NBSIR 82-2625

A Taxonomy of Tool Features for the Ada* Programming Support Environment (APSE)

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Institute for Computer Sciences and Technology
Center for Programming Science and Technology
Washington, DC 20234

Final Report

December 1982

Issued February 1983

Prepared for:

Department of Defense
Ada Joint Program Office
801 N. Randolph, Suite 1210
Arlington, VA 22209

*Ada is a registered trademark of the U.S. Department of Defense

NBSIR 82-2625

A TAXONOMY OF TOOL FEATURES FOR THE Ada* PROGRAMMING SUPPORT ENVIRONMENT (APSE)

Prepared by:
Raymond C. Houghton, Jr.

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
Institute for Computer Sciences and Technology
Center for Programming Science and Technology
Washington, DC 20234

Final Report

December 1982

Issued February 1983

Prepared for:
Department of Defense
Ada Joint Program Office
801 N. Randolph, Suite 1210
Arlington, VA 22209

*Ada is a registered trademark of the U.S. Department of Defense

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, ***Secretary***
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, ***Director***

FOREWORD

The subject of the taxonomy that is presented in this report is the Ada Program Support Environment (APSE) [DoD80]. The categories within the taxonomy are features of the APSE. Two criteria are used in the selection of features:

1. A feature must be within the current state of software development practice, that is, there must be examples of existing tools that are currently being used by the Federal Government or industry.

2. A feature must be useful for the development of Ada software and may, in some cases, be explicitly oriented toward the language.

Note that the second criterion requires a judgemental decision on the usefulness of a feature.

Many valuable comments were received on earlier drafts of this report and the contributions are appreciated. The author would like to thank the following people for their review: P. Powell, M. Branstad, B. Brosgol, L. Weissman, P. Henderson, L. Druffel, A. Wasserman, D. Wallace and P. Freeman. The author would especially like to thank the following people for their in-depth comments: P. Oberndorf, J. Tippett, M. Shahdad, H. Fischer, P. Scheffer, S. Saib, A. Gargaro, V. Castor, F. Cox, J. Ahders, J. Kerner, P. Blackledge, R. Reed, and D. Sen. This work was supported by the Ada Joint Program Office under a special agency agreement with the National Bureau of Standards.

TABLE OF CONTENTS

1. Introduction

A taxonomic classification is an ordered arrangement that begins with the broadest, most inclusive categories and ends with the narrowest, most specific categories. Taxonomic classifications have several uses including: (1) familiarization with the terminology associated with a subject, (2) obtaining a broad overview of an area, and (3) as a starting point for obtaining in-depth information on a certain subject. In this report, a taxonomic classification will be referred to as a taxonomy. Although, strictly speaking, taxonomy is a science. A taxonomic classification is often referred to as a taxonomy, e.g. Taxonomy of Computer Science and Engineering [AFIP80], Software Engineering Standards Taxonomy (IEEE Software Engineering Standards Project, L. Tripp, Chairperson).

In this report, the taxonomy developed by the National Bureau of Standards' Institute for Computer Sciences and Technology (NBS/ICST) is used as a basis for the development of an APSE taxonomy and for the comparison of the features provided by the Ada Language System (ALS) and the Ada Integrated Environment (AIE)*. The ALS and the AIE are Minimal Ada Programming Support Environments (MAPSE's). Therefore, the APSE taxonomy includes many additional features, including many that are specified in [DoD80].

Where the NBS/ICST Taxonomy is concerned with a broad range of features offered by all types of software development tools (see Appendix B), the scope of the APSE taxonomy has been limited to tool features that are within current technology and that are oriented to the Ada language. It is anticipated that the APSE taxonomy will be used as a basis for the development of additional tools for the APSE, particularly in the management area (Appendix A). Consequently, the selected features involve a low implementation risk.

Finally, the features of the APSE taxonomy are expected to support the properties of METHODMAN (Software Development Methodologies for Ada) [Druf82], which are currently being defined by Freeman and Wasserman [Free82]. Of course, some properties of METHODMAN will be more susceptable to automation than others and those that are susceptable to automation may be methodology dependent.

The features in the APSE taxonomy are presented in Section

---

* The ALS is being developed by Softech, Inc. under contract to the U.S. Army. The AIE is being developed by Intermetrics, Inc. under contract to the U.S. Air Force. The implementation of tool features in the ALS may differ significantly from implementations in the AIE. To gain a complete understanding of the differences between these systems, further analysis at lower levels of detail is necessary.

2. All features discussed are oriented at the user level. That is, the user is aware of and can either instantiate or apply the feature directly. Features that can be methodology dependent are indicated. Section 3 includes discussion of the underlying features of the APSE. These are called tool primitives and they are necessary to support the features presented in Section 2. Appendix A contains a prioritization of the tool features. Appendix B contains the NBS/ICST Taxonomy.

2. Classes and Processes of the APSE

The APSE taxonomy is a hierarchical arrangement of software tool features as shown in Figure 1.

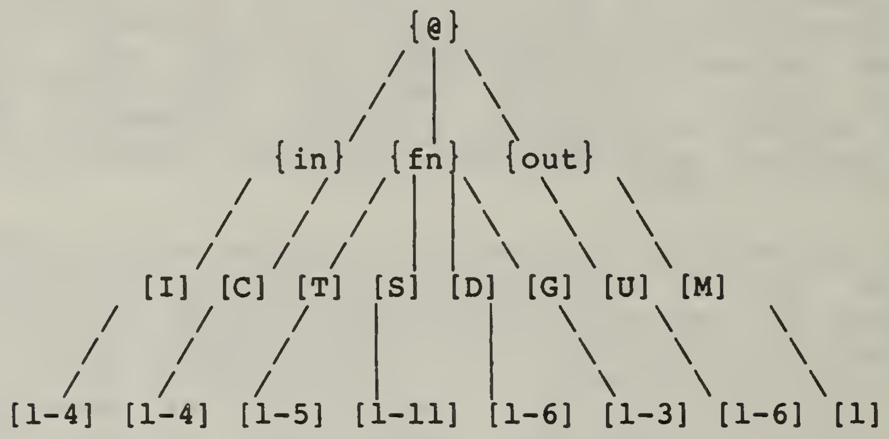

Figure 1. APSE Taxonomy

The highest level (@) is the most abstract and covers all the features below it. The second level of the taxonomy covers the basic processes of the APSE. These are input (in), function (fn), and output (out). At the third level are the classes of tool features. These are subject (I), control (C), transformation (T), static analysis (S), dynamic analysis (D), management (G), user output (U), and machine output (M). The fourth level is the feature level. At this level each of the features of the APSE taxonomy is defined. In some cases, the fourth level has been further expanded to a fifth level. The ranges between brackets signify the number of features in each of the classes.

In the sections that follow, the functional classes (management, static analysis, dynamic analysis, and transformation) are discussed according to their importance and impact on the APSE. This is followed by a brief discussion of the input/output features. Since many of the features in the APSE taxonomy are also in the NBS/ICST taxonomy, the reader is requested to refer to Appendix B for additional description and definition of the features that are common to both taxonomies.

Management.

The most important area that is necessary to support a development methodology is tool support for management. Table 1 shows that the ALS and AIE provide support for configuration management and Ada library management. The APSE Taxonomy specifies many additional features that are important to the management of software development.

MANAGEMENT

ALS	AIE	APSE Taxonomy
Configuration Ctrl	Configuration Ctrl	Configuration Ctrl
Information Mgt	Information Mgt	Information Mgt
Ada Library Mgt	Ada Library Mgt	Ada Library Mgt
-	-	Specification Mgt
-	-	Data Dictionary Mgt
-	-	Ada Package Mgt
-	-	Test Mgt
-	-	Project Mgt
-	-	Cost Estimation
-	-	Scheduling
-	-	Tracking

Table 1. Management

Information Management. In addition to Ada library management, the APSE taxonomy includes design management, requirements management, data dictionary management, Ada package management, and test data management. All of these features can be implemented in the APSE through use of the data base support provided by the Kernel Ada Programming Support Environment (KAPSE' The KAPSE will be accessible through the standard KAPSE interface which is being defined by the KAPSE Interface Team (KIT) [Ober82]. One of these features, specification management, is somewhat methodology dependent.

Specification management refers to the control of requirements and design specifications. Specification management tools are somewhat methodology dependent because associated with these tools are usually requirement and design specification languages with formal procedures for their use. However, there may or may not be controlling elements in the tools that prevent one from deviating from the formal procedures, e.g., review and approval mechanisms, required record keeping of scope changes, strategy changes, justification, and rectification. Examples of specification management tools with associated methodologies are REVS [Alfo80] and SARA [Pena81]. Examples of tools which do not require formal procedures to be used in association with a specification language are MEDL-R and MEDL-D [Houg82]. The latter tools emphasize the management of requirements and design information and are more independent of the development methodology.

Systems which aid the management of data specifications are commonly called data dictionary systems. Data dictionary systems have been available for years [Leon77] but their potential in software development has not been fully utilized. Data dictionary systems can be a valuable tool over the entire life cycle of systems and software [FIPS76]. For example, during the requirements definition phase, information is collected about data type and usage requirements. This information can then be used in the production of data descriptions for individual software packages and procedures, and in conjunction with other analysis capabilities (see cross reference and data flow analysis).

Ada package management provides a way to avoid redundant coding by identifying packages that already perform certain functions. It keeps track of the functionality of packages and retrieves information about packages that provide capabilities of interest to the user. Capturing the functionality of Ada packages is a current objective of research. However, an interim solution to package management can be implemented through keyword specification and retrieval in a manner similar to library systems. A recent paper [Lecl82] describes a browse documentation system that is very similar in concept.

Test data management is a feature that is used throughout the development process. Test data is derived in all phases of software development and is stored and tracked by the management system. When the system under development is executable, either through simulation, stubs, or code, test data management usually works in conjunction with regression testing (see dynamic analysis). Test data and test results are controlled, stored, retrieved, and compared by the system. Programs are initiated through a test driver that implements the regression testing feature. Examples of tools that provide test data management and regression testing features are AUTORETEST [ASDS79], DRIVER [ASDS79], and MSEF [Eane79].

Project Management. The APSE taxonomy includes three types of project management features. The first, cost estimation, has been the subject of concentrated research. Several models [DACS79] [Boeh81] have been developed to estimate cost. Examples of tools that implement models are SLIM [Putn81] and PRICE-S [Frei79]. Because cost estimation is dependent on the practices and experience of a software development organization, the tools include many tailoring parameters that can be set and tuned for the specific organization.

The second and third project management features are scheduling and tracking. Scheduling assesses and predicts cost expenditures, personnel activity, and task relationships for a development effort. Tracking provides a mechanism for recording and retrieving cost expenditures, personnel activity, task relationships, project status reports, productivity measurements, and resource utilization. Output from tools with these features usually includes varied graphical representations including activity diagrams, milestone charts, and histograms. Using the data produced by cost estimation, these tools can show actual costs versus predicted labor and cost expenditures. Using data from the information management features, productivity measures such as lines of code per day, lines of re-used code, changes in error and modification rates, average program size and complexity, and average execution time can be provided. Examples of tools that provide scheduling and tracking features include ASA-PMS [Houg82], PAC II [ASDS79], and PDSS [ASDS79].

Static Analysis

Static analysis is addressed to a limited extent by the ALS and the AIE. Most of the features provided are a result of compiling or linking. Since it is important to uncover errors as early in software development as possible, the APSE taxonomy includes features that provide static analysis earlier in the life cycle, including the analysis of specification languages.

Static analysis should be provided at any level indicated by the user and could include library analysis, package analysis, task analysis, program analysis, or subprogram analysis. The extent of analysis provided is dependent on the information available. That is, analysis may not be complete until code is available and compiled. For example, complete auditing of Ada package utilization requires an analysis of the effects of elaboration. Since this is best handled at the intermediate code level, auditing may not be complete until all Ada code is present in the package.

STATIC ANALYSIS

ALS	AIE	APSE Taxonomy
Type Analysis	Type Analysis	Type Analysis
Interface Analysis	Interface Analysis	Interface Analysis
Statistical Profiling	-	Statistical Profiling
Cross Reference	Cross Reference	Cross Reference
-	-	Auditing
-	-	Complexity Measurement
-	-	Completeness Checking
-	-	Consistency Checking
-	-	Structure Checking
-	-	Reference Analysis

Table 2. Static Analysis

Type analysis and interface analysis are provided by both the ALS and the AIE as a result of the Ada language requirements. However, separate compilation of procedures can delay interface checking until the procedures are linked. This is not a recommended practice because it uncovers errors very late in the development process. With knowledge of the requirements, design, and data definitions, both analyses can be performed earlier in the development process. Most VHLL (Very High Level Language) processing tools, in particular those that process requirements and design languages [Alfo80] [Teic77] [Pena81] provide these features.

Statistical profiling of statement types used by Ada programs can provide useful information to language designers and managers. Managers can use this information to determine programming style among their staff. For example, the data could show that a programmer avoids the package construct, uses many GOTO´s, or often uses the inline pragma. Language designers and standards committees find statistical profiles useful in identifying popular and unpopular language constructs. The usefulness of statistical profiles is emphasized in a paper by Knuth [Knut71]. The ALS provides statistical profiles for compilation units (e.g. separately compiled procedures and packages), but does not provide this feature at higher levels.

Cross reference is a common feature provided by many compilers, including the ALS and AIE Ada compilers. Although cross references are easily generated from symbol tables produced by compilers, they are not often used because of the overwhelming amount of information that is provided. Even cross reference listings of small programs can be quite lengthy. This problem can be alleviated through a retrieval mechanism that could be provided by the data dictionary. Cross references for design and requirement specifications could be provided in the same manner.

Auditing for Ada programs could include such checks as packages opened (WITH) but never referenced, packages referenced (USE) but never used, missing parts of procedures and packages, use of non-standard or non-portable constructs, dangerous practices (e.g., suppression of exception checks), poor programming practice (e.g., excessive use of GOTO's, no introductory comments, oversized modules), and general information (e.g., overloading conditions). Examples of tools which provide this capability for other languages include FCA [Smit76], AUDIT [Culp75], PBASIC [Hopk80], and PFORT [Dona80].

Complexity Measurement is a technique which can be applied to Ada designs and programs. Two recognized techniques for measuring complexity are McCabe's Cyclomatic Complexity [McCa76] and Halstead's Software Science [Hals77]. Because McCabe's measure is based on structure, it can be easily applied to detailed software designs. However, little progress has been made in measuring complexity of requirements or high level designs, areas where the payoff potential may be the greatest. Tools which provide complexity measurement include DARTS [Houg82] and SAP/H [Houg82].

Completeness checking and consistency checking are two features that are applied across and within a development stage of the software life cycle. In order to automatically test for completeness or consistency of requirements and designs, a formal specification language is required. Tools which use formal proofs to demonstrate consistency are available but most are still considered research tools. Tools which perform consistency and/or completeness checking include AFFIRM [Thom81], PSL/PSA [Teic77], and REVS [Alfo80].

A feature often provided by compilers is structure checking. Structure checking is a feature that recognizes flaws in a program or design structure. Results of a survey of compilers [Shad82] found occurrences of the following types of structure checking in Fortran and Cobol Compilers: unreachable statements, null-transfer statements (i.e., a branch to the next statement), null-body loops, empty programs, and self-transfer statements (infinite loops). In addition to these types of structure checking, there are also tools that check for violation of structured programming constructs (e.g., COBOL STRUCT [FSWE80]), and in some cases these tools perform restructuring of the code (e.g., The Engine [Lyon81]).

Reference analysis detects errors in the definition and use of data. To completely check for this type of error, it is necessary to generate a program graph and to check data references on each path. With this level of checking, errors such as variables defined but never used, variables used but never defined, and variables defined and then subsequently redefined before being used can be checked on all paths through a program. The tool which promulgated this technique is DAVE [Oste76].

Dynamic Analysis.

Dynamic analysis is provided by the debugger in the ALS and the AIE. Two performance related features, tuning and timing analysis, are also planned for the ALS. Tuning and timing data are useful for determining how many times and how long parts of programs are being executed. Tuning and timing analysis should also provide data on concurrency of tasks, that is, it would be useful to know the times that tasks are initiated, completed, and executing concurrently. Three other dynamic analysis techni ues that are useful testing features are included in the APSE Taxonomy. These features are assertion checking, coverage analysis, and regression testing. All three of these features are performed in conjunction with the test data management capability which was discussed previously. Examples of regression testing tools were also given previously.

DYNAMIC ANALYSIS

ALS	AIE	APSE Taxonomy
Timing Analysis	-	Timing Analysis
Tuning Analysis	-	Tuning Analysis
Tracing/Debugging	Tracing/Debugging	Tracing/Debugging
-	-	Regression Testing
-	-	Assertion Checking
-	-	Coverage Analysis

Table 3. Dynamic Analysis

Assertion checking was a feature that was at one time part of the Ada language (Preliminary Ada), but was taken out in later revisions. Assertions, which can be implemented as special comments in a program, are useful for several purposes. Their most informal use is as an understanding mechanism. Assertions can be used to declare relationships or states that are assumed to be true at certain points in a program. Thus, a programmer can use assertions for debugging. Assertions can also represent relations or states assumed true at a higher level of abstraction. Using assertions in this manner allows one to test consistency between the design and code. If assertions are placed in the code so that there is a mapping from each assertion to each design specification or to each requirements specification, then assertions can be used to compute design or requirements coverage. Tools which provide assertion checking include RXVP80* [Saib81], Fortran-77 Analyzer [TRW81], and PET [Stuc76].

---

* RXVP80 is a registered trademark of General Research Corporation.

Coverage analysis is a traditional feature provided by dynamic analysis tools. Because of the longevity, many tools have been developed and enhanced to provide coverage analysis. Since coverage analysis provides managers with actual testing metrics, the most important effect that it has is that it encourages programmers to develop testable software. That is, in order to get a high percentage of program coverage, a programmer must try to make all parts of the program accessible for testing. [Houg82] lists 40 coverage analysis tools for FORTRAN, JOVIAL, COMPASS, COBOL, PL/1 and Pascal. Through use of the tuning feature, the ALS partially supports coverage analysis by identifying parts of a program that have not been executed. However, the ALS does not compute coverage percentages.

Transformation.

Because the ALS and the AIE are required to provide the minimal programming support for Ada, they have a majority of the transformation features necessary to support other features in the APSE Taxonomy. Syntax-directed editing is the only exception. This feature should use tool primitives that are necessary to support compilation, i.e. lexical analysis, syntax analysis, and data base management. Syntax-directed editing is a capability that has been the subject of recent research. The result has been the production of several viable prototypes [Arch81] [Feil81] [Fisc81] [Teit81] [Wilc76].

TRANSFORMATION

ALS	AIE	APSE Taxonomy
Formatting	Formatting	Formatting
Optimization	Optimization	Optimization
Compilation	Compilation	Compilation
Instrumentation	-	Instrumentation
-	-	Editing
-	-	Syntax Direction

Table 4. Transformation

Input/Output.

Subject Input. Table 5 shows that the addition of specification languages is the only change in the main input (or subject) of the APSE when compared to the ALS and the AIE. This statement is true only because most of the additional functions that require input are included with other categories. For example, since assertions are represented as comments in Ada code they are included with code input.

SUBJECT

ALS	AIE	APSE Taxonomy
Text Input	Text Input	Text Input
Data Input	Data Input	Data Input
Code Input	Code Input	Code Input
Ada Code	Ada Code	Ada Code
-	-	VHLL Input
-	-	Specification Languages

Table 5. Subject

Control Input. Table 6 presents the control features. The additional functions that require control input are also included with other categories. For example, commands to initiate static and dynamic features are included with the command category. Pipes, the exception, refer to the command piping mechanism that is attributed to Unix (TM) [Ritc74].

CONTROL

ALS	AIE	APSE Taxonomy
Parameters	Parameters	Parameters
Commands	Commands	Commands
Command Procedures	Command Procedures	Command Procedures
-	Pipes	Pipes

Table 6. Control

User Output. Several output features have been added to the APSE Taxonomy. Graphics includes output such as, activity diagrams, milestone charts, tree diagrams, message charts, and state charts. Much of the graphics output is attributed to the added project management features. Other output, except on-line assistance, is included with other user output features.

USER OUTPUT

ALS	AIE	APSE Taxonomy
Diagnostics	Diagnostics	Diagnostics
Listings	Listings	Listings
Text	Text	Text
Tables	Tables	Tables
-	-	Graphics
On-Line Assistance	On-Line Assistance	On-Line Assistance
Command Assist	Command Assist	Command Assistance
-	-	Error Assistance
-	-	On-Line Tutor
-	-	Definition Assistance
-	-	Menu Assistance

Table 7. User Output

On-line assistance is a user interface feature that is part of the input/output process of the APSE. The APSE taxonomy includes on-line assistance capabilities beyond the command assistance that is provided by the ALS and the AIE. It includes error assistance, on-line tutoring, definition assistance, and menu assistance. These additional features tend to make the user interface more friendly to the unfamiliar user and more helpful to the familiar user.

The most important aspect of on-line assistance is that it should be context sensitive. That is, if the APSE has just issued an error message to a user and the user responds with "help error", then the system should assume that the error message requires further explanation. This same technique, when applied to command assistance, allows the first request to be a brief mind-jogging message that aids the more experienced users. The second request can go into more detail. Menu assistance should also be context sensitive in that it should only identify those commands that can be issued from the current state. For example, if a user is working with the editor, the "help menu" should list only editor commands.

Definition assistance is the ability to make queries on the use of terminology (i.e., "help define KAPSE"). Definition assistance should not only include terms associated with the Ada environment but also with the Ada language, such as "generic", "pragma", "overloading", "package", etc. Further options of definition assistance should be a browse capability, an ability to add new terms, and the presentation of an example, if applicable.

One of the most difficult problems for a new user of a system is simply getting started. Lists of commands and command descriptions do not help the new user because for many systems the amount and depth of the information provided can be overwhelming. What is more advantageous for a new user is a step-by-step tutorial introduction with exercises and a capability to try out various commands without doing harm to oneself or any other users on the system.

An example of a system which provides context sensitive, query-in-depth command, error, and definition assistance is discussed in [Rell81]. Example dialog is presented in [Rell81a]. A system which also includes much user assistance along with an on-line tutor is the SIGMA message processing service [Roth79].

Machine Output. Table 8 presents the output from the APSE which is destined for other machine environments. Since the main purpose of the APSE is for the development of embedded code for other systems, object code for the target system is the only machine output. The NBS/ICST taxonomy includes additional classes of machine output, but since the APSE is a self-contained environment, the user is not aware of these other classes and they are not included.

MACHINE OUTPUT

ALS	AIE	APSE Taxonomy
Object Code	Object Code	Object Code

Table 8. Machine Output

3. Tool Primitives

Each of the features in the APSE taxonomy implies the existence of basic tools or tool primitives that support the feature. Tool primitives are the basic building blocks that support the features that are available to the user. All features in themselves imply at least one tool primitive; for example, cost estimation implies a cost estimizer, auditing implies an auditor, coverage analysis implies a coverage analyzer. Some features imply the existence of primitives that are not directly part of the taxonomy or that support more than one taxonomy feature. The following list includes these primitives followed by the features that they support:

Data Base Manager - supports most APSE features especially information management and project management.

Graph Generator, Graph Analyzer, Data Flow Analyzer - support complexity measurement (McCabe´s measure), structure checking, reference analysis, and optimization by constructing, traversing, and analyzing a specification of the program graph.

Profile Generator - supports complexity measurement (Halstead's measure) and statistical profiling.

Comparator - supports regression testing and configuration management.

Compiler primitives (lexical analyzer, syntax analyzer, semantic analyzer, parser, and symbol table generator) - support compilation and other features that deal directly with the languages accepted by the APSE.

Program Instrumenter - supports assertion checking, coverage analysis, timing analysis. Works in conjunction with the compiler primitives.

Command Interpreter - supports control features.

State Tracker - supports context sensitive on-line assistance by keeping track of the current state of the user. Also supports "split screen" capabilities and background tasks.

Because each of these primitives supports more than one feature in the APSE, their output should become part of the database. For example, the coverage analyzer should not perform lexical and syntax analysis of a program if it has already been performed by the compiler. The coverage analyzer should instead use the intermediate form (e.g., Diana) of the program. This information should be available in the database.

4. Summary of the APSE Taxonomy

INPUT

<u>Subject</u>

- Text Input
- Data Input
- Code Input
  - Ada Code Input
- VHLL Input
  - Specificiations

<u>Control Input</u>

- Parameters
- Commands
- Command Procedures
- Pipes

FUNCTION

<u>Transformation</u>

- Formatting
- Optimization
- Compilation
- Instrumentation
- Editing
  - Syntax Direction

<u>Management</u>

- Configuration Control
- Information Management
  - Ada Library Mgt
  - Specification Mgt
  - Data Dictionary Mgt
  - Ada Package Mgt
  - Test Mgt
- Project Management
  - Cost Estimation
  - Scheduling
  - Tracking

<u>Static Analysis</u>

- Type Analysis
- Interface Analysis
- Statistical Profiling
- Cross Reference
- Auditing
- Complexity Measurement
- Completeness Checking
- Consistency Checking
- Structure Checking
- Reference Analysis

<u>Dynamic Analysis</u>

- Timing Analysis
- Tuning Analysis
- Tracing/Debugging
- Regression Testing
- Assertion Checking
- Coverage Analysis

OUTPUT

<u>User Output</u>

- Diagnostics
- Listings
- Text
- Tables
- Graphics
- On-Line Assistance
  - Command Assistance
  - Error Assistance
  - On-Line Tutor
  - Definition Assistance
  - Menu Assistance

<u>Machine Output</u>

- Object Code

5. References

[AFIP80] AFIPS Taxonomy Committee, "Taxonomy of Computer Science & Engineering", AFIPS Press, 1980.

[Alfo80] M. W. Alford, "Software Requirements Engineering Methodology (SREM) at the Age of Four", Proceedings of COMPSAC 80, October 1980.

[Arch81] J. Archer, Jr., R. Conway, "COPE: A Cooperative Programming Environment", Dept. of Computer Science TR 81-459, Cornell University, June 1981.

[ASDS79] ----------, "Software Tools: Catalogue and Recommendations", Applied Systems Design Section, TRW Defense and Space Systems Group, January 1979.

[Boeh81] B. W. Boehm, "Software Engineering Economics", Prentice- Hall, Englewood Cliffs, N.J., 1981.

[Culp75] L. M. Culpepper, "A System for Reliable Software Engineering", IEEE Transactions on Software Engineering, June 1975.

[DACS79] ----------, "Quantitative Software Models", Data and Analysis Center for Software, SRR-1, March 1979.

[DoD80] ----------, "Requirements for Ada Programming Support Environments, STONEMAN", U.S. Dept. of Defense, February 1980.

[Dona80] John D. Donahoo and Dorothy Swearinger, "A Review of Software Maintenance Technology", Rome Air Development Center, RADC-TR-80-13, February 1980.

[Druf82] L. E. Druffel, Letter to the Editor of Software Engineering Notes, April 1982.

[Eane79] Eanes, Hitchon, Thall, and Brackett, "An Environment for Producing Well-Engineered Microcomputer Software", Proceedings of the 4th International Conference on Software Engineering, September 1979.

[Feil81] P. H. Feiler and R. Medina-Mora, "An Incremental Programming Environment", Proceedings of the 5th International Conference on Software Engineering, March 1981.

[FIPS76] ----------, "Guideline for Planning and Using a Data Dictionary System", Dept. of Commerce FIPS PUB 76, August 1980.

[Fisc81] C. N. Fischer, G. Johnson, and J. Mauney, "An Introduction to Release 1 of Editor Allan Poe", Report, Dept. of Computer Science, University of

Wisconsin-Madison, July 1981.

[Free82] P. Freeman and A. I. Wasserman, "Software Development Methodologies and Ada", Ada Joint Program Office, November 1982.

[Frei79] F. R. Freiman and R. E. Park, "Price Software Model - Version 3: An Overview", Workshop on Quantifiable Software Models, IEEE Cat. No. TH0067-9, October 79.

[FSWE80] ----------"Federal Software Exchange Catalog" General Services Administration, GSA/ADTS/C-80/3, FSWEC-80/0118, September 1980.

[Hals77] M. H. Halstead, "Elements of Software Science", Elsevier - North Holland Pub. Co., New York, 1977.

[Hopk80] T. R. Hopkins, "PBASIC--A Verifier for Basic", Software-Practice and Experience, October 1980.

[Houg82] R. C. Houghton, Jr., "Software Development Tools", NBS Special Publication 500-88, March 1982.

[Knut71] D. Knuth, "An Empirical Study of FORTRAN Programs", Software-Practice and Experience, 1971.

[Lecl82] Y. Leclerc, S. W. Zucker, and D. Leclerc, "A Browsing Approach to Documentation", Computer, June 1982.

[Leon77] B. Leong-Hong and B. Marron, "Technical Profile of Seven Data Element Dictionary/Directory Systems", NBS Special Publication 500-3, February 1977.

[McCa76] T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol SE-2, December 1976.

[Lyon73] G. Lyon, "Static Language Analysis", National Bureau of Standards Technical Note 797, October 1973.

[Lyon81] M. J. Lyons, "Salvaging Your Software Asset", Proceedings of the National Computer Conference, May 1981.

[Ober82] P. A. Oberndorf, Chairman, "Kernel Ada Programming Support Environment (KAPSE) Interface Team: Public Report, Volume 1", Naval Ocean Systems Center, NOSC TD 509, April 1982.

[Oste76] L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs", Software-Practice and Experience, October 1976.

[Paig74] M. R. Paige and J. P. Benson, "The Use of Software Probes in Testing FORTRAN Programs", Computer, July 1974.

[Pene81] M. H. Penedo, "SARA as a Tool for Software Design: Building-Block Modelling and Composition", Proceedings of the NBS/IEEE/ACM Software Tool Fair, R. Houghton, Ed., NBS Special Publication 500-80, October 1981.

[Putn81] L. H. Putnam, "SLIM: A Quantitative Tool for Software Cost and Schedule Estimation", Proceedings of the NBS/IEEE/ACM Software Tool Fair, R. Houghton, Ed., NBS Special Publication 500-80, October 1981.

[Rell81] N. Relles and L. A. Price, "A User Interface for Online Assistance", Proceedings of the 5th International Conference on Software Engineering, March 1981.

[Rell81a] N. Relles and L. A. Price, "A User Interface for Online Assistance", Proceedings of the NBS/IEEE/ACM Software Tool Fair, R. Houghton, Ed., NBS Special Publication 500-80, October 1981.

[Ritc74] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System", Communications of the ACM, July 1974.

[Roth79] Rothenberg, J., "On-Line Tutorials and Documentation for the SIGMA Message Service", Proceeding of the National Computer Conference, 1979.

[Saib81] S. H. Saib, J. P. Benson, C. Gannon, and W. R. DeHaan, "RXVP80 (TM): A Software Documentation, Analysis, and Test System", Proceedings of the NBS/IEEE/ACM Software Tool Fair, R. Houghton, Ed., NBS Special Publication 500-80, October 1981.

[Shad82] M. Shadad, E. Libster, "Compiler Features: A Survey", NBS-GCR-82-418, December 1982.

[Smit76] P. Smith, "Fortran Code Auditor: User´s Manual", Rome Air Development Center, RADC-TR-76-395, December 1976.

[Stuc76] L. G. Stucki, "The Use of Dynamic Assertions to Improve Software Quality", McDonnell Douglas Astronautics Company-West, MDC G6588, November 1976.

[Teic77] D. Teichroew and E. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation of Information Processing Systems", IEEE Transactions on Software Engineering, Vol SE-3, No 1, 1977.

[Teit81] R. T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Communications of the ACM, September 1981.

[Thom81] D. H. Thompson, et. al., "Specification and Verification of Communication Protocols in AFFIRM", USC/Information Sciences Institute, ISI/RR-81-88, February 1981.

[TRW81] ----------, "FORTRAN 77 Analyzer: Users Manual", TRW Sales No. 36098.000, Prepared under contract for the National Bureau of Standards, 1981.

[Wilc76] T. R. Wilcox, et. al., "The Design and Implementation of a Table Driven Interactive Diagnostic Programming System", Communication of the ACM, November 1976.

APPENDIX A

PRIORITIZATION OF TOOL FEATURES

In the earlier drafts of this report, reviewers were asked to comment on the importance of the features included in the APSE taxonomy. Most reviewers accepted the high level prioritization that was implied by the report. That is, tool support for management is the most important area. All reviewers felt that all the features mentioned would be of some use in the APSE.

The table that follows presents a feature level prioritization based on the comments received. Each of the features is presented in the same order as the report and is categorized according to the perceived priority.

	Required	Important	Useful
MANAGEMENT	X	-	-
Configuration Ctrl	X	-	-
Information Mgt	X	-	-
Ada Library Mgt	X	-	-
Specification Mgt	-	X	-
Data Dictionary Mgt	-	X	-
Ada Package Mgt	-	-	X
Test Mgt	-	X	-
Project Mgt	-	X	-
Cost Estimation	-	X	-
Scheduling	-	X	-
Tracking	-	X	-
STATIC ANALYSIS	X	-	-
Type Analysis	X	-	-
Interface Analysis	X	-	-
Statistical Profiling	-	-	X
Cross Reference	X	-	-
Auditing	-	X	-
Complexity Measurement	-	-	X
Completeness Checking	-	-	X
Consistency Checking	-	-	X
Structure Checking	-	X	-
Reference Analysis	-	X	-

	Required	Important	Useful
DYNAMIC ANALYSIS	X	-	-
Timing Analysis	-	X	-
Tuning Analysis	-	X	-
Tracing/Debugging	X	-	-
Regression Testing	-	X	-
Assertion Checking	-	-	X
Coverage Analysis	-	X	-
TRANSFORMATION	X	-	-
Formatting	X	-	-
Optimization	X	-	-
Compilation	X	-	-
Instrumentation	-	X	-
Editing	X	-	-
Syntax Direction	-	X	-
INPUT/OUTPUT	X	-	-
On-Line Assistance	X	-	-
Command Assistance	X	-	-
Error Assistance	X	-	-
On-Line Tutor	-	-	X
Definition Assistance	-	-	X
Menu Assistance	-	X	-

APPENDIX B

THE NBS/ICST TAXONOMY OF SOFTWARE TOOL FEATURES

January 19, 1982

Introduction.

The taxonomy is a hierarchical arrangement of software tool features and is illustrated in Figure 1. The highest level (@) is the most abstract and covers all the features below it. The second level of the taxonomy covers the basic processes of a tool: input (in), function (fn), and output (out). At the third level are the classes of tool features: subject (I), control input (C), transformation (T), static analysis (S), dynamic analysis (D), management (G), user output (U), and machine output (M). The ranges in brackets signify the number of features in each of the classes. At the fourth level or feature level of the hierarchy are a total of 64 tool features. In some cases the fourth level has been further expanded to a fifth level to provide further differentiation (not shown in figure 1).

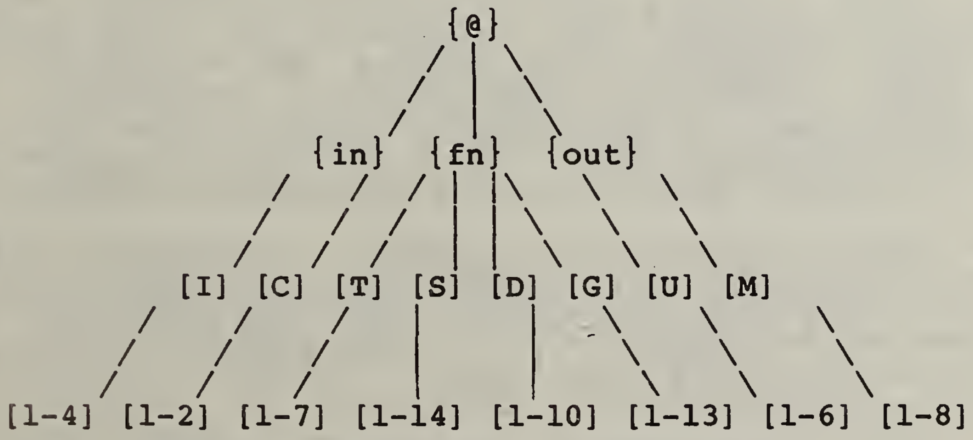

Figure 1. Taxonomy of Tool Features

The Taxonomy.

Input. Tool input features are based on the forms of input which are provided to a tool. These features fall into two classes, one which is based on what the tool should operate on, i.e., the subject, and the other based on how the tool should operate, i.e., the control.

INPUT	
Subject	Control Input
I1. Text	C1. Commands
I2. VHLL	C2. Parameters
I2A. Description Language	
I2B. Requirements Language	
I2C. Design Language	
I3. Code	
I4. Data	

Table 1. Input

a. Subject (Key: I). The subject is usually the main input to a tool. It is the input which is subjected to the main functions performed by a tool. The four types of subjects are text, VHLL (very high level language), code, and data. Although the difference between these types is somewhat arbitrary, the taxonomy has very specific definitions for each.

I1. Text - accepts statements in a natural language form. Certain types of tools are designed to operate on text only (e.g., text editors, document preparation systems) and require no other input except directives or commands.

I2. VHLL - accepts a specification written in a very high level language that is typically not in an executable form. Tools with this feature may define programs, track program requirements throughout their development, or synthesize programs through use of some non-procedural VHLL. There are three recognized types of VHLL´s. Each is briefly described as follows:

I2A. Description Language - accepts a formal language with special constructs used to describe the subject in a high-level non-procedural form. An example of a description language is Backus-Naur Form (BNF).

I2B. Requirements Language - accepts a formal language with special constructs and verification protocols used to specify, verify, and document requirements. Examples of requirements languages include the Problem Statement Language [Teic77] and the Requirements Statement Language [Bell77].

I2C. Design Language - accepts a formal language with special constructs and verification protocols used to represent, verify, and document a design. Design languages are normally procedural, that is, they specify how a program is going to work in an algorithmic manner. An example of a design language is Program Design Language [Cain75].

I3. Code - accepts a program written in a high level language, assember, or object level language. Code is the language form in which most programming solutions are expressed.

I4. Data - accepts a string of characters or numeric quantities to which meaning is or might be assigned. The input (e.g. raw data) is not in an easily interpreted, natural language form. A simulator that accepts numeric data to initialize its program variables is an example of a tool that has data as input.

Some tools, such as editors, operate on any if the four of these input forms. In cases such as this, the input form is chosen from the viewpoint of the tool. Since most editors view the input form as text, the correct subject for this tool is text.

b. Control Input (Key: C). Control inputs specify the type of operation and the detail associated with an operation. They describe any separable commands that are entered as part of the input stream.

C1. Commands - accepts character strings which consist primarily of procedural operators, each capable of invoking a system function to be executed. A directive invoking a series of diagnostic commands (i.e., TRACE, DUMP, etc.) at selected breakpoints is an example. A tool that performs a single function will not have this feature but will most likely have the next.

C2. Parameters - accepts character strings which consist of identifiers that further qualify the operation to be performed by a tool. Parameters are usually entered as a result of a prompt from a tool or may be embedded in the tool input. An interactive trace routine that prompts for breakpoints is an example of a tool with parametric input.

Function. The features for this class are shown in Table 2. They describe the processing functions performed by a tool and fall into four classes: transformation, static analysis, dynamic analysis, and management.

FUNCTION

Transformation	Static Analysis
T1. Editing	S1. Auditing
T2. Formatting	S2. Comparison
T3. Instrumentation	S3. Complexity Measurement
T4. Optimization	S4. Completeness Checking
T5. Restructuring	S5. Consistency Checking
T6. Translation	S6. Cross Reference
T6A. Assembling	S7. Data Flow Analysis
T6B. Compilation	S8. Error Checking
T6C. Conversion	S9. Interface Analysis
T6D. Macro Expansion	S10. Scanning
T6E. Structure Preprocessing	S11. Statistical Analysis
T7. Synthesis	S12. Structure Checking
	S13. Type Analysis
Dynamic Analysis	S14. Units Analysis
	S15. I/O Specification Analysis
D1. Assertion Checking	
D2. Constraint Evaluation	**Management**
D3. Coverage Analysis	
D4. Resource Utilization	G1. Configuration Control
D5. Simulation	G2. Information Management
D6. Symbolic Execution	G2A. Data Dictionary Management
D7. Timing	G2B. Documentation Management
D8. Tracing	G2C. File Management
D8A. Breakpoint Control	G2D. Test Data Management
D8B. Data Flow Tracing	G3. Project Management
D8C. Path Flow Tracing	G3A. Cost Estimation
D9. Tuning	G3B. Resource Estimation
D10. Regression Testing	G3C. Scheduling
	G3D. Tracking

Table 2. Function

a. Transformation (Key: T). Transformation features describe how the subject is manipulated to accommodate the user's needs. They describe what transformations take place as the input to the tool is processed. There are seven transformation features. Each of these features is briefly defined as follows:

T1. Editing - modifying the content of the input by inserting, deleting, or moving characters, numbers, or data.

T2. Formatting - arranging a program according to predefined or user defined conventions. A tool that "cleans up" a program by making all statement numbers sequential, alphabetizing variable declarations, indenting statements, and making other standardizing changes has this feature.

T3. Instrumentation - adding sensors and counters to a program for the purpose of collecting information useful for dynamic analysis [Paig74]. Most code analyzers instrument the source code at strategic points in the program to collect execution statistics required for assertion checking, coverage analysis, or tuning. See D1,D3, and D9.

T4. Optimization - modifying a program to improve performance, e.g. to make it run faster or to make it use fewer resources. Many vendors' compilers provide this feature. There are many tools that claim this feature, but do not modify the subject program. Instead, these tools provide data on the results of execution which may be used for tuning purposes. See D9.

T5. Restructuring - reconstructing and arranging the subject in a new form according to well-defined rules. A tool that generates structured code from unstructured code is an example of a tool with this feature.

T6. Translation - converting from one language form to another. There are five types of translation features. Each is defined as follows:

T6A. Assembling - translating a program expressed in an assembler language into object code.

T6B. Compilation - translating a computer program expressed in a problem-oriented language into object code.

T6C. Conversion - modifying an existing program to enable it to operate with similar functional capabilities in a different environment. Examples include CDC Fortran to IBM Fortran, ANSI Cobol (1968) to ANSI Cobol (1974), and Pascal to Pl/1.

T6D. Macro Expansion - augmenting instructions in a source language with user defined sequences of instructions in the same source language.

T6E. Structure Preprocessing - translating a computer program with structured constructs into its equivalent without structured constructs.

T7. Synthesis - generating programs according to predefined rules from a program specification or intermediate language. Tools that have this feature include program generators, compiler compilers, and preprocessor generators.

b. Static Analysis (Key: S). Static analysis features specify operations on the subject without regard to the executability of the subject [Howd78]. They describe the manner in which the subject is analyzed. There are 15 static analysis features. Each is briefly described as follows:

S1. Auditing - conducting an examination to determine whether or not predefined rules have been followed. A tool that examines the source code to determine whether or not coding standards are complied with is an example of a tool with this feature.

S2. Comparison - determining and assessing similarities between two or more items. A tool that determines changes made in one file that are not contained in another has this feature.

S3. Complexity Measurement - determining how complicated an entity (e.g., routine, program, system, etc.) is by evaluating some number of associated characteristics [Mcca76] [Hals77]. For example, the following characteristics can impact complexity: instruction mix, data references, structure/control flow, number of interactions/interconnections, size, and number of computations.

S4. Completeness Checking - assessing whether or not an entity has all its parts present and if those parts are fully developed [Boeh78]. A tool that examines the source code for missing parameter values has this feature.

S5. Consistency Checking - determining whether or not an entity is internally consistent in the sense that it contains uniform notation and terminology [Walt78], or is consistent with its specification [Robi77]. Tools that check for consistent usage of variable names or tools that check for consistency between design specifications and code are examples of tools with this feature.

S6. Cross Reference - referencing entities to other entities by logical means. Tools that identify all variable references in a subprogram have this feature.

S7. Data Flow Analysis - graphical analysis of the sequential patterns of definitions and references of data [Oste76]. Tools that identify undefined variables on certain paths in a program have this feature.

S8. Error Checking - determining discrepancies, their importance, and/or their cause. Tools used to identify possible program errors, such as misspelled variable names, arrays out of bounds, and modifications of a loop index are examples of tools with this feature.

S9. Interface Analysis - checking the interfaces between program elements for consistency and adherence to predefined rules and/or axioms. A tool that examines interfaces between modules to determine if axiomatic rules for data exchange were obeyed has this feature.

S10. Scanning - examining an entity sequentially to identify key areas or structure. A tool that examines source code and extracts key information for generating documentation is an example of a tool with this feature.

S11. Statistical Analysis - performing statistical data collection and analysis. A tool that uses statistical test models to identify where programmers should concentrate their testing is one example. A tool that tallies occurrences of statement types is another example of a tool with this feature.

S12. Structure Checking - detecting structural flaws within a program (e.g. improper loop nestings, unreferenced labels, unreachable statements, and statements with no successors).

S13. Type Analysis - evaluating whether or not the domain of values attributed to an entity are properly and consistently defined. A tool that type checks variables has this feature.

S14. Units Analysis - determining whether or not the units or physical dimensions attributed to an entity are properly defined and consistently used. A tool that can check a program to ensure variables used in computations have proper units (e.g. hertz = cycles/seconds) is an example of a tool with this feature.

S15. I/O Specification Analysis - analyzing the input and output specifications in a program usually for the generation of test data. A tool that analyzes the types and ranges of data that are defined in an input file specification in order to generate an input test file is an example of a tool with this feature.

c. Dynamic Analysis (Key: D). Dynamic analysis features specify operations that are determined during or [illegible] ion takes place [Howd78a]. Dynamic analysis features differ from those classified as static by virtue of the fact that they require some form of symbolic or machine execution. They describe the techniques used by the tool to derive meaningful information about a program´s execution behavior. There are 10 dynamic analysis features. Each is briefly described as follows:

D1. Assertion Checking - checking of user-embedded statements that assert relationships between elements of a program. An assertion is a logical expression that specifies a condition or relation among the program variables. Checking may be performed with symbolic or run-time data. Tools that test the validity of assertions as the program is executing or tools that perform formal verification of assertions have this feature.

D2. Constraint Evaluation - generating and/or solving path input or output constraints for determining test input or for proving programs correct [Clar76]. Tools that assist in the generation of or automatically generate test data have this feature.

D3. Coverage Analysis - determining and assessing measures associated with the invocation of program structural elements to determine the adequacy of a test run [Fair78]. Coverage analysis is useful when attempting to execute each statement, branch, path, or iterative structure (e.g., DO loops in FORTRAN) in a program. Tools that capture this data and provide reports summarizing relevant information have this feature.

D4. Resource Utilization - analysis of resource utilization associated with system hardware or software. A tool that provides detailed run-time statistics on core usage, disk usage, queue lengths, etc. is an example of a tool with this feature.

D5. Simulation - representing certain features of the behavior of a physical or abstract system by means of operations performed by a computer. A tool that simulates the environment under which operational programs will run has this feature.

D6. Symbolic Execution - reconstructing logic and computations along a program path by executing the path with symbolic rather than actual values of data [Darr78].

D7. Timing - reporting actual CPU, wall-clock, or other times associated with parts of a program.

D8. Tracing - monitoring the historical record of execution of a program. There are three types of tracing features. Each is described as follows:

D8A. Breakpoint Control - controlling the execution of a program by specifying points (usually source instructions) where execution is to be interrupted.

D8B. Data Flow Tracing - monitoring the current state of variables in a program. Tools that dynamically detect uninitialized variables or tools that allow users to interactively retrieve and update the current values of variables have this feature.

D8C. Path Flow Tracing - recording the source statements and/or branches that are executed in a program in the order that they are executed.

D9. Tuning - determining what parts of a program are being executed the most. A tool that instruments a program to obtain execution frequencies of statements is a tool with this feature.

D10. Regression Testing - rerunning test cases which a program has previously executed correctly in order to detect errors spawned by changes or corrections made during software development and maintenance. A tool that automatically "drives" the execution of programs through their input test data and reports discrepancies between the current and prior output is an example of a tool with this feature.

d. Management (Key: G) - Management features aid the management or control of software development. There are three types of management features. Each is described as follows:

G1. Configuration Control - aiding the establishment of baselines for configuration items, the control of changes to these baselines, and the control of releases to the operational environment.

G2. Information Management - aiding the organization, accessibility, modification, dissemination, and processing of information that is associated with the development of a software system.

G2A. Data Dictionary Management - aiding the development and control of a list of the names, lengths, representations, and definitions of all data elements used in a software system.

G2B. Documentation Management - aiding the development and control of software documentation.

G2C. File Management - providing and controlling access to files associated with the development of software.

G2D. Test Data Management - aiding the development and control of software test data.

G3. Project Management - aiding the management of a software development project. Tools that have this feature commonly provide milestone charts, personnel schedules, and activity diagrams as output.

G3A. Cost Estimation - assessing the behavior of the variables which impact life cycle cost. A tool to estimate project cost and investigate its sensitivity to parameter changes has this feature.

G3B. Resource Estimation - estimating the resources attributed to an entity. Tools that estimate whether or not memory limits, input/output capacity, or throughput constraints are being exceeded have this feature.

G3C. Scheduling - assessing the schedule attributed to an entity. A tool that examines the project schedule to determine its critical path (shortest time to complete) has this feature.

G3D. Tracking - tracking the development of an entity through the software life cycle. Tools used to trace requirements from their specification to their implementation in code have this feature.

Output. Output features, which provide the link from the tool to the user, are illustrated in Table 3. They describe what type of output the tool produces for both the human user and the target machine (where applicable). Again using a compiler as an example, the user output would be diagnostics and possibly listings and tables (cross reference), and the machine output would be object code or possibly intermediate code.

OUTPUT

User Output		Machine Output	
U1.	Computational Results	M1.	Assembly Language
U2.	Diagnostics	M2.	Data
U3.	Graphics	M3.	Intermediate Code
U3A.	Activity Diagrams	M4.	Object Code
U3B.	Data Flow Diagrams	M5.	Prompts
U3C.	Design Charts	M6.	Source Code
U3D.	Histograms	M7.	Text
U3E.	Milestone Charts	M8.	VHLL
U3F.	Program Flow Charts		
U3G.	Tree Diagrams		
U4.	Listings		
U5.	Tables		
U6.	Text		

Table 3. Output

a. User Output (Key: U). User output features describe the types of information that are returned from the tool to the human user and the forms in which these outputs are presented. There are six user output features. Each is briefly described as follows:

U1. Computational Results - output that simply presents the result of a computation. The output is not in an easily interpreted natural language form (e.g. text or tables).

U2. Diagnostics - output that simply indicates what software discrepancies have occurred. An error flag from a compiler is an example.

U3. Graphics - a graphical representation with symbols indicating operations, flow, etc. There are seven types of graphics output. Each is described as follows:

U3A. Activity Diagrams - diagrams presenting actions or states of a software development activity and their interrelationships. Also, diagrams representing or summarizing the major aspects of system performance. Examples of activity diagrams are control diagrams, pert charts, and system profiles.

U3B. Data Flow Diagrams - diagrams that represent the path of data in the solving of a problem and that define the major phases of the processing as well as the various data media used. Examples of data flow diagrams are Jackson diagrams [Jack75], DFD's [DeMa78], and Bubble Charts [Your75].

U3C. Design Charts - charts which represent the software architecture components, modules, and interfaces for a software system. Examples of design charts are HIPO diagrams, structure charts and block diagrams.

U3D. Histograms - graphic representations of frequency distributions where the graph height is proportional to a class frequency. Tools that perform statistical analysis or coverage analysis often provide histograms.

U3E. Milestone Charts - a chart which represents the schedule of events used to measure the progress of software development efforts. Milestone charts are often provided by project management tools.

U3F. Program Flow Charts - graphical representation of the sequence of operations in a computer program. Examples of program flow charts include FIPS Flow Charts [FIPS72], Chapin Charts [Chap74], and Nassi-Shneiderman Charts [Nass73].

U3G. Tree Diagrams - diagrams that represent the hierarchical structure of software modules or data and is generated from a root node. A tree diagram does not show iteration or cycles. An example of a tree diagram is a "call tree" or module hierarchical diagram.

U4. Listings - an output from the tool is a computer listing of a source program or data and may be annotated. Many different forms of listings can be generated. Some may be user controlled through directives.

U5. Tables - an output from the tool is arranged in parallel columns to exhibit a set of facts or relations in a definite, compact and comprehensive form. A tool that produces a decision table identifying a program´s logic (conditions, actions, and rules that are the basis of decisions) is an example.

U6. Text - an output from the tool is in a natural language form. The output may be a choice of many different types of reports and the formats may be user defined.

b. Machine Output (Key: M). Machine output features handle the interface from the tool to either another tool or a machine environment. They describe what a machine or tool expects to see as output. There are eight machine output features. Each is briefly described as follows:

M1. Assembly Code - a low level code whose instructions are usually in one-to-one correspondence with computer instructions.

M2. Data - a set of representations of characters or numeric quantities to which meaning has been assigned. A tool generating input to a plotter is an example.

M3. Intermediate Code - code that is between source code and assembly code. A tool producing P-code for direct machine interpretation is an example.

M4. Object Code - code expressed in machine language which is normally an output of a given translation process. A tool producing relocatable load modules for subsequent execution is an example.

M5. Prompts - a series of procedural operators that are used to interactively inform the system in which the tool operates that it is ready for the next input.

M6. Source Code - code written in a high level procedural language that must be input to a translation process before execution can take place.

M7. Text - statements in a natural language form. A tool producing English text which is passed to a word processor is an example.

M8. VHLL - statements written in a very high level language. A tool which produces a requirements language or design language for use by another tool is an example.

Background

The initial development of the taxonomy was performed under contract [1] to the National Bureau of Standards. Since its initial development, the taxonomy has evolved through in-house and public review. The first public review of the taxonomy took place at a workshop held at NBS on 11 April 1980. Continued development by ICST has clarified the definitions and removed several inconsistencies. This taxonomy was released for general public review through two NBS reports [Houg81a] [ICST81] and two conference papers [Houg81b] [Reif81a].

The workshop and the response to the publications led to the formation of a review group composed of people from industry, Government, and academe. Comments from this group plus

---

[1] Contract NB79SBCA0273 to SoHar, Inc. and SoHaR Subcontract No. 102 to Software Management Consultants.

adjustments made as a result of in-house classification experience essentially formed the taxonomy that is reported in this document. This revision was distributed to the review group for approval on July 16, 1981. The comments received with discussion, resolution, and summary statistics for each proposed change are available from the Institute for Computer Sciences and Technology, National Bureau of Standards, Technology Bldg., Room B266, Washington, DC 20234.

Expansion Issues

Although it would be structurally satisfying to have all features expanded to the same level (it would keep the taxonomy looking like a trimmed pine), there are many features which represent relatively new technology that make this task impossible. These areas are maturing and not yet well understood. For example, features such as synthesis, restructuring, complexity measurement, cost estimation, regression testing, and constraint evaluation are the subject of current research. Expansion of these features to the same level as features such as translation, tracing, or project management is premature.

Another reason for expanding to a lower level is to keep the granularity of the taxonomy consistent at the bottom level. For example, translation and instrumentation are both transformation features at the fourth level. Translation, however is broader in scope than instrumentation. Consequently, translation is further expanded to keep the granularity of its features consistent with other transformation features.

Missing Features

On occasion, a tool may have a characteristic that can not be easily classified with the taxonomy. It may be that the taxonomy is missing a feature. When this situation occurs, the classification for this tool remains at the lowest level possible. For example, it has been proposed that test generation and query languages be added to VHLL Input. Since these languages are not currently part of the taxonomy, the classification would not proceed below VHLL Input. In a future review of the taxonomy, test generation and query languages will be reviewed for need, desirability, and appropriateness in the taxonomy.

All occurrences of missing features are important to ICST. The taxonomy reflects tool technology at the time of revision. New features and new tools may require future changes to the taxonomy to keep it up to date. Anyone with comments or problems relating to the taxonomy is encouraged to forward them to ICST. The address is Institute for Computer Sciences and Technology, National Bureau of Standards, Technology Bldg., Room B266, Washington, DC, 20234.

References.

[Bell77] T. E. Bell, D. C. Bixler and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering", IEEE Transactions on Software Engineering, Vol SE-3, No 1, 1977.

[Boeh78] B. W. Boehm, J R. Brown, H. Kaspar, M. Lipow, G. J. MacLeod and M. J. Merritt, "Characteristics of Software Quality", North-Holland Publishing Company, NY, 1978.

[Cain75] S. H. Caine and E. K. Gordon, "PDL: A Tool for Software Design", Proceedings of the National Computer Conference, 1975.

[Chap74] N. Chapin, "New Format for Flowcharts", Software-Practice and Experience, Oct.-Dec. 1974.

[Clar76] L. A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering, Vol SE-2, September 1976.

[Darr78] J. A. Darringer and J. C. King, "Applications of Symbolic Execution to Program Testing", Computer, April 1978.

[DeMa78] T. DeMarco, "Structured Analysis and System Specification", Prentice-Hall, Inc., 1978.

[Fair78] R. E. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software", Computer, April 1978.

[FIPS72] U. S. Dept. of Commerce, "Flowchart Symbols and Their Usage in Information Processing", National Bureau of Standards FIPS-PUB-24, June 1972.

[Hals77] M. H. Halstead, "Elements of Software Science", Elsevier - North Holland Pub. Co., New York, 1977.

[Houg81a] R. Houghton, "Features of Software Development Tools", NBS Special Publication 500-74, February 1981.

[Houg81b] R. Houghton, "An Inverted View of Software Development Tools", Proceedings of the 20th Annual Technical Symposium of the Washington, D.C. Chapter of the ACM, June 1981.

[Howd78] W. E. Howden, "A Survey of Static Analysis Methods", Tutorial: Software Testing and Validation Techniques, IEEE Cat. No. EHO138-8, 1978.

[Howd78a] W. E. Howden,"A Survey of Dynamic Analysis Methods", Tutorial: Software Testing and Validation Techniques, IEEE Cat. No. EH0138-8, 1978.

[ICST81] ----------,"Software Development Tools: A Reference Guide to a Taxonomy of Tool Features", U. S. Department of Commerce, LC-1127, February 1981.

[Jack75] M. A. Jackson, "Principle of Program Design" Academic Press, 1975.

[Mcca76] T. J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol SE-2, December 1976.

[Nass73] I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming", SIGPLAN Notices of the ACM, August 1973.

[Oste76] L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation Error Detection and Documentation System for FORTRAN Programs", Software˜Practice and Experience, October 1976.

[Paig74] M. R. Paige and J. P. Benson, "The Use of Software Probes in Testing FORTRAN Programs", Computer, July 1974.

[Reif80] D. Reifer and H. Montgomery, "Final Report, Software Tool Taxonomy", Software Management Consultants Report No. SMC-TR-004, June 1980.

[Reif81a] D. Reifer, "Tool Standards-It´s About Time", Proceedings of the Software Engineering Standards Application Workshop, IEEE No. 81CH1633-7, August 1981.

[Robi77] L. Robinson and K. N Levitt, "Proof Techniques for Hierarchically Structured Programs", Communications of the ACM, April 1977.

[Teic77] D. Teichroew and E. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation of Information Processing Systems", IEEE Transactions on Software Engineering, Vol SE-3, No 1, 1977.

[Walt78] G. Walters and J. McCall, "The Development of Metrics for Software Reliability and Maintainability", Proceedings of the Annual Reliability and Maintainability Symposium, January 1978.

[Warn76] J-D. Warnier, "Logical Construction of Programs", New York: Van Nostrand Reinhold Co., 1976.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. BIBLIOGRAPHIC DATA SHEET (See instructions)	1. PUBLICATION OR REPORT NO. NBSIR 82-2625	2. Performing Organ. Report No.	3. Publication Date February 1983

4. TITLE AND SUBTITLE

A Taxonomy of Tool Features for the Ada Programming Support Environment (APSE)

5. AUTHOR(S)

Raymond C. Houghton, Jr.

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

Final Report

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

Department of Defense
Ada Joint Program Office
801 N. Randolph, Suite 1210
Arlington, VA 22209

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 categorization of the software development tool features of the Ada Programming Support Environment (APSE) is presented. The features of two Ada environments, the Ada Language System (ALS) and the Ada Integrated Environment (AIE), are compared. The underlying features of the APSE are presented.

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

Ada Programming Support Environment; APSE; software development; software engineering; software tools; taxonomy

13. AVAILABILITY

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

14. NO. OF PRINTED PAGES

31

15. Price

$8.50