NBS

Study of a Prototype Software Engineering Environment

Dolores R. Wallace and D. Richard Kuhn

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Institute for Computer Sciences and Technology
Gaithersburg, MD 20899

June 1986

U.S. DEPARTMENT OF COMMERCE

NATIONAL BUREAU OF STANDARDS

NBSIR 86-3408

STUDY OF A PROTOTYPE SOFTWARE ENGINEERING ENVIRONMENT

Dolores R. Wallace and D. Richard Kuhn

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Institute for Computer Sciences and Technology
Gaithersburg, MD 20899

June 1986

STUDY of a PROTOTYPE SOFTWARE ENGINEERING ENVIRONMENT

Dolores R. Wallace and D. Richard Kuhn

ABSTRACT

A prototype software engineering environment was studied as part of the program to provide information to Federal agencies on software tools for improving quality and productivity in software development and maintenance. The purpose of a software engineering environment is to surround its users with software tools necessary for systematic development and maintenance of software. This report presents the results of the study of the prototype software engineering environment with respect to its features. The report also presents several factors to consider when evaluating a software engineering environment.

KEYWORDS

documentation; extensibility; integration; maintainability; portability; programming environment; software engineering environment; software support; software tools; user facilities.

FOREWORD

Under the Brooks Act, the National Bureau of Standards' Institute for Computer Sciences and Technology (ICST) promotes the cost effective selection, acquisition, and utilization of automatic data processing resources within Federal agencies. ICST efforts include research in computer science and technology, direct technical assistance, and the development of Federal standards for data processing equipment, practices, and software.

ICST has published several documents on software tools as part of this responsibility and the growing recognition that the use of software tools and software engineering environments can reduce the effort necessary to develop and maintain computer software. The guidance is designed to assist Federal agencies in automating and standardizing their software development and maintenance projects.

This report presents the results of a study of an experimental software engineering environment. It discusses the environment and its features to enable readers to gain an understanding of how environments can aid the software development and maintenance process. Future ICST documents will provide guidance in selecting and using software engineering environments.

The experimental software engineering environment prototype and other commercial products are identified in this paper for clarification of specific concepts. In no case does such identification imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the material identified is necessarily the best for the purpose.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 INTRODUCTION

One program of the Software Engineering Group of the Institute for Computer Sciences and Technology (ICST) of the National Bureau of Standards (NBS) provides information to Federal agencies on software tools for improving quality and productivity in software development and maintenance. A recent study involved examining and evaluating the individual characteristics and functions of a prototype software engineering environment. Goals of this study were to provide feedback to the developers; to gain first-hand experience with the tools and uses of environments; and to determine factors that should be considered before purchasing a software engineering environment. Understanding these factors may assist in assessing the need for a software engineering environment, the type of environment needed, and the features that should be included in the environment for a specific software project.

The software engineering environment selected for examination is the result of an on-going research project. The goal of the project's developers is to examine some of the central questions confronting the would-be builder of a software development environment by creating and studying an experimental prototype [OSTE83]. Preliminary versions of this environment, called Toolpack, were released for test and evaluation during 1984. ICST personnel have been involved with the Toolpack project almost since its beginning. This reports describes features of the ICST study from November 1983 to March 1984.

Toolpack is a collaborative effort involving researchers at Argonne National Laboratories; Bell Communications Research, Inc.; Morristown, NJ; Jet Propulsion Laboratory; Numerical Algorithms Group, Ltd.; Purdue University; University of Arizona at Tucson; and the University of Colorado at Boulder. Funding in the United States has been supplied by the National Science Foundation and the Department of Energy; funding in England has been supplied by Numerical Algorithms Group, Ltd.

Under no circumstances does this report constitute an endorsement of Toolpack by the National Bureau of Standards; nor does it imply that Toolpack is necessarily the best product for its purpose. Toolpack is still undergoing development as a research product. Many features described in this report are likely to be different in future versions.

2.0 THE TOOLPACK SUPPORT ENVIRONMENT

The Toolpack support environment was developed for a target community of developers of scientific software written in FORTRAN 77 [OSTE83]. Goals of Toolpack's designers were to provide an effective file management system and a tool set for the development of easily transportable code. Development activities include the creation, editing, testing, analysis, and documentation of code.

The developers of Toolpack intended the environment to support the development and the validation, verification and testing activities of the coding phase of the software lifecycle. Some concepts of the environment, such as the file management system, could be extended to support activities throughout the lifecycle.

When completed, the Toolpack environment (Figure 1) will contain the Toolpack Interface to the Environment (TIE) and the software tools. TIE provides the interface to the host file store and to the portable file store of Toolpack. Toolpack provides extensibility by permitting users to add or remove tools. The integration system provides a central data base of information. The information may be the source files on which some software tools operate, the test data to be executed on these files, or the results of those operations on which other tools perform their functions. Ideally these functions are tied together by an object-oriented "command interpreter" that allows several functions, or combinations of functions, to be performed by one command. This command interpreter is still experimental and was not available for the evaluation. A "command executor" that performs similar functions was provided with the version we received. The command executor allows the user to invoke one tool at a time, in a manner similar to most interactive operating systems.

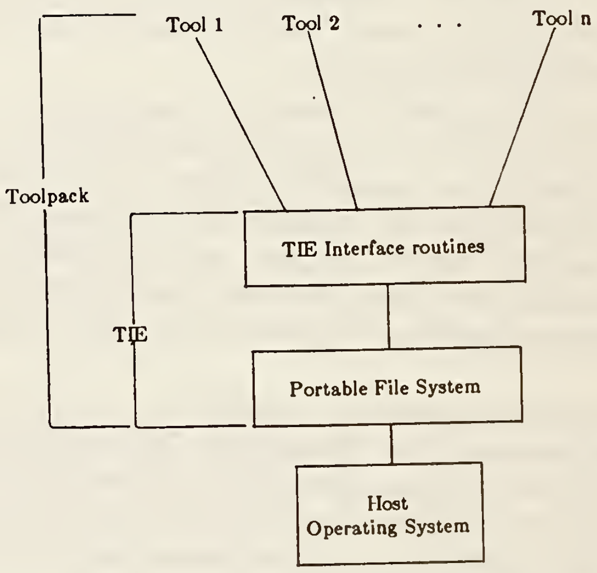

Figure 1. The Toolpack support environment.

Toolpack contains a suite of tools that are supported and integrated by TIE and the command language. The tools and their descriptions are shown in Table 1. Toolpack provides the tool capabilities needed for programming support:

o editor and documentation aids
o display, copy aids
o static analysis tools
o dynamic analysis tools
o a program transformer.

A compiler and link editor are not provided with Toolpack, since these programs are operating system and machine dependent. It is relatively easy to provide access to these host tools from Toolpack by writing a host command procedure that can be called from the Toolpack command executor. General and specific features of the major parts of Toolpack are discussed in the evaluation section. The tools listed in Table 2 are scheduled to be released in the first complete version of Toolpack, but were not available for evaluation. Table 2 is included here to indicate the goal of Toolpack's developers to achieve completeness. We used this information in our evaluation.

Our study of Toolpack is based on several criteria. Functional characteristics were evaluated using test cases developed from our functional requirements. For other factors, we developed criteria from Toolpack's description, from guidance suggested by

Federal software engineering publications, and from other technical journals. Much of the literature on Toolpack describes features that were not available in the version that we received. The evaluation is limited to the parts of Toolpack that were available to us. More information on the evaluation techniques may be found in another document [WALL86].

Our evaluation considered the following issues:

- o Functionality and Completeness
- o Extensibility
- o Tool integration
- o User facilities
- o Documentation
- o Maintainability
- o Portability
- o Resource usage.

3.0 FUNCTIONALITY AND COMPLETENESS

The test plan for Toolpack required the execution of each software tool on sample data consisting of FORTRAN subroutines. The sample data contain features appropriate to testing the functions of each tool. The individual tools to which we had access (Table 1) performed according to the functional descriptions provided in the documentation. The syntax for their use and the format for expected outputs were consistent with their documentation.

Information beyond functionality is needed to gauge how well a software development environment can satisfy user expectations of increased productivity and improved software quality. The tools selected for such an environment must be shown to be of value. The manner in which they interact and combine are important. Finally, the total environment must provide sufficient services so that the programmer need not leave the environment to perform other work on the software product.

Few metrics are available for measuring the effect of tools on software productivity and software quality. To help us determine how well Toolpack meets requirements for individual tools and as a total environment, we incorporated the results of two separate viewpoints into our analysis. One considers the individual worth of tools [HANS85] while the other considers levels of total capability provided by a given tool set [BRAN81]. The latter places value on how well the tools work together and on how easily new tools may be added, a feature that makes initial completeness less important. Interestingly, both studies emphasize similar sets of individual tools; many, but not all, appear in the selection offered by Toolpack.

In the study of individual tools, Hanson and Rosinski [HANS85], of Bell Laboratories, used psychometric scaling techniques to assess the value that programmers place on different types of tools. The authors of this study asked a group of experienced programmers to rate the tools according to their value in increasing productivity. The subjects were also asked to judge the similarity or difference between pairs of tools. The authors used these judgments to estimate sets of tools that are considered essential, and to group tools that are similar and provide equal productivity improvements.

ISTCE - Command Executor, maintains the portable file system and schedules tools.

ISTCT - File Type Changer, changes the type of a file between direct access and sequential.

ISTDS - Declaration Standardizer, standardizes declarative parts of FORTRAN 77 program units.

ISTFD - FORTRAN Intelligent Differencer, compares token streams from two source files.

ISTFL - File Length Calculator, finds the length of a sequential file.

ISTGP - Generalized Pattern Matcher, searches multiple files for the occurrence of a regular expression.

ISTLX - FORTRAN 77 Scanner, a FORTRAN 77 lexical analyzer.

ISTMP - Macro Processor, a general macro processor that can be used for expanding Toolpack macros, Toolpack documentation macros or any macro file.

ISTPL - Polish 77, a parameterized FORTRAN 77 'pretty printer'.

ISTPO - Polish Option File Editor, a menu driven editor for ISTPL option files.

ISTPR - FORTRAN 77 Parser, a FORTRAN 77 parser.

ISTPT - Transform Precision, converts the precision of variables (e.g. single to double).

ISTRF - Text Formatter, a general text (as opposed to source code) formatter.

ISTSP - File Splitter, splits a file into individual FORTRAN 77 source files.

ISTSV - File Save and Restore Utility, a file archiver/restorer.

ISTTD - Text Differencer, a standard text comparison tool.

ISTVC - Version Controller, maintains separate versions of a file in an ISTVC archive file.

ISTVS - View Symbol Table, produces a brief report of the contents of the symbol table for a file.

ISTYF - Flatten ISTYP Parse Tree, converts an ISTYP format parse tree to a token stream.

ISTYP - FORTRAN 77 Parser, a FORTRAN 77 parser.

Table 1. Toolpack tools supplied with test version.

Table 3 shows the tool types and rankings in [HANS85] and lists Toolpack tools from Tables 1 and 2 that represent each type. Tool names shown in asterisks, such as *ISTCI*, were not available in the version that we tested.

ISTCI - Command Interpreter, an experimental programming environment based on the Odin command interpreter.

ISTED - FORTRAN Aware Editor, a basic line based editor with some FORTRAN 77 awareness. This will be the vehicle for development of a language based editor.

Table 2. Toolpack tools not available for evaluation.

Branstad, Adrion, and Cherniavsky [BRAN81] propose levels of support that are based on increased levels of capability. The levels, which are presented in Table 4, provide a way of gauging the total level of support provided by an environment. The Minimal System (D1) contains features common to most operating systems. The Basic System (D2) augments D1 with a database and features to support the management of code and documentation. A tool included in D2, "Make", is a capability in UNIX[1] to configure programs or documentation [FELD79]; in D2 its generic functions are considered to be essential to a fundamental set of programming tools. The Full System (D3) completes coverage for the entire lifecycle. The Advanced System (D4) integrates the Full System represented by D3.

Tool	Relative Choice	Toolpack Tool
Interactive debugger	1.00	none
Screen editor	.98	*ISTED*
Subnetwork checker	.97	none
Process meter	.97	*NBS Analyzer*
Print file	.96	*ISTCI*
Stream editor	.91	ISTDS, *ISTGI*, ISTMP, ISTPT, ISTRF
Data dictionary	.90	none
Configuration manager	.90	ISTSV
Source code control	.90	ISTVC
Test coverage analyzer	.89	*NBS Analyzer*
Auto test generator	.88	none
Process monitor	.88	none
Private library	.88	ISTCE, *ISTCI*
Storage monitor	.88	ISTCE, *ISTCI*
File comparator	.84	ISTFD, ISTTD
Big file splitter	.84	ISTSP
Program cross referencer	.81	none
Display	.79	ISTCE, *ISTCI*
Big file scanner	.79	ISTGP
Source beautifier	.78	ISTPL

Table 3. Tool Rankings from Hanson & Rosinski Study.

Toolpack provides most of the tools listed in Tables 3 and 4 and contains some additional tools. The test version of Toolpack had several of the D2 set of tools; the first complete release of Toolpack is expected to contain the full D2 set of tools. Toolpack developers did not include tools for the earlier lifecycle phases of requirements and design which appear in D3. Effective tool integration, as considered by D4, is one of the important features the Toolpack developers aimed to provide.

Tools not mentioned in the Hanson study but contained in the Toolpack system are static analysis tools such as the lexical analyzer and parser. These tools, or their equivalent, may have been assumed to be included in compilers for the Hanson study. The Minimal System, D1, also contains some static analysis tools (e.g., type analysis) that Toolpack's developers may have assumed in compilers.

[1]UNIX is a trademark of AT&T.

	Standard Level Features	Additional Support for Critical Applications
D1 - Minimal System	Translation Cross-Reference; Trace Audit; File Comparison Optimization; Text Editing	Range Checking Type Analysis Assertion Checking Formatting
D2 - Basic System	D1 with Data Dictionary Information Repository Separate Compilation Make; Version Control Interface Analysis Test Coverage	Data Flow Analysis Structure Analysis Complexity Measurement Performance Monitor
D3 - Full System	D2 Requirements Specification Requirements Analysis Design Specification Design Analysis Test Harness Automated Documentation Automated Project Control	Symbolic Evaluation Proof of Correctness
D4 - Advanced System	D3 with Information Interfaces Specified and Full Integration	

Table 4. Levels of Support Based on Capability

Many of the tools rated by the Hanson programmers are for organizing and controlling code. In addition to code management tools, the Branstad levels emphasize the need for analysis tools for a complete programming environment. Several tools from both studies are not provided in Toolpack. The major shortcoming of Toolpack is the lack of an interactive debugger. A programming environment should increase productivity, as well as aid in producing quality software. No matter how carefully the software is developed, debugging will be needed. An interactive debugger could shorten the time required to locate errors.

The lack of a data dictionary is a serious shortcoming as well. A data dictionary is particularly valuable for maintenance, and Toolpack's users will inevitably need to modify their software. Modifications can be made more quickly and reliably with a dictionary that is kept up to date.

Other valuable test tools not included are an automatic test generator and a complete test harness (command procedure or script to organize and automate test runs). Creating test cases and organizing them manually is a time consuming and error prone task. Toolpack does include a test coverage analyzer. The analyzer can be used to check that the test cases provide complete coverage. A subnetwork checker would be unnecessary for individual programs, and not really essential for very small systems.

With Toolpack's extensibility feature, the lack of any specific tool is not necessarily a major shortcoming. The extensibility feature, discussed elsewhere in this paper, provides a capability to integrate additional tools into Toolpack. However, the features that tie the system together, to provide the user a unified work environment, are necessary for a completeness evaluation. Some of these features (e.g., the file system, the command language, the interfaces to other systems), are described in other sections of this paper. Greater availability of some of these tools during the test period would have enabled us to understand more clearly the benefits of a complete, uninterrupted work session to produce quality software.

4.0 EXTENSIBILITY

To understand how much effort is needed to integrate a tool into Toolpack, we developed a software tool that computes the McCabe Cyclomatic Complexity Metric for Fortran 77 programs. The tool, called METRIC, was developed externally to Toolpack, using only the tools available on Berkeley UNIX. Then the tool was integrated into Toolpack.

METRIC used standard Fortran 77 I/O statements. Conversion to Toolpack required replacing the Fortran I/O, format, and print with calls to Toolpack routines. The process was much like converting an application to a new operating system or access method.

4.1 Development of a New Tool

McCabe [MCCA82] represents the control structure of a program as a directed graph and uses its cyclomatic number to estimate the complexity of the program. The cyclomatic number (V) gives the maximum number of linearly independent paths in the graph (G). It is calculated as

$$V(G) = e - n + 1,$$

where

e = number of edges,

n = number of nodes.

Combinations of paths will generate all possible paths through the graph. McCabe shows that V(G) is equal to one plus the number of decision nodes in a program. To calculate V(G) it is necessary to recognize the decision statements in a program and count the number of conditions.

METRIC uses the parse tree and symbol table files built by the Toolpack parser, ISTPR. No Toolpack routines were used; the parse tree and symbol table files were processed exactly like any other files using standard FORTRAN 77 I/O. METRIC loads the parse tree from a file built by ISTPR into internal arrays, then reads the arrays sequentially to compute the complexity metric.

The Toolpack documentation provided an excellent description of the parser and the parse tree format, and a copy of the context-free grammar used to generate the parser. Having the grammar made it easy to write routines to recognize the statement types needed to compute the metric (IF, IF-THEN, COMPUTED GOTO, etc.). After studying the documentation, the Metric program was simple to write. It was completed in two days, with a third day spent changing the output format and testing it against a collection of files. After this we spent several more days running the metric against a wide variety of source code. The only error encountered was a case where we neglected to increment a statement type counter. The Toolpack lexical analyzer, ISTLX, and the parser, ISTPR, functioned correctly on all tests. The total development time, including studying Toolpack documentation, design, coding, and testing, was about two and a half weeks of concentrated effort.

4.2 Conversion to Toolpack

METRIC was converted to use Toolpack routines in place of FORTRAN 77 I/O statements. This new tool, called "ISTMT" is callable from the Toolpack Command Executor and has a user interface consistent with the other Toolpack tools. The conversion was accomplished in three steps:

(1) Review documentation on Toolpack routines that can be used to access the parse tree and symbol table. Since the documentation is generally good, this step went quickly.

(2) Examine each function of the Metric program to see if it needed to be changed for Toolpack portability. The term function refers to any block of code performing a single operation, not just to Fortran functions and subroutines. In many cases a simple one-to-one replacement was sufficient because of the similarity between the data structures used in our program and in Toolpack. For example, our program contained two subroutines that load the parse tree and symbol table files into internal arrays. Toolpack provides a single initialization routine that loads the parse tree and symbol tables into data structures that can be accessed through 20 other library routines. The only changes needed for initialization were to replace the calls to our routines with a call to the Toolpack initialization routine. Our load routines could then be deleted from the program.

(3) Replace the program functions by Toolpack-conforming code or Toolpack function calls. Only minor changes were needed in most of the program. Creating formatted output using Toolpack routines was difficult because the routines do not provide a convenient way of tabbing.

The original METRIC accessed the parse tree and symbol table files through Fortran unit numbers. To make ISTMT consistent with the other tools, we added a user interface to accept file names from 'stdin'. This was easily accomplished by modifying the user interface routine from another Toolpack tool.

4.3 Evaluation of Conversion

Converting METRIC to Toolpack resulted in a significant decline in performance. Executing the metric tool in the Toolpack embedded environment required over six times as long as the Fortran 77 stand-alone version. The 6:1 ratio was constant across parse tree files ranging in size from 350 to 6000 nodes.

Our Toolpack release contained a library of I/O routines that had been tailored to Berkeley UNIX. These routines can be used to improve performance by replacing some of the portable Toolpack routines. Using the UNIX I/O routines requires no changes to source code. They are incorporated into the load module by the link editor. When these I/O routines were linked into ISTMT, the program performed as well as the earlier version that used standard Fortran I/O statements.

The output subroutine of METRIC required the most extensive changes and was the most difficult to convert to Toolpack. There is no straightforward replacement of WRITE and FORMAT statements by Toolpack routines. Aligning output in columns was particularly awkward compared with the simple tabbing provided in Fortran 77. Toolpack usage might be simplified by providing Toolpack routines that more closely resemble the standard Fortran. An alternative might be to provide print functions similar to those in UNIX. It might also be helpful to have some discussion of formatted output included in the Toolpack documentation.

With the exception of output formatting, the conversion was easy and the basic structure of the program was not changed. One of the few problems with the conversion occurred when we neglected to declare one of the Toolpack functions as EXTERNAL. The compiler did not detect the error and the program did not crash when executed. The call to the routine returned '0' rather than a correct value, so it appeared that the Toolpack routine was not working properly.

After checking the Toolpack code and rechecking our code again, we realized that all the code should be working and only then began looking for operating system and compiler-related problems. This debugging could have been avoided with the help of a static analysis tool to check interfaces and externals.

5.0 TOOL INTEGRATION

Toolpack is tightly integrated. Tools are single-function modules that use the output of other tools whenever possible. For example, the lexical analyzer builds token and comment files that are used by several other tools. The FORTRAN Intelligent Differencer uses the token stream to ignore textual differences between files, while the pretty-printer uses the token and comment files to reconstruct and reformat source code. Figure 2 [OSTE83] shows the dependencies among tools.

One of the most interesting features proposed for Toolpack is the virtual file system. The file system contains a representation of the dependencies between tools and their input and output files. For example, if the parser is invoked, it requires a token stream file. If a current version of the token file exists it is provided by the file system. If it does not exist, or is not current, the system invokes the lexical analyzer to generate it before calling the parser.

This tight integration provides many advantages. The user has no need to keep track of intermediate file names while using the tools, and the virtual file system insures

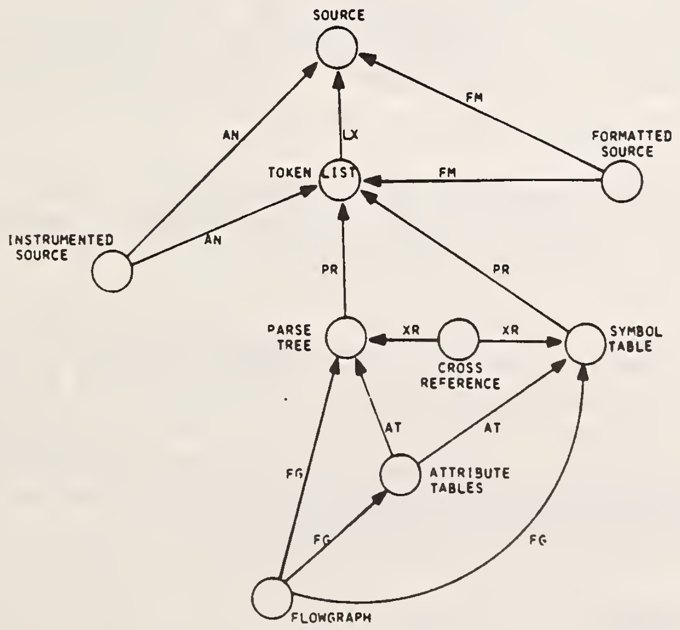

Figure 2. Toolpack tool dependencies

that an up-to-date version of intermediate files will always be used.

The Toolpack version that we tested did not have a working virtual file system, so we could not judge the value or performance of the file system design. We had previously examined a prototype version, ODIN [CLEM84]. ODIN required significant UNIX operating system overhead because the virtual file system was implemented by creating a new directory for each invocation of a tool. This resulted in directory nestings of twenty or more levels and a heavy load on UNIX.

6.0 USER FACILITIES

The acceptance of any software system frequently depends upon its "user friendliness" as much as any other quality.

6.1 Command display

Many of the tools provide a help menu that displays command syntax and function. The displays are accurate, with minor exceptions. One shortcoming is the lack of consistency in invoking help. For example, the command interpreter provides help in response to "??", while the editor uses "HEL" as the help command. There is no menu-driven operation provided, but this is not as important in Toolpack as it would be in many applications. The virtual file system and command interpreter, that are scheduled to be included in future versions of Toolpack, would relieve the user of many decisions, reducing the need for menu driven lead-through.

6.2 On-line documentation

Other than the help screens provided by the tools, there is no on-line documentation such as that provided by the "man(ual)" command in UNIX. This is to be expected in a system still under development, and it is not a serious drawback. The printed documentation provided with the system is complete and very readable.

6.3 Command syntax and abbreviation

The command format used in Toolpack is similar in style to that of UNIX: one to three characters are used for most commands. Most of the abbreviations are intuitively clear. There is no command completion facility. Command names must be spelled out completely. Since the command names are short, the lack of a command completion facility is not a problem.

The similarity to UNIX commands can be confusing at times. For example, the Command Executor uses "CD" for "Create Directory" while in UNIX, "cd" stands for "change directory".

6.4 Prompting

Invoking the tools generally requires a tool name followed by a set of parameters, such as file names. An especially nice feature is the prompting provided by the tools. For example, to run the lexical analyzer ISTLX, the user enters "ISTLX" followed by the names of Source, List, Error, Token and Comment files. If only "ISTLX" is entered, the tool will prompt for the missing file names. If some, but not all, of the file names are entered, the tool will prompt for the missing names.

7.0 DOCUMENTATION

Quality of documentation is an important aspect of a software engineering environment, since even the best environment will be of little value without clear instructions on how to take advantage of its capabilities. We found both the user documentation and system documentation for Toolpack to be of uniformly good quality.

7.1 User Documentation

Although the Toolpack system was not complete when we evaluated it, and had not been officially released, the version that we examined came with a reasonably complete set of documents. There are documents for installation, for writing tools with Toolpack, for understanding the Toolpack interface, for the editors, and for every tool. All are clearly written and appear to be complete. We considered the documentation acceptable for our use. Our only complaint was that documentation described tools not yet in the Toolpack environment.

7.2 System Documentation

An extensible environment such as Toolpack needs good system documentation, because users will want to add tools to the system. The documentation supplied to us contained complete system library interface descriptions, written in a style similar to the UNIX operating system manual pages: routine name, call structure, type and description of each parameter.

The code is well commented and the comments are meaningful. The library subroutines generally contained more comments than the tools themselves. We found ratios of roughly one comment for each three lines of code in the subroutine libraries, and one comment for each six source lines in the tools.

We gained some practical experience using the system documentation when we developed a tool on our own. The documentation was generally good, although in a few cases the functions of a routine were not clear. In these cases we checked the source code to determine what would happen. Generally, comments in the source code and published documentation made it easy to understand the routines.

8.0 MAINTAINABILITY

One of the most important considerations in acquiring a system is the ease with which it can be maintained. This note discusses maintenance aspects of Toolpack following the outline provided in FIPS PUB 106, "Guideline on Software Maintenance".

8.1 Single High-Order Language

Toolpack and the Toolpack software are written entirely in FORTRAN 77, except for three C routines that provide bit shift functions and access to operating system date and time. Because the file system routines are written in FORTRAN 77, they are easily portable, but very inefficient. The Toolpack implementation we used had I/O routines written in C that used UNIX I/O functions to improve performance. Testing showed that these operating system dependent routines perform I/O about twenty times faster than the routines that use FORTRAN for all I/O, so the additional maintenance burden is worthwhile.

8.2 Coding Conventions

Programming style can have a significant impact on maintenance. The conventions discussed below are recommended in FIPS 106 and are frequently recommended as ways of improving the readability of source code.

(1) Keep it Simple - Most of the Toolpack modules are coded in a simple and straightforward style. The cyclomatic complexity metric described in [MCCA82] is also useful to estimate the difficulty of understanding and testing a program. Table 5 and Table 6 show the complexity metric scores of Toolpack software. McCabe [MCCA82] suggests that values of 10 or less represent reasonable complexity for a

module.

Library	# Modules	1 to 10	11 to 20	> 20
access	44	40 (91%)	3 (7%)	1 (2%)
common	106	96 (90%)	9 (9%)	1 (1%)
direct	2	2 (100%)	0	0
embedded	2	2 (100%)	0	0
host	23	23 (100%)	0	0
pfs	91	89 (98%)	2 (2%)	0
string	37	28 (76%)	7 (19%)	2 (5%)

Table 5. COMPLEXITY METRIC: Toolpack Libraries.

Tool	# Modules	1 to 10	11 to 20	> 20
isted	133	95 (71%)	26 (20%)	12 (9%)
istfd	20	13 (65%)	5 (25%)	2 (10%)
istlx	50	40 (80%)	6 (12%)	4 (8%)
istpl	58	44 (74%)	9 (16%)	6 (10%)

Table 6. COMPLEXITY METRIC: Selected Tools.

(2) Indentation - The Toolpack programs are nicely indented. In addition, a pretty-print tool (ISTPL) is provided, making it easy to reformat source code.

(3) Use meaningful variable names - Since FORTRAN limits the length of identifiers to six characters, it is not always possible to create good descriptive names for variables. The names used in Toolpack are generally meaningful, and the code is sufficiently well commented to tell how a variable is used.

(4) Avoid similar variable names - Again, the length restriction of FORTRAN makes this unavoidable sometimes, and some routines and variables do have similar names. This is generally not a significant problem in reading the code.

(5) Pass values using parameters - This is done in most cases. It appears that COMMON blocks are used with restraint and only where there is a very good reason to have them.

(6) Avoid non-standard language features - Great care was taken by the developers to make Toolpack portable, and the code conforms to ANSI standard.

(7) Extensively comment the code - The code is well commented, as discussed in the previous section on documentation.

8.3 Structured Modular Software

The Toolpack modules are small and use structured programming conventions. Table 7 shows average module size for the subroutine libraries.

Library	# Modules	Lines	Avg. Size
access	40	2,416	60
common	106	6,657	63
direct	2	84	44
embedded	2	226	113
host	23	453	20
pfs	91	4,969	55
string	37	2,305	62

Table 7. Average Module Size.

8.4 Standard Data Definitions and Data Dictionary

Toolpack does not have a data dictionary, but variable names are consistent.

9.0 PORTABILITY

Toolpack's target users, mathematical software developers, use FORTRAN almost exclusively with a wide variety of hardware and operating systems. Thus, one of the major goals of the Toolpack project is to provide a system that creates totally portable programs. The solution to this problem is a set of subroutines that replaces the standard FORTRAN I/O statements. All portable file system (PFS) software is written in FORTRAN 77, so it is portable among hardware and operating systems. The documentation clearly indicates which subroutines are likely to require changes for host dependencies.

9.1 Portable Character Set

Toolpack provides its own character set which is mapped onto the host character set by special subroutines. In theory, a program that works on an ASCII machine should work on an EBCDIC machine, provided that data are represented in the Toolpack character set. Since we had no hardware with an EBCDIC code set, we could not test this feature.

9.2 Impact on Performance

An interactive environment should have good response time, but we noticed that I/O was extremely slow using the portable Toolpack I/O routines. We wanted to examine the value of using I/O routines tailored to the host operating system.

The Toolpack I/O routines move data character-by-character and execute conversion routines to map from one character set to the other. This results in a severe performance problem. We compared the portable I/O routines with a set of routines that perform the same function using UNIX system calls. A program that copies one file to another using the portable routines actually becomes CPU-bound because of the deeply nested subroutine calls. This testing is described below.

9.3 Performance Improvements

Toolpack contains I/O routines written entirely in FORTRAN 77 to aid portability. Unfortunately, this results in very slow operation. To speed up processing some implementors have developed a library of I/O routines written in C that use the UNIX system functions. These routines can be used to replace the F77 routines with no modifications to a program's source code. All that is necessary is to re-link the object files using the "unixio" library.

We executed a set of tests to determine the degree of performance improvement possible through the operating system-dependent routines in "unixio". The improvement found is quite spectacular, suggesting that it is well worth the effort to tailor a few routines to the host operating system.

To estimate the improvement in I/O processing, we wrote a small program that copies an input file to an output file. The program was created by extracting and modifying the code for the "CP" instruction in the command executor. The program does one file OPEN, one CLOSE, and uses GETLIN and PUTLIN to copy. Two executable versions of the program were created: one was linked using only the F77 Toolpack routines for I/O; the other was linked with routines in the "unixio" library. Table 8 shows the times of the two versions run on different length files.

file length (bytes)	F77 Toolpack routines (seconds)	unixio routines (seconds)	unixio/F77 times (percent)
5,000	23.9	6.4	18.4
10,000	46.4	4.8	10.3
15,000	63.8	5.2	8.1
20,000	85.5	5.7	6.7
30,000	126.5	6.4	5.0
40,000	168.9	7.3	4.3
50,000	208.1	8.2	3.9
70,000	279.9	9.9	3.5
90,000	382.1	11.3	2.9

Table 8. I/O performance

The improvement factor obtained above will be only approximate. The programs in Toolpack use different mixtures of I/O functions, but most will have one or more OPENs, CLOSEs, many READs, and possibly many WRITEs. Furthermore, some I/O subroutines are called for each character transferred and some are called at the end of each line.

Profiling showed that these subroutine calls produced a bottleneck that kept the transfer rate below 300 bytes/second. The situation was reversed with the 'unixio' version of the copy program. Since the CPU time between I/Os is less than the time for two disk I/Os, the disk limits throughput. The operating system buffers and asynchronous I/O prevent the disk from becoming a bottleneck on the file sizes we copied, which explains why the transfer rate increases with file size.

Another interesting question is how the Toolpack unixio routines compare with the performance of the UNIX FORTRAN 77 I/O library subroutines. We developed two versions of a program that reads the parse tree file produced by ISTPR. The first used standard FORTRAN I/O statements supplied by the UNIX FORTRAN I/O library. The second used the Toolpack unixio routines. Both versions ran in approximately the same amount of time, regardless of the file size.

10.0 RESOURCE USAGE

Toolpack users will normally need only a copy of the portable file system, command executor and tools. The system manager will need to maintain a library of Toolpack source code and documentation. The system library should also contain object code modules that can be linked to create executable tools for the users.

The system library for the version that we tested requires about 11 megabytes of disk storage. Future versions will have a larger set of tools and the command executor will be replaced, so more space will be required.

The version of Toolpack that we tested is designed for a single user environment. Each user must have a separate portable file system , because PFS files are accessed through Toolpack system subroutines that contain hard-coded path names for PFS files. Because the path names are different for each user's PFS files, executable files cannot be shared.

Shown below are the space requirements, in bytes, for the files needed by each Toolpack user. The sizes used are based on the recommendation in the Toolpack Installer's Guide. These figures might be different depending on the user's needs. A sampling of Toolpack source code shows that one line of code requires an average of about 30 bytes.

At this size, the 10,000 block (256 bytes/block) PFS would hold about 81,000 lines of code, after allowing for overhead. The size figures for the tools will of course vary slightly with different versions.

System Files	3,357,845
Tools, executable	3,744,510

	7,102,355

11.0 SUMMARY

NBS's involvement with Toolpack developers has a long history. A session on research for software environments at NBS's Programming Environment Workshop [BRAN78] was led by one of Toolpack's developers. Our original plans for testing and evaluating Toolpack were written from early technical publications describing Toolpack. We worked incrementally, making iterations on our plans and examining improvements in successive versions. We ended our review before this particular environment became a completed product so we could not realize the full effect of a completed environment.

Our goals for this Toolpack involvement were to learn about software support environments and how to evaluate them and to report back to the developers while they worked on the environment. The first part of this summary discusses Toolpack specifically and the second part provides more general information on environments and evaluating them.

In our study we found that the individual tools and the integrating features of Toolpack's tools functioned well. Principles of software engineering have been employed by Toolpack's developers. The software quality features of maintainability, portability, and documentation are excellent. In general, user facilities are provided and would be useful to the Toolpack user. As a total programming support environment, our Toolpack version is incomplete, with many planned features missing.

The extensibility feature is key to the success of an environment. In Toolpack, tools may be readily added so that lack of a specific analysis or text manipulation tool is not a major shortcoming. Some of the tools proposed for Toolpack, such as a FORTRAN-aware editor, are research ideas that once developed can be readily implemented. Lack of higher level services that tie in to the basic system is a fundamental problem. For example, the capability to execute a series of commands in one command sequence or to check the need to rebuild files called by commands in the sequence would be highly useful. Interfaces to other systems on the host computer also would be valuable. The integration of the Toolpack programming tools themselves, (e.g., use of data from one by the other), works well; the higher level integration services still need a lot of work. Once these tools are ready for Toolpack, its extensibility feature should make them relatively easy to add.

The contrast between the lack of completeness of Toolpack and the relative completeness of the operating environment on which we installed it emphasizes two questions. When should a software engineering environment be used? How should an environment be evaluated for a specific use? Our Toolpack experience provided some insights to these questions, based on the following considerations:

o capabilities already available to us vs those of Toolpack o need for individual tools, based on evaluations in technical literature

o evaluation of other characteristics and features.

In our case, the operating system, UNIX, provided us a full range of services included in Toolpack as well as many other utilities to which we were accustomed. Among these services are background processing, profiling, access to UNIX utilities without leaving the current utility, execution of a series of commands via shell scripts. Toolpack may be most useful as an environment that can be built on top of an existing operating system that does not have a full range of utilities. Toolpack provides some features that are often not available on operating systems for large-scale scientific processors. For organizations that do most of their development on UNIX, Toolpack's integration capabilities and file system may not be as valuable, although some of its tools can be quite useful. We have used the scanner and parser in two projects since this study was done and found the parse-tree access routines to be very convenient for developing FORTRAN source code processing tools. An organization needs to evaluate its existing capabilities against those provided by a software engineering environment.

In this study we concentrated primarily on the tool capabilities of Toolpack and used technical evaluations to guide us. Others could apply this approach to determine the minimum tool set needed for their project. The Hanson tool evaluation and the Branstad levels could be used to compare a current operating environment to a software engineering environment to determine which provides the greater capability. Next, the capability could be weighed against the factors affecting how the capability is provided: user facilities, documentation, resource usage, maintainability, extensibility.

There are different ways of looking at a software engineering environment. Some of the features and characteristics to be considered in an evaluation are described in another NBS study [HOUG85] that was completed after the Toolpack examination. Other factors (e.g., cost benefit) need to be considered but discussion of those is outside the scope of this document. Determining when to use a software engineering environment and what services it must provide is not an easy task. The steps suggested in this summary may provide some guidance in performing such a task.

12.0 REFERENCES

[BRAN78]

Branstad, Martha A. and W. Richards Adrion, Editors, "NBS Programming Environment Workshop Report," National Bureau of Standards, NBS SP 500-78, June 1981.

[BRAN81]

Branstad, Martha A., W. Richards Adrion, and John C. Cherniavsky, "A View of Software Development Support Systems," *Proceedings of the National Electronics Conference,* National Engineering Consortium, Inc, 1981.

[CLEM84]

Clemm, Geoffrey M., "ODIN - An Extensible Software Environment Report and User's Manual," University of Colorado, Boulder, CO, CU-CS-262-84, March, 1984.

[FELD79]

Feldman, S.I., "Make - A Program for Maintaining Computer Programs," *Software Practice and Experience,* Vol.9, 1979.

[FIPS106]
"Guideline on Software Maintenance," Federal Information Processing Standards Publication 106, National Bureau of Standards, 1983.

[HANS85]
Hanson, Stephen Jose and Richard R. Rosinski, "Programmer Perceptions of Productivity and Programming Tools," *Communications of the ACM,* Vol.28, No.2, February, 1985.

[HOUG85]
Houghton, Raymond C., Jr., and Dolores R. Wallace, "Characteristics and Functions of Software Engineering Environments," National Bureau of Standards, NBSIR 85-3250, Spetember, 1985.

[MCCA82]
McCabe, Thomas J., "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," NBS Special Publication 500-99, National Bureau of Standards, December, 1982.

[OSTE83]
Osterweil, Leon J., "Toolpack - An Experimental Software Development Environment Research Project," *IEEE Transactions on Software Engineering* Vol. SE-9, No.6, November, 1983.

[WALL86]
Wallace, Dolores R., "An Experiment in Software Acceptance Testing," National Bureau of Standards, NBSIR 86-3407.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** *(See instructions)*	1. PUBLICATION OR REPORT NO. NBSIR 86-3408	2. Performing Organ. Report No.	3. Publication Date JULY 1986

4. TITLE AND SUBTITLE

Study of a Prototype Software Engineering Environment

5. AUTHOR(S)

Dolores R. Wallace and D. Richard Kuhn

6. PERFORMING ORGANIZATION *(If joint or other than NBS, see instructions)*

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS *(Street, City, State, ZIP)*

10. SUPPLEMENTARY NOTES

☐ Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT *(A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here)*

A prototype software engineering environment was studied as part of the program to provide information to Federal agencies on software tools for improving quality and productivity in software development and maintenance. The purpose of a software engineering environment is to surround its users with software tools necessary for systematic development and maintenance of software. This report presents the results of the study of the prototype software engineering environment with respect to its features The report also presents several factors to consider when evaluating a software engineering environment.

12. KEY WORDS *(Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons)*

documentation; extensibility; integration; maintainability; portability; programming environment; software engineering environment; software support; software tools; user facilities.

13. AVAILABILITY

[✓ X] Unlimited
☐ For Official Distribution. Do Not Release to NTIS
☐ Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.
[✓ X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

23

15. Price

$9.95