an approach to software development emphasizing a disciplined methodology that produces reliable software; that utilizes a flexible facility with a set of tools tailored to the software development process; and that allows incorporation of new tools to ease the process of development. The basic components are FACE, the control and status gathering service, IMPACT, the scheduling and resources computation facility, and the Project Development Data Base. Some of the tools available to the software projects are AUTODOC (documentation tool), PATH (program analyzing tool), TCG (a test case generating tool) and TOPS (a design verification tool).

Brown, A.R and W. A. Sampson, Program Debugging, American Elsevier, New York, 1973.

The nature of programming errors and debugging, the different types of debugging, and the debugging aids presently available to a programmer are explained. The authors' own technique, the METHOD, adapted from a management principle, is then introduced. This debugging technique has as its basis the location and definition of a "deviation" from the expected, which forces the programmer to consider on the causes of the deviation, and

Brown, J.R., "Practical Applications of Automated Software Tools," TRW Report no. TRW-SS-72-05, September 1972.

The measurement of "thoroughness" in testing is illustrated with a small program, using automated aids that analyze the code, instrument it to produce the necessary output, and monitor testing. The tools discussed are FLOW, a program execution monitor, and PACE a set of automated tools that support the planning, execution, and analysis of computer program testing. The tools described help in measuring the thoroughness of testing by analyzing all possible paths, and eliminating those that contain mutually exclusive conditions; and by identifying the remaining paths in relation to their use of statement and branches.

Burlakoff, Mike, "Software Design and Verification System," Workshop on Currently Available Program Testing Tools, April 1975, p. 19.

The Software Design and Verification System, SDVS, is an integrated set of software tools intended to reduce the efforts needed in the design, development and verification of software. The software system is partitioned into a number of simulated computer processors, and it uses a set of software modules for its execution in a simulated environment under SDVS control. The design and verification conditions are specified in a Test Case File, and using the SDVS, the software system is exercised according to specifications.

Carpenter, Loren C. and Leonard L. Tripp, "Software Design Validation Tool," Proc. International Conference on Reliable Software, April 1975, pp. 395-400.

Design Expression and Confirmation Aid, DECA, is a set of programs used in conjunction with a top-down dominated design methodology. A design expression consists of a static structure (the design tree), a dynamic structure (the transition diagram), and the relationship between them (the data parcel). Using the transition diagrams, the design can be validated by "walking through" the system's action. DECA consists of five sequentially executed subprocesses: a syntax scanner, a text ordering sort, a consistency-checking document printer, an information-ordering sort, and a global checker. The use of DECA enhances the design process of the software system, and in turn its development, because of the thorough, top-down validation of the design prior to the actual coding of the program.

Chanon, R.N., "An Application of a Specification Technique," Proc. Third Texas Conference on Computing Systems, November 1974, pp. 10.1.1-10.1.2.

The experience resulting from applying the Parnas specification technique [see Parnas, D. L., "A Technique for Software Module Specification with Examples," Communications of the ACM, vol. 15, no. 5 (May 1972), pp. 330-336] in the construction of a semantic analyzer for a compiler construction course is described. The specification technique consists of (1) a description of the possible values of the argument to each function, (2) a description of the possible values of the result, and (3) exactly enough information to completely determine the results, given the arguments to a function and the previous results of the function comprising the module.

Crocker, Steve and Bob Balzer, "The National Software Works: A New Distribution System for Software Development Tools," Workshop on Currently Available Program Testing Tools, April 1975, p. 21.

The National Software Works is a centralized clearinghouse residing on the ARPANET. The NSW plans to increase the availability of software tools to the vastly different types of user on a distributed network environment; it intends to incorporate major types of computers into the Works, and allow the tools to be

used in dissimilar computers; it plans to develop specifications for interfacing new systems; etc.

de Balbine, Guy, "Tools for Modern FORTRAN Programming," Workshop on Currently Available Program Testing Tools, April 1975, p. 27.

Three general development tools purported to aid in the writing of a program are discussed: PDL--a program design language and processor, which allows the writing of a description of what is to be done in simple English; S-FORTRAN--the structured FORTRAN language and processor, which converts the design language into executable code; and the "structuring engine"--a program reformatter for existing FORTRAN programs.

DeVito, A. R., "The PRO/TEST Library of Testing Software," Workshop on Currently Available Program Testing Tools, April 1975, p. 31.

The PRO/TEST library of testing software consists of three modules: The DATA GENERATOR, which generates test files from parameter statements; the FILE PROCESSOR, which is used on the files generated; and the FILE CHECKER, which automatically verifies computer output. This library of testing software is concerned both with obtaining a sufficient volume of comprehensive test data, and with verifying the completeness and correctness of test results.

Fairley, R. E., "An Experimental Program Testing Facility," Proc. First National Conference on Software Engineering, September 1975, pp. 47-52.

The Interactive Semantic Modelling System, ISMS, is an experimental program testing facility that allows experimentation with implementation of a variety of information collection, analysis and display tools. ISMS addresses the following questions with respect to testing: (1) What information is useful? (2) How can the information be collected? and (3) How can the information be analyzed and displayed in a meaningful format? In this context, the design characteristics of the ISMS preprocessor and those of some of the tools being developed with ISMS are described. Salient among the design features are: the syntax driven nature of the preprocessor; the

isolation of the data collection from the data analysis and display processes; and the independence of the collection, display and analysis routines from the internal details of the data base implementation.

Ferrari, Domenico and Mark Liu, "A General Purpose Software Measurement Tool," Software--Practice and Experience, April-June 1975, vol. 5, no. 2, pp. 181-192.

The general-purpose Software Measurement Tool, SMT, is intended for use in the interactive system, PRIME. The SMT is designed with flexibility, and generality in mind. It allows the user to instrument a program, modify pre-existing instrumentation, and specify how the data are to be reduced with a few commands or with a user-written measurement routine. There are three phases to the measurement process of the SMT: the instrumentation phase, the execution phase, and the data reduction phase. At present, only a prototype of the SMT has been implemented.

Fragola, J. R., and J. F. Spahn, "The Software Error Effects Analysis: A Qualitative Design Tool," Proc. 1973 IEEE Symposium on Computer Software Reliability, May 1973, pp. 90-93.

A qualitative method of evaluating a software package, called Software Error Effects Analysis (SEEA), provides systematic, consistent, objective and permanent analysis of the software. The technique requires the definition of the module interdependencies, and the analysis of the effect of error in the data flowing between them, thus yielding a comprehensive picture of how the program can fail and where. It identifies the weak points in the program and suggests where redundancy should be added.

Glassman, B. A. and J. W. Thomas, "Automating Software Development--A Survey of Automated Aids in Support of the MDAC-W Computer Program Management Technique (CPMT)," McDonnell Douglas Report Number MDC G5707, January 1975.

To relieve developers of some of the menial and error-prone tasks in writing software, some automated aids are developed. The tools described include CCP (Configuration Control Program): automates

software configuration and management procedures; MACFLO: constructs flowcharts depicting the logic flow of computer software; JOYCE: constructs glossaries, cross reference maps, and trees defining the program structures and variable usage; PET (Program Evaluator and Tester): gathers information on source statements and inserts code into the source deck to obtain run-time statistics; REPROMIS: aids in documentation; TRAKIT: aids in project control; and FORSEQ and TIDY: accomplish general housekeeping, such as sequence numbering and indentation.

Goetz, Martin, "Soup Up Your Programming with COBOL Aids," Computer Decisions, March 1973, vol. 5, pp. 8-12.

Because of the large number of COBOL users in the computing community, there is a large group of software tools available to the COBOL programmer during the software development cycle. These "COBOL-aids" are most abundant during the coding, testing and debugging, and maintenance phases of program development. A brief discussion of each of these tools, with examples, is presented.

Graham, Robert M, Gerald J. Clancy and David B. DeVaney, "A Software Design and Evaluation System," Communications of the ACM, vol. 16, no. 2 (February 1973), pp. 110-16.

The Design and Evaluation System, DES, is a system that addresses the problem of evaluating the performance of a proposed design before it is implemented. DES has a single hierarchical data base that contains hardware and software information on the object system. DES provides performance information at each of the following stages in the design and implementation of a software system: component, subsystem, and total system. Component evaluation is concerned with the performance and resource usage of individual procedures--mostly static information such as the total number of instructions in the procedure. This stage also produces a simulation model which becomes input to the subsystem and system evaluation stages. Subsystem evaluation builds a composite model of the subsystem, and then it is subjected to the same evaluation as in the components evaluation stage. And finally, the system evaluation uses simulation to determine the dynamic behavior of the system. At each of the above stages, the evaluation results produced by DES are used to validate the performance of a design before it is implemented.

Green, Eleanor, "What, How, and When to Test," Workshop on Currently Available Program Testing Tools, April 1975, pp. 33-34.

During the four major phases of the software life cycle-- design, implementation, verification and maintenance--the questions of what to test for, how to devise adequate tests and when to stop testing should be taken into consideration. Among the automated aids that help in the testing processes are AUTOFLOW II and ROSCOE which help in the testing at the specification level, and MetaCOBOL and LIBRARIAN, which help in testing at the program level. The author suggests that testing should be thought out from the design phase; that testing should be well-designed and where possible use automated aids; and that testing should be an integral part of the life cycle of software.

Grishman, Ralph, "Criteria for a Debugging Language," Courant Computer Science Symposium I (June 29-July 1, 1970) Debugging Techniques in Large Systems, edited by Randall Rustin, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971, pp. 57-75.

Current debugging systems are discussed in terms of the features that a good debugging system language ought to have. In particular, special attention is focused on interactive debugging systems. The author's system, AIDS, is by the author's account a good system. It contains: (1) ON and WHEN statements specifying the occurrences that trigger the initiation of debugging actions; (2) a debugging procedure following an ON or WHEN condition specifying what actions are to be performed; (3) statements to invoke facilities in the debugging system not available in the source language; and (4) a debugging language similar to the source language. AIDS is intended for use with both FORTRAN and assembly language programs; thus an "object code" system was judged most advantageous. A source program is submitted to a FORTRAN compiler or an assembler, which then generates an object program and a listing. AIDS first reads the listing, extracting the attributes of all symbols; it then loads the object program, and asks the user what he would like to do.

Griswold, Ralph E., "A Portable Diagnostic Facility for SNOBOL4," Software--Practice and Experience, January-March 1975, vol. 5, no. 1, pp. 93-104.

In programming systems based on abstract machine-modelling concepts, the underlying structure of the abstract machine can be made available to the software implemented on it. The result is a facility for diagnosis and exploration of software structure. One such facility is " The Window to Hell" (TWTH). TWTH consists of built-in SNOBOL4 functions, operators and keywords, that provide a vertical extension between the SNOBOL4 "externals" and the SIL (SNOBOL4 Implementation Language) "internals". Therefore, SIL level structures are made accessible to a SNOBOL4 program, and the powers of the higher level language are available for analyzing and modifying its own internals.

Howard, Phillip, editor, "Simulation: Its Place in Performance Analysis," EDP Performance Review, vol. 1, no. 11, November 1973.

Simulation as an analysis and evaluation tool is discussed from the users' point of view, with concentration on "commercially"- available simulation packages, e.g., "general purpose" system evaluation tools, such as SCERT, SAM, CASE, which are used as "system" simulators, and are often thought of as a "future planning " tool.

Howard, Phillip C., editor, "Third Annual Survey of Performance-Related Software Packages," EDP Performance Review, vol. 3, no. 12, December 1975.

This series of surveys on commercially available (proprietary) software packages is a good source of information for the software development projects, because it identifies by name, and classifies software tools that help in the measurement, evaluation, and improvement in the quality of computer software, or help in the improvement in the productivity of a computer installation. Although this is purported to include only those software packages that have as their primary function that of enhancing or evaluating the performance of a computer system, it also includes communications tools, precompilers, data management etc, as well as the traditional performance-type tools, like monitors, simulators, optimizers, etc.

Howard, Phillip C., editor, "Bibliography of 1974 Performance Literature," EDP Performance Review, vol. 3, no. 3, March 1975.

This bibliography is organized by subject; and it contains an author's index, a publisher's address list and a subject index. The subjects covered include job accounting, compilers, simulation tools and techniques, and is exhaustive for performance related topics.

Howden, William E., "Methodology for the Generation of Program Test Data," IEEE Transactions on Computers, May 1975, vol. c-24, no. 5, pp. 554-559.

The program test-data generation methodology described decomposes a program into a finite set of classes of paths, in such a way that an intuitively complete set of test cases would cause the execution of one path in each class. This is known as the "boundary-interior" method for choosing test paths. There are five phases in this methodology: (1) the analysis, classification and construction of program-like descriptions of classes of program paths, (2) the construction of the description of input data which cause the different standard classes of paths to be followed, (3) the transformation of implicit description into equivalent explicit description, (4) the construction of explicit description of subsets of the input data set not accomplished by the previous phase, and (5) the generation of input values satisfying the explicit descriptions. The first four phases of this methodology could be useful for a partial program-correctness system. The last phase is the actual test-data generating process for certain classes of programs that must be thoroughly tested.

Howden, William F., "Automated Program Validation Analysis," Proc. Fourth Texas Conference on Computing Systems, November 1975, pp. 4A2.1-4A2.6.

DISSECT is a system in which the user has flexible control over the application of the types of validation analysis carried out automatically by the computer. This analysis can be used to generate test data, prove correctness or check a program against its specifications. DISSECT has the ability to attach attributes to paths and to direct the system to choose paths conditionally based on a path's attributes. The author presents an evaluation of

DISSECT against other systems like it--e.g., SELECT, EFFIGY, RXVP--all of which use the "symbolic evaluation" method for forming the systems of predicates that describe the data causing a path or set of paths to be followed.

Ignalls, Daniel H.H., "FETE--A FORTRAN Execution Time Estimator," Stanford University Report no. STAN-CS-71-204, Computer Science Department, Stanford University, February 1971.

FETE is an automated system that inserts counters automatically into a program during execution. FETE is a three step process: first, it accepts any FORTRAN program and produces an edited file with counters; second, it executes the modified program, while saving the original source program; and third, it re-reads the modified source program, and correlates it with the final counter values in such a way that the executable statements appear with the exact number of executions and approximate computation time. Next to the logical IF's, FETE shows the number of TRUE branches taken, and computes the "cost" of execution, based on a linear scan cost-algorithm. Basically, the tallying counters are inserted when certain control structures are found, and the time estimates are calculated when the modified source program is executed with all the tallying counters.

Ikezawa, M.A., "AMPIC," Workshop on Currently Available Program Testing Tools, April 1975, p. 7.

AMPIC is a tool under development that is claimed to represent a program as a structured program, regardless of how it was originally programmed. Its capabilities include: an assembly language flowchart generator, assembly-source-to-higher functional language translator, semi-automatic path analysis, and details of deductions by which the translations are produced.

Isaacson, Portia, "PS: A Tool for Building Picture-System Models of Computer Systems," Proc. Third Texas Conference on Computer Systems, November 1974, pp. 3.4.1-3.4.5.

PS is a tool for designing computer systems by using picture systems as models. A picture-system model consists of a "picture set" containing a picture for each state of the computer system relevant to the mechanism being modeled. The goal of this research is to automate the production and analysis of picture system models so that these models can be used on a broader basis as a means of communicating computer systems mechanisms at various levels of abstraction.

Itoh, Daiju and Takao Izutani, "FADEBUG-I, A New Tool for Program Debugging," Proc. 1973 IEEE Symposium on Computer Software Reliability, April 1973, pp. 38-43.

FACOM Automatic DEBUG (FADEBUG-I) is a debugging tool for assembly language that is to be used at the early stage of module testing. This debugging tool has two main functions: it checks out automatically the possible execution paths of a program; and it compares the contents of main memory with the desired data after the program has been running. Experimental data using FADEBUG-I are evaluated and results show that with this automatic debugging tool, program performance is improved.

Kerninghan, B. and P.J. Plauger, "Software Tools," Proc. First National Conference on Software Engineering, September 1975, pp. 8-13.

A tool-building concept is introduced whereby programs can be conceived as special cases of more general purpose tools. The authors show that programs can be packaged as tools, and proceed to describe an ideal environment for tool-building and tool-using. This environment is UNIX, where it is possible to have building blocks, or "filters" with the capability for handling input and output redirection, and a mechanism for hidden buffering between output and input, called "pipes." One of the advantages offered by this "filter and pipe" concept is that once a filter is created it can be used as a building block to build other more complex tools using pipes to join them together, so that eventually it will become unnecessary to re-create a tool each time one is needed; rather, one should be able to piece together different filters to perform the desired function.

King, J.C., "A New Approach to Program Testing," Proc. International Conference on Reliable Software, April 1975, pp. 228-233.

Rather than testing a program by choosing an appropriate example, or data, the author proposes that symbols be supplied to the program being tested. He argues that the normal computational definitions for the basic operations performed by a program can be expanded to accept symbolic inputs and produce symbolic formulae as output. Specifically, the interactive debugging/testing system called EFFIGY is discussed. This system was implemented with the principles of symbolic execution in mind.

King, J. C. "Symbolic Execution and Program Testing," Communications of the ACM, July 1976, pp. 385-394.

The principles of symbolic execution are discussed, in an ideal sense, as an alternative to specific testcase-testing. In particular, a general purpose interactive debugging and testing system, the Interactive Symbolic Executor--EFFIGY--is described. EFFIGY provides debugging and testing facilities for symbolic program execution, with normal program execution as a special case. Additional features of this Symbolic Executor include an "exhaustive" test manager, and a program verifier.

Kirchoff, M.K. and R.H. Ryan, "The Need to Salvage Test Tool Technology," Workshop on Currently Available Program Testing Tools, April 1975, p. 3.

The usual procedure with test tools is that of discarding them after they have performed the function they were designed to do, and re-inventing the same tools (together with the mistakes from the previous generation) when the need arises once again. This paper advocates the salvaging of the test- tool technology in order to avoid the "re-invention of the wheel" each time the same tool is needed. To this purpose, the creation of a "test tool specification library" is suggested. This library would include a specification for each tool, stating requirements for that tool's development so that eventually a specification language can be developed. To test the feasibility of this idea, it is proposed that a modest set of tools be specified in this manner; and the success or failure of this kind of experiment should be reported in conferences.

Kulsrud, H.E., "Extending the Interactive Debugging System HELPER," Courant Computer Science Symposium 1, (June 29 - July 1, 1970), in Debugging Techniques in Large Systems, edited by Randall Rustin, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1971, pp. 78-91.

HELPER is an interactive extensible debugging system used for debugging programs that have been compiled previously. HELPER is highly modular and treats many normal systems functions as user programs. It is directed to the problem of debugging at the machine language level. It analyzes the program by simulating the instructions; the user communicates with the system by means of a simple algorithmic command language, and the arguments of this language are the symbols of the user's own source program. HELPER consists of five main elements: a simulator, a compiler, a communicator, a controller, and a set of debugging routines. Of particular interest are the compiler and the set of debugging routines, because these are what provide the extensible capability of the system. The compiler is used to translate the commands given by the user, and the debugging programs are those routines that are accessed when certain switches are set by the simulator and user commands. To introduce changes in the command language, only the new syntax for the compiler is fed to the metacompiler, and the resultant code replaces old code or is added to the compiler. To introduce a new debugging command, the separate debugging routine(s) is added to the system. The flexibility that is obtained with the extensibility of this debugging system allows for operational upgrading of the system software.

Lemoine, M. and J. Y-Rousselot, "A Tool for Debugging FORTRAN Programs," Workshop on Currently Available Program Testing Tools, April 1975, p. 48.

A debugging tool, implemented on a CII IRIS 80, is designed to help the FORTRAN programmer in locating potential sources of error, i.e., static function (before the run), and monitor the execution of the program, i.e., dynamic function (during the run). Basically, it divides the FORTRAN program into "blocks" and then, reconstructs the segmented program as a directed graph. The tool is divided into two parts: (1) manipulation of directed graphs (flowcharts), and (2) manipulation of program semantics and the associated graphs (debug).

Lite, S., "Using a System Generator," Datamation, June 1975, pp. 44-47.

GENASYS is a system generator that exploits the very high degree of commonality that exists among commercial applications. The system has a core library of 300 general system definition macros, which are used to automate the production of code once processing and output specifications are defined. The user supplies the initial macros from a "workbook" of predefined system definition macros, and using the macro-expansion facility of the assembler, these macros are modified by the input parameters, which are then used to produce the final source code and complete documentation.

Lyon, Gordon and Rona B. Stillman, "A FORTRAN Analyzer," NBS Technical Note no. 849, October 1974.

The NBS FORTRAN Analyzer performs both static and dynamic analyses on a program. In the static analysis section frequency statistics on 118 FORTRAN statement types are collected, and accumulated over all programs analyzed. In the dynamic analysis section the execution frequencies of each code segment are monitored, and the flow from segment to segment are recorded. The dynamic analyzer is a two-pass function: in the first pass the source code is instrumented by inserting calls to a tallying function; and in the second pass the instrumented program is executed, causing the program to compute and report execution frequency statistics in addition to performing its normal functions. Specific features of the FORTRAN analyzer are discussed in detail. The report includes a sample run and sample analyses.

Lyon, Gordon and Rona Stillman, "Simple Transforms for Instrumenting FORTRAN Decks," Software--Practice and Experience, October-December 1975, vol. 5, no. 4, pp. 777-888.

The method of instrumenting a FORTRAN program used in this FORTRAN Analyzer consists of inserting calls to a tallying function in the original source program during pass one, and then, in pass two this augmented version of the program is executed, providing frequency- of-execution counts of program segments. During pass one, a static analysis of the source code is performed, by providing a count of statement types. The resulting analysis highlights

unusually heavy usage in certain segments of codes, as well as unexecuted segments. Particular transforms used in this analyzer are explained.

Miller, E. F., "Experience with RXVP in Verification and Validation," Workshop on Currently Available Program Testing Tools, April 1975, p. 26.

RXVP is a commercially available software package that aids in acceptance testing. Included among the functional capabilities of this package are static analysis, dynamic analysis at statement and control-structure level, and test case data generation.

Miller, E. F. and R. A. Melton, "Automated Generation of Testcase Datasets," Proc. International Conference on Reliable Software, April 1975, pp. 51-58.

The use of systematic testing methodology can be a vehicle to insure software quality. One such methodology relates functional test cases to formal software specification as a way to achieve correspondence between software and its specifications. To do this, appropriate test case data generation is required. The automatic generation of test case data, based on a-priori knowledge of two forms of internal structures is discussed: a representation of the tree of subschema automatically identified from within each program text, and a representation of the iteration structure of each subschema.

Pomeroy, J. W., "A Guide to Programming Tools and Techniques," IBM Systems Journal, 1972, vol. 11, no. 3, pp. 234-254.

Different techniques and tools aiding the programmer in the software development process are discussed, and illustrated with descriptions and examples of tools for each category. Tools mentioned in this article are constrained to those that are obtainable through IBM.

Ramamoorthy, C.V. and K. H. Kim, "Software Monitors Aiding Systematic Testing and their Optimal Placement," Proc. First National Conference on Software Engineering, September 1975, pp. 21-26.

Complete validation of large software systems is often economically infeasible; hence partial validation remains the most practical approach, with testing as the most commonly used technique for achieving partial validation. The technique of using software monitors as partial aids to systematic performance of program testing is presented. Typical strategies involving software monitors are: 1) Test-path generation schemes, (2) test-input generation schemes, and (3) test-output evaluation schemes. Two types of monitors are especially well-suited for these techniques: the flow-controlling monitors, and the traversal-markers monitors. The optimal instrumentation of these monitors is analyzed in terms of their placement in the target system.

Ramamoorthy, C. V., R. E. Meeker, Jr and J. Turner, "Design and Construction of an Automated Software Evaluation System," Proc. 1973 IEEE Symposium on Computer Software Reliability, April 1973, pp. 28-37.

The concept of a two-step approach is applied to an organized software validation effort: (1) Analyze the software for well-formation, and eliminate existing anomalies; and (2) Apply testing procedures to check for application specifications. Using the above philosophy of validation, the Automated Code Evaluation System (ACES) is developed. This system is essentially a language processor with capabilities for static language analysis and data generation. ACES performs analysis of program structures, modelled as directed graphs, to allow for detection of structural flaws and examination of critical or interesting flow-paths through the program. Execution monitoring is performed by automatically inserting calls to a monitoring routine. Although "complete" validation of large software systems cannot be performed, systems such as ACES that encourage the systematic analysis of code can play an important role in partial validation.

Ramamoorthy, C. V. and S. F. Ho, "Testing Large Software with Automated Software Evaluation Systems," Proc. International Conference on Reliable Software, April 1975, pp. 382-393.

A general survey and a brief description of software tools are presented as an introduction to the authors' discussion of operational experiences with automated software evaluation systems, such as FACES,

PACE, AIR. A Software Evaluation System is a composite system consisting of various automated tools intended to perform system design analysis, debugging, testing and validation. Automated tools are classified and described as follows: (1) by mode of operation--static and dynamic analysis; (2) by the development phase in which it is applicable--design and analysis, testing and debugging, and maintenance; (3) by the specific function they perform--automated design, simulation, code analysis, run-time behavior monitoring, test generation, documentation, etc. The desirable characteristics of a "good" automated tool are outlined: good resolution power, generality, several levels of abstraction, easy to use, highly automated, well-structured, well-documented and thoroughly tested, and machine-independence to facilitate transferability.

Reifer, D. J., "Interim Report on the Aids Inventory Project," The Aerospace Corporation, Report SAMSO-TR-75-184, 16 July 1975.

The Aids Inventory Project described here serves as a storage and distribution center for software tools (support programs) used in the testing and development of weapon-system software. The Inventory is divided into two parts: The Physical Inventory, which consists of a set of tools that are application-independent, and are used by multiple projects; and the Existing Aids Catalog which contains a list of the programs including their location, limitations, status and capabilities. This report also contains a glossary of aids with their definitions, the Existing Aids Catalog, a glossary of other technical terms used throughout the report, and a bibliography of relevant literature.

Reifer, D. J., "Automated Aids for Reliable Software," The Aerospace Corporation, Report SAMSO-TR-75-183, 26 August 1975.

A guide to automated aids used to increase program productivity by decreasing cost, and increasing software reliability is presented. The aids are divided into the following categories: simulation, development, test and evaluation measurement, and programming support. This report recommends that a systematic basis be established upon which a "core" set of tools is defined, and is used to build a

"tailored" support system for individual projects.

Réifer, Donald J. and Loren P. Meissner, "Structured FORTRAN Preprocessor Survey," Lawrence Berkeley Laboratory, Univ. of California at Berkeley Technical Report UCID-3793, November 1975.

This survey is an inquiry into existing FORTRAN preprocessors and other software packages that are aimed at constructing "structured" control-structures with the FORTRAN language. Developers of these preprocessors were polled with questionnaires, and the results of the inquiry are included in this report.

Reifer, D. and Robert Lee Ettenger, "Test Tools: Are They a Cure-All?" The Aerospace Corporation, Report SAMSO-TR-75-13, 15 October 1974.

Test tools are evaluated in terms of their capabilities, constrained by the requirements that the test tools should be commercially available, and that the developers have the intention of implementing them in JOVIAL. Four tools meeting these two criteria were investigated: QUALIFIER, NODAL, RXVP, PET. The evaluation is based on timing data, obtained using benchmark programs. In conclusion, the theme question is answered negatively, mainly because the test tool technology, as known today, is an evolutionary process, and today's test tool constitutes a "first step" in that direction.

Rizza, John B. and Dennis Hacker, "Quality Assurance Inspection and Test Tools--An Application," Workshop on Currently Available Program Testing Tools, April 1975, pp. 9-10.

The usage of software tools at different stages of software development is contrasted with purely manual methods, and the benefits resulting from the application of software tools are dramatized by the presentation of a method of projecting cost reduction for software projects. Specifically, two examples are given showing the cost-reduction formula at work: the use of software tools to verify compliance with coding standards versus using purely manual methods; and the application of branch-testing standards vs. not using them. The examples illustrate the merits of using software tools by

the project funds saved.

Rochkind, Marc J., "The Source Code Control System," Proc. First National Conference on Software Engineering, September 1975, pp. 37-43.

The Source Code Control System (SCCS) is a software tool designed to help programming projects control the changes made to the source code. It provides facilities for storing, updating and retrieving all versions of modules, for controlling updating privileges, for identifying load modules by version number, and for recording who made each software change, when and where it was made, and why. The key features of the SCCS are: (1) Its storage capability, which allows all versions to be stored together in the same file; (2) Its protection capability, which controls updating, and access privileges; (3) Its identification capabilities, which automatically insert date, time and version numbers to the programs; and (4) Its documentation capabilities, which record who made the changes, what they were, when, where, and why they were made. Presently, the SCCS resides in a facility known as the "Programmer's Workbench," under the UNIX operating system.

Ryder, B. G., "The PFORT Verifier," Software--Practice and Experience, October-December 1974, vol. 4, no. 4., pp. 359-377.

The PFORT Verifier is a standard-enforcing tool that checks a FORTRAN program for adherence to PFORT, a portable subset of ANS FORTRAN. The Verifier checks that intra-program-unit communication, occurring through the use of COMMON and argument lists, is consistent with the standard. Intra-program-unit error diagnostics, symbol tables and cross-reference tables are produced as part of the output for the program being checked.

Satterthwaite, E., "Debugging Tools for High Level Languages," Software--Practice and Experience, July-September 1972, vol. 2, no. 3, pp. 197-217.

The design of a programming system that supports a range of debugging aids and techniques using ALGOL W is described. These tools are based upon the source language, are efficiently implemented, and are useful in verification, analysis and diagnosis.

There are four design criteria for this debugging system: (1) the information presented to the user should be in terms of his program and of the source language, ALGOL W; (2) The compiler produces machine level code, and this code should not be degraded by the requirements of the debugging system; (3) Resource requirements should be limited, for I/O especially; and (4) the debugging features should be easy to invoke. Some examples of tools in this debugging system are presented, among them, a selective trace that is automatically controlled by execution frequency count, and an assertion capability that allows for specification of redundant information at critical points in the program.

Shomer, Jon A., "Improving Program Reliability Using COTUNE II," Workshop on Currently Available Program Testing Tools, April 1975, p. 30.

COTUNE II is a program execution monitor for COBOL that produces an execution count and processor-time histograms. The resulting COTUNE II reports can be used for documenting the source program as well as for giving an execution profile of the program. This tool can be used in testing and validating a program to ensure its correctness, and to control its reliability.

Stillman, Rona B, and Belkis Leong-Hong, "Software Testing for Network Services," NBS Technical Note no. 874, July 1975.

This report is a first step towards identifying effective software tests and measurement tools, and developing a guide for their use network-wide. Two tools are studied experimentally: the NBS FORTRAN Test Routines, which collectively form a useful tool for FORTRAN compiler validation; and the NBS FORTRAN Analyzer, which is a useful testing tool that provides static and dynamic analyses of the source code. Indications of their roles in systematic testing in a network environment are given.

Stucki, L. G., "Testing Impact on the Future of Software Engineering," Proc. Fourth Texas Conference on Computing Systems, November, 1975, pp. 4A-1.1 - 4A-1.6.

While self-metric software analysis suggests an attempt to explore control and data semantics of program behavior, the notion of an embedded assertion language is introduced that extends the ability to carry out "systematic programming." These assertion concepts are based upon the premise that there is a need to "think-through" the actual and expected behavior of an algorithm; they are also designed to encourage the development of algorithmic validation criteria from the beginning of the software development cycle, i.è., the definition of requirements, to the final program code. The assertion language is discussed in terms of its characteristics, such as, the generalized local assertion construct, which may be embedded in comments at any point within the executable code of a program; different assertion control options, such as instrumentation control, dynamic control and threshold control; and specialized local assertions and global assertions.

Stucki, L.G., and Gary L. Foshee, "New Assertion Concepts for Self-Metric Software Validation," Proc. International Conference on Reliable Software, April, 1975, pp. 59-65.

A user-embedded assertion capability is included as an extension to the Program Evaluator and Tester (PET). PET is a validation tool that allows for automatic verification of the dynamic execution of the user's programs. The assertion capabilities would allow the user to establish his own assertions at key points within his algorithm in the language he is using, as well as the ability of the user to obtain dynamic analysis and feedback on the validity of the assertion. These assertion capabilities would provide for two levels of control: global assertions and monitor commands have effect over the whole length of their enclosing module or block; and local assertions are position dependent, and consist of any legal logical expression of the host language. The assertions are transparent to the normal language compiler, and must be pre-processed in order for the dynamic execution checking to take place. The preprocessor instruments the assertion by augmenting the original source program with self-metric instrumentation. The augmented code is then executed, and also run-through a post-processor, finally yielding summary reports containing assertion violations and execution statistics.

Youngberg, E. P., "A Software Testing Control System," Computer Program Test Methods Symposium, June 21-23, 1972, in Program Test Methods, edited by William C. Hetzel, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973, pp. 205-22.

One of the ways of achieving thoroughness in testing is by using a systematic approach. A "test control system" provides for systematic testing procedures that ensure thoroughness. The control exercised on the test applications allows failure areas to be easily identified, and it also allows system degradation to be evaluated. One such test control system is the Validation Control System (VCS). The VCS consists of seven basic "tasks," these are: a "control nucleus;" "self-checking" modular test "kernels" containing service request calls to the control nucleus; a "job interrogator"; a parameter-driven "structurer"; a parameter-driven "JCL generator"; a data generator; and a "result reporting routine." Together, these tasks give the VCS the capability to interact with the software being tested at the unit level, the component level and the system level , and it provides the mechanism for validating software under a testing control system.

Wong, K.K., editor, "Computerguide 5: Concepts in Program Testing," and "Factfinder 8: Program Testing Aids," vols. 5 and 8 of Computers and the Professional, The National Computing Centre Limited, Manchester, England 1972.

In volume 5, different concepts and techniques in program testing are discussed, and some examples of testing aids are given. In volume 8, basic information about different testing aids is given.

NBS-114A (REV. 7-73)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET	1. PUBLICATION OR REPORT NO. NBS SP-500-14	2. Gov't Accession No.	3. Recipient's Accession No.

4. TITLE AND SUBTITLE

COMPUTER SCIENCE & TECHNOLOGY:

Software Tools: A Building Block Approach

5. Publication Date: August 1977

6. Performing Organization Code

7. AUTHOR(S)
I. Trotter Hardy, Belkis Leong-Hong, and Dennis W. Fife

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. 640-1125

11. Contract/Grant No.

12. Sponsoring Organization Name and Complete Address (Street, City, State, ZIP)
Partially sponsored by
The National Science Foundation
18th and G Streets, N.W.
Washington, D. C. 20550

13. Type of Report & Period Covered: Final

14. Sponsoring Agency Code

15. SUPPLEMENTARY NOTES

Library of Congress Catalog Card Number: 77-608213

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.)

The present status of software tools is described; the need for special-purpose tools and for new techniques with which to construct such tools is emphasized. One such technique involving the creation of general-purpose "building blocks" of code is suggested; an initial application of the technique to the construction of a text editor and syntax analyzer tool is described. An annotated bibliography of current literature relevant to software tools is provided.

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)

Building blocks; programming aids; software tools; syntax analysis; text editing.

18. AVAILABILITY

[X] Unlimited

[ ] For Official Distribution. Do Not Release to NTIS

[X] Order From Sup. of Doc., U.S. Government Printing Office Washington, D.C. 20402, SD Cat. No. C13.10:500-14

[ ] Order From National Technical Information Service (NTIS) Springfield, Virginia 22151

19. SECURITY CLASS (THIS REPORT): UNCLASSIFIED

21. NO. OF PAGES: 66

20. SECURITY CLASS (THIS PAGE): UNCLASSIFIED

22. Price: $ 2.10

USCOMM-DC 29042-P74

☆U.S. GOVERNMENT PRINTING OFFICE: 1977-240-848/275

ANNOUNCEMENT OF NEW PUBLICATIONS ON COMPUTER SCIENCE & TECHNOLOGY

Superintendent of Documents,
Government Printing Office,
Washington, D. C. 20402

Dear Sir:

Please add my name to the announcement list of new publications to be issued in the series: National Bureau of Standards Special Publication 500-.

Name _____

Company _____

Address _____

City _____ State _____ Zip Code _____

(Notification key N-503)

NBS TECHNICAL PUBLICATIONS

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 particular 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, $12.50; Foreign, $15.65.

To begin publication as a single journal, July-August, 1977

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.

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.

Order above *NBS publications from: Superintendent of Documents, Government Printing Office, Washington, D.C. 20402.*

Order following *NBS publications—NBSIR's and FIPS from the National Technical Information Services, Springfield, Va. 22161.*

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).

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 Services (Springfield, Va. 22161) in paper copy or microfiche form.

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, $25.00; Foreign, $30.00.

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

Superconducting Devices and Materials. A literature survey issued quarterly. Annual subscription: $30.00.

Send subscription orders and remittances for the preceding bibliographic services to National Bureau of Standards, Cryogenic Data Center (275.02) Boulder, Colorado 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