(11)



# STRATEGY FOR A

# DOD SOFTWARE INITIATIVE

## VOLUME II: APPENDICES


Department of Defense

1 October 1982

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| REPORT DOCUMENTATION PAGE | READ INSTRUCTIONS BEFORE COMPLETING FORM |
|---|---|
| 1 REPORT NUMBER | 2 GOVT ACCESSION NO. AD-A121738 | 3 RECIPIENT'S CATALOG NUMBER |
| 4 TITLE (and Subtitle) Strategy for a DOD Software Initiative Volume II: Appendices | 5 TYPE OF REPORT & PERIOD COVERED Planning document |
| | 6 PERFORMING ORG. REPORT NUMBER |
| 7 AUTHOR(s) Larry Druffel | 8 CONTRACT OR GRANT NUMBER(s) |
| 9 PERFORMING ORGANIZATION NAME AND ADDRESS Office of Under Secretary of Defense (Research and Advanced Technology) Ada Joint Program Office | 10 PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS |
| 11 CONTROLLING OFFICE NAME AND ADDRESS | 12 REPORT DATE October 1, 1983 |
| | 13 NUMBER OF PAGES 321 |
| 14 MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) | 15 SECURITY CLASS. (of this report) Unclassified |
| | 15a DECLASSIFICATION DOWNGRADING SCHEDULE |

16 DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

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

18 SUPPLEMENTARY NOTES

Refined planning document will be forthcoming in 1983.

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software
Computer technology
Military systems
Strategic planning
Programmable digital computers

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

This volume contains appendices providing background information related to the strategy and plan in the main volume. It includes a brief history and acknowledgements, assessments of the opportunities the Initiative might pursue, technical visions of two possible futures in the early 1990's, summaries of related Defense Science Board recommendations, summaries of foreign initiatives, summary of Joint Task Force identified software problems, summary of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Software Technology Initiative questionnaire results and an estimate of potential software productivity gains. This volume is background material and does not necessarily reflect the opinion of DOD.

## FOREWARD TO APPENDICES

These Appendices provide background detail to the Strategy for a DoD Software Initiative. The appendices were prepared by individual experts and represent their particular points of view. In most cases, these appendices have been reviewed by other experts, and sometimes revised by yet others. However, the views expressed in these appendices are not necessarily extensions of the plan and must be read as representing particular individual or composite points of view. In some cases the point of view is expressed by an individual expert who has little or no experience with DoD systems but has extensive technical expertise. Characterizations of DoD practice may be expressed as extensions of general practice and not accurately reflect the actual DoD practice on life critical systems. The views expressed in these Appendices do not necessarily reflect the views of the DoD.

TABLE OF CONTENTS

# A P P E N D I X  I

## HISTORY OF THE SOFTWARE

## INITIATIVE/ACKNOWLEDGEMENTS

## APPENDIX I

## HISTORY OF THE SOFTWARE INITIATIVE/ACKNOWLEDGEMENTS

The concept of a software-oriented initiative was first introduced by Dr. Ruth Davis, former Deputy Under Secretary of Defense for Research and Engineering (Research and Advanced Technology), in her testimony before the House Armed Services Subcommittee on Research and Development on 5 April 1979. Since then, many related activities have occurred.

As a result of these activities, a strategy for a broadly based, yet carefully coordinated, software-oriented initiative has emerged and is presented in this document. The strategy is the product of critical examination of input from many sources. The activity preceding and related to this strategy is briefly reviewed in this Appendix to show the diversity and amount of effort involved.

The first activity was a DoD-sponsored workshop, held at Fort Belvoir, Virginia, in May 1980. The purpose of the workshop was to identify a basis for undertaking the proposed "software technology initiative." The results of the workshop were a proposed management structure and proposed preliminary goals and thrusts.

Subsequently, a Software Technology Coordinating Committee was formed by the Office of the Secretary of Defense to provide direction and oversight to the proposed software initiative. The formation of the Coordinating Committee brought together key individuals from throughout DoD and the individual services. A management-oriented Program Definition Plan was drafted by the Coordinating Committee and distributed for comment in July 1980. Additionally, the Coordinating Committee placed an announcement in the 30 July issue of the Commerce Business Daily (CBD) requesting recommendations and qualification statements by 31 August 1980. Over eighty replies were received, and the comments and criticisms were reviewed for their merit. In an effort to reach a greater, more diverse audience—especially the academic and private research communities—the CBD announcement was repeated in the December 1980 issue of the Communications of the ACM. Approximately forty replies were received from this announcement.

Ideas extracted from replies to the announcements and other sources were collected in Candidate R&D Thrusts for the Software Technology Initiative[1] and circulated among all segments of the DoD/academic/industry community in the summer of 1981. A

questionnaire was included for readers to evaluate the effectiveness and desirability of the candidate thrusts and suggest other possibilities. The questionnaire was returned by 146 respondents; an analysis of the responses and the resulting rankings of the activities was published in Summary of Responses to the Software Technology Initiative Questionnaire[2].

The R&D Technology Panel of the Management Steering Committee for Embedded Computer Resources conducted a series of planning sessions and drafted a plan in January 1982, which was given limited circulation.The concept of a software initiative received considerable support, but reaction to the details was mixed, primarily due to the plans relatively narrow scope and conservative focus. A revision of the plan, completed in May 1982, served as input to this document.

In parallel with DoD activity, a group of leaders in the computing field organized the ideas from Candidate R&D Thrusts for the Software Technology Initiative into twelve categories and produced technology assessments and recommendations in A Report on DoD's Software Technology Initiative[3]. Another industry group proposed that the initiative adopt an application-oriented approach, in A New Approach to Lowering DoD Software Costs[4]. Together these recommendations represent thirteen opportunity assessments.

Dr. Edith Martin, the newly appointed Deputy Under Secretary of Defense for Research and Engineering (Research and Advanced Technology), was concerned that the evolving plans might not address all critical DoD problems. With support from Army, Navy, and Air Force Assistant Secretaries, she chartered a Joint Service Task Force in April 1982 to report on DoD's problems in using computer technology and to focus on software and systems issues. This task force's report was published on 30 July 1982. Dr. Martin also directed a review of DoD-sponsored software and systems related R&D that would develop an understanding of the current support profile. Finally, she directed that this software initiative planning document be prepared.

---

1. Samuel T. Redwine, Jr., Eric D. Siegel, and Gilbert R. Berglass, Candidate R&D Thrusts for the Software Technology Initiative, OUSDRE(E&PS) (DTIC/NTIS#AD-A102 108), May 1981.
2. Eric D. Siegel, Summary of Responses to the Software Technology Initiative Questionaire, MITRE, McLean, VA (MTR-82W00085), May 1982.
3. Raymond T. Yeh et. al. A Report on DoD's Software Technology Initiative, University of Maryland (Computer Science Technical Report 1611), 1982.
4. L. R. Weisberg, A New Approach to Lowering DoD Software Costs, Honeywell Aerospace and Defense Group, March 1982.

Recognizing that the opportunities and needs are not all technological, the scope was increased to include non-technical concerns. The thirteen opportunity assessments were sent for review to three to five nationally recognized experts in each area of concentration. Their comments and the earlier responses to the questionnaire were used to revise, and in some cases completely rewrite, the assessments resulting in the opportunity assessments in these appendices.

During this series of activities, the need for concentrated attention to software problems was continually reinforced. In particular, several Defense Science Board Task Forces and Independent Review Committees recommended that DoD substantially increase its software-related activity. The strategy presented is, therefore, not only the product of a great deal of thinking, but also a response to a well-recognized and often articulated need.

Many people have influenced the plan in a variety of ways, including offering comment on one of the drafts, responding to the questionnaire, writing one or more of the appendices, or helping to write the plan itself. It is virtually impossible to acknowledge specific contributions properly. There were several preliminary drafts, written by different people, and there are several appendices, written by different people and sometimes revised substantially by second parties based on comments from still others. There simply is no clean way to acknowledge these contributions specifically, yet there is a serious responsibility to those who have given freely of their considerable talents and expertise.

The names of those who contributed in any identifiable way are listed below in alphabetical order. The contributions may date back to 1979 and may have been critical of the effort. The inclusion of a name on this list should not be interpreted as an endorsement of the plan. Although I have freely used material or ideas offered by others and claim no originality, I made the selection and accept full responsibility for the plan. I regret the inadequacy of this method of acknowledgement and sincerely apologize to anyone whose contribution has been inadvertently overlooked.

Larry Druffel

## ACKNOWLEDGEMENTS

Adams, Duane
Amoroso, Serafino
Anderson, Phillip A.
Andrews, Michael
Andrews, Philip J.
Babiak, Nicholas, J.
Balzer, Robert
Banks, Frederick H.
Barbe, D.
Barnes, John W.
Barstow, David R.
Basili, Victor
Batchelder, Merton J.
Bate, Roger R.
Batz, Joseph C.
Belady, Laszlo A.
Bellas, R. J.
Berglass, Gilbert R.
Bergstrom, Deane F.
Betz, Gene A.
Blackmon, J. T.
Blomberg, Charles D.
Boehm, Barry
Boslaugh, D. L.
Boroughf, Holmes
Brown, Stan L.
Browne, Jim
Browning, Dural W.
Bukauskas, Lou
Burrows, J.
Buxton, John
Campbell, J. Frank
Carrio, Miguel A.
Cashman, Paul M.
Cheatham, Tom
Christianson, Neil B.
Clapp, Judith A.
Clark, Kenneth G.
Clary, James
Cleaveland, Richard G.
Coambes, Bob
Cohen, Paul M.
Cohen, V.
Cole, John M.

Comer, E. R.
Conn, Herbert C
Conrad, Thomas P.
Converse, Robert A.
Cooper, Lee
Cooper, Rob
Corder, David R.
Crostley, Eugene F.
Cunningham, R. J.
Davis, C.
Dawson, Theodore E.
DeMillo, Rick
DiNitto, Samuel A., Jr.
Dorfman, M.
Douglas, Frank E.
Dove, William
Dowdell, Charles E.
Druffel, Larry E.
Duvall, Lorraine
Earnest, Robert M.
Egolf, Jim
Ehrenreich, Sam
Ellis, James O.
Epstein, M.
Fallon, R. A.
Farrar, B. L.
Fischer, Herman
Fisher, Dave
Fowler, Northrup, III
Fox, Joseph
Frager, David S.
Frank, Geoffrey A.
Fredette, Richard C.
Gaffney, James J.
Gannon, John
Garcia, Genovevo
Gardner, Clifford C.
Gerhart, Susan
Gerner, Bert J.
Giese, Clarence
Gimble, Eugene P.
Glick, Norman S.
Goodman, Seymour E.
Grafton, Robert B.
Green, Cordell
Green, D.
Greene, Charles John Jr.
Greene, William R.

Gross, Leonard S.
Grove, H. Mark
Gustafson, Robert E.
Henderson, Margaret W.
Herrelko, David A.
Hesser, Wa
Heyliger, George E.
Hodell, Chuck
Howden, William
Irwin, Allen T.
Jackson, Gary
Jackson, L. R.
Jennings, Michele
Jones, Anita
Jones, Robert R.
Jordan, George P.
Kahn, Robert E.
Kaiser, Robert A.
Kastler, Kenneth D.
Kelley, R. D.
Klos, Larry C.
Knight, John
Kramer, J.
Kruesi, Elizabeth
Kuruma, Carole
Kvenvold, Dan
Landry, Bertrand C.
Lane, John J.
Leath, J. Robert
Lehman, M. M.
Levitt, Karl N.
Lewis, Floyd F.
Lieblein, Edward
Lomas, Leighton
Lopkoff, Arthur L.
Lorenzi, D. S.
Lowry, Edward S.
Lugo, Raul S.
Machado, John
Madsen, Nels
Mahen, E.
Mall, Vance
Maloney, Patrick G.
Marciniak, John J.
McBurnett, Steven
McIlroy, Doug
Mellor, Fred W.
Meyrowitz, Alan L.

Michel, E. D.
Miller, Edward F., Jr.
Mills, Harlan D.
Mordhorst, Ronald
Moriconi, Mark S.
Morris, James B., Jr.
Munson, John B.
Murch, Walter G.
Musa, John
Nau, Dana
Neuman, Al
Newbern, David
Noble, L.
Oberndorf, Patricia
Osterweil, Leon J.
Paige, K. K.
Parker, Ken
Patterson, Gerald G.
Patrick, Charles C.
Peele, Shirley
Phillips, Frank A.
Pleasure, James W.
Pollari, Arthur J.
Presser, Leon
Probert, Thomas H.
Pusateri, V.
Radhakrishnan, N.
Raslich, Vaclav
Redwine, Samuel T., Jr.
Reifer, Don
Rice, John R.
Riddle, William
Riley, James M.
Riski, William A.
Roberts, Alan J.
Rosen, Robert
Rothauser, Charles H.
Rothnie, James B.
Sammet, Jean
Saputo, John
Scheiderman, Ben
Schell, J.
Schneck, Paul
Schottle, Tom
Scoville, John T.
Sgro, Rudy
Shirey, Robert W.
Shook, John B.

Sibley, Edgar H.
Siegel, Eric D.
Small, Albert W.
Smith, Diane C.P.
Smith, T. M.
Smith, Thomas D.
Smith, W.
Squires, Stephen L.
Standish, Thomas
Stengle, Lawrence A.
Stevens, Lawson R.
Straeter, Terry
Stroup, Opal
Stucki, Leon
Stuebing, H. G.
Summers, John K.
Szkody, Ronald
Taupeka, Norman J.
Taylor, Robert F.
Teates, H. B.
Tucker, Marc
Vajo, Victor S.
Vicen, Paul M.
Wade, Gerald
Wald, Elizabeth E.
Walker, Kevin B.
Walker, S.
Warner, W.
Weisberg, Len
Weiser, Mark
Welty, Mik
West, I.
Whitaker, William
White, Douglas
Wildman, Charles, J.
Wileden, Jack C.
Wood, David C.
Woody, Toyse O.
Wright, Keith
Wulf, Bill
Yeh, Raymond T.
Zave, Pamela
Zempolich, Bernie

TYPING BY:

Clemency, Cathryn P.
McDonald, Catherine W.
Turner, Shelly M.
Winter, Sandra L.
Wright, Alice

# A P P E N D I X  I I

## OPPORTUNITY ASSESSMENTS

NOTE: The thirteen Opportunity Assessments included here result from the work of many individuals (Appendix I). These assessments served as a source for designing the software initiative described in Volume I. However, the opinions represent those of the individual authors and not necessarily those of the DoD.

## APPENDIX II.0

## SUMMARY OF ASSESSMENT RECOMMENDATIONS

1. Integrated Support Environments

- Support for Integration
  - Develop Framework for Coalescing Existing Tools
  - Unify Tools
- Extended Capabilities
  - Extend Support for Entire Lifecycle
  - Develop Support for Highly Complex Systems
  - Prepare Support for Making Changes to Systems
  - Allow Extension/Customization of Environment
  - Support Exploratory Development of Software
  - Extend Support to Maintenance and Management
- Software Componentry
  - Develop Standards for Software Modules
  - Allow "Warehousing" of Components
  - Support Abstract Components
- Interface and Human Factors
  - Develop Graphical Interfaces
  - Provide Sophisticated Information Retrieval Facilities

- Provide Intelligent Assistance

2. System Definition Technology

   - Languages and Representation Schemes
     - Techniques for Requirements Change Analysis and Change Tracking
     - Develop Techniques for Describing History and Alternate Futures of a System
     - Develop Guidelines for Specification Technique Use
     - Develop Techniques for Controlling Precision and Scope of Descriptions
     - Develop Techniques for Checking Consistency and Completeness of Descriptions
     - Develop Methods for Smooth Evolution of Descriptions
   - Provide for Rapid and Early Construction of Simulation Models
   - Allow Rapid Prototyping of Systems

3. Maintenance

   - Develop Integrated Environments Supporting Maintenance
   - Develop Design Techniques that Accomodate Change and Information Hiding
   - Provide Techniques for Database Maintenance
   - Maintenance of Distributed Systems
   - Prepare Program Understanding Aids
   - Develop Knowledge-based Systems
   - Support Project Management

4. Reliability

- Short-term
  - Develop Prototyping Technology
  - Improve Requirements Language
  - Support Ada Evolution
  - Perform Experiment Evaluation of Design Methods
  - Support Experimental Development and Assessment of Fault Tolerance Techniques
  - Develop Formal Approaches to Testing
  - Develop Testing and Analysis Tools
  - Provide Life Cycle Software Quality Assurance Techniques
  - Perform Proofs of Correctness for Major Design Algorithms and Critical Properties
  - Support Quality Assurance Management
- Long-term
  - Develop Program Transformation and Improvement Techniques
  - Partially Automate Development and In-service Support Techniques
  - Include Quality Assurance Considerations in Development of New Techniques and Tools
    - Requirements Languages
    - Specification Techniques
    - Specification Analysis Techniques
    - Design Methods
    - Programming Methods

- Programming Languages

5. Database Technology

   - Design and Build Project Information Repositories
   - Investigate Information Structures
   - Develop Real-time, High Availability DBMS
   - Improve ADP and $C^2$ DBMS's
   - Improve Database Integrity Techniques

6. Distributed Systems

   - Develop Models of Distributed Computation
   - Develop System Decomposition Techniques
   - Develop and Standardize High-level Protocols
   - Transfer Technology into Community
   - Develop Techniques for Subsystem Interaction
   - Provide Techniques for Exploratory Development
   - Provide Support for Distributed System Management
   - Develop New Procurement Policies

7. Knowledge-based Systems

- o Develop Problem-specific Knowledge-based Systems
- o Generic Systems
  - o Develop Techniques for Program Specification
  - o Develop Knowledge-based Techniques for:
    - Algorithm Design & Analysis
    - Requirements Definition & Analysis
  - o Develop Domain-specific Knowledge-based Systems
  - o Prepare Techniques for Reasoning About Systems
  - o Provide Knowledge-based Support for Management and Communication
  - o Prepare Knowledge-based Environments

8. Hardware/Software Synergy

- o Develop Specification Languages for Systems
- o Provide Tools for Deriving Software and Hardware Specifications from Systems Specifications
- o Provide Tools for Software-Hardware Tradeoff
- o Encourage Software Modules that are Candidates for Hardware Implementation

9. Human Factors

- o Individuals
  - o Study Software Development and Maintenance as a Problem-solving Process
  - o Provide Support for:

- Personnel Selection
- Training
- Managing
- Personnel Evaluation

o Group Dynamics
  o Study Group Dynamics of Software Teams
  o Develop Group Formulation Techniques
  o Develop Computerized Support for Group Interactions
  o Develop Techniques for Evaluating Groups
  o Investigate Inter-group and Inter-organization Issues
o Develop Well-engineered Man-machine Interfaces
o Develop Techniques for Effective, Efficient Entry and Display of Information
o Study Psychological Effects of Man-machine Interaction Modes

10. Technology Transfer
  o Experimentally Investigate Transfer Techniques
  o Study Organizational Strategies
  o Enhance Education/Training Programs
  o Provide Experimental Techniques for Skill Development
  o Provide Development and Maintenance Tools Supportive of Technology Transfer
  o Provide for Transforming Existing Software to Ada
  o Develop Techniques for Evaluating Usefullness of Methods or Tools for Specific Projects

11. Measurement

- Establish a Descriptive Framework
- Establish Common Definitions
- Standardize a Basic Set of Metrics
- Standardize Measurement Procedures
- Construct Theories and Test Hypotheses

12. Management

- Develop Management Tools and Techniques Supporting:
  - Planning and Control
  - Organizing
  - Staffing
  - Directing
  - Control
- Develop Motivation Tools and Techniques
- Develop Measurement Tools and Techniques
- Develop Acquisition Management Tools and Techniques
- Support Management Development
- Utilize Organization Theory
- Aid Transfer of Technology by Managers

13. Applications-Oriented Techniques and Reuse

- Study & Formulate Basic Concepts
- Define Preliminary Designs
- Demonstrate Effectiveness of Technologies in Several Application Areas

- Reusable Software Parts
- Parts Composition Systems
- Application-oriented Languages
- Application Generators
- Knowledge-based Systems

## APPENDIX II.1

## INTEGRATED SUPPORT ENVIRONMENTS

### 1.0 INTRODUCTION

The life span of a software system, from its initial conception as a set of user needs to its ultimate retirement, is comprised of an initial production period preceding delivery of a suitable version and a subsequent ownership period following delivery. These periods can be subdivided into stages, the production period into definition, design, construction and quality-assurance stages, and the ownership period into installation, acceptance and maintenance stages.

These various stages differ in terms of their focus. During the definition stage, for example, the focus is on transforming the users' rough concepts of what is needed into a complete and unambiguous description of the user-visible characteristics of the software system meeting these needs. As another example, the focus of the design stage is upon elaboration of the description prepared during the definition stage so that it provides a "blueprint" for a software system that will exhibit the desired user and customer visible characteristics.

These two examples indicate that the stages are similar in basic nature. In each stage, the task is to further evolve the description by modifying existing information or by adding new information. The intent is to make the description account for the concerns of importance during the stage. In serving this intent, it is imperative that the resulting description be internally consistent and not contain items of information that are in conflict.

Much of the work in the software engineering area has attempted to produce notations, guidelines and techniques that aid the creation and modification of software descriptions. Notations have been developed to provide media that are appropriate for the rigorous statement of information concerning a software system. Guidelines (i.e., principles, practices and procedures) have been developed to assure that the description evolution is both smooth and speedy. And techniques have been prepared to either relieve practitioners of mundane activities or augment the native abilities of practitioners so that difficult tasks can be performed more quickly and accurately.

Many and varied notations, guidelines and techniques have been developed and these aids are proliferating very rapidly. Evaluation and assessment of the aids is underway and will continue well into the future. Some aids will be found to be more suitable for some application areas, while others will be found to be more suitable for other application areas. It is unlikely that a single set of aids will emerge as universally suitable for all application areas and situations.

Nonetheless, it can be expected that subsets of compatible notations, guidelines and techniques can be found for particular application areas. To date, several trial subsets have been defined and embodied as a set of programs, called tools*, that either process descriptions in the notations, encourage observance of the guidelines, or implement the techniques. This embodiment of a set of notations, guidelines and techniques forms a support environment, a "workshop" in which practitioners may carry out their work more effectively and efficiently.

---

* In this Appendix, we use the terms "environment" and "tool" in their technical sense of "automated environment" and "automated tools".

At the core of a support environment is usually found a database system. This is needed, in addition to the other programs, to provide help in retaining and structuring the information that is accumulated about a software system during its life span. With the database forming the core of the support environment, the other programs function to add, massage or remove information in the database. The information structures used in the database are obviously quite tightly linked to the basic concepts underlying the notations, guidelines and techniques provided by the support environment.

Integration of the collection of tools is necessary for the support environment to be truly effective. Such an integrated support environment is based upon some coherent set of concepts, thus assuring that the aids can be used together conveniently and easily and that "the whole is greater than the sum of the parts." A very high level of integration can be achieved by having the unifying concepts follow from a well-defined method that introduces discipline into the production and ownership of software. Because of the lack of widely accepted methods, however, it has been more usual to follow an alternate approach to integration by adopting a general approach (philosophy) for production and ownership. In this case, the unifying concepts are the set of concepts common among the notations or guidelines provided by the support environment.

## 2.0 CURRENT STATUS

Support environments have been an active area of research and development in the last decade. Rudimentary support environments are common, as "extended" operating systems, to aid primarily during the construction stage of development -- coherent collections of aids to programming have quite naturally been developed first. In addition, several support environments have been defined and at least partially implemented which attempt to provide help during the earlier stages

of the production period. These production support environments are still, however, research rather than production-quality systems. Few, if any, support environments have been defined, let alone implemented, that provide help during the ownership period.

One extensive and highly-visible support environment development project has been the DoD-sponsored work on the Ada Programming Support Environment (APSE) [Stoneman]. Two industrial projects are currently underway to implement initial versions of an APSE. These environments will focus primarily on supporting program construction. In fact, the most detailed and well-developed parts of the APSE definition document concern this stage of production and extensive research will be needed to extend the APSE concept to other stages.

No currently available environments are integrated on the basis of a method for disciplined production and ownership. Rather, integration has been on the basis of notations with the level of integration sometimes being very strong (as in a MAPSE where the Ada language is the sole notation) and sometimes being very weak (as in Unix Programmers Workbench which uses only the simple concept of streams of data to unify the aids). Most usually, however, there is a medium level of integration with a small number of notations which share a set of semantic concepts (as in the SARA support system [Estrin 78] which currently has four notations that share semantic concepts pertaining to message-based synchronization among concurrent processes).

While much of the work on support environments has taken place in the academic sector, work in the other sectors is becoming quite strong. Some basic environments are available commercially to support primarily the construction stage, the most well-known of which is the Unix Programmers Workbench. Many companies are developing programming support environments for in-house use, in a very few cases with the intent of eventually marketing the environment.

## 3.0 SPECIFIC RECOMMENDATIONS

### 3.1 Support for Integration

There are two set of activities in this category. One is a short-term effort to coalesce diverse existing notations, guidelines and tools into hospitable collections. The second is a longer term effort to produce truly uniform form collections leading to more deeply and thoroughly integrated environments.

Standardization Framework. This involves a short-term effort to coalesce diverse existing aids into hospitable collections. It involves defining common interfaces for the programs in the environment, a standard command language for the environment, and a standard information format and database structure for use in storing internal representations of the information about the software being developed using the environment. Aids that conformed to these standards could be used together in a relatively harmonious fashion. As a result, the benefits provided by individual aids would be multiplied.

Environment Uniformity. This involves a longer term effort to produce truly uniform collections of aids leading to more deeply and thoroughly integrated environments. It requires comprehensive redesign of aids based on a consistent set of concepts. The result of such an effort would be a set of descriptive notations that span the stages of a software system's life span and a collection of analysis aids tailored for use with these notations. (Several such efforts should probably be undertaken to generate a range of uniform environments based on various methods and underlying abstractions and directed toward various application areas.) Standardized interfaces, internal representations, and database formats would be a natural by-product of such efforts. Uniformity of notations would facilitate

transitions between stages while uniformity of analysis aids would allow more extensive consistency checking during all stages.

### 3.2 Extended Capabilities

The capabilities that would be provided by a basic, primitive integrated support environ- ment should be extended in a variety of ways that would lead to sig- nificant additional payoffs. This area encompasses five activities, each having a strong, clear importance to achieving effective sup- port environments.

Full Life Cycle Support. The majority of existing support environments focus on supporting construction (programming) and post- construction (verification and validation) activities. Extending the capabilities of an integrated support environment to support pre- construction (definition and design) activities should be a high priority. The added capabilities would include both notational and analysis aids. Such extensions will be particularly effective in the context of a uniform, highly-integrated environment.

Support for Change. Software systems inevitably must change because of changes to their requirements or designs or because of the need to correct errors or make enhancements. Capabilities are needed to make it easy to isolate parts that must be changed, introduce new pieces of code, modify existing pieces of code, check consistency, and generate new versions.

Highly Complex Systems. The vast majority of existing aids and environments are geared toward sequential, time-independent software for relatively non-critical applications (e. g., data processing software). Software reliability is most important, however, for highly complex software systems in critical applications. Such highly complex software is typically employed in distributed and/or real-time systems where fault tolerance is vital. Therefore, integrated support environments should be extended to

provide capa- bilities supporting the production and ownership of software that must be distributed, time-dependent, and fault-tolerant. Both notational and analysis capabilities must be developed, and the greatest benefits will accrue if this is done in the context of a uniform environment.

Extensibility/Customizing. An integrated support environment will be most useful to practitioners if it can be extended with additional capabilities and aids or can be customized for use in specialized application areas. A standardization framework such as discussed above is only a partial solution; new constructs and tool generators will be required to permit true extensibility and customizing. The problem is even more difficult in the context of a uniform environment, since the consistency of the environment's underlying concepts must be maintained when the integrated support environment is extended or customized.

Exploratory Development. Aids providing for early visibility of behavior in a developing software system would permit new modes of software production and ownership. Exploratory development, where developers can "try out" approaches and alternatives and maintainers can also "experimentally" investigate alternatives, is feasible given such aids. The benefits from this mode of software development and enhancement would be most dramatic if these aids were incorporated within a uniform environment, but could still be substantial even in a non-uniform environment.

Management and Maintenance. Both management of software projects and the maintenance stage of the software life span are important areas requiring a great deal of exploration in their own right — each is the subject of one of the other software technology assessmen. Appendices. Both of these areas also have important connections to integrated support environments which merit examination as a subthrust under this heading. The collection and analysis of

data about a software project that is relevant to managers should be a function that is automated in an integrated support environment. Similarly, version control, configuration management and the maintenance of consistency among the various representations of an evolving software system are functions that an integrated support environment should provide. Providing these functions in an integrated support environment will demand suitable data representation, storage and retrieval capabilities, as well as a set of tools and other aids.

### 3.3 Software Componentry

"Software Componentry" refers to efforts towards constructing a software system from pre-existing parts or components. This encompasses everything from macro and subroutine libraries to standardized data abstractions and databases. The approach is appealing, but promises relatively less payoff than the other activities mentioned in this Appendix.

Standards for Software Modules. One necessary prerequisite for software componentry is standardization of the structure and documentation for software modules. Standardized use of the Ada package construct is one obvious area in which this could be pursued. Certainly within a software development organization, and especially within an integrated support environment, such standardization should be beneficial and easy to achieve. Standardization on a wider scale will be more problematic.

Component Warehousing. Widespread use of software components, even within an organization or environment and certainly on a wide scale, will demand a warehouse or library system. Primarily this involves the development of classification and cataloging schemes for software components, followed by the assembly and dissemination of catalogues.

Pre-construction Components. Expanding the scope of software componentry to include pre-construction descriptions of components as well as coded software modules would increase the benefits. Such an expansion demands both some standardization of pre-implementation description notations (design languages, etc.) and a cataloging and classification effort similar to that mentioned above.

### 3.4 Interface and Human Factors

The concept of providing workstations to practitioners is becoming fairly popular. But workstations are a means to an end and their real value comes from the extent to which the capabilities that they provide, such as sophisticated graphics and forms processing or the power and response of a dedicated processor, are exploited in an integrated support environment. Three activities concerning the provision of a sophisticated and well-engineered interface are discussed in this section.

Graphical Representations. As in other areas of computing, integrated support environments can benefit from the high information content of graphical input and output. Such features as menu-driven information retrieval or windowing are widely touted for their dramatic contributions to productivity. The primary problems in this area involve developing appropriate graphical representations for software system description.

Information Retrieval. An integrated support environment is essentially a system for creating and manipulating information about a software system that is being developed. Sophisticated aids for retrieving information could provide significant assistance to a practitioner using an integrated support environment. Browsing or selective searching through the accumulated information would be extremely useful modes of interaction with an environment.

Intelligent Assistance. Artificial intelligence techniques should eventually mature to the point that they can be usefully employed in an integrated support environment or in the interface to the environment. Specifically, ideas such as those in the Programmer's Appendice [Rich and Shrobe 78] may one day be incorporated into techniques for guiding and monitoring the activities of a practitioner using an integrated support environment. As discussed in the technology assessment Appendix on knowledge-based systems, the techniques could also be of use as the basis for the environment. This work would seem to be relatively long range and it is therefore difficult to predict its payoff.

## 4.0 RELATIONSHIPS TO OTHER AREAS

This area is very closely related to the other areas under assessment. Advances in Software Definition Technology are needed to enrich the notations available through an environment. Better understanding of Software Maintenance will allow environments to be built to support maintenance. Advances in Reliability, Hardware/Software Synergy, Measurement and Management will provide the techniques that can be supported by tools and thereby significantly extend the capabilities of environments. As Human Factors issues are addressed and understood, environments can become significantly more friendly. The solution of distributed Database problems is critical to obtaining effective and efficient workstation-based environments. And better understanding of Distributed Systems will aid the preparation of environments that support the development of complex systems.

Environments have a particularly strong relationship to the topic of Technology Transfer. They provide an extremely effective means of delivering new technology to practitioners. Thus they are an important means by which technology can be transferred.

## 5.0 PAYOFF ASSESSMENT

Many of the payoffs of providing integrated support environments are impossible to predict or quantify since the environments may significantly change the ways in which software is produced and owned. In a study of productivity, measured in terms of lines of code per person months, needed for a variety of projects, Boehm [Boehm 82] found that the use of sophisticated aids increased productivity by about 50% over that required when only primitive tools were used. It can be expected that productivity will further increase, probably significantly, as aids are integrated. The rest of this section gives several arguments to indicate the nature and extent of this potential increase.

Integration upon some unifying set of concepts can, for example, reduce the overhead for learning and adapting to each new aid. If there is a 50% overhead for learning a new aid and this learning cost is amortized over five projects, then a reduction of 10% can be achieved by eliminating or significantly reducing the overhead. Reduction or elimination of the time needed for context switching among non-unified aids -- which can be reasonably estimated as a half-day of effort per week of work -- could lead to a reduction in effort of 5-10%.

Extending the coverage of a support environment to stages other than construction could also lead to payoff. Boehm and others have found that testing and maintenance constitute at least 75% of the effort involved in software production, while over 60% of all errors discovered in a software system are made prior to construction. Assuming (quite conservatively) that the availability of suitable aids would permit practitioners to discover and correct two-thirds of their pre-construction errors during the pre-construction stages instead of allowing them to persist into the post-construction stages, the net result would be a 30% savings in production effort. When compounded with the benefits of a uniform environment, an

increase of 100% in productivity (i.e., a 50% reduction in effort) could be possible.

A similar argument demonstrates the payoff that can follow from developing environments that support auxillary activities. If we make the reasonable assumption that a manager of a technical project spends 60% of available work time in managing and that each of the five, say, people being managed spend 10% of their work time in individual and group meetings with the manager, then the project personnel each spend an average of about 18% of their time attending to management activities. If this can be reduced by 30%, then the average productivity would be increased by about 7%.

The point is that numerous arguments can be formulated indicating reasonable payoff from developing various parts or aspects of a support environment. Some of the payoffs are dramatic, but most are on the order of 10%. But the payoffs are many and the combined effect of a variety of them can be quite significant -- ten such reductions of 10% each will give an overall reduction of over 60%. Even more reduction can be expected because the payoffs are not totally independent and tend to amplify each others' effects.

## 6.0 REFERENCES


1. Boehm, B., "Keeping the Lid on Software Costs," Computerworld, January 28, 1982.

2. Estrin, G., "SARA at the Age of One," Proc. International Conference on Software Engineering, Atlanta, Georgia, March 1978.

3. Rich, C., and H. E. Shrobe, "Initial Report on a Lisp Programmer's Apprentice," IEEE Trans. on Software Engineering, SE-4, 6 (November 1978), pp. 456-467.

4. Stoneman, Requirements for Ada Programming Support Enviornments, DARPA, Arlington, VA, February 1980.

## APPENDIX II.2

## SYSTEM DEFINITION TECHNOLOGY

### 1.0 INTRODUCTION

Software definition is the first and most important link between the "idea" for a system and the computer system which embodies that idea. Before this link exists, there is no public record of any system at all. After it exists, a document, notation, or computer program is available to be criticized, verified, and changed.

Software definition technology is usually divided into several stages: requirements, specification, and design. A requirement is a documentation of need. It explains the system environment and the problem to be solved, but not what should be done to solve it.

A specification answers the requirements problem with a solution in the language of the requirement and system environment. It does not address the detailed implementation of the solution but rather its operational and functional role in the system as a whole.

A design says how a specification will be implemented on a specific computer system with specific algorithms and data structures. Design is the stage which first departs from the language of the application system and enters the language of the software engineer.

The term "software definition" includes all three areas. A system definition for a completed system would include the components: (1) the overall system mission (requirements), (2) the user's guide (specification), and (3) the implementor's guide (design).

Some of the problems encountered in software definition are:

- o unrealistic requirements -- often the individuals writing the requirements are unaware of the implementation consequences of the stated requirements.

- o inability to predict the operational "feel" of a system -- often requirements documents have specified an interface that, with hindsight, are not ideally humanly engineered and/or do not span the actual operational needs of its users.

- o failure to communicate requirements -- even good requirements are not always implemented well. A good example of this arises where the implementor must make tradeoffs between functionality and performance and, although satisfying the letter of the requirements, the resulting system is not what the user anticipated.

- o conflicting requirements -- too often, apparently reasonable requirements are conflicting when implemented.

- o imprecise requirements -- even the best existing requirements documents are subject to a great deal of interpretation.

- o overly precise requirements -- too often "requirements" have become too precise, specifying details to the level that dictates an implementation (sometimes a non-optimal one) or aspects of an interface that, taken together with other requirements, leads to an awkward design.

## 2.0 CURRENT STATUS

Requirement technology is almost non-existent. There are a few specification languages (e.g., [Zave 82], [Balzer 81]) which claim to be useful as early as requirements, but are primarily intended for later stages of system definition. The difficulty is that requirements must be stated in the user language, and so the requirements technology must either be specific to a given application (such as accounting or network protocols) or be universal and therefore imprecise (such as structured English [Heninger 80]).

Specification technology has been extensively researched, but only occasionally put into practice. SREM [Bell et al. 77], SADT [Ross 77], and PSL/PSA [Teichroew and Hershey 77] are three systems that have been widely used. Some other proposed systems are: for parallel systems DREAM [Riddle et al. 78], and for real-time systems Paisley [Zave 82]. Knowledge engineering has been looking at direct translation of English into specifications. [Balzer et al.].

The design area has had the most research devoted to it. All of the systems mentioned above are applicable to design, and there are many others which can only be applied at the design stage or beyond. Some examples in wide use include abstract datatypes ([Parnas 72], [Guttag 77], [Liskov & Zilles 75]), HIPO [Stay 76], and Jackson Design [Jackson 75]. The EL1 system takes the approach of using a single language from requirements to code [Wegbreit 76], although it is most suitable from design onwards.

## 3.0 SPECIFIC RECOMMENDATIONS

### 3.1 Languages and Representation Schemes

Improved capabilities are needed in the following areas:

- Techniques which facilitate requirements change analysis and change tracking.
- Techniques for specifying a system's likely evolution as well as its initial realization: software systems are constantly changing during their lifetime, yet a software definition is usually just a snapshot of a system at some moment. A software definition should be able to specify where the total implemented system has come from, and where it is going.
- Guidelines for determining which specification techniques will work best in a given application domain.
- Techniques for controlling the precision and scope of a definition, keeping it at the proper level of decision (requirements, coding) while at the same time enforcing necessary rigor.

- o Methods of checking software definitions both for internal consistency and for agreement with the desired system.

- o Methods for evolving a system description from requirements through design and beyond, without switching languages or representations.

## 3.2 Rapid Simulation Capabilties

Capabilities are needed for the rapid simulation of newly conceived systems to judge their probable throughput, error rate costs, etc. Rapid simulation will assist in early conceptual understanding, and speed decisions concerning implementation.

Rapid simulation of a proposed new system or of a proposed change to an existing system will speed decision-making while reducing risk. Relevant performance and cost-effectiveness measures will be obtainable. Expected values and measures of risk and sensitivity usually must be supplied. Both hardware and software considerations are relevant; however, because the simulation of software is less well understood, R&D efforts should emphasize software issues.

Examples of steps in the right direction are the Rand Extendable Computer System Simulation (ECSS), and the SREM SIMGEN capability to generate a rapid simulation directly from a requirements specification.

## 3.3 Rapid Prototyping Capabilities

Since requirements are often difficult to formulate when similar automated applications do not exist, a quickly constructed prototype of a new application can be used to derive the full system requirements. Rapid prototyping will alleviate the problems users and ana ysts encounter in trying to specify fully the requirements for a system. Rapid prototyping can both reduce the time needed to produce requirements, and improve the quality of the requirements eventually produced.

Rapid development of quality system and software requirements is possible with the use of prototype systems. To provide a useful rapid prototyping ability, there is the need to identify processing functions, data collected and stored, and user interfaces. These may vary among applications and with user tolerance and imagination.

Questions to answer are: What can best be entirely ignored in prototypes or handled without concern for details? For what requirements issues is prototyping a suitable aid in defining answers? In what language or other medium should the prototype system be represented? How should the prototyping process proceed? When and how should prototyping be terminated and a production quality system produced?

Examples of steps in the right direction are general database management application generators with simple but powerful interfaces; e.g., GIST, NOMAD, RAMIS, FOCUS. In the software development area, the Unix toolset provides a powerful rapid prototyping capability.

## 4.0 RELATION TO OTHER AREAS

System definition technology interacts deeply with all of the other areas. It requires their technical inputs to ensure the definition of technically-feasible, cost-effective systems. Also, it sets the conceptual framework for the subsequent stages of the life-cycle covered by the other areas. Example interactions are:

- Integrated Support Environment
  - Current specification techniques would be more powerful if integrated with powerful text-processing capabilities.
  - Tools to support later life-cycle phases would be more powerful if they were able to take advantage of information available in the requirements specification. For example, most test tools work backwards from the code; a test tool working forwards from the requirements

specification would catch many more mismatches between the code and the requirements.

- o Software Maintenance

  Some of the most promising techniques for improving software modifiability (e.g., information hiding) depend on one's ability to define not just the initial system to be developed but also the aspects of the system most likely to change.

- o Knowledge-Based systems

  Because problems are informally specified at the early stages of system definition, methods are needed for (1) dealing with human beings, and (2) applying heuristics rather than rigor to problems. Knowledge-based systems using artificial intelligence methods become useful in decreasing the complexity of interaction with a software definition tool, to the point that relatively unskilled users can supply their own definition. This leads to the possibility of users directly programming their own application.

## 5.0 PAYOFF ASSESSMENT

System definition technology reduces software life-cycle effort, improves software quality, and increases software functionality.

### 5.1 Reduced Effort

The major sources of reduced effort achievable via improved system definition technology are:

- o Early error detection and correction.

  Eliminating errors early via completeness and consistency checks, rapid simulation and prototyping, and user-validation procedures will eliminate the much larger expenditures required to correct such errors later in the life-cycle.

- o Error avoidance.

The above capabilities will also avoid many sources of error currently encountered in unstructured natural-language specifications.

- o Effort avoidance.

  Improved technology will not only make the system definition process more efficient, but also eliminate many later sources of effort in developing unnecessary capabilities or unusable products.

## 5.2 Improved Quality

The major sources of improved quality will be:

- o Better maintainability, portability via support of information hiding techniques.
- o Better reliability via specification verification techniques.
- o Better human interfaces via rapid prototyping techniques.

## 5.3 Functionality

The major sources of improved system functionality will derive from the ability to explore more system options via improved problem definition, conceptual modeling, rapid simulation, and rapid prototyping capabilities.

## 6.0 REFERENCES


1. [Balzer 81] Balzer, R., Design specification validation, Rome Air Development Center Report RADC-TR-81-102 1981.
2. [Balzer et al.] Balzer, R. M., Goldman, N., and Wile, D., Informality in program specifications, IEEE Trans. on Software Engineering, SE-4, 2, pp. 94-103,
3. [Bell et al. 77] Bell, Thomas E., Bixler, David C, and Dyer, Margaret, An extendable approach to computer-aided software requirements engineering, IEEE Trans. on Software Engineering, SE-3, 1, pp. 49-59, January 1977.
4. [Guttag 77] Guttag, J., Abstract data types and the development

of data structures, Communications of the ACM, 20, 6, pp. 396-404, June 1977.

5. [Heninger 80] Heninger, K.L., Specifying Software Requirements for Complex Systems: New techniques and their applications, IEEE Trans. on Software Engineering, SE-6, 1, pp. 2-12, January 1980.

6. [Jackson 75] Jackson, M.A., Principles of Program Design, Academic Press, Inc., London, 1975.

7. [Liskov & Zilles 75] Liskov, B. H. and Zilles, S. N., Specification techniques for data abstractions, IEEE Transactions on Software Engineering, March 1975.

8. [Parnas 72] Parnas, D. L ., On the criteria used in decomposing systems into modules, Communications of the Association for Computing Machinery, 15, 12, pp. 1053-1058, December 1972.

9. [Riddle et al. 78] Riddle, W. E., Wileden, J. C., Sayler, J. H., Segal, A. R., and Stavely, A. M., Behavior Modelling During Software Design, IEEE Transactions on Software Engineering, SE-4, 4, pp. 283-292, July 1978.

10. [Ross 77] Ross, D. T., Structured Analysis for Requirements Definition, IEEE Trans. on Software Engineering, SE-3, 1, pp. 6-15, January 1977.

11. [Stay 76] Stay, J. F., HIPO and integrated program design, IBM Systems Journal, 1976.

12. [Teichroew and Hershey 77] Teichroew, Daniel and Hershey, Ernest A. III, PSL/PSA: A computer-aided technique for structured documentation and analysis of information processing systems, IEEE Trans. on Software Engineering, SE-3, 1, pp. 41-48, January 1977.

13. [Wegbreit 76] Wegbreit, B., Goal-directed program transformation, IEEE Transactions on Software Engineering, SE-2, 2, pp. 69-80, June 1976.

14. [Zave 82] Zave, P., An operational approach to requirements specification for embedded systems, IEEE Trans. on Software Engineering, SE-8, 3, pp. 250-269, May 1982.

## APPENDIX II.3

## SOFTWARE MAINTENANCE

### 1.0 INTRODUCTION

Many DoD embedded systems have large, long-lived software which evolves throughout its operational life. The changes to the software can result from repairs, functional enhancements, environmental changes, performance enhancements, and quality improvements. Altogether the cost of these maintenance actions usually exceeds the development cost for software systems. Thus software maintenance is an area with substantial potential for savings.

Improvements in software maintenance may result from developing more maintainable systems, reducing the amount of changes, or improving the capability to make a change. Developing more maintainable systems is also a goal of several other thrusts including Integrated Software Support Environment, Distributed Systems, Data Base Technology, and Systems Definition. Software Reliability aims at reducing the number of repairs. Not all these other area's points involving maintenance will be repeated here. Rather emphasis will be on improving the capability to make changes.

### 1.1 PRODUCT IMPROVEMENT

The software product should be designed, built and maintained such that its composition facilitates evolution. Among the possibilities are configuration independence, reusable components, built-in testing, and the use of Very High Level Languages (VHLL). The product might include better documentation by completely capturing all the information about the software including items, such as design rationale, not normally documented, and by providing advanced docu-

mentation aids-- possibly including built-in training and documentation integral to the system.

Automatic tuning of the data validation functions of a system could reduce the number of manual changes required in this area. "User Programming" possibly by a VHLL or data modification might allow users to make many of the changes which would otherwise require scarce skilled software manpower.

The use of these tools to improve the product have potential for decreasing the cost of maintaining it. In addition, many of the improvements mentioned below under Process and Management Improvements may also be reflected in the product and its increased maintainability.

## 1.2 PROCESS IMPROVEMENTS

Requirements, definition and design should address potential future changes and their likelihoods, resulting in easier future changes. This might include preplanned product improvements. The same integrated support environment should be used for both development and maintenance.

While change is the common denominator of software maintenance, the different types of changes—repairs, enhancements, etc.—may involve different activities and may require different processes and aids. For example, performance enhancement may involve hardware as well as software changes as discussed under Software/Hardware Synergy (see Appendix II.8).

Generally development and maintenance involve the same kinds of low-level operations, but during maintenance they may be performed in different sequences or for different purposes. Because change is also a frequent occurrence during development and because maintenance often involves partial "redevelopment," tools that aid one are normally useful for the other. Nevertheless, a number of specific tools

or techniques might be developed primarily to aid maintenance. Tools include conversion aids, retest aids, advanced debugging aids, change impact analyzers, performance measures and displayers, software transformers, and knowledge-based systems. Software transformers might be used to improve quality, improve performance, or to tailor a program to fit a configuration. A knowledge-based system might be used which incorporates knowledge of the system being maintained. Thus a member of tools might help the maintenance process.

The capabilities of personnel involved in maintenance might be increased by special training in maintenance analysis and programming. Efficiency might also be improved by significant continuity of personnel from development into maintenance.

### 1.3 MANAGEMENT IMPROVEMENTS

Policy concerning software maintenance might be further developed; as might standards and methodologies. For example, DID (Data Item Descriptions) might be made tailorable. Metrics for maintainability and performance would be useful. Contracts might have maintainability incentives based on maintainability metrics. Periodic operational audits or evaluations could also utilize them.

Configuration management is an important part of maintenance management. Especially useful would be cost and schedule estimation for proposed changes--particularly if users could understand them.

Persons concerned with maintainability might be involved throughout development but at least in each of the reviews. Maintainability is not something that can be added-on; it must be specified and designed-in.

## 2.0 CURRENT STATUS

Considering its large share of the costs, software maintenance has been the subject of relatively little R&D. Nevertheless, maintainability has been a frequently stated motivation for modern

programming techniques. The efforts divide into metrics, theory, tools, and management.

Much of the metrics research has aimed at measuring "quality" of complexity [Rustin, 79; McCall, 80]. Complexity might be considered an inverse metric for maintainability. A few efforts proposing metrics for measuring maintainability have been reported [Yau, 80].

Theory related to maintenance has involved such thoughts as "program slicing" [Wieser, 81], and patterns of evolution [Belady, 76]. Data abstraction of information hiding has been proposed as a means of facilitating changes. Database design techniques have likewise been proposed as reducing changes and change effort. But, in general, little theoretical work has been performed. A rewriting of the A-7 software by the Naval Research Laboratory is a combination of theory and practice [Henniger, 80].

Maintenance-oriented tools include configuration management aids and a change impact analyser [SRA, 81]. File comparers for comparing outputs from regression testing are in use. Code library systems are in even wider use as are data base management systems and data dictionaries. Some limited restructuring tools exist, and word processing is widely used for documentation. Considered altogether, however, there are few tools aimed at the maintenance process.

The first books on software maintenance management are just appearing [Glass, 81; Perry, 81]. The state-of-the-art is still relatively primitive, and much work remains to be done in this area. Standardization efforts such as those for Ada are aimed, in part, at aiding maintenance.

Little is understood about maintenance of Distributed Systems. Poorly understood aspects extend from methods for investigating failures to controlling dissemination and application of changes involving multiple nodes.

## 3.0 SPECIFIC RECOMMENDATIONS

Recommended maintenance activities come under seven headings, five of which are related to other major technology assessment areas. The headings are

- o Integrated Support Environments
- o Design for Changes and Information Hiding
- o Database Maintenance
- o Distributed Systems
- o Program Understanding Aids
- o Knowledge-based Systems
- o Management.

Not surprisingly many of the maintenance issues have some parallel with development issues, but maintenance issues have more emphasis on evolution and change.

### 3.1 Integrated Support Environment

An integrated support environment used for both development and maintenance will have numerous benefits as discussed in Section II.1. From a maintenance standpoint it must provide a coherent framework for maintenance tasks and tools. Support must be as good for changes as for original development--re-evaluation, respecification, redesign, reprogramming, retesting, etc., will be required to assure this. And, if the R&D is not done, then a very substantial opportunity for improving systems and reducing costs will have been neglected.

### 3.2 Design for Change and Information Hiding

The concept of information hiding was originally developed by Parnas [Parnas, 72]. Along with its achievement through data abstraction, it has been one of the most active research areas of the

last decade. Ada provides a programming language mechanism for it in the package.

The refinement and application of this concept to DoD applications has significant potential for improving maintenance. Information about parts of the system likely to change could be hidden behind interfaces using only the minimal abstract representation which should be relatively invariant. One strong possibility is hiding configuration details to provide configuration independence for most of the system.

While the concept is fairly straightforward, applying it to designs in the various application areas is only just beginning. Appropriate refinement and utilization could aid greatly in reducing the maintenance burden.

### 3.3 Database Maintenance

Database technology has its own assessment, as discussed in Appendix II.5. From a maintenance orientation it is important that the original database structure be one least likely to change and that change, when it does occur, requires little effort. Tools and techniques may help in reducing the number of changes and in the effort required to perform them.

One technique that has potential for reducing the number of changes is automatically adjusting validation routines, which track past data values to help establish acceptable limits for new values. Another area is better reconfiguration—automatic physical reconfiguration and automated aids for logical reconfiguration.

### 3.4 Distributed Systems

The special problems of maintaining distributed systems need to be solved including repair, performance enhancement, and functional enhancement. Distributed systems introduce problems for software maintenance because one must deal with multiple, concurrently

operating processors, but they also offer additional potential for performance enhancement through parallelism and simply adding more processors.

### 3.5 Program Understanding Aids

Tools are needed to help maintenance personnel analyze and understand the software they are maintaining and the changes to it. Analysis tools such as global cross references, control flow mappers, data flow analyzers, change impact analyzers, profilers, test coverage analyzers, and tract history recorders would be useful. Display techniques, particularly graphical ones with interactive user guided control, could increase the ability to understand and correctly modify software.

Not only would these aids reduce the labor required, but they would potentially be useful on the great mass of existing software as well as on new systems.

### 3.6 Knowledge-based Systems

Prior to developing an "intelligent" maintenance support system, knowledgeable about the system being maintained, a number of steps could be taken in that direction. These include VHLL's, user programming, and reusable package sets. The key to all of these is the organizing of knowledge concerning the application, its software, and the software maintenance process.

This activity has the potential to reduce greatly the skill level required for persons performing maintenance. Indeed, software personnel might be eliminated entirely for some changes.

### 3.7 Management

Almost everything about software maintenance management could use study and improvement. While software development management is still somewhat poorly understood, software maintenance management is

even more poorly understood. The potential for improvements is very substantial.

## 4.0 RELATIONSHIP TO OTHER AREAS

As evidenced by the names of the maintenance activities in Section 3, close relationships exist with Integrated Support Environment, Database Technology, Knowledge-based Systems and Management. In addition, maintenance is related to Software Reliability (the more reliable, the less maintenance) and System Definition Technology (the better the requirements, the less that must be added or changed later). In fact, there is little about software that does not impact its maintenance in some way.

## 5.0 PAYOFF ASSESSMENT

Better software maintenance could impact all facets of utility-user responsiveness, level of effort, quality, and functionality. Modifications are necessary to retain user responsiveness as conditions change.

Maintenance involves more than half the effort currently expended by skilled software personnel. The potential for cost savings are enormous.

Higher quality can be retained in software systems through better maintenance. Presently maintenance changes seem to degrade the quality of systems fairly rapidly.

Software maintenance is half the jackpot and a number of ideas and opportunities exist for wining significant portions of it. Maintenance, change and evolution should be key emphases of the Software Initiative.

## 6.0 REFERENCES

1. Belady, L. A., and M. M. Lehman [76], A Model of Large Program Development, IBM Systems Journal, vol. 15, No. 3, pp. 225-252.

2. Boehm, B. W. [81], Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

3. Glass, R. L. and R. A. Noiseux [81], Software Maintenance Guidebook, Prentice-Hall.

4. Henniger, K. L. [80], "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications," IEEE Trans. on Software Engineering, Vol. 6, No. 1, pp. 2-13.

5. McCall, J. A. [80], and M. T. Matswmoto, Software Quality, Vol. 1 and Vol. 2, Rome Air Development Center, Griffiss AFB, NY, Tech. Report TR-80-109 April.

6. Parnas, D. L. [72], "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, Vol. 15, No. 12, pp. 1053-1058.

7. Perry, W. E. [81], Managing the Systems Maintenance Function, QED Info. Science.

8. Rustin, R. [79], chairman, Workshop on Quantitative Software Models, IEEE New York, NY, Catalog No. TH0067-9 October.

9. SRA [81], Examples of Interactive Semantic Update System (ISUS) Use, Software Research Associates Technical Note TN-74912 March.

10. Wegner, P. [79], Research Directions in Software Technology, MIT Press.

11. Wieser, M. [81], "Program Slicing," Fifth International Conference on Software Engineering, IEEE Computer Society, pp. 439-449.

12. Yau, S. S., and J. S. Collofello [80], Performance Ripple Effect Analysis for Large Scale Software Maintenance, Rome Air Development Center, Griffiss AFB, NY, Tech. Report RADC-TR-80-55 March.

13. Yeh, R. T. [77-78], Current Trends in Programming Methodology (ed), Vol. 1 Software Specification and Design, Vol. 2 Program Validation, Vol. 3 Software Modeling, Vol. 4 Data Structuring, Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

## APPENDIX II.4

## RELIABILITY

System reliability can be considered from many different points of view. To obtain a broader perspective, two assessments were done in this area, one from the point of view of achieving the very high reliability needed for critical software and the other from the point of view of achieving reliability through testing and formal verification.

APPENDIX II.4

PART A

# BUILDING SOFTWARE

## 1.0 INTRODUCTION

This Appendix addresses the problems of building software for systems requiring high reliability, i.e., critical software. The goal is to concisely survey the state of the art and make recommendations for the Software Initiative. This report is necessarily restricted by the author's own knowledge and experience. Also, in the time available to write this report and because of the limited length, it is impossible to be as thorough as one would like.

There are other factors which influence software quality that are omitted here because they are covered in other sections of Appendix II. For example, software environments (Appendix II.1), the relationship between hardware and software (Appendix II.8), and knowledge-based systems (Appendix II.7) all can contribute to software reliability.

How good is the state of the art? In general it is excellent but not good enough. An example that demonstrates the need for research is the first launch of the Space Shuttle. The launch was delayed for two days by a software failure. Very competent people built that software using good (but not great) tools. An in-depth examination of the fault indicates that it is typical of the "it cannot possibly happen" variety.

## 2.0 DEFINITIONS

Assume the following definitions:

(a) Requirements.
The statement of what a program is to do.

(b) Reliability.
Software is reliable if it complies with its requirements most of the time.

(c) Failure.
Software is said to have failed at a point in time when it no longer complies with its requirements.

(d) Error.
An error (more accurately known as an erroneous state) is a state of a system which, in the absence of any corrective action by the system, could lead to failure which would not be attributable to any event subsequent to the error.

(e) Fault.
A fault is the adjudged cause of an error.

(f) Fault Intolerance.
The methodology of software construction in which all possible measures are taken to avoid introducing faults into software.

(g) Fault Tolerance.
The software design approach which assumes that even if software is built using fault intolerance, some residual faults will reman. Provision is then made to cope with these remaining faults.

## 3.0 SOFTWARE RELIABILITY ESTIMATION

In the definition of reliability above, the word 'most' is not defined. This is intentional - quantification of reliability is not discussed in this report. Any technology that provides an improvement in reliability is considered of interest.

There are two points which should be made. First, an examination of the literature reveals that those who are qualified are not in agreement about how to estimate reliability [1]. Second, several government agencies including the FAA and NRC have specified reliabilities (in terms of failures per unit time or similar measures) for systems which include software. If software for critical systems is

going to have to meet these reliability figures, then it is important that an acceptable software reliability estimation method be determined.

## 4.0 ORGANIZATION OF THIS REPORT

The state of the art and the associated research directions break somewhat conveniently into two categories - short term and long term. Under short term, the classical software development cycle is addressed and the various issues which arise in trying to make that cycle work better are covered. Thus, short term really means making improvements and enhancements to current methods over the next few years to clear up existing deficiencies.

Under long term, less traditional methods, that show promise as being able to provide substantial improvements in reliability, are discussed. The key to substantial improvements in reliability is the extensive application of automation and the removal of human intervention. Thus, long term really means throwing away current methods in favor of radical new methods as they are developed over future years.

## 5.0 SHORT TERM

Short term topics divide conveniently into the general categories of fault intolerance and fault tolerance [2]. This section is divided accordingly.

### 5.1 Fault Intolerance

The Software Initiative is concerned with ways of improving technology. Those areas most likely to have a high payoff in the short term are components of the classical software development cycle or reasonably mature technology areas that are on the fringes of that cycle. Conclusions about these components and technologies are discussed briefly in the following sections.

### 5.1.1 Software Prototyping

Software prototyping technology is becoming a useful tool that should be pursued with a view to applying it to critical software. The New York University implementation of Ada [3] (a trademark of the US Department of Defense), using the SETL system, is a superb example of prototyping. SETL is not particularly application specific although it is clearly more appropriate for prototyping compilers than say real-time control systems. Systems oriented to control systems' prototypes could probably be constructed on the SETL model.

### 5.1.2 Requirements

There are many requirements languages and some requirements analyzers. For surveys see [4,5]. In general, they are in better shape than is admitted although there is still plenty of room for improvement. An example is the SREM system [6] which has been used extensively for describing real-time missile tracking radar systems. SREM is a very powerful system that has received relatively little use outside the organization which developed it. Another example, which demonstrates that application of available technology can work well, is the project undertaken by Parnas and his colleagues on the Navy A7 [7].

An important point in the requirements area is that those systems which exist are not widely used, yet requirements definition is the key to building critical software. Reliable software cannot be built unless it is absolutely clear what the software is supposed to do. Many systems (including military systems) use English as a requirements language despite its imprecision. An example from the designers of SREM [6] illustrates the point. The requirements document for a Ballistic Missile Defense system (apparently written in English) contained 8248 requirement and support paragraphs in a 2500 page document.

### 5.1.3 Design Methodology

Design methodologies are much more formal and well defined than many software engineers appreciate. There are, for example, the general notions of structured programming and hierarchical design, and the rather more specific techniques of stepwise refinement [8], interactive enhancement [9], information hiding [10], structured design [11], and the Jackson methodology [12].

The difficulties with software design seem to be:

(1) the methods are not well known,

(2) there are few good tools to aid in their application and enforcement,

(3) there have been no really large-scale experiments with valid statistical results showing the benefits of careful design in terms of improved reliability of the product and it terms of which methods are best for which applications.

Data abstraction provides a powerful tool for the routine use of many design methodologies. The facilities in languages like Ada (see below) will probably have a substantial effect on the practical methods in the short term.

### 5.1.4 Programming Languages

The design of programming languages is not a short-term problem and is not a suitable topic of investigation in the short term - there are many languages that are suitable for describing critical software. Rather, the difficulties lie in finding a language:

a) which is of modern design,

b) which received sufficient care and analysis during its design,

c) which has a precise formal definition,

d) for which compilers exist for the machines of interest,

e) for which validation of compilers and run-time support systems (within the current state-of-the-art) are available,

f) and for which configuration control of the language design exists.

In the short term, these apparently minor issues are the really important issues. Differing opinions on what a language construct means, or subtle errors in compilers, are major causes of faults in programs which have nothing to do with the programming language itself.

In practice, the only programming language which has faced all these issues and attempted to solve all of them is Ada [13]. In addition, Ada is the only widely known and soon to be widely available language to include facilities for data abstraction. These facilities make the more modern design methodologies (such as those mentioned above) far easier to use, and far easier for their use to be enforced. The conclusion is that Ada is the only choice of programming language for constructing critical systems in the short term.

### 5.1.5 Static Analysis

Static analysis is important because it works reasonably well and it is basically an automatic process. It involves checking a program and reporting anomalies and possible anomalies to the programmer. Some existing systems (e.g., Dave [14]) have been criticized on two grounds. First, they use a great deal of computer time and second they tend to produce spurious messages, i.e., signalling as a possible anomaly something which is "clearly" correct. Both areas are being actively researched and good progress is being made.

The first criticism is not really valid in the context of critical software. If an essentially automatic process can help to reveal faults then almost any amount of computer time is acceptable. Naturally, rapid execution is preferred but it is hardly essential. The

second criticism is more important because large numbers of spurious anomaly reports can confuse programmers and cause them to overlook meaningful messages. The situation is not unlike diagnostics from compilers and static analysis has a great deal in common with compiling. In fact, more modern systems integrate static analysis and compiling into one comprehensive set of tools (e.g, TOOLPAK [15]).

### 5.1.6 Testing

"We know less about the theory of testing, which we do often, than the theory of program proving, which we do seldom". [16]

When a programmer finishes "testing" a program, he/she knows virtually nothing except that the test cases work. They represent an infinitesimal amount of the input space in most cases, and consequently no formal statement about a program's future behavior can be made.

A lot of excellent work has been done on testing but it is for the most part either informal or impractical. The interesting thing is that programs which have been "tested" by ad-hoc methods do tend to work quite well. This implies that a formal theory of testing which guides test data selection in practical cases might be feasible.

### 5.1.7 Software Development Cycle Management

Executing computer programs in a test mode is so simple that programmers tend to do it frequently. Software engineering is the only branch of engineering where failure of the product during test usually has no physical consequences. Software is also very easy to change and so failure is usually followed by rapid modification and re-execution. This is not the way to build software yet it frequently occurs.

Other engineering disciplines follow precisely defined procedures at all stages of product development and the procedures are rigidly enforced. The quality of software would improve considerably if all the elements of the classic software development cycle were used and in the correct sequence. When programs are tested, the results of the test should be carefully documented. If a program fails, the requirements and the design must be checked to ensure that the problem does not lie there before any changes to the program source text are made, and so on.

Rigidly enforcing a set of procedures for software development is difficult manually and likely to be circumvented by programmers anyway. The solution lies in automation of the management process, and some excellent work at the University of Illinois on the SAGA [17] system and management grammers seems likely to provide a practical method of controlling software development.

#### 5.1.8 Correctness Proving

With the focus being on critical software, two well-known comments should be repeated:

(1) The proof is just as likely to be wrong as the program.

(2) Proof techniques for parallel programs and for floating point computations are weak to nonexistent.

Finally, it should be noted that Geller [18] published two different proofs of the same program in one paper and the program is wrong.

### 5.2 Software Fault Tolerance

Software fault tolerance is ready to be applied to real systems. It is not, however, part of the classic software development cycle.

There are two general approaches to software fault tolerance. They are Recovery Blocks [19] and N-version Programming [20]. The

two techniques are not unrelated, but they do have somewhat different approaches. They are discussed separately in the section below.

### 5.2.1 Recovery Blocks

Recovery blocks are really part of a general approach to software fault tolerance developed at the University of Newcastle, England. It is impossible to summarize that work here. It has been reported in depth in the literature. An important point to note is that although the University of Newcastle technology of fault tolerance has been developed extensively, there has been little opportunity to evaluate it in practice. Thus there is very little data available showing just what kind of improvement in reliability will be possible with fault tolerance.

### 5.2.2 N-version Programming

N-version programming is another approach to software fault tolerance that has been developed mainly by Avizienis at UCLA. It is somewhat less developed than the University of Newcastle technology but has been the subject of somewhat more experimental evaluation. These experiments have apparently been carried out in an academic environment with the associated limitations.

It is surprising to discover that no specific provision is made in Ada for software fault tolerance. A comment that is frequently made is that the Ada exception handling mechanism provides for fault tolerance - it does not. Attempts to extend Ada using the package facility to provide Recovery Blocks, and support for Safe Programming [21] and N-version Programming have only been modestly successful.

## 6.0 LONG TERM

In the long term, major improvements in the reliability of software will only be achieved if the 'ad-hoc' methods of construction in which humans are involved can be eliminated. The problem in

the classical software development cycle is that humans make decisions at every stage and thereby introduce errors at every stage.

## 6.1 Program Transformation

Software requirements definition is the most important technology area which needs to be addressed in the long term. It is the link between the "idea" or "concept" for a system which exists in a human's brain and computer processing of that idea. Once a complete set of formal requirements exist in machine readable form, they are amenable to many formal methods of analysis. In principle, these methods can be used to build an executable computer program directly from the requirements with either no human intervention or just human guidance. Thus, it is potentially possible to derive a program from its requirements and thereby "prove" that the resulting program complies with is requirements. Note that no proof in the classic sense of program proving is needed. Also, software fault tolerance need not be incorporated because any faults which exist must be in the requirements (assuming the transformation technology is perfect) and so will not be handled, by definition. (If the requirements include software handling of hardware failures, this is a separate problem.) Where formal methods do not yet exist or are not yet sufficiently powerful (such as program design), additional research can be expected to yield satisfactory new or improved techniques.

The ideal situation would be one in which the requirements are entered into a computer by a human at the highest practical semantic level and the process of producing an executable program would be left to the computer. The only testing that would be needed would be that which convinced the human that the requirements as initially entered corresponded to the "idea" in his/her head. Emphasis must be placed on notations which allow requirements to be expressed in a form where the semantics can be determined by processors which will be responsible at least for analysis and possibly for constructing

the executable program. For the most part, this eliminates natural language as a notation for writing requirements.

The practical implementation of these notions is termed Program Transformations [22]. A modest version of the technology has been in use for some time in the form of high-level languages. Programs written in high-level languages are really specifications for machine-language programs. These machine-level programs are not written by humans but are derived automatically from the specifications by a computer program; namely a compiler. This application of program transformation is readily accepted and needs to be extended to higher-level constructs to allow more of the translation process to be automated.

The state-of-the-art in program transformations at the level needed to eliminate human programmers from all but the requirements phase is very far from practical use, as would be expected. However, the systems that have been built and reported are quite impressive. For example, with a little human guidance, the eight queens problem has been solved from its specifications [23].

The existing systems for program transformations are based on syntactic analysis and transformation for the most part. However, possibly there is considerable potential benefit to basing transformations on semantics.

## 6.2 Expert Systems

Expert Systems are a specific area for consideration in the Software Initiative and will not be discussed here in detail. However, it is important to note that the technology of expert systems may be directly applicable to the construction of software in the long term.

Since the requirements are the first machine readable version of an "idea" or "concept", the translation from the "idea" to the

requirements cannot be automated and subjected to completely formal methods. Thus it will never be possible to prove that the requirements correspond precisely to the original "idea". Many errors occur because of the necessarily informal (and thus inadequate) translation of the "idea" into requirements. However, if program transformation systems become available, they could be coupled to an expert system designed to interact with the user in order to determine the user's requirements. Expert systems have already been applied to finding faults in conventional programs.

## 7.0 RECOMMENDATIONS

It is very difficult to make recommendations. All of the subjects discussed under both short term and long term would benefit from an aggressive research program. Similarly, all the subjects would contribute to improvements in the reliability of critical software.

By definition, these recommendations are subjective. They are made under the assumption that the Software Initiative requires recommendations of areas that are mainly high payoff ones. The order in the lists is not a ranking.

### 7.1 Short Term

(1) Development of software prototyping technology.

(2) Evaluation, enhancement and distribution of existing requirements languages, and the development of more advanced ones.

(3) Continued development of Ada.

(4) Performance of large scale, statistically meaningful experiments to evaluate design methodologies.

(5) Development and assessment by experiment of fault tolerance techniques.

(6) Development of a formal approach to testing such that testing would be systematic and the results would lead to general conclusions about the program that is tested.

7.2 Long Term

(1) Development of program transformation and associated code improvement technology.

(2) Pursuit of any technology which can contribute to the reduction of human involvement in the construction of software.

## 8.0 REFERENCES

This is an incomplete, inadequate set of references that might be of value as a starting point for anybody who is interested. The University of Virginia has extensive bibliographies on some of the subject areas in machine readable form.


1. Littlewood, B., "Theories of Software Reliability: How Good Are They And How Can They Be Improved?", IEEE Transaction on Software Engineering, Vol. SE-6, No. 5.

2. Avizienis, A., "Fault Tolerant Systems", IEEE Transactions on Computers, Vol. C-25, No. 12.

3. Dewar, R. B. K., et al, "The NYU Ada Translator and Interpreter", Sigplan Notices, Vol. 15, No. 11.

4. Lauber, R. J., "Development Support Systems", Computer, Vol. 15, No. 5.

5. Entire issue, IEEE Translations on Software Engineering, Vol. SE-3, No. 1.

6. Alford, M. W., "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Translations on Software Engineering, Vol. SE-3, No. 1.

7. Heninger, K. L., "Specifying Software Requirements for Complex Systems: New Techniques and Their Application", IEEE Translations on Software Engineering, Vol. SE-6, No. 1.

8. Wirth, N., "Program Development By Stepwise Refinement", Communications of the ACM, Vol. 14, No. 4.

9. Basili, V. & J. Turner, "Interactive Enhancement: A Practical Technique for Software Development", IEEE Transactions on Software Engineering, Vol. SE-1, No. 4.

10. Parnas, D. L., "On Criteria to be Used in Decomposing Systems into Modules", Communications of the ACM, Vol. 15, No.. 12.

11. Yourdon, E. & L. Constantine, "Structured Design" Prentice-Hall, 1979.

12. Jackson, M., "Principles of Program Design", Academic Press, 1975.

13. "Reference Manual For The Ada Programming Language", U. S. Department of Defense, July 1980.

14. Osterweil, L. J. & L. D. Fosdick, "DAVE - A Validation, Error Detection and Documentation System for FORTRAN Programs", Software - Practice and Experience, Vol. 6, pp. 473-486.

15. Osterweil, L. J., Personal communication.

16. Goodenough, J. B. & S. L. Gerhart, "Towards a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2.

17. Campbell, R. H. & P. G. Richards, "SAGA: A System To Automate the Management of Software Production", ICASE Report No. 80-36, ICASE, NASA Langley, Hampton, Va, 23665.

18. Geller, M., "Test Data As An Aid To Proving Program Correctness", Communications of the ACM, Vol. 21, No. 5.

19. Randell, B., "System Structure For Software Fault Tolerance", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2.

20. Chen, L. & A. Avizienis, "N-version Programming; A Fault-Tolerance Approach to Reliability of Software Operation", Digest of papers, FTCS-8, Toulouse, June 1978.

21. Anderson, T. & R. W. Witty, "Safe Programming", BIT, Vol. 18, No. 1.

22. Entire issue, IEEE Transactions on Software Engineering, Vol. SE-7, No. 1.

23. Balzer, R., "Transformational Implementation: An Example", IEEE Transactions on Software Engineering, Vol. SE-7, Vol. 1.

APPENDIX II.4

PART B

# SOFTWARE QUALITY ASSURANCE

## 1.0 INTRODUCTION

The constructive phase of an engineering project results in the creation of one or more products. The confirmative phase involves the examination of products for errors introduced during the constructive phase. The development of high quality products depends on the effectiveness of both constructive and confirmative methods. The term "software quality assurance" refers to confirmative methods that are used in the development of software.

Complex projects result in intermediate and partial products, and may also involve versions of the final product. In software, the intermediate product at one stage of development may be a specification, or description, of the product to be built at the next stage. Verification is the confirmation that the product of one stage of development meets the specifications of earlier stages. Validation is the confirmation that intermediate and final products meet the perceived expectations of the users of the product. Validation may require actual use of parts of, prototypes of, or the complete finished system, and may not involve specifications or other formal system documents.

In many production environments, quality assurance is associated only with the final products which are to be delivered to the user. Quality assurance is used to detect breakdowns in the manufacturing process and flaws in materials. It is assumed that the specifications for, and the design of, the product have previously been found

to be acceptable, and are not to be changed except when a new product line is to be developed and released. In software development each new product is a new product line, requiring new specifications and new designs. Software quality assurance for software must encompass the whole development process.

The software life cycle is an idealized, convenient classification of different software development activities. It divides development into requirement definition, specification, design, programming, testing and maintenance. This classification can also be used to classify software quality assurance methods and tools. There are two principal ways in which a quality assurance method may be associated with a particular phase of the life cycle: analysis and testing. Its purpose may be for analyzing the products of that phase for correctness properties. Alternatively, its purpose may be for generating test data which is used in the verification and validation of products that are produced in later phases (e.g., the actual code).

## 2.0 CURRENT STATUS

Software quality assurance has been an active area of research and development over the past ten to fifteen years. The current status of quality assurance analysis is discussed below, followed by a discussion of quality assurance testing, and quality assurance management.

### 2.1 Requirements Analysis

Relatively little progress has been made in software quality assurance of requirements. Two kinds of methods are used. Both involve the analysis of a set of requirements to determine if it has certain necessary properties. One kind uses a list of general properties that all requirements should have (e.g., consistency, completeness, testability, etc.) [1]. The effective use of such a test

depends entirely on the knowledge and experience of the analyst. The second kind occurs as part of a requirements construction and analysis methodology. The rule, for example, that "data which flows out of a process node in a data flow diagram must either flow into that node or be derivable by the process from data flowing into that node" is an example of the kind of property that might occur in a quality assurance property list for data flow requirements.[2]

The completeness and consistency properties of requirements written in particular requirements languages are often not identified as being part of quality assurance analysis, and are contained in different parts of the guidelines for using the method in a casual and unsystematic way. Quality assurance at the requirements stage can be made more effective by requiring that the descriptions of proposed requirements methods contain sections which are devoted exclusively to quality assurance of requirements developed using that method.

## 2.2 Specification Analysis

Requirements are used to identify the capabilities of a system. Specifications are used to formally define the system's capabilities after the requirements stage has been completed.

One of the most widely distributed specifications tools is PSL/PSA (Program Statement Language, Problem Statement Analysis) [3]. PSL requires the user to write a specification using a pre-determined set of object classes and object relationships. Certain classes of objects are required to have particular relationships with other classes. Other classes are allowed to, but not required to, be part of specific relationships. The PSL rules which constrain the relationships which objects enter into allow the PSA analyzer to check a PSL specifications "well-formed-ness" rules.

One of the principal disadvantages of the PSL language is that its semantics are not formally defined and it is not always easy to determine which PSL object classes or relationships to use when writing a specification. The object classes, relationships and well-formed-ness rules in PSL were chosen on the basis of experience with system specifications but they are at the same time too broad, and too specific. They are too broad because, in the attempt to accommodate a wide class of systems, there are too many similar, only vaguely defined constructs in the language. They are too specific in the sense that they are based on a model of systems which is not general enough, and does not allow the convenient representation of some kinds of system properties.

The advantage of PSL/PSA, from the quality assurance point of view, is that it allows some form of well-defined quality assurance analysis of specifications. This advantage is shared by similar specifications methods such as SREM[4]. Current research in specifications is aimed at the development of specification languages with well-defined semantics[5]. Emphasis should be placed on retaining this advantage by recognizing the need for quality assurance, and by designing the languages not only for their constructive properties.

## 2.3 Design Analysis

There has been significantly more recognition of the need for, and the development of, techniques for quality assurance of design than of requirements or specifications.

One of the areas of most concentrated study in quality assurance is proofs of correctness. Most work in this area uses examples in which the logical correctness of algorithms rather that the operational correctness of programs is proved. Issues of finite number representation, for example, are ignored. For this reason, proofs of correctness is identified as a design analysis method, useful for verifying algorithms which occur as part of the design of a program.

Proof of correctness is a powerful, effective method for quality assurance. Its disadvantages are that it requires formal algorithm specifications, is error-prone, and often requires more effort than the original design of the algorithm. The second and third difficulties can be averted if the method is applied to small enough algorithms by experienced, qualified analysts.

The problem of size in proofs of correctness can be avoided in several ways. One is through the use of abstractions, such as data abstractions, for dividing up the different things that have to be proved into separate subproblems [6]. Another is by attempting to prove properties of programs, rather than prove total correctness [7].

Design analysis methods have been developed for system design as well as algorithm design. System design deals with the overall system or module architecture of a large program. Algorithm design deals with algorithms which are often implemented as single modules. TRW carried out a large errors analysis project in which the sources of design errors were identified [8]. The project resulted in a set of design analysis tools which can be used to find inconsistencies and incompleteness in module interfaces. Other design analysis tools have been built for other kinds of designs. Each tool appears to be effective for finding a small, but significant class of errors.

More informal design analysis methods include design inspections and structured walkthroughs [9]. Design inspections involve the use of checklists of error properties. The analyst reads through the design looking for errors described in the checklist. Structured walkthrough is a peer-group analysis technique which is used as part of the constructive design process as well as for quality assurance. Impressive results have been reported for both methods.

## 2.4 Program Analysis

Quality assurance analysis of programs is sometimes called static analysis. Both formal and informal methods have been investigated. The informal methods use checklists like those used for design analysis [10]. The formal methods have been automated with static analysis tools [11].

The static analysis tools for automatable static analysis are largely experimental, and there are difficulties with their portability and their efficiency. They are limited in the programming languages to which they can be applied, and are often restricted to FORTRAN. One class of methods analyzes code for anomalies such as the referencing of uninitiatlized variables. Another class allows the user to add type-checking to languages like FORTRAN, or to check for non-standard language constructs that are not detected by the compiler.

Symbolic evaluation can be used to construct symbolic expressions that describe the computations carried out along paths through programs and to construct symbolic systems of predicates for paths that describe the input data which causes the associated paths to be transversed when the program is executed. Symbolic evaluation is expensive and its applicability is restricted to relatively short program paths in programs which use operators, like arithmetic operations, which can be used to construct concise, easy to read expressions. Several experimental symbolic evaluation systems have been built [12-14].

## 2.5 Quality Assurance Analysis and Maintenance

The analysis methods which are described above are designed for finding errors in software development products. They can be applied to those products when new versions, due to errors or modifications, must be produced. Analysis tools have been proposed to assist in the

re-application of quality assurance to revised versions which can be used to detect dependencies between different parts of a software product. If part A depends on part B and part B is changed then the tools will report that it may also be necessary to change part A. The effectiveness of these tools depends on the use of a database for storing software products, on a modularization of products into components, and on a formal definition of the properties of products upon which the dependency is based.

## 2.6 Requirements-Based Testing

It has been recognized that it is important to generate test data which is based on requirements, but there is little information on how to do this. Standards for the functional testing of requirements have been devised but they are very general and the notion of a "complete" set of tests is not well-defined. It may not be possible to construct more rigorous standards of requirements-based testing without associating them with particular methods of requirements representation.

## 2.7 Specification-Based Testing

This is also a relatively undeveloped area in testing. There is more potential for test completeness standards for specifications written in a formal language than for informal requirements documents. The standards are, like most test standards, likely to be heuristic in the sense that they are based on a systematic model of where errors are likely to occur. If the specification language is based on a formal semantic model then it may be possible to construct completeness models using theoretical results about the effectiveness of testing.

## 2.8 Design-Based Testing

The current status of design-based testing is similar to that for requirements and specifications-based testing. Preliminary

guidelines have been constructed [15] and studies completed which indicate that the detection of large numbers of program errors will only be reliably discovered if the design concepts that are used to build programs are taken into account [16].

In the above discussion, requirements, specifications and design testing refer to the generation of test data which is based on requirements, specifications and design. It has also been proposed that requirements, specifications and designs be built which can themselves be executed over test data. There are several interpretations of the idea. One involves the use of models based on directed graphs. Execution of the requirements, specification or design involves the tracing out of paths through the model.

More substantial notions of executability can be developed for models which have an algebraic or formal computational basis such as abstract or algebraic data type specifications [17]. These are used both in specifications and in design and interpreters can be built to execute them over these very test cases.

## 2.9 Program Testing

Most work in test data generation, and in standards for the completeness of test data sets, does not involve requirements, specifications and design. Program-based testing consists of techniques in which the information which is used for constructing test cases come from the program.

Two classes of systematic program-based testing methods have been developed. In one class tests are constructed to satisfy a coverage measure. Branch testing, for example, requires that tests be constructed in such a way that every branch is executed at least once over some test [18]. Other coverage measures are based on other, usually more demanding, measures of coverage (e.g. LCAJS [19], functional testing [20] and data-flow testing [21]). Branch testing

requires the use of a tool for keeping track of branch traversals. Several tools have been built, some of which are now commercially available.

Empirical studies indicate that many errors are not necessarily revealed by branch testing and that other methods, including, for example, both analysis and design-based testing, are necessary for finding these errors 98,16]. Branch testing should be considered a minimal form of test coverage.

Another class of test methods involves using tests which are selected for their effectiveness in revealing particular classes of errors. A number of people have suggested the use of "error-based" testing but there is little, if any, substance to many of these suggestions since they fail to describe any systematic way of doing this. The most notable exceptions are mutation testing [22] and weak mutation testing [23]. Mutation testing requires the selection of test data that will distinguish a program P from another program P* where P differs from P* by one of a well-defined class of differences (e.g., wrong variable, off-by-one constant, wrong arithmetic operation, etc). The drawback to mutation testing is the large numbers of possible differences between P and P*, even when this class is restricted to very "small" common differences. This disadvantage is partially overcome in weak mutation testing at the expense of reliability in finding errors. Other systematic error-based methods, such as domain testing [24] are under investigation, but the question of their practicality is not clear.

Most work in program testing focuses on the problem of generating test data and ignores the problem of determining if the output which results from a test is correct, i.e., it is assumed that there is an oracle for verifying correctness of test output [25]. This is not because, as has been suggested, researchers were ignorant of this problem. The test data generation problem is the primary problem in

testing and it has been tackled first. Several simple suggestions for dealing with the oracle problem have recently been suggested, including the development of a "fast and dirty" prototype whose purpose is to be used for verification of output from the production version. More sophisticated approaches to the problem include the use of dynamic assertions [26], which allow the user to describe and check not only properties of the expected output but properties of intermediate variable values also.

### 2.10 Management

Management of software quality assurance is a relatively unexplored subject, due in part, undoubtedly, to the lack of, or little of, substance to manage. Several test management tools have been built [27] and an attempt at standards completed [28]. The test tools allow the user to automatically run a program against a file of test data and to have its output checked. They are extremely useful for re-testing modified programs and modules. It is clear that management of quality assurance will have to be integrated with the management of the constructive phases of software development, particularly if the life cycle approach is to be adopted.

## 3.0 RECOMMENDED AREAS FOR FURTHER DEVELOPMENT

Several important areas for research and development can be identified which will lead to short and medium term improvements in software quality assurance and increases in productivity. Some of these are described below.

### 3.1 Program Testing and Analysis Tools

Branch testing is one of the few testing methods for which commercially available testing tools are used. Short-term benefits can be realized through the expanded use of branch testing tools as well as the development and use of other testing tools for methods such as functional, mutation, weak mutation and data-flow testing. Tools for

these methods can be built using existing technology. Their use will not only enhance the reliability of programs but it will result in important practical experience with testing methods.

Only simple, largely experimental static analysis tools have been built. Short term benefits can be realized through the development of production static analyzer tools and through their use in the development of production software. Extensible analyzers can be built which allow the introduction of new static rules without the rebuilding of the static analyzer.

### 3.2 Life Cycle Software Quality Assurance Techniques

The adoption of quality assurance goals in the early parts of the software life cycle will increase the quality of the products of these early phases. This will require the recognition of quality assurance as a major component of those phases, the continual development of new requirements, specifications and design quality assurance testing and analysis methods, and the design of a set of systematic life cycle quality assurance guidelines [29].

The development of requirements and specifications based testing methods is particularly important for acceptance testing. For large systems, it may be infeasible to require the acceptance team to test every branch. All that can be done at this stage is to test the functions implemented by the system, as documented in the requirements and specifications.

It is likely that detailed life cycle testing and analysis methods will have to be customized for different life cycle methods. It is recommended that particular quality assurance agendas be developed for several widely used development methods such as Structured Design.

### 3.3 Proofs of Correctness

The practicality and effectiveness of this approach continues to be a controversial question. It is recommended that its use be adopted for the verification of major design algorithms. Longer term benefits will be achieved through continued research into the use of proofs for properties of programs such as security protection in operating systems or consistency in distributed databases.

### 3.4 Quality Assurance Management

Management includes both the overall coordination of life-cycle quality assurance as well as the management of the data associated with individual techniques. Productivity could be significantly enhanced through the use of an integrated database for the maintenance of all software development products. The database would maintain relationships between modules, design-components, test data, test results, and analysis results. It would be useful for retesting, for generating progress reports and for schedule/cost estimation.

Techniques and standards for parts of the quality assurance process have been in use for some time. Test-management systems which apply programs to sequences of test cases stored in a test input-file, and which compare the test results with expected output file, have been in use by some development groups for several years. Their use should be encouraged. The IEEE has recently been involved in the development of quality assurance standards. It is recommended that several sets of standards be developed by experts in the field (rather than survey shufflers) and that those standards be coordinated with integrated life cycle quality assurance/software development strategies which involve the use of particular development and quality assurance methods.

## 3.5 Fundamental Research

Long-term increases in productivity depend on continued fundamental research in requirements languages, formal specifications, automated specifications analysis, design methods for different kinds of systems, and programming methods and languages. The importance of quality assurance is such that it should be part of, or taken into account during, the development and refinement of new methods. Work on formal specifications, for example, should involve consideration of the need for establishing the completeness and consistency of the specifications in both a formally exact and an intuitively meaningful way. Testing and analysis methods should be developed as part of the specifications methodology.

# 4.0 SUMMARY

Several themes for improved quality assurance have been suggested. One is the development and widespread use of existing and new test coverage and program static analysis tools. A second major theme is the necessity of the recognition of quality assurance for all products of the constructive phases of software development (e.g., requirements, specifications, design). In addition, systematic methods must be adopted which use information derived from one product (e.g., test data derived from a design) to perform quality assurance on related products (e.g., testing of a program). The production use of proof methods for design algorithms, or other computational design models of a system, has been suggested as well as long term fundamental research on formal specification methods with a "built in" quality assurance component.

Finally, the importance of management was noted. Short term benefits can be realized by the adoption of standards, perhaps several levels of standards, based on the knowledge of quality assurance experts. Longer term increases in productivity will

require the use of a quality assurance database which is integrated with the development database.

## 5.0 REFERENCES


1. M.S. Fuji, Independent Verification of Highly Reliable Programs, Proc. COMPSAC-77, Chicago, 1977.

2. T. DeMarco, Structured Analysis and System Specification, Yourdon, New York, 1978. .sp

3. D. Teichrow and E. A. Hershey, PSL/PSA: A Computer-Aided Technique for Documentation and Analysis of Information Processing Systems, IEEE Transactions on Software Engineering, SE-3, 1977.

4. M. Alford, A Requirements Engineering Methodology for Real-Time Processing Requirements, IEEE Transactions on Software Engineering, SE-3, 1977.

5. W. E. Howden, The DATAPROSE Specifications Language for Data Processing Systems, Comp. Sci. Tech Report, EE and CS, University of California at San Diego, 1982.

6. M. Mellier-smith and R. Swartz, Proof of the SIFT Program, IEEE Transactions on Computers, July, 1982.

7. J. Guttag, Notes on Data Abstraction, IEEE Transactions on Software Engineering, SE-6, 1980.

8. B. W. Boehm, R. K. McLean and D. B. Urfrig, Some Experience with Automated Aids to the Design of Large Scale Reliable Software, IEEE Transactions on Software Engineering, SE-1, 1075.

9. E. Yourdon, Structured Walkthroughs, Yourdon Press, New York, 1977.

10. M. E. Fagan, Design and Code Inspections to Reduce Errors in Program Development, IBM Systems Journal, 19, 1976.

11. L. J. Osterweil and L. D. Fosdick, DAVE - A Validation Error Selection and Documentation System for FORTRAN programs. Software Practice and Experience, 6, 1976.

12. W. E. Howden, Symbolic Testing and the DISSECT Symbolic Evaluation System, IEEE Transactions on Software Engineering, 4, 1977.

13. L. A. Clarke, A System to Generate Test Data and Symbolically Execute Programs, IEEE Transactions on Software Engineering, SE-2, 215-222.

14. J. C. King, Symbolic Execution and Program Testing, CACM, 19, 1976.

15. W. E. Howden, Functional Testing and Design Abstractions, Journal of Systems and Software

16. W. E. Howden, Applicability of Software Validation Techniques to Scientific Programs, ACM Transactions on Programming Languages and Systems, 2, 1980.

17. J. Goguen and J. J. Tardo, An Introduction of OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications, Proc. Conf. Specifications of Reliable Software, Boston, 1979.

18. L. G. Stucki, Automatic Generation of Self-Metric Software, Proc. IEEE Symposium Computer Software Reliability, New York, 1973.

19. M. R. Woodward, M. A. Hennell, D. Hedley, Experience with Path Analysis and Testing of Programs, IEEE Transactions on Software Engineering, SE-6, 1980.

20. W. E. Howden, Functional Programming Testing, IEEE Transactions on Software Engineering, SE-6, 1980.

21. K. Bogden and Jamaszw Laski, Data Flow Oriented Program Testing Strategy, T. R. CS-JW-11, School of Engineering, Oakland University, 1980.

22. T. A. Budd, R. A. DeMillo, R. J. Lipton and F. G. Sayward, The Design of a Prototype Mutation System for Program Testing, Proc. AFIPS 78, AFIPS Press, Alexandria, Va., 1978.

23. W. E. Howden, Weak Mutation Testing and Completeness of Program Test Sets, IEEE Transactions on Software Engineering

24. L. J. White and E. I. Cohen, A Domain Strategy for Program Testing, IEEE Transactions on Software Engineering.

25. W. E. Howden, Reliability of the Path Analysis Testing Strategy, IEEE Transactions on Software Engineering

26. L. G. Stucki, New Directions in Automated Tools for Improving Software Quality, in R. T. Yeh, Current Trends in Programming

Methodology, vol. 2, Prentice-Hall, Englewood Cliffs, 1977.

27. D. J. Panzl, Automatic Software Test Drivers, Computer, 11, 1978.

28. IEEE Software Engineering Standards Subcommittee, A Standard for Software Quality Assurance Plans, November, 1981.

29. W. E. Howden, Life Cycle Software Validation, Computer, 15, 1982.

## APPENDIX II.5

## DATABASE TECHNOLOGY

### 1.0 INTRODUCTION

A database management system (DBMS) is a mechanism for storing, manipulating and retrieving interrelated data. Associated with such a system are interfaces for specifying the structure of the data and the operations to be performed on it: Schema (Data) Definition Languages for specifying structure, Data Manipulation Languages for specifying low level (record-at-a-time) manipulation operations, and Query Languages for specifying higher level manipulation and retrieval operations. In addition to these basic operations of storage, update and retrieval, a DBMS may also provide features to enhance its performance and utility:

- o concurrency control - coordinates the interactions of multiple users who are accessing a database at the same time.
- o access control - verifies that the user of a database has been authorized by the owner of the data to make an access request.
- o integrity control - verifies that an update to a database will leave it in a consistent state with respect to some set of integrity constraints defined over the database.
- o query optimization - in the most general case, uses information about access control, integrity constraints and physical access paths to select the best strategy for implementing a query.
- o restructuring mechanisms - support changes made to the logical or physical structure of a database. These changes may include the addition or removal of items or relationships, and changes to access paths.

- o a data dictionary/directory (DD/D) - is a tool for recording and processing information about the structure and use of data. The information that would appear in such a system can range from the documentation of simple physical file structures to the design requirements, database structures, and operation protocols of the underlying enterprise.

- o database design aids - are routines for (semi)automatically producing logical and physical database designs.

- o report writers - format data retrieved from a database for output as hard-copy reports.

- o reliability features - for example recovery mechanisms such as commit points, and rollback and rollforward.

- o view mechanisms - support the definition and implementation of alternate logical structures, or views, of an underlying database.

In term of operating environments, DBMSs can be centralized or distributed over a computer network. The goal of distributed DBMSs is to present to their users the illusion of using a centralized DBMS with the advantages of increased reliability and performance expected from a distributed system. Transparent to these users should be the location of the data and directories, the number of times data has been duplicated at remote sites (for reliability or efficiency), and which sites are currently operational.

DBMSs can run on conventional computer hardware or be supported by special-purpose machines. Special-purpose hardware can be based on electronic disk technology or on microprocessor-based architectures. Electronic disk technology provides a basis for building associative memories where retrievals are performed on the basis of content rather than memory address. Microprocessor-based architectures provide a basis for distributing DBMS functionality over parallel processors.

The database management system is important in the development of large scale Defense software systems for two reasons:

- They represent a large building block that may be reused in many different applications;
- The applications today and in the future are becoming data management problems—ones in which there is a need for continuous availability of large and integrated databases: future military systems will be "information based."

Database techniques span a wide area, from data dictionaries and directories to the generalized DBMS. The metadata in a DD/D is one possible outcome of requirements statements analyses, but an active DD/D system will aid in generation of schema for the DBMS and even as the configuration management system (Sibley, Scallan, and Clemons [1981]), thereby acting as a potential control mechanism over the entire software life cycle (Lefkovits [1977]).

## 2.0 CURRENT STATUS

A number of commercial database products are available. Some generalized DBMSs are mature, commercial products that have been in the marketplace for more than 15 years. However, they have made only modest gains in market penetration. This has been due in part to the high cost of commitment to the early versions of these systems in terms of people, dollars, and hardware (e.g., the total cost of installing and operating IBM's IMS). Only the largest application problems merited a DBMS solution.

Only in the past few years has there been a reasonable selection of products that are relatively easy to use, moderately priced and available on minicomputers and smaller mainframes. These systems feature high-level query languages, and adequate concurrency and reliability mechanisms. View mechanisms, and integrity and access control mechanisms have yet to appear as significant features in commercial products.

Special purpose database hardware has just begun to make its appearance within the past year as viable products. Off-the-shelf

distributed DBMS products are still promises of the future. The most advanced of current products allows queries submitted at one node to be executed at remote nodes. No existing products provide data location, data replication and node failure transparency.

Although database management systems have existed since the research efforts of the early sixties as discussed in (Fry and Sibley [1976]) and the early commercial systems of the late sixties, many active research questions still exist. Research problems in DBMS technology arising from conventional database applications (e.g., inventory and personnel) are essentially solved. However, challenging new database applications are pushing existing centralized database technology beyond its limits and posing new and important research problems. These new applications are characterized by one or more of the following requirements:

- embedding more "application knowledge" into the database to simplify and support the job of the end-user.
- integrating the storage and processing of multiple data media (text, photographs, speech, telemetry data, formatted data) into a single database.
- programming environments to support the rapid and reliable development of sophisticated database application programs.
- handling extremely large databases, particularly when highly aggregated numerical data is involved.
- meeting stringent multi-level security requirements for data integrated into a single database.
- providing "workstation" environments with integrated user interfaces to a variety of information processing facilities.

This last bullet also points to the vigorous current research area of distributed database systems. Here the research focus of recent years has been concurrency control theory and query optimization. With the design and publication of almost 100 concurrency con-

trol algorithms, this can be considered a mature research area more in need of an illuminating reevaluation than additional algorithms. This state has also been reached for query optimization. However, issues of reliability such as network partitioning remain significant problems for the future. With distributed DBMS products possible in the near future, the problems of effectively utilizing them are becoming more critical. Distributed database design aids are of first priority in this area.

For some users, the investment in existing DBMS products and application programs will limit or delay their exploitation of the new distributed DBMS technology. This creates an overwhelming need for a database system that can access a network of hetergeneous DBMS nodes without requiring users to know the network protocols, the local DBMS languages, data transfer protocols or the interrelationships of data across the different nodes.

FIPS and ANSI Standards may soon be expected to appear. The "CODASYL" type (or network) model, as exists in UNIVAC's DMS 1100, Cullinane's IDMS, and DBD Systems' Seed, are under consideration through X3H2, X3J4, etc.; NBS expects to provide a functional specification for a relational calculus based system within ANSI when a new committee is formed.

## 3.0 SPECIFIC RECOMMENDATIONS

The specific recommendations fall into five categories ranging from improving the traditional ADP and $C^2$ usage of DBMS's to real-time DBMS's containing unusual types of data such as radar data. Of special interest is the development of a model data base for software project data.

### 3.1 Project Information Repository

At the center of an Integrated Support Environment should be a project information repository containing both technical and

managerial data. This database needs to be defined, used, and evaluated. Possibly the data models or architectures used will require advances in the DBMS state of the art. This is a key activity if software tools are to have standard interfaces and thus allow a evolutionary and possibly marketplace-based development of a rich integrated support environment.

### 3.2 Information Structures

A number of types of information or concepts concerning data structures merit attention,

- o non-formatted data such as sensor and satellite data
- o multi-media databases including graphics, text, numerical data, maps, speech, and non-formatted data
- o data structures explicitly dealing with time
- o the concept of data abstraction shows some promise, but it must be married to better concepts in classification theory (as a part of information storage and retrieval--ISR--technology). Indeed this crossover from ISR to DBMS technology may be the most important. The further fitting of good configuration management techniques would allow for truly exceptional improvements.
- o knowledge-based systems information structures incorporating concepts used in expert systems
- o distributed data bases with distributed information structures
- o database machines to support data structures.

### 3.3 Real-time, High-availabiltiy DBMS

A DBMS suitable for real-time military systems and programmed in Ada potentially could be reused in a number of future systems. High performance and high reliability are critical for these types of systems. The use of DBMS in real time applications with short bursts of high activity (such as weapons or satellite controls) leads to dif-

ferent needs, still mainly solved today by older "access method" driven systems. Modeling of the problem and development of software and possibly hardware DBMS modules is needed.

### 3.4 ADP and $C^2$ DBMS's

A number of issues in the traditional application areas for DBMS's still deserve attention.

- o very large data base systems are of even more interest with new digital video disk type hardware available
- o query language usability-menus, graphics presentation, natural language interfaces
- o conventional query optimization still has the potential for many worthwhile second-order improvements
- o multi-level security requirements in integrated databases
- o better user interfaces including "workstations"
- o better management procedures for introducing and controlling integrated databases
- o standardization could reduce training needs and ease interorganization data transfer

### 3.5 Database Integrity Techniques

The provision of good methods for integrity (validation, security, and disaster/error recovery) is somewhat limited. Currently integrity constraints on data generally provide only primitive support. There is no obvious way for checking consistency of a large number of such constraints. A theory of constraints would help to solve this--especially if it was a valid system for duplicated and distributed (partitioned) data. Not only theory but improved practices in this area are needed.

## 4.0 RELATIONSHIP TO OTHER AREAS

DBMS's and Dictionaries (DD/D's) as tools, can be used by many

other groups. The integrated software environment may use a configuration management system implemented on the dictionary (DD/D) for use in software development and software maintenance. The distributed data processing effort often relies on DBMS's and DD/D's with the directories containing the information for network access, etc. Management could also be facilitated in a variety of ways using DBMS's.

## 5.0 PAYOFF ASSESSEMENT

In the short term, much of the DBMS payoff will result from improving current systems and finding how to use them more effectively. But this payoff requires that information structures and management problems be addressed soon. The reusability aspect of DBMS also need standardization, and the reliability requirements need good integrity techniques.

In the mid term the project information repository and the real-time DBMS should provide significant benefits. Data integrity is a key issue in user confidence and acceptance of systems and improvements here could also make a contribution.

In the far term distributed databases with many types of data combined with knowledge-based techniques and powerful human interfaces are a potent possibility.

## 6.0 REFERENCES


1. Delobel, C. and Litwin, W. [80], eds., Distributed Data Bases, Proceedings of the International Symposium on Distributed Data Bases, March 12-14, 1980, North Holland Publishing Co., 367.

2. Fernandez, E.B., Summers, R.C., and Wood, C. [81], Database Security and Integrity, Addison-Wesley, 1981, 319.

3. Fry, J.P. and Sibley, E.H. [76], Evolution of Data-Base Management Systems, Data-Base Management Systems Issue: ACM Computing Surveys, vol. 8, No. 1, March 1976, 7-42.

4. General Accounting Office [79], Database Management System--Without Careful Planning There can Be Problems, FGMSD-79-35, June 29, 1979, 48.

5. Lefkovits, H.C. [77], Data Dictionary Systems, QED Information Sciences, 1977, 450.

6. Sibley, E.H. [77], The Impact of Database Technology on Business Systems, Proceedings in IFIP Congress 77, Information Processing, 589-596.

7. Sibley, E.H., Scallan, P.G. and Clemons, E.K. [81], The Software Configuration Management Database, NCC 1981 AFIPS Conference Proceedings, 1981, Vol. 50, 249-255.

APPENDIX II.6

# DISTRIBUTED SYSTEMS

## 1.0 INTRODUCTION

In a distributed system, functions are apportioned among several processors that cooperate to complete a task. In the most straightforward of situations, fundamental components of the solution to information processing problems are distributed to specialized processors such as file servers or array processors. In other relatively simple situations, basic tasks, such as resource allocation or file management, are pre-assigned to permanently execute on specific, perhaps general-purpose, processors. More complex situations arise when redundancy is introduced to enhance the overall system's reliability or survivability, when tasks are dynamically distributed across the available processors, or when the processors are dynamically reconfigurable. Further complexity is introduced in distributed, cooperative problem solving in which tasks (that are generally application-specific) are distributed across the processors and must harmoniously and cooperatively interact to solve some global, overall problem with each task solving only some sub-problem.

Communication protocols are a critical part of every distributed system. At the very least, a low-level protocol is needed for processor-to-processor communication. In general, one or more higher levels of communication protocol are needed to support simple message transfer, message exchange, or extended "conversations" among tasks.

## 2.0 CURRENT STATUS

There has been a great deal of interest in large-scale distributed systems composed of general-purpose processors [VanDam 78, Chu

80]. DoD has actively participated in this work with its development of the ARPAnet and its support for the investigation of a system-wide operating system, the National Software Works. The effective application of such a distributed system has been extensively investigated through a number of projects, many of them oriented toward use of the system as a communication basis for distributed applications.

The use of microcomputers in special-purpose distributed systems is still a research topic [Berglass, 81]. A project to investigate the rapid construction of special-purpose VLSI processors is underway at Stanford [Mason, 80]. These processors could be components of micro-distributed systems, but much work remains to be done before effective and efficient software for such systems can be developed.

Commercial endeavors have tended to focus on distribution within local area networks. A prime example is the XEROX 8000 Network for office automation which includes intelligent workstations, file servers, print servers, communication servers and gateways to "foreign" processors. New ventures in this part of the computer industry are quite numerous and several companies are attacking the problem of an integrated system for handling digitized voice and picture data as well as "traditional" data.

A critical problem, particularly in the commercial area, is compatibility among system components, both hardware and software, developed by different groups. National and international standards groups are actively working on the specification and standardization of protocols that will make developer-independent distributed systems possible.

High-level, task-oriented protocols are also a subject of active research as is the subject of high-level languages for distributed processing. Insight into the scope and progress of this research may be gained by perusing the proceedings of the International Distri-

buted Processing Conferences and the SIGPLAN/SIGOPS-sponsored symposia on distributed processing.

Distributed, cooperative problem solving is also an area of active current research. Much of the work in this area (for example, the work being done by Lesser at the University of Massachusetts) is attempting to combine results from the area of distributed processing and the area of artificial intelligence to provide techniques for solving the more difficult problems encountered in distributed systems.

## 3.0 SPECIFIC RECOMMENDATIONS

Usable results in this area cannot be expected in the short term; too little is known about the costs, risks and behavior of distributed systems. Usable, sophisticated systems are even farther off into the future.

The topic of distributed systems is so important, however, the opportunities in a number of areas should not be missed. These areas are discussed in this section and an indirect justification, in terms of the potential benefit, of investing in these areas is given in section 5.0.

### 3.1 Models of Distributed Computation

Only a few models on which to base the design of a distributed system have been defined, much less rigorously exercised by being used as the basis for the implementation of actual systems. Such models would, for example,

- define where responsibility and authority resides for different classes of actions,
- explain when, if at all, such responsibility or authority is transferred between sites,

- o derine what inconsistencies could arise among tasks, dictate what inconsistencies were tolerable, and indicate how the tasks would cope with inconsistencies.

## 3.2 System Decomposition Techniques

Currently, systems are decomposed along simple and well-understood boundaries, for example, along a boundary coincident to a file server's interface. This is more or less a physical partitioning of a system since it reflects interfaces between pieces of special-purpose hardware that we know how to build.

Effective distributed systems require a much more extensive knowledge of a system's decomposition along logical, applications-oriented boundaries, especially in the case of highly complex distributed, cooperative problem solving systems. Knowledge is needed about typical and natural decompositions and techniques for both determining and evaluating alternative decompositions.

## 3.3 Standard High-Level Protocols

Research concerning alternative communication protocols is essential if satisfactory, efficient interactions between tasks running on different processors are to be achieved. In addition, this work is required so that independently developed subsystems or systems are able to communicate.

Research in this area must also investigate alternative protocols that provide for different needs. For example, it is probable that different protocols will be needed for distributed task allocation, distributed debugging, and the handling of multiple priority-class messages.

## 3.4 Technology Transfer

Although DoD, through DARPA, has supported much research and development in protocols, a major problem remains in the transfer of this technology. For example, there is no mechanism to encourage

commercial implementation of DoD standard protocols (TCP/IP). DoD also does not work effectively to gain acceptance of its ideas or products as national or international standards; for example, TCP/IP, although the first standards in their area, and the subject of most experimentation, are likely to be overtaken by NBS' standards.

Similar problems are happening in local networks. DARPA's experience and emphasis on inter-networking is being ignored even in major defense systems (e.g., WIS) through lack of effective technology transfer. There is no strong user influence in local network standards, looking at the total system requirements, and concerned with cost. Current efforts (e.g., IEEE 802) do not consider the total system, e.g. host interfaces to local networks.

### 3.5 Interaction Techniques

Techniques must be devised for routinely encountered classes of system, or subsystem, interaction. Many of the interactions among tasks in distributed systems are analogous to interactions that occur between people. For example, how does one find the person responsible for making a certain decision? How does industry replace one person with another, who perhaps is located at a different physical site, and still allow other people to consult the new person without disruption over the change? What organization permits a few people to provide a service when the request for that service is not great, yet allows many people to provide that service when the request rate substantially increases? These are a few example questions for which the term 'task' or 'computer' can be substituted for the word 'person' or 'people' and the question becomes a technical problem that must be solved for distributed systems.

### 3.6 Exploratory Development

Substantial progress in the area of distributed systems can only be made by building a varied collection of systems, based on dif-

ferent models and used for a variety of services and in a variety of situations. History shows that it takes roughly 5 years to build a real and substantial system to the point that it works at all, and it takes 8 to 10 years before the system works effectively and efficiently to the satisfaction of its users. There are no shortcuts. Progress will be made most rapidly if a number of technologically interesting systems are constructed in parallel.

It is imperative to explore real systems for:

- a variety of applications (for example, office automation, corporate management functions, banking, and magazine publication), and
- a variety of situations (for example, highly reliable service or real time response to critical requests).

### 3.7 System Management Support

A key issue for the acceptance of distributed systems is management support. Because these systems are inherently more complex than the average system, aids are needed to, for example, display the system's status, help in dynamic reconfiguration, and determine suitability of distributed structures for systems. Many of these management issues are analogous to management control of networks of large-scale computers (e.g., the ARPAnet), but are much more complicated when there are sophisticated forms of interactions among tasks.

### 3.8 Procurement Policies

Procurement or systems and upgrades must foster attempts at innovative systems. Procurement policies could be used to, for example, allow attempts at distributed system alternatives in parallel with standard system designs. This would produce valuable experience and allow more objective comparisons of designs and implementations.

## 4.0 RELATIONSHIP TO OTHER AREAS

Advancements in the distributed processing area will complicate

many of the problems to be attacked in other areas covered in the technology assessments: maintenance, databases, measurement, applications-oriented technology, and management.

Advancements in this area also depend critically on the solution of problems in other areas. For example, advancements are needed in the areas of hardware/software synergy, system definition technology and technology transfer in order for distributed system advancements to be rapid and have a wide-spread effect.

Finally, the solution of some of the problems in this area can aid work in other areas. For example, better understanding of distributed systems can provide the techniques needed to enhance environments so that they can adequately support the development of complex software systems. And redundancy in distributed systems provides an alternative approach to system reliability.

## 5.0 PAYOFF ASSESSMENT

The payoff to be derived from use of distributed systems is nearly impossible to predict - too little is known about their cost and their effectiveness. We can, none the less, predict a number of potential benefits and taken together, these benefits provide ample support for concentrated work in this area.

First, many systems valuable to Dod are just not possible without the availability of sophisticated distributed systems. Examples of such systems include distributed sensor networks, distributed situation-assessment systems, and distributed battle field control systems.

Second, when a task is implemented on a distributed system of autonomous, cooperating processors, entire system components (either hardware or software) may be replaced to achieve required levels (up or down) of computing power, or reuse in new configurations to build other systems. This is an extension of the concept of replaceable or

reusable software modules, but potentially on a much large scale. Existing systems may become units of distributed systems, allowing some functions to migrate gradually to newer hardware. This could increase the life span of older systems in which huge software investments have been made.

The potential savings if distributed systems can be used to replace obsolescent systems gradually with new systems are enormous. Huge systems involving millions of lines of code would not need to be replaced all at once; users and operators could be retrained gradually, or exiting user interfaces could be preserved; and the risks of depending on wholly new systems would be greatly reduced. Further, a program running on a particular machine could be used in a system with other systems, but without having to be recoded for other machines. Operating system or hardware dependencies of programs would become less critical measures of transportability, if the hardware and operating system could be transported with the programs.

Third, distributed system concepts could contribute to the design of systems with enhanced reliability. Processors could be paired to watch each other (the "buddy" system), reporting discrepancies to a controlling processor. Multiple processors performing the same critical task could be combined in voting architectures in which one processor's anomalous output is "outvoted" by two or more processors delivering the same output.

Fourth, distributed systems could be models for trusted systems. Many software faults result from unwanted contention among tasks, but when tasks run on physically separate processors, the potential for error-causing side-effects is reduced. A distributed system could take advantage of verified VLSI-coded functions. Inexpensive but powerful microcomputers, each performing a single function, could be combined into highly trustable systems; special memory management

hardware for these microprocessors could increase overall system trustworthiness.

Distributed systems could also be models for survivable systems. Geographically dispersed systems, multiple copies of databases, and computer networks with many alternative paths contribute to the system's ability to survive natural and man-made disasters.

Fifth, distributed system design would facilitate hardware tailoring to individual site requirements. In a monolithic system, usually the entire system must be upgraded when one critical task's performance is unacceptable. But in a distributed system, a single system component could easily be upgraded or downgraded without change to components performing adequately. This could result in substantial cost savings when several sites have widely disparate performance requirements.

Finally, distributed systems provide additional flexibility in system design. Obsolescent systems could be phased out gracefully, processing power could be directed to just those functions needing it, and whole systems could be reused in new designs. There is also the real possibility of achieving vendor independence, especially with the advent of high-level inter-processor protocols.

## 6.0 REFERENCES


Berglass, G., C. Hisgen, and E. Siegel [81], Multi-Microprocessor Designs for a Secure Packet Switch, MITRE Corporation, McLean, VA, Technical Report MTR-81W00086, April 1981.

Chu, W. W. [80], Special Issue on Distributed Processing Systems, IEEE Transactions on Computers, Vol. C-29, (December 1980), pp. 1037-1163.

Mason, J. F. [80], VLSI Goes To School, IEEE Spectrum, Vol. 17, No. 11 (November 1980), pp 48-52.

Van Dam, A., and J. Stankovic [78], eds., Special Issue on Distributed Processing, Computer, Vol. 11, No. 1 (January 1978), pp. 11-57.

# APPENDIX II.7

## KNOWLEDGE-BASED SYSTEMS

### 1.0 INTRODUCTION

Software development and maintenance requires the use of a great deal of knowledge. Much of this is application independent and pertains to the process of creating and modifying an executable description of the software. Other pieces of knowledge are specific to the domain of systems within which the application falls (for example, the domain of signal processing systems). Still other pieces are specific to the particular problem being addressed.

In knowledge-based systems, the intent is to capture some of this knowledge and codify it in machine usable form. The ultimate would be to completely automate the process of capturing system requirements and generating and changing software. More realistically, a knowledge-based system can provide sophisticated, intelligent assistance so that the tasks still done by humans are at a higher and more natural level and the human is relieved of having to specify much of the detail of a software system or having to keep track of the relations among the details.

There is obvious potential in using knowledge-based techniques for supporting and partially automating the development and maintenance of software. But, the state-of-the-art is such that many topics must be more completely investigated in the near term and then consolidated to provide a truly effective system. In the remainder of this appendix, we review current work, discuss several areas that can be profitably investigated, and then provide some rough estimates of the payoff of pursuing work in this area.

## 2.0 CURRENT STATUS

Extensive work has been performed in the knowledge-based systems area. In this section, we give a cursory overview of this work—a more extensive review appears in Chapter X of the AI Handbook [Barr 82].

A number of current knowledge-based systems can be characterized as systems that are given a collection of facts (rules, productions, axioms, or what have you) in first-order predicate calculus (FOPC), or in some production rule language, and have the job of establishing new facts by proving various theorems. Differences among knowledge-based systems then stem from the areas of knowledge the facts are concerned with, the way the facts are actually represented, and in the strategies employed to establish (derive) a given theorem. In principle, therefore, a general set of mechanisms for representing predicates and proving theorems in FOPC--combined with appropriate interpretations of the meaning of the predicates and theorems—forms the basis of each member of a class of expert systems. The issue is that, while adequate in principle, the cost of using such a general approach is often orders of magnitude too expensive. (An interesting analogy is that or using production rules to do parsing—essentially generating a set of possible parse trees, continuing until we finally generate one that fits. Parsing is a "solved problem" exactly because we have developed techniques for avoiding the exponential amount of work such an approach entails for a restricted but, practically, very large class of problems that can in fact be done with work measured as n log n.)

Thus, the issues in developing a particular knowledge-based system of the class that we might term rule-based systems are to somehow take advantage of the particular problem at hand and develop methods for representing the facts and deriving the resulting theorems that are efficient for the particular problem. Typically this involves

employing certain interpreted predicates and efficient algorithms for dealing with them as well as search strategies that will insure that the amount or work involved in finding a reasonable path through the search space for the particular problem is closer to linear than exponential.

In certain of the rule-based systems a human being (or equivalent), capable of gathering (or initiating the gathering of) further data, is closely coupled to the system. Essentially what this permits is the dynamic augmentation of the database of facts (and not until there is a reasonable likelihood that some fact is actually needed). If it is the case that several possibilities exist for requesting new facts (that is, there are several paths through the search space that are currently feasible) it may well be that there will have to be some mechanism for ordering these possibilities with respect to some metric (e.g., what is the likelihood that some particular test will be expensive in time or dollars, will annoy the patient, will wipe out some component being tested, and so on).

Another example that deserves mention, because of its direct relationship to software production, is that developed by Barstow and Kant, as part of the PSI system [Green 79]. Barstow developed a rule-based system capable of deriving one or more concrete equivalents for some abstract program [Barstow 79]. Kant developed a companion system that could compare the cost of competing code fragments during the derivation [Kant 81]. Coupled, the two systems were capable of producing <u>good</u> programs.

There have been a number of rule-based systems that have been quite successful. Most of the systems reported to date have been concerned with the medical areas and thus would presumably have a rather narrow interest in DoD. However, it seems very likely that we will soon see the development of expert systems dealing specifically

with problems of interest to DoD. As an example, it is clear that such systems could have important applications to such areas as maintenance (as in aircraft maintenance) and testing (as in hardware testing); further development along the lines Barstow and Kant investigated might result in systems that could produce non-trivial programs.

There is a great deal of similarity among the rule-based systems developed to date. It can be argued that future knowledge-based systems for maintenance, testing, and so on will also, viewed sufficiently abstractly, be very similar. Ideally, we can imagine having one (or at most a small number of) knowledge-based system models that can be customized for each particular application by choosing efficient representations for the knowledge database required for the application, providing efficient interpretations of various predicates and functions peculiar to the application, and determining efficient search strategies and costing algorithms appropriate for the application. While doing this is clearly a much harder problem, there is a definite analogy with the problem of providing a "front end" for a compiler—a problem for which the customized model approach is presently very successful.

## 3.0 SPECIFIC RECOMMENDATIONS

### 3.1 Problem-Specific Systems

Knowledge-based systems can potentially be oriented toward specific problem domains. Such systems do not provide tools to create or maintain software. Instead, they incorporate a new technology, that of knowledge-based systems, to solve new problems and solve old problems better. In such systems, a complete knowledge-based system, built by hand solves a particular problem.

Of necessity, such systems must be very narrow in scope. Areas that, at present, seem amenable to this approach are: undersea sur-

veillance (acoustical signal processing), tactical situation assessment (intelligence report or sensor data processing), equipment maintenance diagnosis, target allocation, strategic reasoning, optical or radar image processing, etc.

Several non-production versions of such systems have already been built and are now undergoing transfer into usable, production-quality systems. These usable versions could be operational in three to five years. Continued investment in this area should provide significant benefit, but will advance software tools only indirectly.

### 3.2 Generic Systems

Knowledge-based systems of broader applicability can be obtained by providing intelligent support or by partially automating the development and maintenance of software. Such systems can be obtained by capturing knowledge about solution techniques and about the decision-making process used to create or modify solutions.

Several areas must be pursued in order to be able to produce generic knowledge-based systems. Current results in these areas demonstrate the plausibility of trying to achieve generic knowledge-based systems. They also indicate the interdependency among the work—coordinated advancement and integration of the work will be needed to obtain truly effective generic systems. Finally, they reveal the need to continually question the efficacy of current software development and maintenance paradigms and practices.

Here, we discuss several areas of high potential. As noted, work in these areas must eventually be integrated and changes to development and maintenance paradigms and practices must be considered. Thus, while these opportunity areas are discussed separately, we make the additional recommendations that integrative work, possibly leading to new paradigms and practices, be pursued concurrently.

Program Specification: The first area is program specification. We are talking about the class of program specification methods that will need the support of knowledge-based systems. Two examples are VHLL's (very high level languages) and natural language. VHLL's can have some existence as specification methods without knowledge-based compilers, but not nearly the payoff.

We can define a generic VHLL to be a formal language that includes abstract data types such as sets, mappings, sequences and relations and abstract control structures including transformation rules and enumerators and constructors for sets, sequences, mappings, and relations. They allow declarative and procedural programming. A domain-specific VHLL would also include domain-specific constructs such as, for example, for real-time.

A VHLL can be used as a programming language directly. But VHLL's are so high level that knowledge-based compilers are needed to produce efficient programs from them. Conventional compiling techniques are inadequate. A VHLL with a conventional compiler or interpreter might be called an executable specification language. It helps in prototyping but without smart compiling is too inefficient to be the final version. Our assumption about the payoff of VHLL's is that they will be coupled with knowledge-based compilers that will generate efficient programs.

Other specification methods include natural language, graphics, examples, traces, and any other means that is useful. Studies have shown that there is, in general, no "best" specification method. Every method is well suited to a class of problems and not as well suited to others. Some problems lend themselves to certain specification methods and not to others. Each method has a place and each method needs further research. The best specification systems would allow a mix that suits the problem at hand. Some current systems use

a formal VHLL as a starting point, but it is clear that informal methods will be important.

The use of high-level specification techniques will allow reusability to a much greater extent. High-level specifications will look more like program plans or specifications than like normal software. They probably will not be reused intact (because new applications will need variants of the old specifications), but will be modified and integrated with other plans with some help from a smart environment. And the particular implementation resulting from these specifications would not be used in general. The knowledge-based compiler would make a store-recompute decision about whether it would be cost effective to use the old code or generate new code.

Algorithm Design: We use the term algorithm design to refer to the more complex and creative aspect of the programming process in which new algorithms are designed, modified and debugged. This part of programming may be contrasted to the more straightforward aspects such as data structure selection (through of course a well-defined boundary is difficult to draw). The knowledge-based approach can be extended into more difficult areas in order to develop an intelligent set of tools for algorithm design.

The algorithm-finding portion of the implementation process requires sufficient inference and creativity that it will not be automated in the near future. But some tools can be provided to the algorithm designer to help with deductive inference, data structure selection, algorithm analysis, etc. This area is one of the most scientifically exciting, in that principles we call creative are codified.

The major payoff here is in the portion of the software process that is concerned with very sophisticated algorithms. This is a small part of programs but important because it is where the programs

spend a lot of time. Typical areas are those addressed by special purpose processors such as array processors or database machines.

Requirements Definition: The definition of software requirements can also be assisted by knowledge-based techniques. In the medium term, it is reasonable to expect knowledge-based tools to help with building prototypes and checking the consistency of requirements statements.

Domain-specific Systems: The knowledge base can be augmented with information about specific applications domains or about domain-specific methods. This domain-specific information would provide detailed information about problems in the domain or their solution--this would reduce the information that the user would have to supply and could lead to more efficient derivation of solutions.

Reasoning Aids: To date, most of the work that has been done in the area of reasoning about programs has been carried out under the rubric of program verification. And it would appear that, at the present time, hope for and support for producing practicable program verification systems is waning rapidly. A major reason for this would appear to be the considerable difficulty of verifying that a non-trivial program is correct, and a good portion of this difficulty derives from the fact that the statement that the program is correct can be considerably more complex than the program itself. There are a number of ways that we might be able to get around this problem (for example, using program verification tools to demonstrate the equivalence of an abstract formulation of some task, itself proved correct by conventional mathematical techniques, with a concrete refinement of that formulation, and so on). However, whether or not the verification tools bear fruit in proving correctness of a practical program, there are numerous other areas where techniques and tools for reasoning about programs will have important application.

We give a brief appraisal of the current state of such tools and the directions in which further development of them should proceed.

Broadly speaking, there are two basic components to a tool that can reason about some program—we shall refer to these as the interpreter and the simplifier. It is the job of the interpreter to take some program (or program fragment) as input and to produce as output a set or symbolic expressions that usually denote relationships among the various program variables over some fragment of the program. The simplifier has the job of "simplifying" the symbolic expressions produced by the interpreter.

The present state of the art or the interpreter and simplifier components of a tool for reasoning about programs provides many opportunities for improvements. As we noted earlier, the major impetus for such tools to date has been program verification. One result of this is that the interpreter components have usually been designed specifically to generate verification conditions and have usually approached this goal by employing Hoare-type logics. It has become increasingly clear that a rather more general approach that can deal directly with sharing relations and the like and that provides a database of facts about a program that can be accessed by a wide variety of reasoning tools is in order.

There are a number of expression simplifiers that are reasonably sophisticated for relatively narrow classes of expressions. However, few of these deal directly with quantified variables (in the sense that variables can occur as universally or existentially quantified in formulas in FOPC), have the capability to "solve" sums or more general recurrence relations, or deal with domains other than integers, arithmetic and logical relations, arrays, and the like. It seems clear that a concerted development effort could result in significant improvements in the sophistication of present day simplifiers.

Project Management and Communication: A very important activity in the software process is project management and communication during the software life cycle, including the software's development and maintenance. We could extend the knowledge base available in a knowledge-based system to create a programming environment that understands the program being developed, the decisions being made, and communications among system developers and users. One advantage would be to provide a continuing record, or "corporate memory", for decisions made during development and maintenance activities. Another advantage would be to assist with communication since one can view the major activities in software as communication between designers, programmers, managers, users and the system itself. The communication may be questions, documentation, bug reports, requests, deadlines or promises to other users, programmers, or the system. By using the system itself for communication, all messages are available for future use. As the system develops a better model of the project it can better deal with the messages. Another good example of this type of tool is an intelligent help facility that helps managers, newcomers or workers to understand what's going on. An important research issue is how to use the available programming knowledge to assist the user in understanding the program and the project.

Another advantage of the project management system is that any management discipline that isn't automated might not be enforced, since it requires more effort on the programmer's part. But to automate such disciplines will require intelligence. Thus knowledge-based support may allow the automatic inference of the information needed to follow a good project management plan.

Knowledge-based Environments: A knowledge-based environment is like a knowledge-based compiler. It uses programming knowledge to make smart editors, debuggers, libraries, consistency testers, etc. An example: with a smart editor the user can say "find the part of

the specification that does such and such a task or causes such and such an effect" (not "find a part of the program that syntactically looks like this"). A smart debugger would point out an inconsistency in a specification or would point out that a piece of a specification that would be customary for this type of program seems to be missing. A system that finds inconsistencies or maintains consistency during modifications would be valuable during maintenance. A smart library and knowledge management system would find, not suitable programs, but suitable program specifications or VHLL schemas that might be modified for a new task. As we move to higher levels, the traditional boundaries around components of an environment tend to dissolve and we get a smart help system that integrates these capabilities to form a good assistant to the entire life cycle.

Another aspect of a smart environment is a system that can paraphrase specifications. This helps in validation and program understanding. A common use might be to paraphrase specifications into English to help newcomers to read them (automatic documentation). Another type of help would be to generate examples of how a program would perform on typical and atypical data.

## 4.0 RELATION TO OTHER AREAS

The knowledge-based systems technologies can have a direct impact on several of the other software technology areas. We cite a number of these in the paragraphs following:

### Integrated Software Environments

Future integrated support environments should include a spectrum of tools that can reason about programs. At one extreme are very simple tools that essentially develop cross-reference dictionaries that can serve to determine what program entities are and are not defined, what program constructs make use of or modify other components, and so on. At another extreme, we may wish to include tools that are capable of verifying that a program is correct (that is, meets its

specifications). We can also imagine tools that are between these two extremes. One example would be a tool that inspects some collection of program constructs and detects and/or certifies the construct free of a certain class of faults (for example, that a selection index is not or is always valid, that parallel processes are not or are free of such problems as starvation, deadlock, and so on). Similarly, we can imagine tools that respond to semantic queries about a program and tools that aid the programmer in testing the program.

System Definition Technology

If the definition of a system is couched in a formal language (formal in the sense that the language permits a formal semantic definition) then the same kinds of tools that develop and use analyses of concrete programs would be applicable to the definitional model of a program, assuming, of course, that the domains over which they were capable of reasoning were extended to include those employed in the (presumably more abstract) definitional language.

Software Maintenance

Knowledge-based tools could play a number of roles in software maintenance. For example, tools that can reason about a program could be a significant aid in exploring the ramifications of some proposed program change or enhancement (for example, what is, is not, or may be affected by some proposed change). Also an important future application of the "expert system" technology will likely be in providing coordination among the activities of groups of programmers, managers, and the like engaged in program modifications, trouble reporting and bug fixing, configuration and re-configuration of system releases and so on. The permission to carry out some activity (for example, modify some system component) could derive from a database of rules and could be controlled by a rule-based protocol that insured that all the steps required to be carried out were in fact carried out. (For example, after modifying x you must perform tests

t, log the modification, inform documentation control of the change, and present the tested result to the configuration control manager for system s).

Reliability

Program verification techniques are, of course, directly applicable in this area. Note that the application may not be verification of correctness but, rather, the certification of the absence of certain classes of faults. Similarly, program reasoning tools are applicable to such tasks as insuring that all paths in a program have been executed, insuring that various extreme cases have been tested, and the like.

Database Technology

The knowledge based system technology is directly applicable to various database technology issues; two examples are enforcing integrity constraints and providing for retrieval of inferred facts rather than raw data. Conversely, database technology is related to rule-based systems in that certain rules may actually be stored in a database and we will want efficient mechanisms to couple the rule-based system with the database.

Human Factors

A major problem in presenting information to users (here we have in mind those who are using a programming support system, but the same remarks apply to application system users) is just what to show them and how to present (display) what we have to show. The problem is complicated by the fact that the needs and desires of a given user may change over time (the user may become more sophisticated, play a different role, operate in a different context, and so on.) It seems clear that rule-based systems could be used to sort out just what and how to display certain information and easily respond to situations that change over time or that are context (e.g., role and activity) dependent.

Measurement

The term measurement has been used with regard to both measuring various aspects of the life cycle as well as in the narrower context of measuring the programs produced with regard to space or time consumed, and so on. Most of the work to date on program measurement has been based on statistical analysis or raw data derived from probing a program in various ways during its execution. If one wishes to develop measurements of a program that are symbolic (that is, functions of tokens that represent quantities that are unknown but for which we might be estimates, probability distributions, and so on) the tools and techniques for reasoning about programs are directly applicable.

## 5.0 PAYOFF ASSESSMENT

The ultimate benefit of work in this technology area is very difficult to assess. Nonetheless, the extent of this payoff can be glimpsed by considering the payoff of high-level specification techniques and knowledge-based management support. Although we do not have estimates at this time, the payoff from other aspects of knowledge-based environments is also expected to be very high.

Experiences with VHLL's indicate that they are five times more concise than LISP, which is itself a relatively compact language compared to, say, Ada or PASCAL. The compactness is real, i.e., there is less information to communicate, not just shorter symbols for the same information. The VHLL allows much more detail to be unspecified and the knowledge-based compiler fills in the detail. The detail includes data structure choices, access functions, control structure details, and various optimizations.

Is this factor of five realistic? Other researchers have reported a factor of four increase in productivity, just due to a formal VHLL language. As another example consider database systems. Query languages provide a limited class of VHLL programs, those that

retrieve information. Query languages provide at least a factor of five improvement over conventional languages such as PASCAL for expressing queries, and often greater. This can be measured by real compactness which, as in other software areas, correlates with productivity.

The payoff of a VHLL includes other factors. They are not only shorter, but also easier to understand. The reason is that they are saying less, i.e. there is less information (or details, or constraints). They are easier for humans to understand and for programs to understand. For example, it is easier to understand the sort program specified by "the same set put in alphabetical order" than the sort program specified by a detailed quicksort algorithm assuming both are specified in appropriate languages.

What does such high-level comprehensibility buy? Many things. It is easier to prove correctness. Compare the two sort programs. One is trivial to prove correct. The other is a challenge. The VHLL version is easier to explain, debug, maintain, verify, validate, etc. There is nearly an across-the-board beneficial effect, of course dependent on knowledge-based compiling being feasible. This all means that for a given development effort, a programming environment can be more effective. (Or a programming environment of given effectiveness requires less effort to use.)

The payoff from knowledge-based management support is similarly large. Suppose a knowledge-based system supplied all information needed for project management. One estimate is that managers spend half their time communicating. If half of the communication is in information gathering, then it cuts costs by 1/4. Suppose the other half of management time is spent in analyzing, summarizing and decision making. Suppose the knowledge-based management system supplies all reports, charts, analyses, decision support systems etc. that are needed. Might that save half of the other half of a manager's time?

If so, then add another 1/4 and say it cuts cost/time by 1/2. But you probably would not be too interested in saving that money—instead it would allow the management of more complex tasks. Another view is that the part of maintenance that goes into understanding the project and understanding the program would be automated by this system when used in conjunction with the rest of the environment. And a payoff of 17% to 81% increase in productivity has been given by Standish for program understanding. It might be increased if we include project understanding and management understanding. A problem with these estimates are that they are not independent of the effects of the rest ot the environment.

## 6.0 SUMMARY

Knowledge-based systems can play a very important role in many areas of software technology. There will likely be numerous rule-based systems developed over the coming years both in the role as application systems as well as in various roles in systems supporting program development and maintenance.

## 7.0 REFERENCES


1. [Balzer 81] Robert Balzer, "Transformational Implementation: An Example," IEEE Transactions on Software Engineering, Jan. 1981, pp. 3-14.

2. [Barr 82] Barr and Feigenbaum, The Handbook of Artificial Intelligence, Volume 2, Chapter 10, "Automatic Programming", William Kaufmann, 1982.

3. [Barstow 79] Barstow, David, Knowledge-based Program Construction, North Holland, 1979, New York.

4. [Erman 80] Erman, L.D., Hayes-Roth, F., Lesser, V. R., and Reddy D. R. The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty, Computing Surveys 12:2 (June 1980), 213-253.

5. [Green et al. 81] Cordell Green, Jorge Phillips, Stephen

Westfold, Tom Pressburger, Susan Angebranndt, Beverly Kedzierski, Bernard Mont-Reynaud, and Daniel Chapiro, Towards a Knowledge-Based Programming System, Kestrel Institute, Technical Report KES.U.81.1, 1981.

6. [Green 79] C. Green, R. P. Gabriel, E. Kant, B. Kedzierski, B. McCune, J. Phillips, S. Tappel, and S. Westfold "Results in Knowledge-Based Program Synthesis," Proceedings of The Sixth International Joint Conference on Artificial Intelligence, 1979.

7. [Green 82] Cordell Green, and Steve Westfold, "Knowledge-Based Programming Self Applied," Machine Intelligence 10, Ellis Forward and Halsted Press (John Wiley), 1982.

8. [Kant 81] Elaine Kant, Efficiency in Program Synthesis, UMI Research Press, Ann Arbor, Michigan, 1981.

9. [Nii 79] Nii, H.P. and Aiello, N., AGE (Attempt to Generalize): A Knowledge-Based Program for Buildings Knowledge-Based Programs, Proc. Sixth Internat. Joint Conf. Artif. Intell. (1979), 645-655.

10. [Rich 78] Rich and Shrobe, The Programmer's Apprentice, IEEE Transactions on Software Engineering, November 1978.

11. [Rich 81] C. Rich, Inspection Methods in Programming, Ph.D. thesis, MIT/AI/TR-604. Dec. 1980.

12. [Teitelman and Masinter 81] Warren Teitelman and Larry Masinter, "The Interlisp Programming Environment," Computer, Vol. 14, 4, April 1981.

13. [Waters 82] R. Waters, The Programmer's Apprentice, IEEE Transactions on Software Engineering, January 1982.

14. [Waters 78] R. Waters, Automatic Analysis of the Logical Structure of Programs, Ph.D. thesis, MIT/AI/TR-492, Dec. 1978.

## APPENDIX II.8

## SOFTWARE/HARDWARE SYNERGY

### 1.0 DEFINITION AND SCOPE

A system is a synergy of its components if the performance of the system is better than is predicted by examining the individual parts. In this context, a system exhibits software/hardware synergy if its software and hardware components work in conjunction to perform a task more efficiently than could be performed either totally in software or totally in hardware. Efficiency may refer either to run-time performance, to design time, or to life-cycle costs.

Many cases of software/hardware synergy are situations where the cost of software design has been substantially reduced by implementing in hardware functions common to many software modules of a system. The development of floating point hardware is a good example of software/hardware synergy. In the early days of numerical computing, floating point operations were done by software. Gradually, common formats for floating point numbers were developed, along with subroutine packages for manipulating these numbers. As demand for fast floating point computations increased and hardware costs decreased, floating point operations appeared in the instruction sets of main-

frame computers. The availability of high-speed floating point operations increased the number of applications that were able to use floating point operations (for example, real-time graphics packages were able to use floating point operations instead of scaled integer arithmetic for doing coordinate transformations). Now many minicomputers offer floating point accelerators which provide increased performance of numerically intensive tasks without requiring software changes.

A software module is a candidate for implementation in hardware if (1) the module is used by a large number of higher level modules and (2) the speed of the module is critical. The major impacts of VLSI are the reduction in cost of replications of a hardware module and the increase in complexity of the function of a module. However, VLSI does not imply a reduction in design costs; they are increasing at a rate comparable to software design. In order for the conversion of a software module into hardware to be cost-effective, it must be possible to amortize the hardware development costs over several system modules, and preferably several systems. Thus the payoff for software/hardware synergy must come from identifying and standardizing speed-critical software routines.

The potential for using special purpose hardware to perform software tasks increases the burden on the system designer. He must configure a system that connects the various special purpose hardware devices in a way that allows the system to benefit from the increased potential performance of the "silicon subroutines."

## 2.0 CURRENT STATUS

One of the most successful examples of software/hardware synergy to emerge in recent years was the introduction of virtual memory, implemented by the use of what is commonly called demand paging. The implementation of a virtual memory requires software and hardware working together cooperative in order for the implementation to be practical. As a virtual memory is usually implemented today, every address seen by the hardware is a virtual address, which must be translated into a physical address in physical memory. Translation of a virtual address to a physical address is performed completely by hardware, as follows: A virtual address is split into two parts. These parts are called the page table index and the offset within a page. A page table mapping is used to translate the page table index into a physical page frame number which, when concatenated with the offset within a page, gives a physical memory address. The location specified by the physical memory address may not be accessible, however. If the page table mapping indicates that the referenced physical page frame is not accessible, i.e., is not currently resident in

physical memory, then a page fault occurs. This is where software (in the operating system) takes over for the hardware. The page fault calls into operation an operating system routine that brings the references virtual page into physical memory. Under certain conditions this may also require the software to "swap out" to backing storage a physical page that (hopefully) will not be needed until a much later time.

It is totally impractical to implement demand paging either completely in hardware or completely in software. The mapping from a virtual address to a physical address is performed very often as a basic, low-level operation in any machine that supports virtual memory. To revert to software in order to perform address mapping in such a machine is unthinkable and totally impractical. On the other hand, it is usually the case that the number of page faults is small compared to the number of virtual to physical address transformations. It is thus practical to implement the handling of page faults in software. In fact, it is not only practical but necessary to handle page faults in such a manner. The combined tasks or swapping in a new page, deciding which page to swap out and swapping out the old page are, in general, complex algorithms that require sophisticated programming solutions. To place such algorithms in hardware is effectively impractical. Besides the complexity involved, the placing of such algorithms into hardware would result in a machine that

would be most inflexible in a market that included widely varying applications.

For many applications a machine that implements a virtual memory provides a highly productive programming vehicle. The user with a large application program for the machine is freed from any concerns about whether his program will fit into memory or not (although other programming concerns may surface). Almost all machines that implement virtual memory provide a hugh address space that is far larger than required for even the most avaricious of users. The user is freed to worry about problems other than how to make his program fit into a physical memory that is too small for his requirements.

In an implementation of a virtual memory, we have all the ingredients required for software/hardware synergy. Without the cooperation of hardware and software, a virtual memory system is impractical. The total effect of this cooperational support is a working virtual memory system that can greatly increase programmer productivity.

In another example of hardware/software synergy, Intel, foreseeing the day when 300,000 transistors would reside on a single silicon chip, recognized early that significant research would be required to effectively utilize the rapidly increasing functionality provided by

advancing VLSI technology. The result of this research is the IAPX 432 system. Probably the most important goal of the 432 design that relates to software/hardware synergy is the decision to draw a rather explicit line of separation between system policy and system mechanism. A 432 system mechanism is a function that must be performed by the system that is relatively simple, is algorithmic (rather than heuristic) and is unlikely to differ in its implementation regardless of the application to which a 432 is exposed. A 432 system policy is any function that is not a mechanism, i.e. requires relatively complex programming, may require some determination of which of several alternatives is "best" at some instant in time, and may even depend on a specific application to determine which of several policies should be invoked. The architects of the 432 have been very careful to ensure that, when possible, mechanisms are implemented by hardware and, always, policies are implemented in (operating system) software. An example of a mechanism in our earlier demand paging example was the translation of a virtual address into a physical address (when no page fault occurs) while an example of a policy is the determination of which of several candidate pages is to be swapped out to make room for a page that needs to be swapped in.

From the preceding paragraph it should be evident that the 432 architects have not attempted to provide a machine with a complete operating system in hardware. In fact, the 432 operating system

resides in both hardware and software. The kernel of the 432 operating system is considered to "complete" the bare 432 architecture, which implements time-critical operating system mechanisms only. This is a perfect example of software/hardware synergy. The bare 432 hardware does not provide a usable, complete operating system by itself. Rather, a set of software functions must be provided that interact closely with the 432 architecture to complete the operating system. On the other hand, without the high-performance hardware support supplied by the 432 architecture, the 432 system would probably not provide satisfactory overall performance. Software and hardware cooperate with each other and complement each other in the 432 system. This fact is often illustrated as in Figure 1.

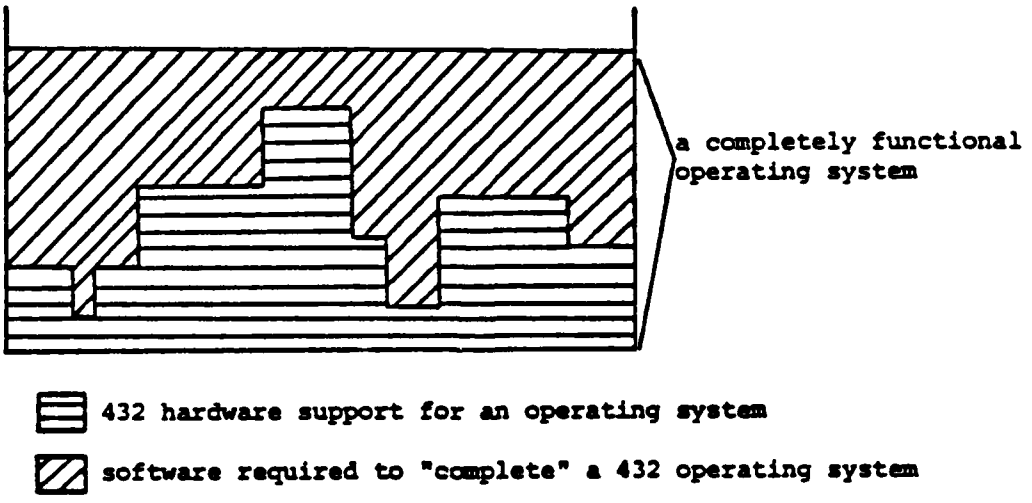



Figure 1: Hardware/Software Synergy in the Intel 432

Failure to strike a balance between software and hardware is an important issue in computer design because it may mean that software/hardware synergy is not being used to advantage. For example, an alternative to the 432 architecture would provide a fully functional operating system in hardware. Although there are real advantages and good reasons for taking this approach (such as increased system performance in some areas), it is difficult to take advantage of software/hardware synergy in an operating system implemented totally in hardware. In forcing policy considerations into hardware, an operating system implemented fully in hardware provides for little flexibility in case an application of the architecture requires a policy different from one chosen by the system's architects. The approach taken by 432 system in partitioning policy from mechanism takes advantage of the synergistic effect of cooperation between software and hardware.

As we alluded to earlier, the advent of VLSI technology has significantly changed the software/hardware synergy picture. Until it became practical to mass produce substantial computing processors on silicon wafers, software/hardware synergy was not really an issue. Because every hardware device had to be handmade using relatively expensive human labor, computer manufacturers required very strong arguments in favor of a concept before the concept found its way into hardware. Machines tended to consist primarily of support for the

most basic of computing operations.

In today/s VLSI age the hesitancy to design innovative hardware has almost completely reversed itself. During the last twelve years the number of transistors that can be placed on a silicon chip has doubled every two years. This phenomenal growth of available functionality has prompted manufacturers of VLSI microprocessors to demand of their architectural designers more and more innovation in order to fully utilize the rapidly increasing available functionality of a silicon chip.

The desire to utilize available silicon functionality has transcended architectural design. Today, software designers have begun to ask themselves whether functionality that historically has been considered software functionality can be considered for translation into hardware. In asking which software functions should be placed into hardware, the answer should invariably be given that software functionality placed into hardware should promote synergy between the resulting hardware and the software than will interact with the hardware. One can even ask whether there exist functions previously in the software realm that, if placed into hardware, would produce a software/hardware system that is synergistic with respect to the further development of software for the system.

The potential implied by VLSI for implementing complex functions has encouraged the design of high performance algorithms for VLSI [2,3,4]. As these algorithms are implemented and their performance evaluated in the context of prototype systems, the operations that they perform become candidates for software/hardware synergy. For example, the development of systolic arrays to perform in real-time basic linear algebra operations such as matrix multiply, solution of systems of linear equations, and matrix triangularization makes them a natural candidate for software/hardware synergy in real-time systems.

Another area that is capitalizing on software/hardware synergy is raster scan graphics systems. Time-critical and heavily used functions required by line-drawing raster scan graphics, such as line plotting and the scan conversion of rectangular figures, are placed into hardware, while software provides the flexibility required in user interface graphics routines. The software/hardware synergy demonstrated by the raster scan graphics field promises to have a major impact in the database technology and human factors thrust areas.

The military computer family project is an example of attempts to encourage software/hardware synergy by concentrating on the definition of a standard interface between the two. The concept of an

Instruction Set Architecture (ISA) defines the instruction set of a computer, which is the interface between the software and the hardware. By specifying only the ISA and some performance, reliability, and power limits, the MCF project allows a variety of implementations. By standardizing on a particular instruction set architecture makes it difficult to adapt to new architectures.

## 3.0 FUTURE DIRECTIONS

The successful use of software/hardware synergy will require careful attention to a decision process for moving software into hardware:

Step 1: An algorithm for performing some noteworthy or marketable function is developed totally in software. The algorithm is studied intensively and is slowly but surely improved.

Step 2: The algorithm begins to reach a "consensus" steady state and is deemed important enough to be a candidate for placement into hardware.

Step 3: If the algorithm is simple or highly "regular" and geometrically well-behaved then it can usually be implemented totally in hardware. In all other cases the algorithm must be partitioned as described in step four to take maximum advantage of software/hardware

synergy.

Step 4: The algorithm is partitioned into two parts: the time-critical portion and the complex portion. The time-critical portion is characterized by its simplicity, its being that part of the algorithm where most of the execution time is spent, and its independence with respect to its implementation over a wide range of applications of the entire algorithm. The complex portion of the algorithm consists of that portion of the algorithm that is not in the time-critical portion. The complex portion is characterized by its complexity, its being that part of the algorithm where only a relatively small part of the algorithm's execution time is spent and its requirement to be adaptable to new configurations that might result from the algorithm suddenly being thrown into a new, unforeseen application area.

When VLSI made its debut several years ago, predictions were given of such magnitude as the impending appearance of an entire Pascal compiler in hardware on a chip set. These predictions were made by people with very little experience in software development, mostly from the hardware industry. It is folly to believe, as some still do, that a highly complex algorithm which should have proceeded to

step four can be implemented completely in hardware (step three) and be developed correctly (or even more closely to correct) or in a shorter period of time. Complexity can not be dismissed by waving a "magic hardware wand" at the complexity. Whether an algorithm is developed in hardware or software, the complexity remains because it is inherent in the complexity of the algorithm itself.

If anything, development of correct, complex algorithms in hardware is more difficult than in software because software developers have higher level development tools than do hardware developers. If one is capitalize on software/hardware synergy, one must be very sophisticated about when to abandon step three in favor of step four. Similarly, once in step four, one must be very careful about how the algorithm is partitioned into its time-critical portion and its complex portion. It is this partitioning that is at the heart of successful software/hardware synergy.

The payoffs from software/hardware synergy will come from effective system design. The increasing complexity of hardware function implied by VLSI must be harnessed by better system design. System design is an iterative process. Better system design will be possible if the designer is able to explore a larger part of the design space. The designer needs tools to allow him to explore that space. He must be able to construct many system models, and to get feedback

on the correctness and the performance of the models. Automated tools for constructing and modifying models, and predicting system performance based on the models must be developed. These tool should support hierarchical models, and allow top-down refinement of the system design. Finally, the models should generate specifications of the system to be constructed, particularly the interface between the software and the hardware.

## 4.0 RECOMMENDATIONS

1) Develop specification languages for systems. Software/hardware synergy is achieved at the system design level.

2) Provide tools for assisting the designer in deriving software and hardware specifications from the system specification.

3) Provide tools for performing software/hardware tradeoff studies. Encourage the development of high level hardware description languages that can support software/hardware tradeoff studies by (a) predicting system performance, and (b) allowing rapid redesign.

4) Encourage the development of software modules in the automated software support environment that are candidates for hardware implementation.

## 5.0 PAYOFF ASSESSMENT

It is important for one to realize two facts about software/hardware synergy: the payoffs are largely indirect and it is currently difficult and expensive to perform effective, large scale (i.e. many transistors) VLSI development. To illustrate what we mean by indirect payoff, it is convenient to reconsider the concept of virtual memory. The existence of a virtual memory capability may not benefit a particular technology area, say, software maintenance, explicitly. It can, however, provide a more user-friendly environment in which to maintain software. In this sense, a virtual memory capability contributes indirectly to a more productive software maintenance environment. To the second point concerning the expense of VLSI development, it should be noted that 300,000 transistors per chip is technically feasible today only for the large semiconductor companies, such as Intel, Motorola, National Semiconductor, etc. Universities and government organizations do not have access to such state-of-the-art silicon technology. This situation may change somewhat if the so-called "silicon foundry" project comes to fruition.

Government-sponsored research projects must, over the next several years, become the driving force behind the incorporation of exotic software into VLSI components. The large semiconductor companies themselves have only begun to scratch the surface of software/hardware synergy. This is true to a large extent because, up until now, the microprocessor and memory markets have been

extremely lucrative for semiconductor companies. These companies have been so busy tapping these markets that they have felt little pressure to be innovative in the placement of exotic software into hardware. This situation is changing, however. The recent Japanese entry into the semiconductor memory market is forcing American companies to become more innovative in order to produce the new products that will allow their companies to keep pace with high technology advances.

The payoff outlook for software/hardware synergy is enormous. For the large semiconductor companies it may mean the difference in whether or not they survive in the semiconductor industry. The payoff for entry into this arena by government institutions, however, cannot be expected to be large immediately. Even with a sophisticated silicon foundry at its disposal, government sponsored research into specialized VLSI components will be expensive and relatively slow to develop fully. It will require several years before a research investment into large scale VLSI development will begin to pay for itself. On the other hand, placing appropriately sophisticated software into silicon is vitally important, because it is <u>the</u> future of large scale VLSI development.

Given our rapidly increasing dependence on the computer, especially in national defense, we can ill afford to expect the private sector to provide leadership for the future development of exotic, large scale VLSI devices. Although governmental entry into this technology arena will be expensive and slow to bear fruit, such entry is inevitable because of its importance. The sooner this inevitability is recognized, the stronger and more productive our national defense effort will be.

## 6.0 REFERENCES


1. Proceedings of ACM Symposium on Architectural Support for Programming Languages and Operating Systems, March 1982, Palo Alto, California.
2. Mead, C. and Conway, L. Introduction to VLSI Systems, Addison-Wesley, Reading, Mass., 1980.
3. Proceedings of the Caltech Conference on VLSI, Caltech, California, January, 1979.
4. Kung, H.T., Sproull, R., and Steele, G. eds. VLSI Systems and Computations, Computer Science Press, Rockville, Maryland, 1981. Edited proceedings of CMU conference on VLSI Systems and Computations, Oct. 1981, Pittsburg, Pennsylvania.
5. Kung,H.T. "Why systolic architectures?" Computer 15, 1 (1982), 37-46.
6. Fuchs, H. and Poulton, J. "Pixel-Planes: A VLSI-Oriented Design for a Raster Graphics Engine," VLSI Design, Oct. 1981.

APPENDIX II.9

# HUMAN FACTORS

## 1.0 SCOPE AND DEFINITION

Human factors is becoming recognized as a central issue in the design, development, and maintenance of computer systems. It is concerned with the human element in computer systems, notably the person:person interaction in computer user groups or software trains and the person:machine interaction with users or software personnel. If these interactions can be made faster, simpler, and more rewarding, then the total cost in both monetary and human terms of using computer systems will decrease markedly.

At the present time, many computer system interfaces with people are designed by computer analysts or programmers without any special training in such work. The resulting systems may be difficult to understand and frustrating to use. Error rates rise and efficiency drops because of poor design decisions. Placing such decisions on a sound basis will result in substantial cost savings to the government in the production and use of computer systems.

### 1.1 Product Improvement

There is a great potential for product improvement as a result of human factors research. In the area of physical interfacing, improvements in keyboard design, in non-keyboard methods of communications, in graphics, and in oral, aural communications could greatly increase speed and ease of use as well as reduce the error rate. For example, spoken commands and touch-sensitive screens may be more effective for certain applications—but the tradeoffs must be made

explicit to allow designers to use them, and guidance must be provided for their design.

The conceptual level, which deals with the logical interface between person and machine, is a wide open area for improvements in comprehension and usability. Design rules for creation of command menus and command hierarchies are needed, as are experimentally-validated rules for design of command syntax and semantics.

Many current systems produce output that is difficult to understand or remember. Research is needed into the best presentation methods for various types of data. Person:machine dialogs need to be refined to avoid excess verbiage while maintaining exact communications. Graphic presentation of output also needs research to assist in the design of presentations that combine rapid user comprehension with a very low error rate. As an example, research into presentation of database structures may be useful.

Another level of person:machine interaction is the emotional level. Improvements here may ease the acceptance of computer systems, decrease tension in the workplace, reduce the error rate, and increase productivity. Design of interactive dialogs, error messages, etc., to take emotional and perceptual factors into account will probably have a large positive effect on the working environment of the system user. A number of users today feel that they are the servants of a judgemental computer system. This feeling could be changed by proper interface design.

Work has barely started in these areas of product improvement, yet it is obvious that as systems come into wider use in DoD environments where the users are under tension, it will be crucial that the computer contribute to a lessening of tensions and an increase in accuracy rather than the reverse. Even in low-tension environments, good design of the human interface may result in large improvements in productivity as well as marked decreases in the error rate.

## 1.2 Process Improvements

All of the improvements listed under "product improvements" apply to process improvements because of the growing use of tools in the computer system design, construction, and maintenance environments. Not only can the standard library of editors, compilers, linkers, etc. to be improved, but the new tools being developed can be designed to take advantage of human interface design rules.

In addition to those improvements, improvements can be made in the functioning of programming groups through the application of human factors research to group and individual dynamics. Human factors can help managers select and train design, implementation, and maintenance specialists. It can also help create team structures appropriate for the tasks at hand. Different tasks and different groups of people may need different structures to perform at their peak; it is crucial that management be aware of these structures and be able to implement them effectively if high productivity is to be maintained.

## 1.3 Management Improvements

As mentioned above, human factors work can ease the task of management by helping in the creation of a viable group dynamic. It can also be invaluable in the selection of personnel. Some people are far more productive than others, yet productivity is not easily predicted. A method of personnel evaluation is needed both for selection and for recognition.

The impact of human factors research on training will probably ease some of the problems caused by the shortage and turnover of highly-productive programmers, designers, and managers.

Finally, research in group dynamics of computer workers and users will help the scheduling and resource allocation process

because of the better understanding of the dynamics of system creation and use.

## 2.0 CURRENT STATUS

There has been a marked upsurge in interest in human factors and software psychology in recent years. An excellent current reference is Software Psychology by Ben Shneiderman (Cambridge, MA: Winthrop, 1980). Besides the standard references, such as the proceedings of the Human Factors Society, the Association for Computing Machinery's special Interest Group on Computer Personnel Research, and journal articles, human factors articles are being published in the primary computer science journals, such as Computing Surveys (see the special issue on "The Psychology of Human-Computer Interaction," March 1981) and IBM Systems Journal (see the special issues on "Human Factors," vol. 20, n. 2 (1981) and on "Man-Machine Interaction," vol. 20, no. 3 (1981)). Bell Labs has recently published a survey of the area for designers [Bailey, 1982].

The Institute for Certification of Computer Professionals has made substantial efforts at defining the body of knowledge required by DP managers and senior programmers. Efforts elsewhere have addressed examinations for EDP auditing, computer security and specific programming language knowledge. In addition, curriculum design efforts for a variety of programs have been performed; of particular interest are those for software engineers.

## 3.0 SPECIFIC RECOMMENDATIONS

The two major human factors activity areas are person:person interaction and person:machine interaction. The first major area is subdivided into an area concerned with the individual ("individual characteristics"), and an area concerned with the group ("group dynamics"). The second major area is subdivided into an area concerned with physical design of keyboards, screens, etc. ("physical

interface"), an area concerned with logical, clear, usable presentation of information ("conceptual interface"), and an area concerned with the emotional impact of computer systems on the user ("emotional interface").

It is important to note that in all these areas there is a strong need for rigorous psychologically oriented controlled experimentation. Funding is needed for basic and applied laboratory studies and field trials of individual performance, and person:person and person:machine interactions. This is related to the experimental study of the software process which is discussed in the measurement thrust.

## 3.1 Individual Characteristics

The subarea of individual characteristics includes study of the psychological processes, selection, training/management, and evaluation of personnel.

Attention needs to be directed at the underlying individual psychological processes of software professionals involved in software development and maintenance activities. Of primary concern are the cognitive and problem solving processes. Not only do these need to be understood but tools and techniques need to be developed to facilitate them.

Selection of computer systems personnel has long been a problem. Little work has been done to define the personality types and knowledge base that combine to create the best analysts, programmers, programming managers, etc. The personality type most common today in the field may not be the best, and the desired type may change as the field evolves. In addition, it is known that there are huge differences in staff productivity; what is not known is an inexpensive, dependable way to find high-productivity personnel.

The problem of training and managing the staff or the users must be faced once the groups are assembled. Study is needed to find more effective techniques of training both staff in individual effectiveness and in communications skills, to raise their productivity as individuals and as members of the design or programming team. Finally, methods are needed to ensure that software professionals keep up with the state of the art as it rapidly advances.

Without dependable, generally accepted methods of evaluation it is very difficult to be fair during hiring, performance reviews, and promotion evaluation. Research is needed in such evaluation methods, and such research will be strongly tied to software metrics that measure the quality of produced software.

### 3.2 Group Dynamics

Research in group dynamics should focus on the selection/composition of a programming group, its management/facilitation, and its evaluation. Work should also be done in inter-group relations, such as the relations between the government and a contractor, between system users and system builders, and between builders and maintainers.

It may be that selecting group members with different personality types and cognitive styles results in the highest productivity and the best working environment. If so, guidelines to assist managers in the task of assembling such groups would be very valuable and could go a long way towards reducing the productivity-decreasing strain of personal conflict within the programming group. Well-selected groups might also create better programs because of a balance among different people who can see different facets of a problem, et have been trained to communicate with one another and with other groups.

Indeed, assembling a group is only a beginning—work needs to be done to find the best way of team buildings to facilitate productive, pleasant group interaction and to find the best way of teaching the technique of such facilitation to managers.

In the increasingly common case where groups interact mainly through computerized media, for example, electronic mail, research needs to be done on how a sense of community develops and is maintained. Techniques of interest include electronic mail, on-line user consultants, electronic bulletin boards, and teleconferencing.

There is also a need for evaluation of group productivity and group cohesiveness. it is not sufficient for a group to be highly productive if the turnover rate is high, or to create a pleasant working environment if productivity is low. Evaluation of experimental results and industry experience will be crucial to development of a reliable basis for assembling and managing groups.

Inter-group and inter-organizational issues are also important. Among the areas needing investigation are client-centered, participatory design; contractor: DoD interaction; product reviews, and human factors in technology transfer.

### 3.3 Physical Interface

This area is concerned with the physical interface between person and machine. There is a considerable body of literature in this field that can be applied to computers. Studies have been done on workstation design, keyboard design, glare reduction, ambient lighting, etc. Response time and display rates can have surprising impacts on user performance and satisfaction. Such studies need to be evaluated and disseminated within the computer field.

There is also a need for study of new devices and interface methods that have been introduced with computers. Examples are touch panels, other position indicators such as joysticks and lightpens,

voice response and recognition, use of tones and bells, use of color or three-dimensional displays, headgear that is sensitive to eye movement, etc.

### 3.4 Conceptual Interface

Major study is needed in this area, which is concerned with finding the best way to interchange information between user and machine to ensure accuracy, ease of use, and efficiency.

Research is needed to find the best way of handling computer:user dialogs to ensure that accurate transfer of information takes place with the minimum amount of work. Dialogs need to be tailored to the amount of information that the person can handle in short-term memory without becoming confused. Methods of structuring written, graphical, or aural presentations to avoid confusion need to be developed and codified for use by interface designers. Experimentally-verified guidelines for designing command syntax and semantics are needed, as are guidelines for designing hierarchies of commands.

Reliable, accurate complexity measures of presentations to the end-user and to the computer systems designer (e.g., presentations of computer systems designs, data base structures, and program structures) are needed. Development of such measures will have a huge impact on contract management and on programmer evaluation, as well as on programmer and analyst productivity, because maximum acceptable complexity as measured by standard tools can be written into contracts and used as an input for personnel evaluation.

Specialized work in the area of data base and query language use is probably justified because of the growing importance of these areas and their special environments.

### 3.5 Emotional Interface

Many people have a strong emotional reaction to computers. Research in this area will have as its objective to increase user acceptance of computers, reduce tension and error rates, and increase productivity.

One area already being studied is that of response time. The answers are not yet in on whether response time should be constant, should reflect the cost of the command, should reflect the amount of time needed to prepare the next input, or should be a function of some other factors.

People often feel a need to be in control of the interaction with the computer; research is needed to assist in the design of dialogs to reinforce this feeling.

Work is also needed to evaluate ways of modifying dialogs and command syntax/semantics depending on user expertise and style. An inappropriate interaction schema can result in user irritation, leading to high error rates and lower productivity.

Guidelines for preparation of error messages are also important, as the user must be left with a pleasant feeling instead of one of being continually unfavorably judged by a machine.

All the above paragraphs can be summarized by the statement that the computer:person interface must be friendly and productive. Guidelines, design, and evaluation tools, and standards and acceptance criteria must be designed to ensure that introduction of a computer into a work environment results in a more pleasant environment as well as a more productive one.

## 4.0 RELATIONSHIP TO OTHER AREAS

This area is closely related to the Measurement in part because measurement of system complexity and human performance are quite

important. It is also closely related to Integrated Support Environment, Reliability, Software Maintenance, Knowledge-based Systems, Technology Transfer, and Management. Indeed, there are few areas where Human Factors will not make its mark.

## 5.0 PAYOFF ASSESSMENT

As discussed above, the payoff potential of work in the human factors area is extremely high.

- o Personnel selection, evaluation, and team building have enormous potential to improve productivity.
  - In all the published experiments with programmers the variance is huge. A typical finding is a 20:1 difference between programmers earning the same salary with the same length of experience working on the same program. In contrast, a 20-30% improvement from a particular tool or method is considered large. Tremendous gain could be made if we really understood what make good programmers so good and could use this understanding in selection or training.
  - Schmidt (1979) contains a detailed analysis estimating that the federal government could save between $5 to $100 million per year through the consistant use of an existing selection test.
- o Effort involved in computer system use will be greatly decreased because of clearer, simpler, easier-to-learn ways of interacting with the computer.
- o Computer system quality will increase because of the improved environment in which it was constructed. In addition, the improved human interface will decrease errors and increase productivity, thereby improving the quality of the user's job as well.
- o Functionality of computer systems may be increased because user comprehension of complex systems will be manageable by new human factors techniques.

Work in human factors is crucial if major improvements in computer systems are to be obtained, and the work has just started. Funding and focusing of efforts are needed.

## 6.0 REFERENCES


1. Bailey, R. W. [82], Human Performance Engineering, A Guide for System Designers, Englewood Cliffs, N.J.: Prentice-Hall, 1982.

2. Curtis, B. [81], ed., Human Factors in Software Development, Los Angeles, CA: IEEE Computer Society, 1981.

3. Moran, T. P. [81], ed., Special Issues: The Psychology of Human-Computer Interaction, ACM Computing Surveys, Vol. 13, No. 1 (March 1981).

4. Nichols, J. A. [82], chair., Proceedings Human Factors in Computer Systems, March 15-17, 1982, Gaithersburg, MD: National Bureau of Standards, 1982.

5. Ramsey, H. R. and Atwood, M. E. [79], Human Factors in Computer Systems: A Review of the Literature, Science Applicants Inc., Englewood, CO, report SAI-79-111-DEN, 21 Sept. 1979. (NTIS # AD-A075-679).

6. Ramsey, H. R., Atwood, M. E., and Kirshbaum, P. J. [78], A Critically Annotated Bibliography of the Literature on Human Factors in Computer Systems, Science Applications Inc., Englewood, CO, report SAI-78-070-DEN, 31 May 1978. (NTIS # AD-A058-081).

7. Shneiderman, B. [80], Software Psychology, Cambridge, MA: Winthrop, 1980.

8. Theil, C. [81], ed., Human Factors, IBM Systems Journal, Vol. 20, No. 2 (1981).

9. Theil, C. [81], ed., Man-Machine Interaction, IBM Systems Journal, Vol. 20. No. 3 (1981).

10. Wasserman, A. I. [81], ed., Tutorial: Software Development Environments, Los Alamitos, CA: IEEE Computer Society, 1981.

11. Zientara, M. [81], "Looking for Staff with 'Potential'? GE Has a Suggestion: Profile Sheet," Computerworld, Vol. XV No.50 (Dec.14, 1981) p.1.

APPENDIX II.10

# TECHNOLOGY TRANSFER

## 1.0 INTRODUCTION

Technology transfer involves the moving of methods and state-of-the-art technology into practice in software organizations. Advances have been made in developing good software development techniques as well as solving large-scale design problems. Unfortunately, there has been minimal progress in moving this information from research environments into practice, from organization to organization, and within organizations. Unless there is a concerted effort to spearhead this transfer of information, it will not happen. Part of the problem is due to the growth of the number of people working in this business and the relative late arrival of technology.

### 1.1 Technology Transfer Into Use

There are four basic issue areas within technology transfer is important — planning, education, practice and enforcement. The first area is planning and organization. It is worth noting that without the push of government it is doubtful that a real technology transfer effort will happen in the short run. Areas that fit under planning and organization:

> Identification of candidates and selection of items for transfer.
>
> Involvement of the ultimate target organizations and personnel in the research and development of the technology.
>
> The planning of technology transfer strategies.
>
> The preparation of the organization for the change in technology.
>
> The organization of the technology transfer, as well as

mechanisms for maintaining and updating it.
Evaluation and demonstration of the technology transfers.

Education is obviously an important issue. Education must cover all phases and personnel in the software business. It includes the training of the technical personnel and managers, as well as the users and contractors. If the contracting officers are not trained in what to expect and plan for, we will not get quality products in return. Specific education needs across all areas of technology include:

Intensive advance programmer training

Management training

User training

Contractor training.

Learning knowledge is not enough. What we need is skill development and, therefore, a fair amount of practice is needed before developers become highly skilled. There are many ways in which this can occur. Some of them include:

Programmer laboratory (a controlled environment for practicing technology)

Publication of standard designs and specifications

Measurement feedback and evaluation.

The last issue in technology transfer is the enforcement of that technology. This is to guarantee that good technology and good ideas are used in software development. Issues here involve:

Standards

Tool environments

Contractor incentives toward quality products

Widespread use of software R&D products is essential to achieving the productivity and quality goals. Technology transfer of insertion

plans must be part of all the initiative's new product and procedure development plans, and develop in parallel with them. Each product must be developed to a stage where it can be put into general use. This means not only that systems must work and that documentation and training assistance and materials must be provided, but also that systems must embody the best human factors and that documentation and training must be clear and effective. These concerns are essential to technology insertion success, considering the number of organizations and individuals involved in the DoD and defense contractor software industry.

New products and procedures will be accepted and used if they are better than available alternatives, and if sufficient incentives are offered. DoD policy requirements that products and procedures "shall be used" will not be sufficient if contractors perceive the new methods to be costly or risky; they will campaign for waivers. Incentives are needed to defray the costs and reduce the risks of converting from older environments and methodologies to Ada. Incentives are justified by DoD's desire to minimize (or eliminate) diverse, incompatible methodologies. The fewer methodologies, the more software and expertise can be transferred (reused) among defense organizations.

The initiative must face up to the difficulties involved in changing organizations. An organization's adoption of a new technology architypically proceeds through seven stages, as the organization gradually increases its awareness of the technology and its confidence in the technology's benefits.

1) Become aware. The organization must become aware of the technology, typically through publicity in professional and trade journals and newspapers.

2) Notice/Find. Once aware of a new technology, the organization must locate and contact a supplier, obtain brochures, look at demonstrations.

| STEPS OR ASPECTS IN ADOPTION OF INNOVATION / APPROACHES TO TECHNOLOGY INSERTION | BECOME AWARE | NOTICE/FIND | CONSIDER FOR USE | DECIDE TO USE | PREPARE TO TRY | SUCCESSFULLY TRY | PROPAGATE |
|---|---|---|---|---|---|---|---|
| EARLY & WIDESPREAD USER INVOLVEMENT | x | x | x | x | x | x | x |
| INFORMATION TAILORED FOR DECISION MAKERS | | x | x | x | | | |
| SOFTWARE PRODUCTS CONSISTENT WITH ADA/APSE | | | x | x | x | | x |
| TRAINING | | x | x | x | x | x | x |
| CLEAR AND COMPLETE DOCUMENTATION | | | x | x | x | x | x |
| HUMAN ENGINEERED PRODUCTS | | | x | x | x | x | x |
| INTEGRATED PRODUCTS | | | x | x | x | x | x |
| DEMONSTRATIONS | x | x | x | x | | | |
| QUANTITATIVE EVALUATIONS | | | x | x | | | |
| FULL SUPPORT FOR SUCCESSFUL PRODUCTS | | | x | x | x | x | x |
| INDEPENDENT VERIFICATION AND VALIDATION | | | x | x | | | |
| WIDESPREAD PUBLICITY AND INFORMATION | x | x | x | | | | |
| STANDARDIZATION POLICY | x | x | x | x | | | |

# IMPACT OF APPROACHES ON STEPS IN TECHNOLOGY ADOPTION

x SIGNIFICANT IMPACT

LITTLE OR NO IMPACT

FIGURE 10-1

3) Seriously Consider. Once the technology is understood, the organization evaluates the costs and benefits of using it. A committee may be formed to examine alternatives and make recommendations.

4) Decide to Try. Management decides to try the technology, based on in-house evaluation, consultant's advice, and competitor's positions.

5) Prepare to Try. In preparation for using the technology, management may create a pilot project or designate a department to bring the technology into the organization.

6) Successfully Try. Successful in-house experience in a pilot project or by a special group is important to the organization's acceptance of the technology. Formal and grapevine reports of the technology's benefits foster confidence in the technology within the organization.

7) Propagate. If all these steps are successful, the organization will be ready to adopt the new technology. Management can then be reasonably certain that the technology will be accepted and used, and can propagate if throughout the organization.

The figure shows how initiative activities, if done properly, will influence the acceptance of R&D products. Such activities are described below.

- User Involvement. Users will be involved from the earliest stages, when user groups will be formed. Workshops, publications, prototypes, and demonstrations will keep users aware of progress. Users will be involved in generic tool efforts through several phases of public comment and refinement on proposed methodology requirements.

- Information Tailored for Decision Makers. Executives and managers need management summaries, cost/benefit analyses, and management presentations to help them decide to try initiative products. Technical people need detailed documentation, analyses of experimental results, and technical support to respond to management's requests for technical evaluation of products.

- Software Products Consistent With Ada. All initiative products will be consistent with Ada and its supporting environment. Organizations that have already adopted Ada,

through MAPSE and later developments, will find that products readily fit into their environments. Tools will be written in Ada to facilitate transportability among Ada hosts.

- Training. Product developers will provide training materials and trained instructors to teach users. Training effectiveness must be demonstrated.

- Clear and Complete Documentation. Documentation includes user guides, reference manuals, and maintenance manuals. Good user guides and reference manuals assure that answers to questions can be readily found. Clear maintenance manuals assure that products are well designed and can be well supported. The effectiveness of documentation will be part of product demonstrations.

- Good Human Interfaces. A goal of all product development will be to make all interfaces with users compatible. This implies consistent syntactic and semantic conventions (e.g. similar meanings for similar expressions and only one way to get on-line help or abort a runaway job). Meaningful error indications are a necessity, and products should incorporate the latest ideas on error recovery.

- Integrated Products. Because the initiative's products will fit well together, the use of some products (perhaps early ones) will encourage the use of others. New products can be incorporated into the software environment without concern about input or output acceptability.

- Demonstrations. Each contractor must propose realistic demonstrations for his products in operational system upgrades, experiments paralleling actual system programs, or new system programs, if possible. Products must be demonstrated in at least two settings to demonstrate the effectiveness of training. One demonstration must involve a contractor group other than the creating group (though perhaps within the same contractor organization).

- Quantitative Evaluations. Quality metrics developed will help to evaluate product usefulness. Quantitative evaluation of demonstrations will be required to provide measurable evidence of a technology's value.

- o Full Support for Successful Products. Potential users need to be convinced that products they are asked to adopt will be supported (i.e. maintained and improved) until superseded by improved products that will be supported equally well. (Transition assistance will be necessary when upward compatibility is unfeasible.) This is probably the most important consideration in an experienced user's decision to accept a new product.

- o Independent Validation and Verification. Independent validation and verification (V&V) of products assures potential users that products have been well designed, well built, and well tested. Independent V&V can reduce the guinea-pig anxiety that affects decisions to adopt new products.

- o Widespread Publicity and Information. Information on products and demonstrations will be publicized throughout the DoD community via direct mail, the trade press, professional journals, workshops, and conference presentations.

- o Standardization Policy. Although promulgation of DoD standards does not compel their use, it does compel each contractor to be aware of, and consider the use of standard products and procedures. The "big stick" policy approach must be supplemented with incentives and means to encourage adoption.

Early involvement of users and experts should be sought to guide development and promote acceptance of products when they become available. Informal user groups could evolve into formal, continuing organizations to determine operational needs and evaluate proposed solutions. Several different groups, for software professionals and for various application areas, might be formed.

Groups of experts could review interim proposals and products (with allowance for possible conflicts of interest) and advise how to accomplish technology insertion within DoD and defense contractor software organizations. Industry expert reviewers will also be channels for technology transfer into their organizations. In addition, mechanisms are needed for coordination among DoD personnel involved in short, medium, and long term software R&D.

## 1.2 Full Technology Pipeline

While the most important and difficult steps in achieving early, widespread use of new technology are the final ones of transferring it into and throughout the using organization, earlier steps in the innovation process are also important.

This innovation process includes the entire evolution of an idea from the first conception through its widespread utilization. This process involves all the DoD organizations in software R&D, technology insertion, and propagation as well as those organizations in standards, acquisition, management, and performance of the work using the innovation. This process might be characterized as the "technology pipeline" through which innovations flow.

Figure 10-2 shows a diagram of this technology pipeline. The innovations which flow through this process may be technical, managerial, procurement, or even educational. Generally, innovations flow one way and requirements the other. However, innovations may originate anywhere in the flow, including within organizations composed of practitioners. In addition, changes in DoD acquisition practices and standards usually require their own special subflows for reaching agreement. Finally, educational institutions offer a potential aid to the DoD community's technology transfer effort which is for software currently receiving little support and limited use.

## 1.3 Objectives

The overall objective of using an innovation sooner has, on a trivial level, two subobjectives: (1) conceive of the innovation earlier, and (2) reduce the delay between conception and widespread use. While the first of these can be influenced, the second offers the more straightforward opportunities for improvement. Note that a year saved during R&D is potentially just as worthwhile as a year saved

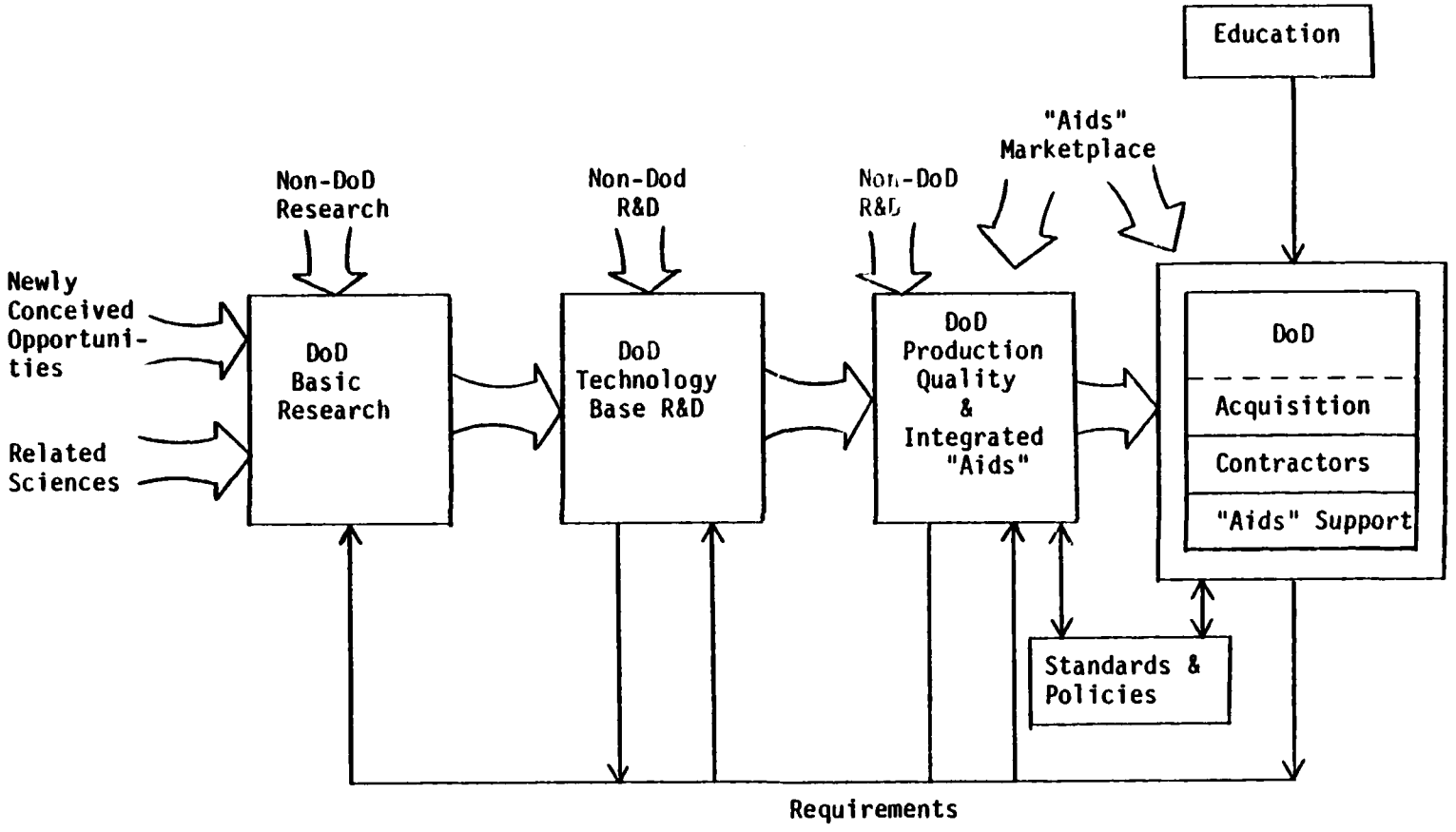



FIGURE 10-2: TECHNOLOGY PIPELINE

during technology transfer. A list of objectives for technology pipeline improvement certainly should include objectives to

- o Initiate the right long-term R&D
- o Nurture the most beneficial innovations, i.e., good decision making
- o Obtain rapid, smooth transitions through and between R&D stages
- o Achieve rapid, smooth injection and propagation of innovations into use
- o Use innovations throughout DoD community
- o Have near-, medium-, and long-term improvement, i.e., keep pipeline balanced.

On the more negative side, DoD must be concerned to also

- o Limit undesirable international technology transfer
- o Avoid unwarranted duplication of effort
- o Achieve all these objectives for an affordable price.

Another possibility is to encourage innovations which are not entirely DoD funded. DoD might establish structures to encourage individual efforts which would lead to benefit for the DoD community via marketplaces for truly reusable application software or compatible software tools. Situations where existing software or environments can be improved incrementally and cumulatively by many independent efforts, each providing some "added value," would be extremely useful.

Measurables can be associated with many of these objectives. These measurables fall roughly into two groups; one set addresses the R&D pipeline and the other addresses the use of innovation. The set of measurables related to the R&D pipeline includes

- Elapsed years in R&D pipeline
- Fraction of projects at each stage which are picked up by the next stage
- Profile of levels of effort in stages of pipeline in aggregate and by subareas (as gauge of pipeline fullness or balance)
- Other stage by stage measures, e.g., time in stage, evaluation done, lack of interstage transfer delays/gaps, stability of funding, etc.

The set related to use in the DoD community and the state of practice includes

- Level of dissemination and use of products, e.g., advanced DoD environment (see Appendix II.1) tools, management aids, or model contracts
- Dollars spent across DoD on application projects using an advanced DoD environment, other modern environments, and primitive environments
- Dollars spent by more inclusive up-to-dateness class measures for project practice
- Product user satisfaction surveys
- Surveys of up-to-dateness of personnel in community
- Organizational state of practice rating instruments for contractors
- Percent of reusable software in application systems
- Volume of non-DoD-funded but environment compatible tools available in marketplace
- Delay in years before undesirable international technology transfer occurs.

Together these lists of objectives and measurables indicate that DoD can establish meaningful measures which will guide and measure progress in pursuing technology transfer.

## 2.0 CURRENT STATUS

There exists very little organized DoD effort towards transferring software technology. A notable exception is the Data Analysis Center for Software at the Rome Air Development Center operated by IIT. Technology transfer requires organization, planning, and evaluation concerning what technology to choose. Very few results exist in this area, although there has been some experimentation and evaluation.

Unfortunately, these results follow no standard, are offered at varying levels of rigor, and are often given just for specific techniques that do not involve full methodology; that is, they do not cover the full life cycle nor are they tailored to any specific project or organization. When there is education, it is often geared only to programmers and not to managers and users.

There is almost no effort to permit the training and skill development of programmers and/or managers and users. The only alternative available is on-the-job training in real projects. This often is dangerous because the methodology then receives low priority since the effort to "get the job done" takes precedence. This lack of practice causes poor implementation of methodology.

There is practically no publication of standard designs or specifications or code. The noted exception is well-developed areas such as compilers. This is an extremely important issue if we are to stand on each other's shoulders to build a more advanced technology in software engineering.

There is almost no feedback to developers on effective use of methodology. Very little data is collected and very little information about how well people have performed is provided as feedback.

There do exist standards for development. Unfortunately, they are usually at a very low level and tied to poor and outdated tech-

nology. It is very difficult to provide reasonable, up-to-date standards, but it is essential.

There are tools that help in encouraging (rather than enforcing) good methodology; for example, use of high level languages. But unfortunately, it is just too small a part of the life cycle and a more comprehensive tool environment would go a long way toward encouraging and enforcing technical standards.

There are no incentives given to organizations for technology transfer and the use of good methodology. In fact, sometimes the opposite occurs; organizations are rewarded financially for taking a long time to develop the software. They are given maintenance contracts to maintain badly produced and faulty software. Part of the problem is the cost-plus contracting mechanism. Besides, there are very few measures for evaluation and very few rewards for an organization to deliver a project of high quality.

The state of technology transfer is quite poor and much needs to be done. The problem is that technology takes a long time to get into the field, but this is exaggerated in the software area by the fact that the field has grown so dramatically in the past thirty years. Original programmers of thirty years ago are still in the business. Much has happened in technology in that thirty years, but it has not affected many of the programmers. There is no common education base from which to start the retraining.

## 3.0 SPECIFIC RECOMMENDATIONS

Technology transfer is an area where substantial innovation and increased emphasis is needed. Organizationally and technically an aggressive and well-focused innovative effort is required. More of the same will simply not suffice.

There are several specific thrust areas that are recommended for improvement. First, experiments are needed in understanding local

milieus; that is, the effect of the innovations on specific applications, management organizations, etc. There are also experiments needed in evaluating methods for those situations. What are the best techniques to apply for a specific application and a specific organization subject to the specified constraints?

There is a need for proposals for technology transfer organizational strategies that include the full education practice, evaluation, diagnosis, remedial education, and enforcement.

We need more trained people. There is a need for consistent training programs for programmers, analysts, managers, users, contractors, etc. In this way, they may speak a common language and understand their role in the application of the full life cycle methodology.

Techniques are needed for providing practice and skill development. This is in addition to the normal education program. Some approaches are:

1. The publication of standard designs so that existing knowledge can be shared and reused (and not reinvented every time). These standard applications and designs could then be built upon or improved. Then we can stand on each other's shoulders in the development of technology. These standard designs can be used in the education program. It obviously requires a mechanisms for publication and dissemination, as well as a good set of notations of expression.

2. The development of programmer laboratories. A programmer laboratory is an environment in which different methods can be tried and various skills developed. Although this concept is primarily experimental, it could be used to build real products, such as tools. The idea of skilled development in any technology, the software engineering methodology specifically, is underrated but extremely important.

3. Measurement feedback is absolutely important in evaluating how well the methods are being applied, in explaining what has gone right or wrong, and in modifying the method for the environment. Evaluation is a key issue both in skill development and enforcement and must be treated as such.

Education and practice, at least in the beginning, are not enough. Good methods and techniques need to be enforced. This enforcement requires guidelines and standards and support in the forms of tools and incentives. Specific proposals here might be:

1. The guidelines and standards must support technology transfer and not become outdated. We need something like living standards; e.g., a set of guidelines and interpretation board that will translate the guidelines, modify them to specific projects, and update them as technology changes. This group can act as a "supreme court" in interpreting and "congress" in modifying the laws as required.

3. Contractor incentives toward quality products is essential. We must require methods and evaluate quality to assure good methods are being used. Here metrics and measures for evaluation are extremely important, both as incentives in the original contract as written, and as evaluation after the product is delivered.

Transition of software to Ada must be addressed. Many billions of dollars worth of non-Ada operational, military software exists and will continue to exist for many years. This software is written in several languages, without standards, etc. Current development projects and those beginning before Ada is ready will face similar problems. The dilemma is that it is impractical to convert operational software and software environments to Ada all at once, yet it is undesirable to continue to maintain (i.e. correct, change) all this non-standard software in non-standard environments, which would then have to be supported too.

Many initiative results could be used or adapted to facilitate maintenance (e.g., correction of errors and adaptation to new requirements of pre-Ada software). Many of the tools and practices developed will be language-independent, and could help current projects directly. Some other transition possibilities are tools to convert programs from older languages to Ada, compilers whose output would be compatible with Ada compiler output, and old-language-

dependent tools compatible with Ada and developed in parallel with the Ada tools. Figure 10-3, "Transition Strategy Labyrinth," graphically illustrates the various ways to get from today's situation to the integrated Ada environment. (Note that "Mixed Languages" means that relatively unchanged parts of operational software remain in old languages, but significantly changed parts are recoded in Ada. Similarly, "Mixed Environment" means that software is partially supported in current environments and partially supported in Ada. Implementation of any of these possibilities not only supports maintenance of existing software, but also facilitates transition of personnel to Ada.

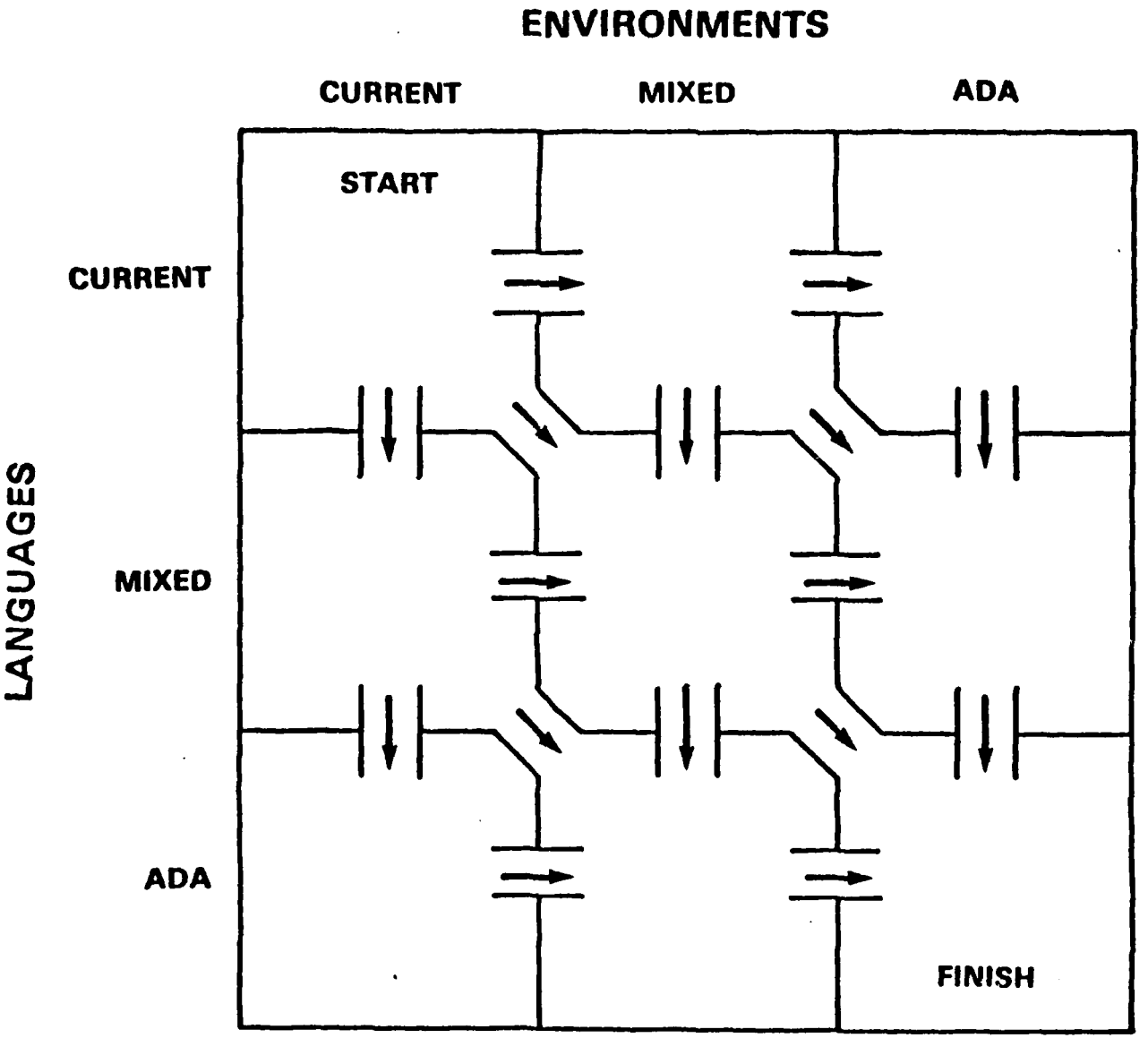
The cost of making a tool or practice available and the impact of its introduction on costs, schedules, and personnel need to be evaluated for each project.

## 4.0 RELATIONSHIP TO OTHER AREAS

Good technology transfer is a necessary condition for the success of all other areas in this Appendix. It is essential that whatever other technology areas are chosen that their transfer into practice be accounted for. It does no good to develop new technology and not intuse it into the development and maintenance organizations.

## 5.0 PAYOFF ASSESSMENT

Applying good methodology should clearly reduce the effort in development and maintenance and improve the quality of the product. It should permit the development of more functions simply by permitting an organization to (1) build on old ideas (from publication of designs), (2) better understand what has been done, and (3) better control what is being done. Many products are currently built that are unusable. Good technology transfer should minimize that number and in that way increase function and user responsiveness.




TRANSITION STRATEGY LABYRINTH

FIGUR 3

Substantial initial payoff can come from simply propagating best practices and reducing the delay between research state of the art and the state of practice. Estimates range up to 250% productivity gains although a more modest but still large percentage, such as 50-100%, seems reasonable.

As the advances in the state of the art accelerate under the STI, faster technology transfer becomes even more valuable. If the state ot the art is advancing such that a 20% per year potential increase in productivity is occurring, then a reduction of three years in the delay in getting R&D results into widespread use would itself yield a whopping 73% increase in productivity.

## 6.0 REFERENCES


1. Baber, R. L., Software Reflected, North Holland, 1982.

2. Gansler, J. S., The Defense Industry, MIT Press, 1980.

3. Glass, R. L. (ed.), Modern Programming Practices: A Report from Industry, Prentice Hall, 1982.

4. Lane, H.W., R. G. Beddows, and P. R. Lawrence, Managing Large Research and Development Programs, State University of New York Press, 1981.

5. Martens, J. and L. Duvall, "The Role of an Information Analysis Center in Software Engineering Technology Transfer," NCC 1980, Vol. 49, pp.677-682, A FIPS Press, 1980.

6. Mercer, J. L., and R. J. Philips, Public Technology, AMACOM, 1981.

7. Roman, D.E., Science, Technology, and Innovation: A System Approach, Gird Publishing, Inc., 1980.

8. Redwine, S. T., and G. R. Berglass, A Proposed Plan for the DoD Software Technology Initiative, MITRE (MTR-82W-0091), May 1982.

9. Salasin, J. (ed.), The Management of Federal Research and Development: An Analysis of Major Issues and Processes, MITRE (MTR-7657), 1977.

# APPENDIX II.11

# MEASUREMENT

## 1.0 INTRODUCTION

Progress in any scientific discipline depends upon the ability of practitioners in the field to objectively measure the phenomena of interest. The importance of measurement in understanding an area of inquiry is succinctly expressed by Lord Kelvin's well-known statement,

> "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the stage of science."

Our current lack of understanding of the software development and maintenance process is reflected in the absence of generally understood and applied measures.

### Stages of the Scientific Method

The scientific method, which has led to dramatic advances in other fields, can be applied to the realm of software development and maintenance with similar expectation of substantial benefits. The scientific method involves a sequence of stages which rest on measurement as a foundation. The first stage involves describing the area of study. In this descriptive stage, the objects and characteristics of interest are defined and categorized. In addition, measures are developed to record values of these characteristics. The major purpose of the descriptive stage is to establish a baseline

for the phenomena under investigation and to discover the basic relationship among different sets of variables. In choosing the metrics to quantify the variables, it is important to consider their reliability (i.e., Will two people obtain the same value for this metric when it is applied to the same object?) and validity (i.e., Are we really measuring the characteristic that we think we are?). When applied to software, the descriptive stage involves an attempt to provide a complete and useful characterization of the software process and product and to discover relationships among the primary variables of interest.

The second stage involves a process of induction in which the relationships observed in the descriptive stage are used as the basis for constructing theories to explain these relationships. A theory should account for these relationships with some generalizable assertions, that is, with assertions that extend beyond the specific conditions under which the relationships were observed.

The third stage involves a process of deduction in which specific predictions or hypotheses are generated from the theory. The predictions often involve outcomes resulting from novel or extreme combinations of conditions.

The fourth stage involves a process of hypothesis testing in which empirical evidence (data) is obtained in an attempt to confirm or disconfirm the theory. This stage is particularly difficult in the world of software where conducting a controlled experiment using any reasonably large-sized program is prohibitively expensive. Often the hypothesis testing must be performed within a laboratory environment with the hope that the results will be generalizable to the industry at large.

A fundamental characteristic of the scientific method is the continual interplay between the empirical and the theoretical: empirical observations are used in constructing the theory; the theory, in

turn, guides the collection of further data; these results are then used to further refine the theory.

## 2.0 CURRENT STATUS

Since the emergence of software engineering as a discipline, there has been a growing recognition of the importance of meaningful measures to describe and characterize software and software development. In 1979, the Office of Naval Research commissioned a panel from industry and academia to assess the current state of the art and to make recommendations for research areas that are likely to realize a benefit in the near future. The conclusions of the panel were threefold: (1) data-collection should be routinely incorporated into the software development process to help satisfy the dearth of data about software development, (2) there is a need for careful experimentation in the field, and (3) we need to understand how to map results obtained with one language and environment onto results obtained with other languages and environments (Perlis, Sayward, & Shaw, 1981).

More recently, the IEEE Computer Society has established a working group to develop a standard for software reliability measurement. The goal of this project is to specify metrics which can be applied throughout the software life cycle to predict and measure the reliability of the software product.

While the need for measurement has been clearly recognized, actual progress in this area has been sporadic and not substantial. Most of the work falls into the descriptive stage of discovery and has involved the identification of possible measures to be applied to the software product or process. Less common have been attempts to collect actual data from software projects or to document the relationship between two or more sets of measures. There have been even fewer attempts at true theory construction and hypothesis testing.

In summarizing the current status of software measurement, it is important to distinguish between a metric, a distribution, and a theory. A metric is a quantitative measure of the degree to which the software process or product possesses a certain characteristic; the term "metric" and "measure" are used here interchangeably. A distribution is a vector of metrics which represents a sequence of values across time or across components of a system or even across different systems. A theory is an abstraction which attempts to explain the relationship between two or more distributions in the form of general assertions which allow one to extrapolate beyond the immediate conditions of observation.

## Identification of Measures

As noted above, most of the measurement work to date has involved the identification of possible measures. The general types of measures which have been suggested can be categorized into process and product metrics (Basili, 1980). Process metrics provide information about the software development or maintenance process and can be sub-categorized into measures of resource consumption and various change and error statistics. Product metrics provide information about the actual software. These can be sub-categorized into size, control, and data metrics. Product metrics also include various execution statistics.

A substantial amount of effort has been directed at attempts to measure more abstract characteristics of the software product. The measurement of software complexity, in particular, has received a great deal of attention. The sheer number of proposed complexity metrics has led one researcher to comment that there are more complexity metrics than practicing computer scientists" (Curtis, 1979). In an even more ambitious vein, McCall, Richards, and Walters (1977) suggested an entire framework for the measurement of software qual-

ity.

Data Collection

In addition to various proposals for metrics, there have been a number of data-collection efforts reported in the literature. These have involved attempts to collect and categorize one or more types of measures from an on-going software project to obtain a distribution of the measure across one or more life cycle phases or across modules of a software system. Endres (1975), for example, reported a frequency distribution for different types of errors discovered during the testing of a major system enhancement. Several studies have taken a more goal-directed approach in which measures were obtained to answer specific questions about the effect of a method or tool (Basili & Weiss, 1980).

Discovering Relationships

Finally, there have been a number of attempts to describe the relationship between two distributions of metrics, most notably between project size and human effort. The size-effort relationship has been of particular interest because it can serve as the basis for later resource estimation (that is, one can predict resource requirements from an estimate of project size). These empirically-based resource estimation techniques (Walston & Felix, Bailey & Basili, 1980) fall within the descriptive stage of science since they represent attempts to discover the nature of the function relating effort to size rather than attempts to explain or go beyond the observed function.

Theory Construction, Deduction, and Hypothesis Testing

There are a few examples of true theory construction and hypothesis testing in the software industry. These include Halstead's Software Science (1977), several theoretically-based resource models (e.g., Musa, 1980). The construction of these

theories marks the appearance of the second stage of scientific discovery in software engineering; they are, however, still in a developing state.

It should be noted that there have been a number of controlled experiments to evaluate specific tools and techniques (e.g., Sheppard & Kruesi, 1980). While many of these have produced interesting results regarding the relative effectiveness of the specific tools and methods being evaluated, they have not, for the most part, been conducted within the framework of a guiding theory. As such, they have contributed to an understanding of a portion of the software development process, but they have not built upon a more general framework of theory construction and hypothesis testing.

## 3.0 RECOMMENDATIONS

The fundamental recommendation is to follow the principles of the scientific method which have been successfully applied in other fields. This means that the general framework of description, theorizing, hypothesizing and evaluation should be viewed as a point of departure for all research and all work should fit into one or more of these stages.

### Establish A Descriptive Framework

A solid descriptive foundation is the proper basis for theory construction; it will also provide immediate practical benefits. These benefits include an understanding about the types of activities that consume the most effort and an understanding about the types of errors that occur most frequently. In the short term, this understanding can lead to more cost-effective investments in tools and techniques.

The establishment of such a descriptive foundation will require contributions from multiple sources. In order for these contributions to be mutually beneficial, it will be necessary to establish a

common set of definitions and measures. This commonality will be essential to promote meaningful communication and to prevent unnecessary duplication of effort.

Establish Common Definitions. As a first step, it will be necessary to establish common definitions of the measurable products and phenomena associated with a software development effort. Currently, no such commonality exists. Even conceptually simple measures such as lines of code are not computed consistently and are affected by such extraneous factors as programming style.

Standardize a Basic Set of Metrics. Beyond establishing a common definition for each metric, it will be necessary to standardize a basic set of metrics to be collected from any software development. Typically, a software development process can be characterized by the distribution of effort and by the types and numbers of changes and errors that occur during the development. The software product can best be described by some measure of its size, complexity, and function. It is also important the characterize the environment in which the development occurred. This means that the use, or even the degree of use, of development methods and tools must be quantified in some communicable way. Finally, it will be necessary to expand measurement beyond the design and code phase of the life cycle into the requirements specification phase and into the operation and maintenance period so that the longer-term goal of theory construction can cover these activities as well.

Standardize Measurement Procedures. When both a set of common definitions and an essential data set for describing a development have been adopted, it will become necessary to unify the procedures with which to collect the desired data. It should be possible to capitalize on the advent of Ada and the required commonality of the APSE in order to help fulfill this requirement. In fact, a minimum set of standard data could become a deliverable with any contracted

development. With proper configuration control and through the use of specific tools (such as will likely be found within an APSE) the collection and classification of these data should incur little or no overhead for the developers.

Construct Theories and Test Hypotheses The next stage, which must be built upon a solid descriptive foundation, involves the construction of theories to explain the observed distributions and relationships. This is not a trivial step and is more involved than generating statistical models through regression techniques (which is a part of the first stage). While a theory should be based upon observed patterns and relationships, it should allow one to extrapolate beyond these observations.

Hypothesis testing is also a non-trivial endeavor. One important consideration when designing a test for a given hypothesis is the tradeoff between internal and external validity. A controlled laboratory experiment maximizes internal validity by controlling all potential confounding variables; at the same time, its artificial nature reflects a lack of external validity (i.e. generalizability to other, non-laboratory situations). On the other hand, case studies, which have near-perfect external validity, can not be conducted with the control possible in a laboratory and, hence, lack internal validity.

At this time, it is not known how to best conduct hypothesis testing in the software industry in a way which will result in an appropriate balance between internal and external validity. In all probability, no one experimental paradigm is "best-suited" to all research in the area of software technology. Rather, there is a need to explore alternative experimental paradigms and select the one that is most appropriate for the situation. Research techniques borrowed from experimental psychologists are quite useful for making evaluations such as the adequacy of the human-computer interface, because

they involve observations on a single programmer. On the other hand, engineering concepts may be more appropriate for evaluating other qualities, such as software reliability. In addition, new techniques such as the "quasi-experimental" designs which have been successfully applied in evaluative research in education (Cook & Campbell, 1979) may be useful for evaluating the effectiveness of tools and techniques on larger projects where groups of programmers are involved.

As was suggested, these stages of scientific discovery are interactive. A successful test of a hypothesis will likely promote some augmentation of its underlying theory while the unsuccessful test of a hypothesis may suggest some alternative theory. This application of the scientific method of discovery to software engineering will provide the foundation for a body of knowledge upon which tools and techniques, methods and methodologies, and software models can be built and evaluated.

## 4.0 RELATIONSHIP TO OTHER AREAS

Measurement will provide a foundation and a focus for the entire Software Initiative since it provides the means for assessing the impact of advancements in the other activities of the initiative. By defining, selecting and gathering relevant data from a large number of software developments, baselines can be established which will serve to describe the current state of the software industry. Only then can we know what constitutes an improvement from those baselines.

The arena of software development and maintenance is far too complex and its description too nebulous for us to understand the effects of its numerous interacting variables without a systematic measurement program. Responsible measurement can allow us to discern some important patterns which will, in particular, assist the Management, Reliability, Human Factors, and Technology Transfer areas.

Relationship to Management

The development, validation and refinement of life-cycle models will provide valuable tools for project management. Of particular relevance to management are resource models which can be used at the beginning of a project to support cost estimation, scheduling and staffing. Ideally, a model will allow management to understand not only the tradeoff between costs and schedule but tradeoffs between these two factors and specific quality goals as well. Such a model can also serve an important control function since performance deviations can be quickly detected and the causes determined.

It is worth pointing out that there is a symbiotic relationship between management and measurement. Not only is measurement key to the success of the management area, the management area will be key to the success of measurement due to increased control over the software development process. This will serve to reduce the noise (unaccountable variance) in the data collected which will, in turn, promote greater predictability.

Relationship to Reliability

With respect to reliability and quality assurance, measures such as mean-time-between-failures and error density provide useful inputs to reliability models. Most of the measures that currently exist, however, are product metrics and they cannot be taken until the product is nearly complete. The continued development of process metrics will allow measurements to be taken at an earlier point in the life cycle. These metrics can indicate whether quality goals are being met or where problems may be occurring at an earlier point in the process than is currently possible.

It is also noteworthy that with an improved understanding of the various software quality factors and, in particular, an understanding of how to measure these factors, it will be possible to specify qual-

ity goals in objective, quantitative terms. Incentive awards can then be directly tied to the degree to which these goals are actually met.

Relationship to Human Factors

Measurement will be instrumental to advances in the human factors area. Issues arising in the design of the human-computer interface are particularly suited to small-scale experimentation; however, work in this area has only recently begun to appear. While the current methodology borrows techniques from both the psychology and engineering disciplines, the experimental designs used by psychologists will probably prove to be the most appropriate for this activity in the long run.

Relationship to Technology Transfer

The development of metrics is particularly crucial for technology transfer. Meaningful cost/benefit analyses will be required to demonstrate to project and company management that a particular change, tool, or technique is worth considering. Along with knowing where a given change may help, we will also know where it may not. However, measurement to establish baselines will be required before these analyses can be carried out.

## 5.0 PAYOFF ASSESSMENT

The ability to measure attributes of the software product and process throughout the life cycle lies at the heart of software engineering and management. There can be no true science of software engineering, no real discipline, without the ability to attach numbers to characterize the product and process and examine relationships among these. a major payoff to be derived from measurement is a cumulative knowledge to support advances in all areas of software technology.

In the short term, a solid descriptive foundation will provide a better understanding of a number of aspects of the development of software which are incompletely understood at this time. Information such as what activities consume unexpected amounts of effort or what types of errors are the most expensive to correct will emerge from these data. This information can lead to a more cost-effective investment in tools and techniques which are specifically designed to reduce the time spent on the most costly activities and minimize the most expensive types of errors. Measurement will also allow for an assessment of whether the chosen tools and techniques are having the desired effect. In addition, industry-wide information about software developments will help direct research efforts and tool development efforts in areas where there is the greatest need and, hence, the greatest potential for payoff.

In the longer-term, measurement should be carried out within the context of theory construction and validation. The benefits resulting from these activities will apply to several important areas of software technology as outlined in the previous section (Relationship to Other Activities).

In the final analysis, the biggest payoff for meaningful measurement will be the ability to quantify and assess the impact of advances in the other thrusts of the Software Initiative. Without measurement, and the establishment of baselines, no one will ever be able to determine whether changes undertaken in any of the areas outlined in the initiative have had an effect. Without a theoretical framework within which to view these observed changes, they will represent, at best, isolated improvements rather than an integrated program of advancement. The use of measurement, followed by the construction and testing of theories in accordance with the scientific method, will ultimately ensure a further understanding of the

processes underlying software development and maintenance and continual improvements in the quality of the final product.

## 6.0 REFERENCES


1. Bailey, J. W., & Basili, V. R. "A meta-model for software development resource expenditures," Proceedings of the Fifth International Conference in Software Engineering, New York: IEEE,1981, 107-116.

2. Basili, V. R., (Ed.) Tutorial on models and metrics for software management and engineering, New York: IEEE, 1980.

3. Basili, V. R., Weiss, D. M. "Evaluation of a software requirements document by analysis of change data," Proceedings of the Fifth International Conference on Software Engineering New York: IEEE, 1981, 314-323.

4. Cook, T. J., & Campbell, D. T., Quasi-Experimentation: Design and analysis issues for field settings, Chicago: Rand McNally, 1979.

5. Curtis, B. "In search of software complexity," Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost, Silver Spring, Md.: IEEE Computer Society Press, 95-106.

6. Endres, A. "An analysis of errors and their causes in system programs." IEEE Transactions on Software Engineering, June 1975, 140-149. Halstead, M. H. Elements of software science, New York: Elsevier, 1977.

7. Kelvin, William Thompson Lord, from Popular Lectures and Addresses, 1891-1894.

8. McCall, J., Richards, P., & Walters, G. Factors in Software Quality, Three volumes: NTIS AD-A049-014, AD-A049-015, AD-049-055, 1977.

9. Musa, J. D. "Software reliability measurement." The Journal of Systems and Software, 1980, 1, 223-241.

10. Perlis, A. J., Sayward, F. G., & Shaw, M. (Eds.) Software Metrics, Cambridge, Mass.: The MIT Press, 1981.

11. Putnam, L.H. "A general empirical solution to the macro software

sizing and estimating problem," IEEE Transactions on Software Engineering, July 1978, 345-361.

12. Sheppard, S. B., & Kruesi, E. "The effects of the symbology and spatial arrangement of software specifications in a coding task," Proceedings of Trends and Applications 1981, New York: IEEE, 1981.

13. Walston, C., & Felix, C. "A method of programming measurement and estimation," IBM Systems Journal, 1977, 16, 54-73.

## APPENDIX II.12

## MANAGEMENT

### 1.0 INTRODUCTION

The management aspects of software engineering build upon and extend the contributions of each of the other assessment areas. Management, as used within this section, deals with the ability of people to motivate other people to build "quality" products on-time and within budget. It encompasses the functions of planning, organizing, staffing, directing and controlling. It involves many people at many levels within an organization. It responds to both strategic and tactical issues and needs. It employs project and acquisition management principles. It uses the software engineering technology base to minimize technical risk and achieve project, organizational and national goals. It develops its own technology base in an attempt to address the management issues that limit the potential utilization and effectiveness of new technology.

The ultimate aim of this management initiative is to improve overall DoD software productivity and quality over the next five years. Achievement of this goal will be accomplished using a three-pronged attack which disciplines the products and professionalizes the people involved. The key investments are in people and in providing them with the technology they need to improve their judgment and their decision-making capabilities.

### 1.1 Process

The life cycle processes involved in software development and maintenance embody the work activities conducted to generate products integral to the success of organizations and weapons systems. The

processes involved are broken into distinct steps to which people, products and technology can be assigned. Management's task is to use the software engineering technology base to identify each step's work activities, order their accomplishment, identify and reduce their risk, predict their cost and schedule, measure their progress, control their accomplishment and to integrate their products into a working system. The process discipline is important to management because it allows the evolutionary, time-related work flows associated with software developments to be modeled and controlled. Improved understanding and regulation of these processes is needed so that management can establish meaningful, time-related cost, schedule and technical performance measures and models that are both predictable and predictive.

1.2 Product

Ultimately products of the software development process should be well-documented, simple, functional packages designed to be reusable by multiple organizations for a range of applications. Much of this initiative deals with developing the technology base that allows this desire to happen. Management's task is to use this software engineering technology base to identify meaningful work products, define their content, order their development, measure their quality, control their cost and ensure that minimum standards of workmanship are met. Product standards need to be defined for use throughout the life cycle so that everyone involved knows what is expected, when, and in what form. The product discipline is important to management because it defines what the results of each step of the software development process are and what each should contain. Improved definition of the product is needed so that management can establish realistic, time-related, quantitative measures of "goodness" for use in evaluating developments throughout the life cycle. The measures

must relate quality to cost so that managers can make effective tradeoffs between technical and programmatic variables.

### 1.3 People

The people involved in the software life cycle are creative professionals. Each must be armed with the skills, knowledge and abilities needed to get his/her work done. Each must have their communications and interpersonal skills developed so that he/she can work with others and can form teams. Management's task is to develop a people-oriented technology base that allows them to define each software job's skill requirements, develop their people by providing them with the knowledge and abilities to meet these requirements, motivate their people singly and in teams to do the work, make their people feel accountable for results, reward their productive people, champion innovation and provide an environment condusive to growth, achievement and recognition. Management development must also be addressed so that those empowered to make decisions that affect people understand their psychological as well as the technical ramifications. Management needs to improve its understanding of the people aspects of the software job so that it can foster the professionalism needed to get the job done.

## 2.0 CURRENT STATUS

The state-of-the-art of software management has progressed rapidly over the last few years. Proven project and people management techniques have been adapted to software to provide the basis for organizational improvement. Much of the mystique of software management has been dispelled and managers know that they can apply classical management approaches founded upon "good" engineering principles to realize success on their software projects. Unfortunately, the state of the practice has not kept up with the state of the art. As a result, much of the management technology base that exists for software is not being used. New and innovative ways of delivering

this technology base to managers must be pursued to remedy this unacceptable situation.

## 3.0 RECOMMENDATIONS

The following specific recommendations will provide DoD managers with the ability to utilize the technology base to deliver quality software products on-time and within budget.

### 3.1 Management Tools and Techniques

The recommendations categorized under this heading address management's need to utilize the software engineering technology base to improve its capability to plan, organize, staff, direct and control software projects and personnel. Short-term and long-term efforts are both described.

Planning and Control. This subarea will provide management with the technical performance measures they need to manage progress and risk throughout the software life cycle. Developments in this area will furnish management with the closed-loop feedback systems they need to monitor the temperature of a project and understand when they are in trouble. Improved methods and tools will be pursued to relieve the tedium associated with the planning process and increase efficiency and effectiveness of controls. As an example of a project that could be pursued, a risk simulator which employs lessons learned models could be pursued to help managers quantify the consequences of risk and make better decisions throughout the life cycle.

Organizing. The subarea will develop team-building tools and techniques that managers can use to harness the power of teams and cope with skill shortfalls in the software area. Teams can positively influence productivity and quality by using peer pressure to affect behavior. Teams can also help solve interdisciplinary problems that often plague system development. As an example of a project that could be pursued, a knowledge-based conferencing tool could

be developed to expand software development environments to host team-directed problem solving sessions.

Staffing. This subarea will develop the personnel systems managers need to identify, grow, appraise and evaluate their software people resources. Career classification ladders will be formulated so that managers can identify the skills, knowledge and abilities required to perform wide ranges of jobs. Personnel development approaches will be defined so managers can nurture their people quickly and expertly. Key performance indices will be developed so people can be evaluated quantitatively based upon their work instead of emotion. As an example of a project that could be pursued, a personnel skill model could be developed to match prospective candidates to career positions using the job/success attributes defined as part of this subthrust.

Directing. This subarea will adapt existing management theory from other fields to the software area so that managers can improve communications (i.e., speaking, writing and listening) and motivate software engineering professionals to set and realize meaningful personal, project and organizational goals. As an example, a management by objective system could be employed to objectively evaluate people performance in terms of quantitative goals established for productivity and quality. Feedback in such areas would be invaluable because it would identify what makes people more productive.

Control. This subarea will develop process, product and people measures, models and metrics that managers can use to predict and assess productivity, quality and personnel performance. Managers also require data to calibrate their models and metrics so that comparisons can be made at all levels of the organization using a common and consistent measurement basis. This subarea will, therefore, address the data collection issue and will develop tools and techniques for automating the measurement process. As an example of a

project that could be pursued, a decision-support system could be built that incorporated models, measures, metrics and automated data collection routines into an integrated software support environment tc provide managers with trend analysis and quantification of parametric status and performance.

### 3.2 Motivation Tools and Techniques

The focus of this area will be placed on identifying those factors which management can use to motivate its software professionals to improve organizational productivity and product quality. Factors known to influence motivation include achievement, growth, recognition, advancement opportunities, responsibility, interpersonal relations and the work itself. What does each of these terms mean when it pertains to software and how do managers use them to get their people to produce quality products quicker and smarter? What investments are needed to improve productivity and what relationships exist between motivation, methodology, mechanization, management and measurement? How does productivity relate to our cost and quality measures, models and metrics? What are the differences between cost and productivity factors? These and other questions will be addressed by this thrust.

As a first step in answering these questions, a motivation model could be developed, married to our software development process model, validated and used to evaluate the effects of motivational factors on organizational cost, quality and productivity performance. Experiments could be conducted and the results could be fed into our model to validate their fidelity. Other innovative approaches could also be pursued.

### 3.3 Measurement Tools and Techniques

The focus of this area will be placed on developing decision-support systems that relate measures and models to cost so that

managers can evaluate the cost, schedule, technical performance and risk ramifications of their software engineering decisions anytime during the life cycle. These systems should be built to help managers make better decisions and should provide considerable "what if" processing. They should be easier to use than not to use and should be designed to provide managers with meaningful information. The vast store of experience derived in the fields of operations research and management science should be transitioned for use in these systems in a cost-effective manner.

### 3.4 Acquisition Management Tools and Techniques

The focus of this area will be placed upon investigating approaches that help acquisition managers better manager their contracted software development and maintenance efforts. What makes a good acquisition manager? What tools and techniques do you use to improve your visibility into contractor cost, schedule and technical performance? What contractual provisions do you employ to increase the Government's leverage throughout the software development or maintenance effort? What incentives do you use to reward competence? How do you get your contractor to commit, and how do you make him/her honor these committments? These and other questions will be addressed.

### 3.5 Management Development

The focus of this area will be placed upon developing software managers quickly so that they can handle their job expertly at any level of the organizational hierarchy. Innovative approaches to management development will be explored so that managers placed in a "sink" or "swim" situation will "swim." Management skills will be developed and judgment will be perfected before a good technical person is given major project responsibilities. Innovative training approaches will be pursued to determine how to make management development fun. For example, a management simulator that delivers

training using real-world scenarios as a game in an interactive, self-paced environment could be pursued.

### 3.6 Organization Development

The focus of this area will be placed upon transitioning progress in the field of organizational psychology and interpersonal relationships into software groups. Psychological concepts, techniques and methods used to increase the effectiveness of manager and organizations will be explored. Heavy emphasis will be placed upon the study of groups, the design and evaluation of training programs, performance measurement (including merit ratings and appraisal interviews) and the measurement of employee morale and job satisfaction. Sensitivity training and other proven approaches to improve people to people interactions will be introduced to foster the openness needed to make teams work in an interdisciplinary environment.

### 3.7 Technology Insertion

The focus of this area will be placed upon developing techniques that allow managers to introduce software engineering technology to their projects and/or organizations with a minimum degree of risk and disruption. Approaches that help managers assess the cost/benefit/risk tradeoffs related to the use of the technology base will be developed. Innovative technology insertion mechanisms (e.g., on-line training will expert advice) will be formulated and guidance for their use will be provided.

## 4.0 RELATIONSHIP TO OTHER AREAS

Management is the key ingredient to the success of the Software Initiative. "Good" management recognizes the need to develop and exploit the software engineering technology base. "Good" managers know how to harness the technology base to deliver acceptable products on-time and within budget. Both "good" management and managers

are developed as a product of the management area activities described within this section.

## 5.0 PAYOFF ASSESSMENT

The potential payoffs associated with moving the state of the practice of software management closer to the state of the art are many and varied. First, a higher percentage of DoD software projects will be delivered on-time and within initial budgetary estimates. Second, rework costs will decrease because the products delivered will be of higher quality. Third, overall organizational productivity would increase resulting in substantial cost avoidances. Fourth, growth in personnel requirements could be reduced enabling existing organizations to do more with the staff resources they already have. Fifth, turnover would decrease as morale improved and recruitment and training costs would diminish.

We could quantify the payoffs resulting from this management initiative many ways. For example, the EIA estimates that the cumulative cost for embedded computer system software during the 1983-1988 timeframe for the DoD will exceed $182 billion. A one percent increase a year in productivity attributed to better management would result in a cost avoidance of over $1.8 billion over this timeframe. This figure doesn't even account for any improvements in the ADP side of DoD or in the commercial sector where cost growth for software is spiraling. As another example, a decrease to 10% a year in turnover for a single organization like the Air Force Logistics Command where 1600 personnel (civilian and military) are working software projects would result in a cost avoidance of $1.6 million annually assuming the cost of recruiting and training a professional is $10,000. Extrapolate such savings throughout the DoD and it is likely that twenty to fifty times that amount would be saved. The leverage in management is large in terms of cost/benefits no matter which method you use to substantiate the investment.

## 6.0 REFERENCES


1. Basili, V. R. (ed.), Tutorial on Models and Metrics for Software Management and Engineering, IEEE Computer Society, 1980.

2. Boehm, B. W., Software Engineering Economics, Prentice-Hall, 1981.

3. Couger, J. D., and R. A. Zawacki, Motivating and Managing Computer Personnel, John Wiley & Sons, 1980.

4. Jensen, R. W., and C. C. Tonies, Software Engineering, Prentice-Hall, 1979.

5. Metzger, P. W., Managing a Programming Project Prentic-Hall, 2nd edition, 1981.

6. Perry, W. E., Managing Systems Maintenance, QED, 1981.

7. Reifer, D. J., Tutorial: Software Management, IEEE Computer Society, 2nd edition, 1981.

8. Semprevivo, ... C., Teams in Information Systems Development, Yourdon Press, 1980.

9. Yourdon, E., Managing the Systems Life Cycle, Yourdon Press, 1982.

## APPENDIX II.13

## APPLICATION-ORIENTED TECHNOLOGIES AND REUSE

### 1.0 INTRODUCTION

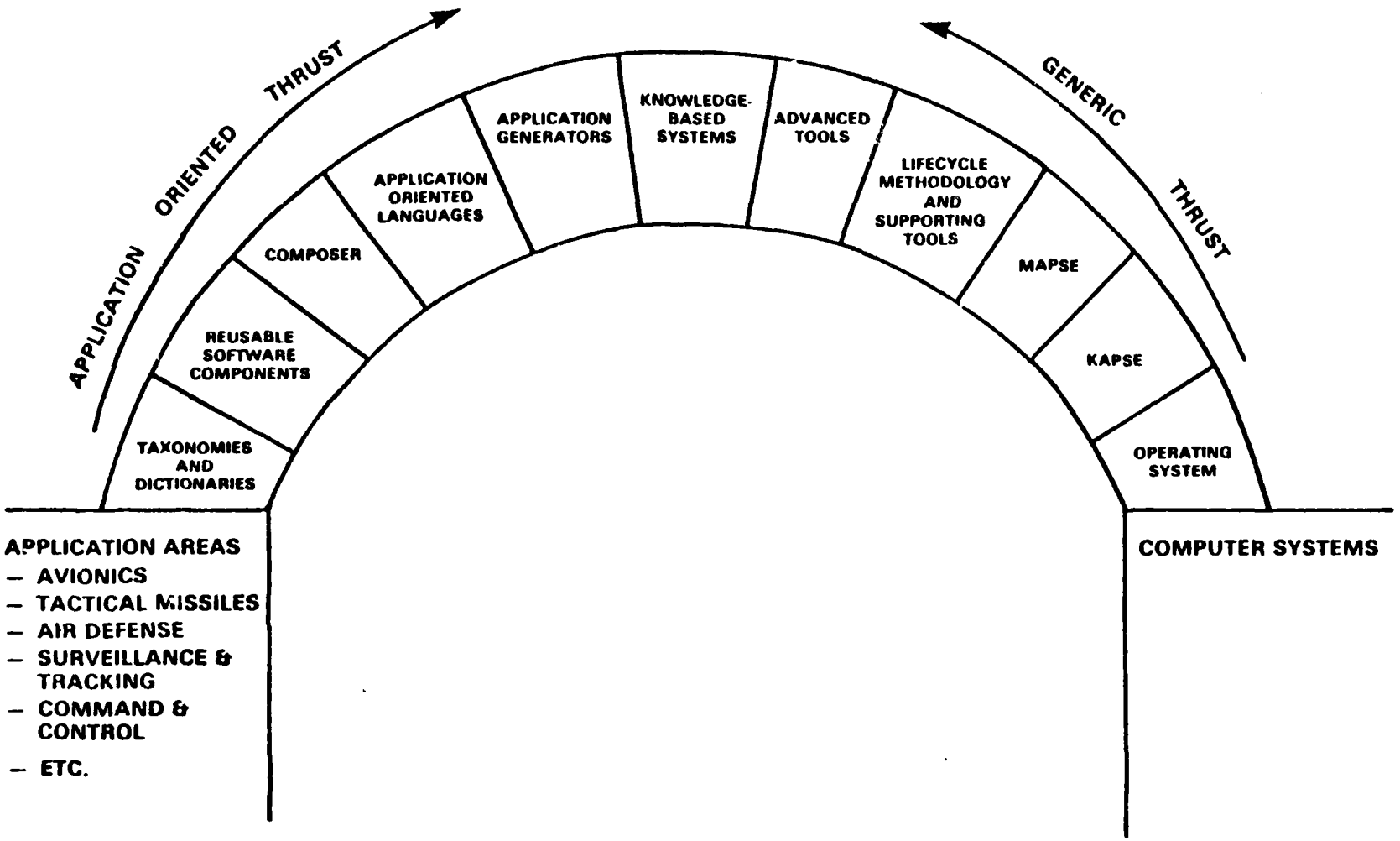
Software, like other products, is composed of parts; however, unlike physical products, the cost of reproducing software parts is negligible, regardless of size or complexity. The major costs are in design, development, and validation of the first version, and in maintenance (adaptation to new requirements) of operational software over its lifetime.

One way to increase productivity and lower costs is to reuse parts from previous projects in new projects. It should be possible to use parts from previous projects in new projects. It should be possible to make parts produced by one DoD contractor available to other contractors, greatly increasing the inventory of reusable parts and the productivity of those who use them.

DoD needs standards for products, procedures, and practitioners to fulfill military missions. Better tools, standard practices, and reuse of software parts will yield less expensive, more versatile military software sooner and in the necessary quantities. The Software Initiative should attack these needs in two ways:

- An application-oriented activity which will provide reusable software parts for military applications; both simple and sophisticated tools for combining parts are needed.
- Generic activities which will provide new tools, standard practices, and effective means to evaluate both tools and products built with these tools.

The two types of activities formally and informally support each other. Figure 13-1 shows how both thrusts combine to bring people




# BUILDING THE SOFTWARE BRIDGE TO JOIN APPLICATION AREA AND COMPUTER

3194 2

FIGURE 1

and computers closer together.

Application-Oriented Technologies Promote Reuse

The application-oriented activity involves five technologies to promote software reuse:

- Reusable Software Parts. First, a taxonomy (a classification scheme) of functions for reusable software parts will be developed. Then, inventories of parts will be built. Next, the use of parts inventories will be demonstrated. Demonstrations will involve the retrieval of parts from inventories and their use in software development and maintenance.

- Parts Composition Systems. A parts composition system automates the assembly of parts retrieved from reusable parts inventories. Each part has formal descriptions of its inputs, outputs, and functions that the parts composition system uses to fit parts together.

- Application-Oriented Languages. An application-oriented language (AOL), sometimes called a very high level language, is a general-purpose programming language with application-specific control and data structures. Common application-specific functions are built in. An AOL automates the retrieval of parts from inventories and their combination into larger elements.

- Application Generators. An application generator generates an application program or solution from non-procedural (what-to-do) statements or requirements. The conceptual model is filling out a form to get something done. By comparison, an application-oriented language is more likely to require procedural (how-to-do) statements. Application generators also depend on parts composition.

- Knowledge-Based Systems. A knowledge-based system uses codified general problem-solving techniques and codified specific application knowledge to solve problems in the application area. It is an "expert" assistant to the application worker, reducing his need for skilled human assistance.

Figure 13-2 shows how the more sophisticated techniques embrace the simpler, near-term technologies to increase automation.

A tool's ability to deal with application complexity and evolving requirements will be a major factor in its usefulness and acceptance. In complex applications, no matter how carefully requirements are specified, some needs are discovered only after some experience is gained. Evolving requirements inevitably require software adaptation or replacement. In a well-designed system, the cost of a modification should be commensurate with the magnitude of the functional change.

## 2.0 CURRENT STATUS

This section summarizes the current status in terms of the technical and non-technical factors affecting reuse.

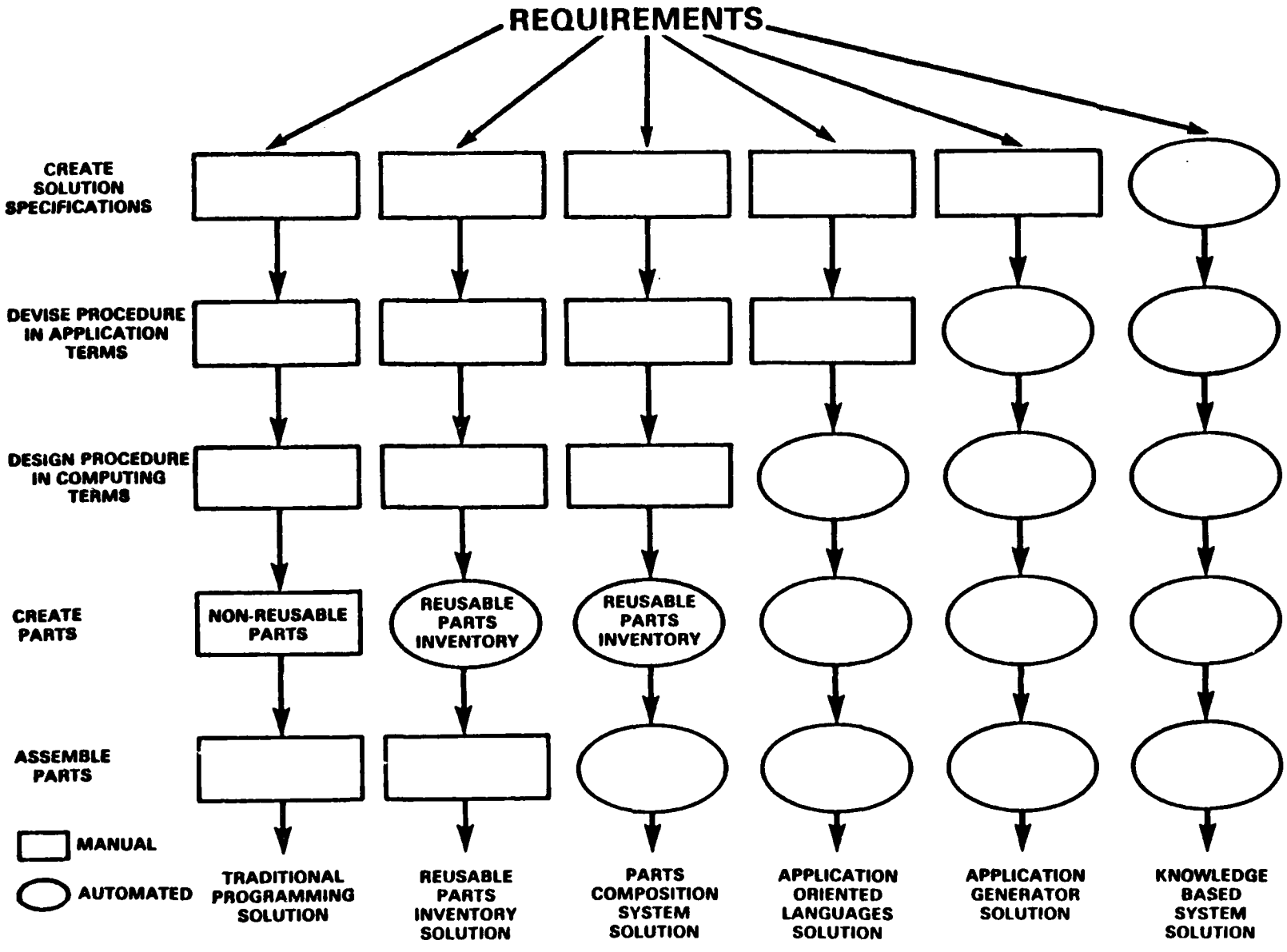
### 2.1 Technical Factors Make Software Reuse Difficult

There are many technical obstacles to software reuse. However, because some contractors have achieved limited success within their own organizations, the prospect for lowering or eliminating these obstacles through R&D is promising. A few of the obstacles are explained in the following paragraphs.

- Software is rarely designed for reuse. Design for reuse requires a more disciplined approach and better documentation. It requires better trained people, and it may cost more.

- Even if software is designed for reuse, a potential reuser may have trouble finding it. There is a great deal of software, and no available, effective classification system for organizing most of it. Often a project could use a piece of existing software except for some minor changes, easily made. The problem of automating the search for suitable software then becomes even more complicated.

- Using someone else's software is risky, because it may not work. Even though a piece of software may be widely used, perhaps it has never been tried in cases that arise in a new




# HIERARCHY OF APPLICATION ORIENTED TECHNOLOGIES

3194-2

FIGURE 12-2

application. A test environment for a software package may have been built, but it may be undocumented, it may no longer exist, or it may be proprietary to some other organization. Existing formal verification techniques are not practical for software parts of any useful size.

- Software can be written in any of several languages. Because there are no effective language standards, even programs in the same language (perhaps from different vendors' implementations) are often incompatible. Efforts have been made to automate conversion between different vendors' versions of the same language, and between languages, but some human intervention is usually necessary. Often, features of one language or implementation have no counterparts in another language or implementation.

- Even when standards exist, one group's standards may differ from another's. Standards in different organizations are likely to differ even more. Standards evolve, so that current standards may be incompatible with standards used in earlier work.

The upshot is that software reuse is risky, that design for reuse is costly, and that reuse probability is questionable. However, the possible payoff from application-oriented R&D is significant. The Joint Logistics Commanders "Joint Policy Coordinating Group on Computer Resource Management" Software Workshop (June 1981), wrote:

> "Those with a DoD embedded application orientation declared a more modest payoff (25% cost reduction) [in software costs, from reuse] than those from a business background (up to 66%) where reuse has been a reality for several years. There was clear consensus that reuse would improve the project schedule and reduce the risk of new system developments."

The percentage payoff for embedded systems will increase as parts inventories grow and more sophisticated technologies evolve.

## 2.2 Non-Technical Factors Affect Software Reuse.

Current DoD procurement policies discourage software reuse by contractors. For software reuse to succeed, DoD procurements must provide incentives for maximum production, support, and use of reusable software. DoD does not always retain complete rights to software it pays to develop. Where a contractor is free to profit from improvements he makes to such software, DoD should be entitled to benefit from the improvements, if DoD paid for the development of the basic product. Otherwise, after the basic product is changed, DoD and DoD-sponsored organizations may have to pay full price for software largely conceived and originally developed with DoD funds. Procurement regulations should be studied, suggestions for improvements made, and model contracts developed and demonstrated.

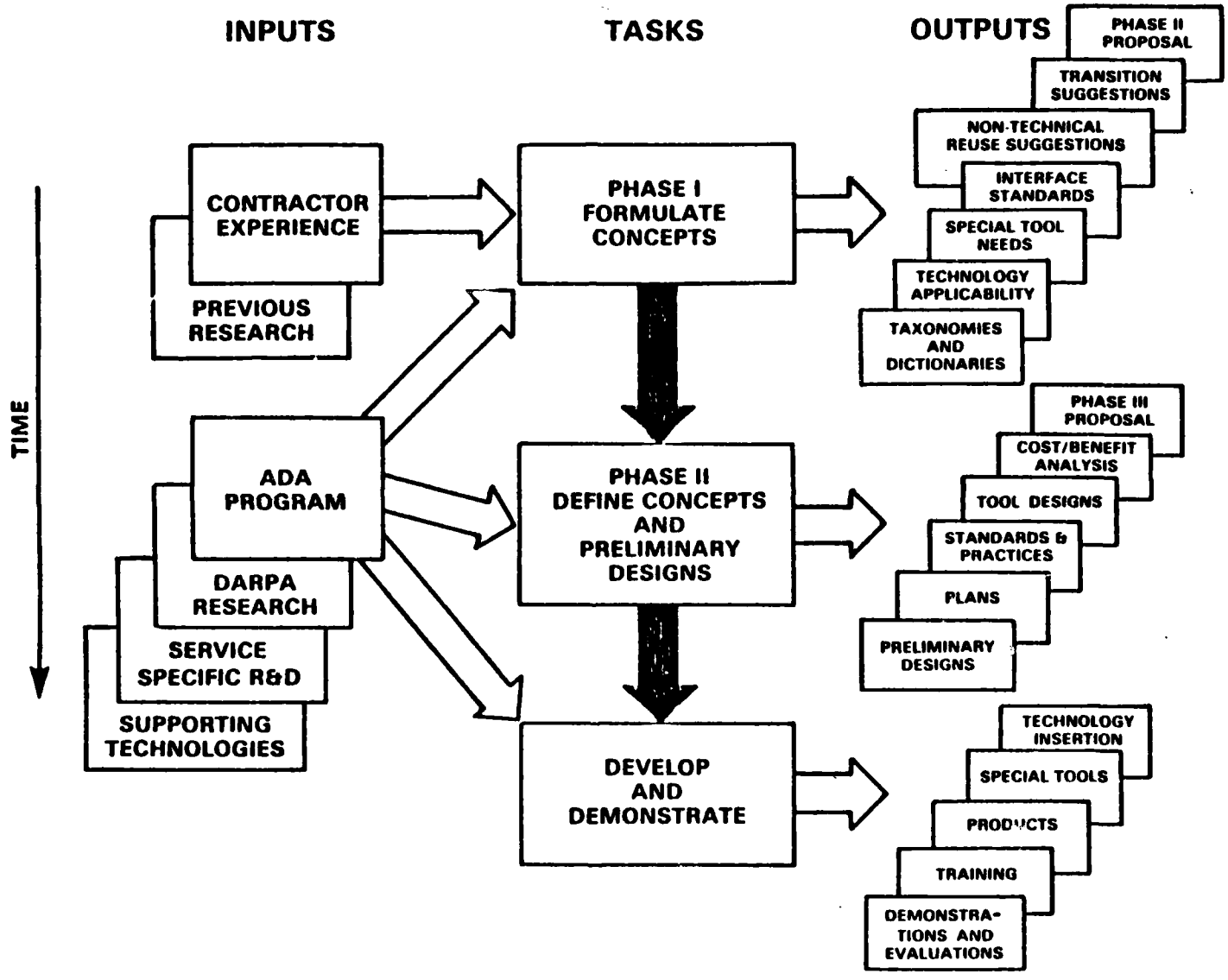
Reusability must be designed in at the earliest stages of development, which may significantly increase development costs. The return from these increased costs will only be realized when and if the software is used in other projects. Analysis is needed to identify software parts for which the extra development costs are worthwhile.

## 3.0 SPECIFIC RECOMMENDATIONS

The recommended application-oriented activity consists of three phases--concept formulation, concept definition and preliminary design, and full development and demonstration. Figure 13-3 shows the overall structure of the activity.

## 3.1 Formulate Concepts.

In Phase I, DoD might award a number of nine-month Concept Formulation contracts. DoD would draw up a list of military system areas likely to benefit from application-oriented technologies (e.g. digital avionics, tactical missiles, smart munitions, ground-based air defense systems, aircraft and missile detection and surveillance,




# STRUCTURE OF THE APPLICATION-ORIENTED THRUSTS

FIGURE 3

tanks and other land warfare vehicles, artillery fire control, and tactical command and control). Each bidder then would propose to work in such areas. Government-industry user groups should be convened to assess the potential for success in these areas, establishing a basis for continuing user involvement in the Software Initiative. For each chosen application area, the contractor would do the following:

- Assess the contribution that reusable software and application-oriented technology could make (referencing specific planned or forecast weapons systems or $C^3I$ programs).
- Develop a dictionary and taxonomy to describe and classify application concepts (i.e. objects, events, data, and functions).
- Propose interface standards for modules (i.e. software parts); these will be needed for parts composition.
- Propose an automated parts composition system.
- Study the feasibility of application-oriented languages, application generators, and knowledge-based systems.
- Identify and describe any special tools that must be built to support the development and demonstration phase (Phase III).
- Suggest approaches to reduce non-technical obstacles to software reuse, e.g. contractual arrangements and incentives, validation and verification, retention and transfer or rights.
- Suggest strategies for transition of existing software and software support to Ada.
- Perform preliminary cost/benefit analyses. Outline an approach to detailed cost/benefit analysis for Phase II.
- Describe the approach (including standards and practices) to be used in the next phase.

- Prepare a detailed proposal and plan for Phase II.

Proposals must demonstrate awareness of the state of the art, including Ada developments, current research results, past and current DoD application-oriented efforts (e.g. the Naval Research Laboratory's redevelopment of the A-7 software), commercial ADP application-oriented tools, and existing documentation standards for retrieval for reuse (e.g. FIPS PUB 30 and ANSI X3.88).

Some useful suggestions for improving the procurement process should be obtained from contractors' suggestions received during Phase I. Analysis of the results of Phase I will determine what to continue to Phase II.

3.2 Define Preliminary Designs.

Some fraction of the Phase I contracts might be continued into Phase II, Concept Definition. These will be selected on their technical merit, their probable contribution to meeting the STI's goals, and their cost. Phase II contracts would last one year. In Phase II, for each selected application area, contractors should do the following:

- Improve their taxonomies and refine their dictionaries.
- Recognize and incorporate recent Ada and other R&D developments into their proposals.
- Develop preliminary designs for each proposed technology.
- Perform a detailed cost/benefit analysis for each proposed technology.
- Propose demonstrations and technology insertion plans.
- Establish education and training plans to prepare their people for full effectiveness with Ada and modern software engineering practices.
- Develop preliminary designs for any special automated tools agreed on.

- o Describe arrangements for quality assurance and independent validation and verification.
- o Describe the practices and standards to be used in the next phase.
- o Prepare a proposal for the development and demonstration phase (Phase III).

Each contract must contain terms and conditions protecting government rights to concepts and products developed during the program. (These are expected to include arrangements for rights to improvements made in this software after contract termination.) Phase II will show which application areas will benefit from the use of application-oriented tools and methods, and which technologies are appropriate for these applications.

3.3 Develop and Demonstrate.

The contractors for Phase III, Development and Demonstration, will be selected on the basis of their proposals submitted during Phase II. Phase III might last up to five years, depending on each contract's goals. Contractors could be required to make early results available to the DoD community when it is clear that significant benefits would result.

Each Phase III contract might be self-contained (vertically integrated). Then each contractor would have complete responsibility for any necessary special tools and processes, and would not have to depend on other contractors' schedules. Phase III efforts would vary, but would follow the same outline. For each selected application, the contractor would do the following:

- o Recognize the latest Ada and integrated support environment developments, other R&D results, and experience, and incorporate them into his plans.
- o Train its staff in Ada, the environment, and in the standards and practices proposed for Phase III.

- o Develop agreed upon special implementation tools.
- o Develop detailed designs for, and implement initial versions of agreed upon technologies.
- o Experiment with and improve initial implementations. (This may involve prototyping of tools and techniques)
- o Demonstrate the effectiveness of developed application-oriented tools. (This may involve rapid prototyping of applications.) Figure 13-4 outlines guidelines for effective demonstrations.
- o Provide initial versions of technology insertion materials and assistance, experiment with and improve initial versions, and provide demonstrably effective technology insertion materials and assistance.

Successful Phase III projects would yield application-oriented tools that could, along with generic tools, significantly influence software development in these areas. Required consideration of Ada efforts at proposal time and continuous prompt use of new releases of the integrated support environment will ensure that the application-oriented tools and other results will be integrated into the environment.

## 4.0 RELATIONSHIP TO OTHER AREAS

Activity in this area will help demonstrate Ada and the Integrated Support Environment thereby helping Technology Transfer. Application-oriented technologies help address the Systems Definition problem by dealing in terms close to the application area. Knowledge-based Systems are one of the technologies proposed. End-user languages or interfaces have often been associated with databases and can lead to user performed maintenance. Distributed Systems and Hardware/Software Synergy both contain possibilities of application-oriented hardware which might make very productive combinations with application-oriented software. Acquisition management issues are extremely important if reuse is ever to become common.

- Realism
  - Size of developed portion
    - Complete System or Subsystem
    - Significant units of manipulation
    - 10K-200K lines of code
  - Testbeds
    - Actual or planned realistic contexts and exercises
  - Multiple Uses
    - Demonstrate in a variety of uses (at least two)
    - Demonstrate in unplanned (i.e. unknown/unexpected to developers) situation (optional)
  - Evolving Systems
    - Demonstrate ability to evolve as requirements change
  - Level of Complexity
    - Not toy or oversimplication
  - Schedule
    - Tight deadlines during a demonstration
    - Rapid Prototyping (optional)
  - Use by other than development team
- Design Excellence
  - High quality and disciplined plans and procedures
  - High quality approach to application
- Evaluatability of Demonstration
  - Quantitative evaluation
  - Conclusion on merit of approach and effort will result which can guide future DoD decisions
  - Conclusions on parts of approach and effort will result
- Chance of Success
  - Reasonable chance of full success
  - Substantial residual benefits if not full success
- Direct Technology Insertion Effects
  - Significant number of units and persons in DoD community obtain experience and capability
- Direct Future Benefit to DoD
  - Important or costly (application) area to DoD
  - Existing or firmly planned future system can use products (optional)

FIGURE 13-4: DEMONSTRATION GUIDELINES

As discussed at the beginning, application-oriented efforts form a complement to the other areas which are mainly generically oriented.

## 5.0 PAYOFF ASSESSMENT

Significant side benefits will result from the technology transfer and feedback affects from timely demonstration of Software Initiative generic products. These effects alone might justify a modest application-oriented activity. However, significant direct benefit should also result.

The direct benefit will result from reuse. The application-oriented projects supported by Software Initiative depending on the level of effort will produce some volume of reusable software. In addition, once the technical and non-technical impediments to reuse have been reduced, reusable software should start to be generated by some of the regular DoD development efforts.

Existing examples in the ADP area have shown improvements from 66% reuse mentioned above to 1000% increase in productivity [Martin 82]. Honeywell has reported a factor of 11 improvement in the area of automatic test equipment through the use of the application-oriented language Atlas and related tools [Weisberg 82]. Thus real potential seems to exist particularly in applications areas which are relatively stable and well understood.

In addition to savings through reuse, additional benefit may come from the increased reliability provided by using "experienced" software.

## 6.0 REFERENCES


1. Hasse, V.H, and G.R. Koch (eds), Special Issue on Application-Oriented Specifications, Computer, Vol.15, No. 5, May 1982.
2. Martin, J., Application Development Without Programmers, Prentice-Hall, 1982.

3. Redwine, S. T. and G. R. Berglass, A Proposed Plan for DoD Software Technology Initiative, MITRE, MTR-82W-0091, May 1982.

4. Weisberg, L. R., A New Approach to Lowering DoD Software Costs, Honeywell Aerospace and Defense Group, March 1982.

# APPENDIX III

## VISIONS OF THE FUTURE

The opinions expressed in this appendix are those of the authors and do not necessarily represent the views of the DoD.

# APPENDIX III

## VISIONS OF THE FUTURE

Movement into the future can be pursued in two ways. A conservative way is to continue to develop and support software in much the same way, but use as much technology as possible to increase productivity, reliability and adaptability. A more radical way is to seek new and better ways of developing and supporting software, particularly ones which can be automated as much as possible. These approaches to achieving a better software development and support world can be called evolutionary and revolutionary since one attempts to evolve the software development and support paradigm and the other seeks to revolutionize it.

The rest of this Appendix contains two reports. The first provides a vision of what will be accomplished by evolution and the second proposes a new software development and support paradigm and discusses what the world would be like by adopting it and how it can be achieved by using knowledge-based techniques.

## APPENDIX IIIA

## SOFTWARE TECHNOLOGY IN THE 1990'S USING THE CURRENT LIFE CYCLE PARADIGM

### 1.0 INTRODUCTION

In this paper we offer a vision of the future.

It has become evident that there are serious problems to address concerning the production and support of computer software and that a significant effort is needed to improve the situation.

The demand for computer software is skyrocketing and the means to meet this demand are glaringly inadequate. This demand is partly propelled by a rapidly growing need to incorporate software as an essential component of military systems, in which software increasingly establishes and controls system functionality. Elsewhere in the national economy, the need for software is growing no less rapidly as organizations strive to increase productivity through automation and to improve product versatility and market appeal through the use of computers.

In military systems in which software is a critical component, it is essential to obtain software which performs its mission well. This means that military software must be reliable, affordable, and adaptable.

Reliability is the most critical requirement, since if the software component of a military system doesn't do what is needed, or can't provide dependable performance, or can't support the capacity of a system to accept battle damage and keep on fighting, or stops functioning in the presence of an error at a critical moment in a military engagement, our future military systems could fail in ways which adversely affect our national security.

Military software must not only be reliable, it must also be affordable. The DoD operates in an environment of constrained resources and must be able to obtain reliable software for its military systems at a price it can afford to pay. In the body of the paper, we argue that a successful attack on improving software productivity is the key to obtaining affordable software, and, in our vision of the future, we sketch how this could be accomplished.

Finally, experience indicates that military software must be adaptable. Many embedded computer systems are large (i.e., over 100,000 instructions) and long-lived (with service lifetimes of 15 to 25 years). During their lifetimes, they undergo significant change in response to upgraded requirements and they must adapt, often rapidly, to changing circumstances of usage. Recent evidence indicates that the number of instructions changed per year is often on the order of the number of instructions in the whole system, and that 70% to 90% of the total amount spent on software over its entire lifespan is attributable to the cost of the operational phase during which it is upgraded, repaired, and adapted to new uses. Thus, it is critical that military software exhibit the capacity to respond to change in an affordable fashion and without loss of critical system reliability.

Further, an extrapolation of current trends indicates that these properties will be even more essential to successful software support of military operations in the 1990s. The use of advanced, highly flexible electronic warfare capabilities in the recent Middle East and Falkland Islands conflicts gives an idea of the type of rapid reprogrammability and surgical precision of operations which will characterize successful military operations in the future.

Our goal, then, is to achieve a state-of-practice in the 1990s in which we can build embedded system computer software with

adequate levels of productivity, adaptability, and reliability --- that is, we want to achieve PAR by 1990.

What will it take to achieve PAR by 1990? That is the central question we address in this paper.

To answer this question, we first take a look at the quantitative dimensions of the software demand-supply gap --- how big is it?, and what are the numbers? Then, we investigate quantitative dimensions of software productivity --- what mix of component productivity improvements are available?, and quantitatively, how much can we expect each component improvement to contribute? Can we sketch the rough outlines of some strategies that will cumulatively improve productivity sufficiently to close the demand-supply gap? We think we can.

Our quantitative investigation of productivity matters leads us rather forcefully to the conclusion that there are no simple panaceas for the software problems we face. To the contrary, our economic analysis strongly indicates that we must make a wide-ranging, substantial attack on many constituent problems over a long period of time if we are to be successful. In short, the overall improvement we need must come from a large number of component improvements, each of which contributes meaningfully but not overwhelmingly to the whole.

We are convinced that success can come only if we learn to manage a large number of variables skillfully, and if the components to the overall solution integrate well. Completeness and integration are, therefore, two key words in our vision.

Thus, we provide a discussion of the many components of the software production infrastructure that need to be improved and integrated, using our so-called "smokestack diagram" to provide an overview, and we examine various stages of growth of the computer

industry with an eye toward identifying catalytic forces that can be employed to stimulate adoption of greater levels of shared infrastructure on which value-added production methods can be based and which enable productivity to be improved.

This sets the stage for introducing elements of our vision of the future: how the Ada program can be shaped to contribute highly-integrated, complete software life cycle support environments of high potential payoff; effective policies for technology insertion; policies for catalyzing the adoption of greater levels of shared infrastructure; and how a newly founded Software Engineering Institute can play a key role in collecting, integrating, and spreading improved tools, practices, and skills.

Finally, we present a word portrait of how the future might be different if the improved elements we have envisaged were in place and operating successfully in the 1990s.

## 2.0 QUANTITATIVE DIMENSIONS OF SOFTWARE DEMAND

There is a serious and rapidly widening gap between software demand and the capacity of software suppliers to meet this demand. Figure 1 shows that software demand is rising at least 12% per year while the supply of people who produce software is increasing about 4% per year and the productivity of those software producers is increasing at about 4% per year. This leaves a cumulative gap of 4% per year.

The magnitude of the current gap is roughly the equivalent of fifty to one hundred thousand people.

If the demand-supply gap were to continue to widen at a compound rate of 4% per year, then by the year 1990 the gap would have increased as shown in Figure 1, leaving us with a shortfall that could be measured in terms of between 860,000 and a million software personnel.

# TRENDS IN SOFTWARE SUPPLY AND DEMAND

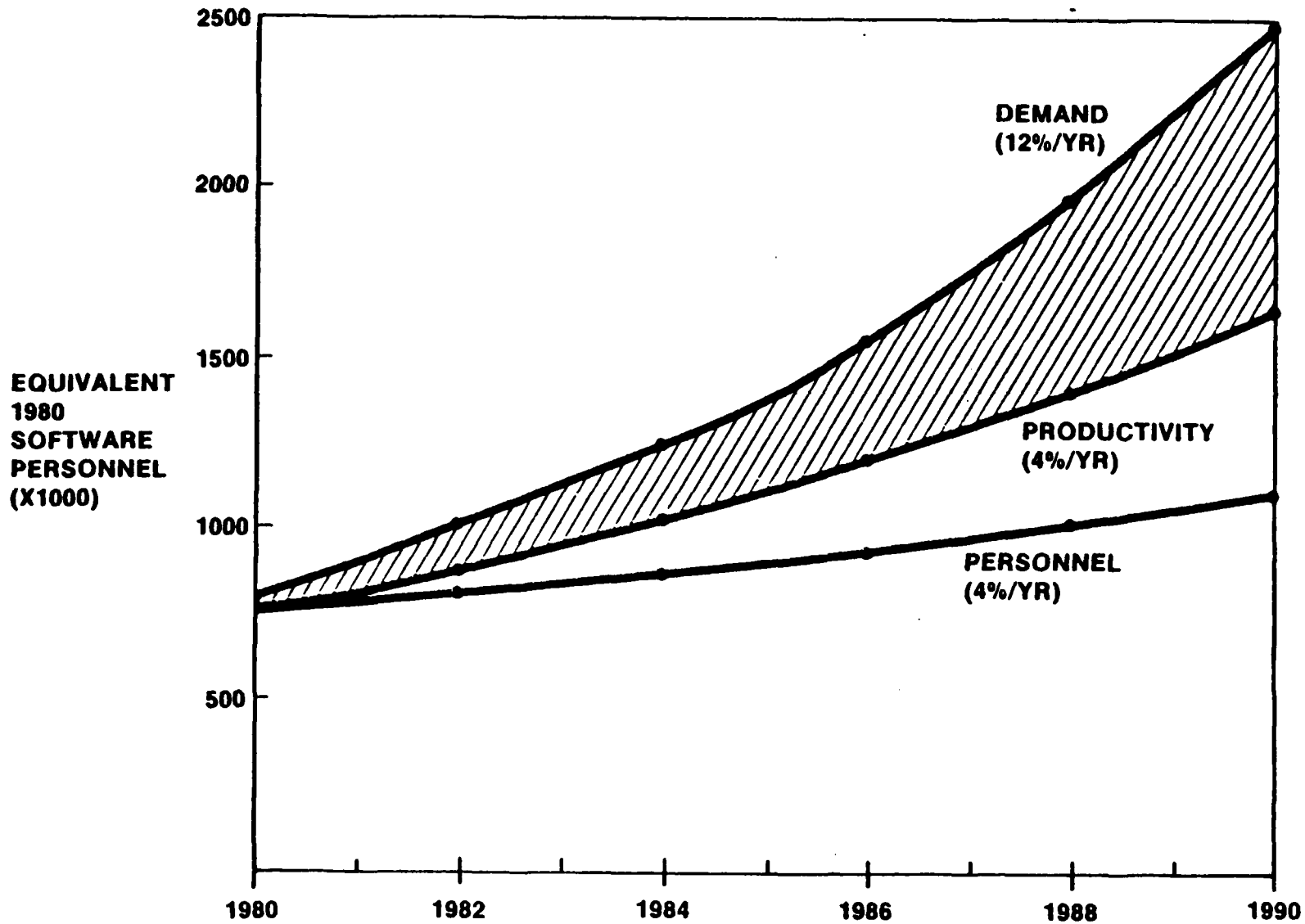



FIGURE 1

Of course, in a demand-supply crunch, something has to give. Already the price of software personnel has been bid up noticeably in the marketplace, increasing the cost of software, stimulating the exodus of university computer science faculty to industry, and increasing software personnel turnover in industry as software personnel use job-hopping to boost themselves up the rungs of the salary ladder. It is noteworthy that some other industrial nations have neither a current nor a projected shortfall of software personnel. Under the circumstances, if the United States does not take effective remedial measures, it could be priced out of the software market and could forfeit shares of that market to other nations as a result.

Given the poor prospects of increasing the output of new software personnel from an educational system that appears to be under serious economic strain, there may be no alternative but to increase software productivity if the demand for software is to be met. In any event, given the magnitude of the demand-supply gap, we are persuaded that unless we improve software productivity substantially by 1990 we may not have affordable software.

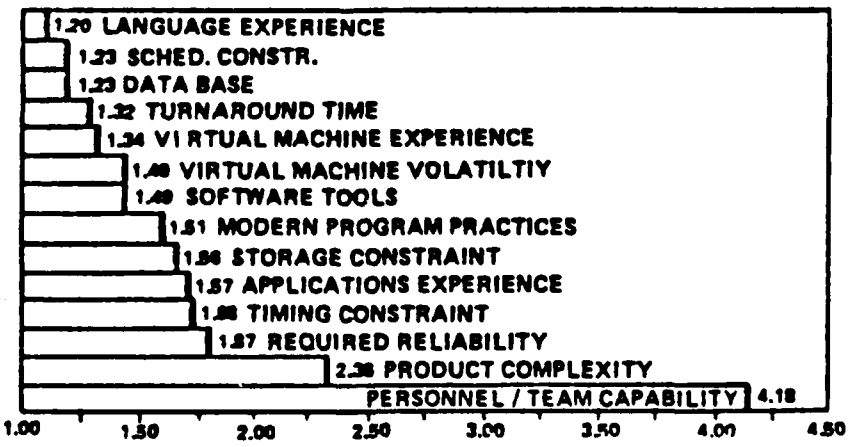
## 3.0 IMPROVING SOFTWARE PRODUCTIVITY

How can we address the problem of increasing software productivity? One approach is to analyze the cost drivers that contribute to the cost of software products. Some of these cost drivers are controllables, and by investing to provide software project resources or by following selected policies and disciplines, we can control factors that improve productivity.

For example, the COnstructive COst MOdel (COCOMO), described in detail in [Boehm 1981a], estimates the cost of a software project as a function of program size in Delivered Source Instructions (DSI) and a number of other cost driver attributes summarized in Figure 2. Figure 2 shows the Productivity Range for each attribute: the

relative productivity in DSI/man-month attributable to the given attribute, after normalizing for the effects of other attributes. Thus, the 1.49 productivity range for the Software Tools attribute results from an analysis indicating that, all other factors being equal, a project with a very high level of tool support will require only 0.83 of the effort required for a project with a nominal level of tool support, while an equivalent project with a very low level of tool support would require 1.24 times the effort required for the nominal project, or 1.24/0.83 = 1.49 times the effort on the "very high" tools project. The "very high" and "very low" ratings correspond to specific levels on a COCOMO rating scale for tool support [Boehm 1981a].

Given rating scales for the Software Tools attribute and for the other cost driver attributes, it is possible to perform a productivity audit of a software project to determine the weighted-average productivity multipliers characteristic of it in relation to the nominal measures in the COCOMO model. Then, one can investigate several future scenarios representing varying levels of investment into productivity-improving measures. Figure 3 gives a table summarizing the analysis of the potential impact of the Software Initiative on DoD software productivity (given in Appendix X to the report "Strategy for a DoD Software Initiative").



Estimated DoD Software Initiative Impact
on Software Productivity

| COST DRIVER | DoD Average Effort Multipliers | | |
|---|---|---|---|
| | 1976 | 1982 | 1990 |
| Use of Software Tools | 1.05 | 1.02 | 0.85 |
| Use of Modern Programming Practices | 0.98 | 0.95 | 0.85 |
| Programming Language Experience | 1.03 | 1.02 | 0.98 |
| Software Environment (Virtual Machine) Experience | 1.05 | 1.03 | 0.95 |
| Computer Execution Time Constraint | 1.25 | 1.18 | 1.11 |
| Computer Storage Constraint | 1.22 | 1.15 | 1.06 |
| Computer Turnaround Time | 1.03 | 1.01 | 0.90 |
| Reduced Requirements Volatility | 1.17 | 1.15 | 1.00 |
| Retooling Avoidance | 1.06 | 1.06 | 1.00 |
| Software Re-use | 0.93 | 0.90 | 0.50 |
| Relative Effort | 2.01 | 1.54 | 0.36 |
| Productivity Gain | | 1.30 | 4.34 |

Figure 3

Figure 3 shows that a productivity improvement program achieving several cost driver attribute improvements in parallel could improve productivity by a factor of 4.34 between 1982 and 1990. (The potential gain for maintenance may be overstated somewhat as the last three factors apply less to maintenance than to development. On the other hand, the lower COCOMO maintenance multipliers for modern programming practices partly compensate for this effect.)

Besides providing an estimated productivity gain, this analysis provides insights for determining which tools and policies to emphasize in a selected productivity improvement strategy. Moreover, it provides a valuable framework for tracking the actual

progress of a productivity program and for determining whether its goals are actually being achieved.

Another form of analysis is reported in [Alford et al 1981] which assesses the likely reduction of software project effort devoted to each software activity during each software development phase as a result of a software environment improvement program. This analysis is based on the COCOMO model's software effort distribution by phase and activity [Boehm 1981a, Chapter 7] and is shown in Figure 4.

Activity - Oriented Estimates of Effort Reduction

| | PLANS & REQUIREMENTS | PRODUCT DESIGN | PROGRAMMING | INTEGRATION & TEST | DEVEL-OPMENT TOTAL | MAINTENANCE |
|---|---|---|---|---|---|---|
| PHASE EFFORT PI (%) | 7.4 | 16.7 | 44.4 | 31.5 | | 100.0 |
| ACTIVITY | AE, AS, ES | AE, AS, ES | AE, AS, ES | AE, AS, ES | ES | AE, AS, ES |
| REQUIREMENTS ANALYSIS | 0.42 0.25 0.8 | 0.10 0.50 0.8 | 0.03 0.50 0.07 | 0.02 0.50 0.2 | 2.8 | 0.05 0.40 2.0 |
| PRODUCT DESIGN | 0.16 0.15 0.2 | 0.42 0.20 2.1 | 0.06 0.50 1.3 | 0.04 0.50 0.8 | 4.2 | 0.11 0.20 3.3 |
| PROGRAMMING | 0.10 0.10 0.1 | 0.14 0.50 1.2 | 0.55 0.25 6.1 | 0.48 0.50 0.1 | 16.5 | 0.41 0.50 20.5 |
| TEST PLANNING | 0.06 0.25 0.1 | 0.08 0.20 0.4 | 0.08 0.20 1.1 | 0.08 0.25 0.4 | 2.0 | 0.06 0.25 0.1 |
| V & V | 0.10 0.20 0.1 | 0.10 0.40 0.7 | 0.12 0.50 2.7 | 0.20 0.55 3.5 | 7.0 | 0.14 0.55 0 |
| PROJECT OFFICE | 0.06 0.20 0.2 | 0.07 025 0.3 | 0.06 0.20 0.4 | 0.5 0.20 0.4 | 1.3 | 0.06 0.20 1.5 |
| CM / QA | 0.03 0.35 0.1 | 0.02 0.20 0.1 | 0.06 0.55 1.5 | 0.08 0.55 1.4 | 3.1 | 0.06 0.50 3.0 |
| MANUALS | 0.06 0.55 0.2 | 0.07 0.45 0.5 | 0.06 0.45 1.0 | 0.07 0.45 1.0 | 2.7 | 0.11 0.55 6.0 |
| TOTAL | 1.8 | 6.1 | 14.8 | 16.7 | 39.4 | 46.1 |

Figure 4

In Figure 4, the Phase Effort row shows the percentage of the total development effort devoted to each phase. For each phase the AE column shows the COCOMO effort distribution by activity. For example, Figure 4 indicates that 7.4% of the development effort is devoted to the Plans and Requirements phase. Within this 7.4%, Requirements Analysis activities consume 42%. The AS (Activity Savings) column gives an estimate of the percentage of effort that could be saved by adopting productivity improvement measures. For example, 25% of the Requirements Analysis effort was estimated to be

saved during the Plans and Requirements phase. The ES (Effort Savings) column shows the resulting percentage of the total effort saved. When summed over all phases and activities, the overall results show a development savings of 39% and a maintenance savings of 46%, exclusive of any savings due to reduced requirements volatility, retooling avoidance, and software reuse. These savings are not as great as those estimated by the cost-driver model, but they are reasonably comparable and quite significant: as shown in Appendix IX, this estimation approach yields an overall productivity gain of a factor of 3.93 by 1990.

## Productivity Study Conclusions

In a 1980 productivity study, reported on in [Boehm et al 1982], the following conclusions were reached.

1. Significant productivity gains require an integrated program of initiatives in several areas. These areas include improvements in tools, methodology, work environments, education, management, personal incentives, and software reuse. A fully effective software support environment requires integration of software tools and office automation capabilities.

2. An integrated software productivity program can have an impressive payoff. Productivity gains by factors of 2 in four years and factors of 4 in nine years are generally achievable, and are worth a good deal of planning and investment.

3. Improving software productivity involves a long, substained effort. While the payoffs are large, they require long-range commitment. There are no easy, instant panaceas.

4. Software support environment requirements are still too incompletely understood to specify precisely. Software support environments fall into an extensive category of man-machine systems whose user requirements are not completely understood.

Based on these conclusions, it is possible for contemporary software organizations to initiate significant long-range efforts to improve software productivity which could, for example, improve productivity by:

- o a factor of 2 by 1985;
- o a factor of 4 by 1990.

A significant quantitative fact is that if 70% of the software producers could quadruple their productivity by 1990 (while the remaining 30% continue to increase productivity at the current nominal rate of 4% per year), the software demand-supply gap could be closed. (This is based on an analysis in which 70% of the producers improve productivity at a compound rate of 22% per year for each of the remaining years in this decade).

The analysis indicates that productivity increases of the required magnitude are achievable by sufficiently widespread pursuit of policies of judicious investment and adoption of software practices of proven effectiveness. Whether the leadership and collective will power needed to accomplish this can be mustered is a different question. A national Software Initiative could play a significant catalytic role in stimulating awareness of the problem and its feasible solutions, and in enlisting the collaboration and cooperation of enough people to close the gap. Although stiff challenges must be faced, it is clearly possible to succeed.

## 4.0 INTEGRATED SOFTWARE INFRASTRUCTURE FOR THE 1900'S

Successful production of software rests on many layers of infrastructure. Figure 5 illustrates how applications software in DoD mission areas such as Avionics and C3I relies on layers of support infrastructure such as application-specific technology, general purpose support technology, process technology, education

and technology transfer capabilities, and management skills and techniques.

Suppose we envisage a software support environment of the future encompassing all of the elements in Figure 5 and suppose they are well-integrated. How would such a support environment differ from the environments we have today?

In fact, nearly all of the components of the environment illustrated in Figure 5 have some sort of implementation today. The main problems with these existing components are:

1. They are largely immature and incompletely developed. Examples are rapid prototyping capabilities, requirements aids, APSE master data bases, portable KAPSEs, etc.

2. They are poorly integrated with each other. Examples are tools supporting different phases of the life cycle; obsolete acquisition regulations, specifications, and standards; environments requiring the mastery of 2 distinct tool command languages to do one's job; people not trained or qualified to use advanced capabilities, etc.

If these shortfalls in the existing DoD and support contractor environment could be corrected, most of DoD's current software problems could be eliminated. Furthermore, as the analysis in the previous section has suggested, DoD could reap an estimated software life cycle productivity gain of a factor of 4 if it could achieve the fully integrated environment depicted in Figure 5.

Thus, a key element in our vision of the state of software practices in the 1990s is to achieve the fully-populated, fully-integrated software environment whose structure is illustrated in Figure 5.

Certainly some of the effort of the current decade must be devoted to developing new techniques and capabilities, in areas such as rapid prototyping, requirements aids, reusable software component libraries, etc. But the major effort must be a carefully planned and

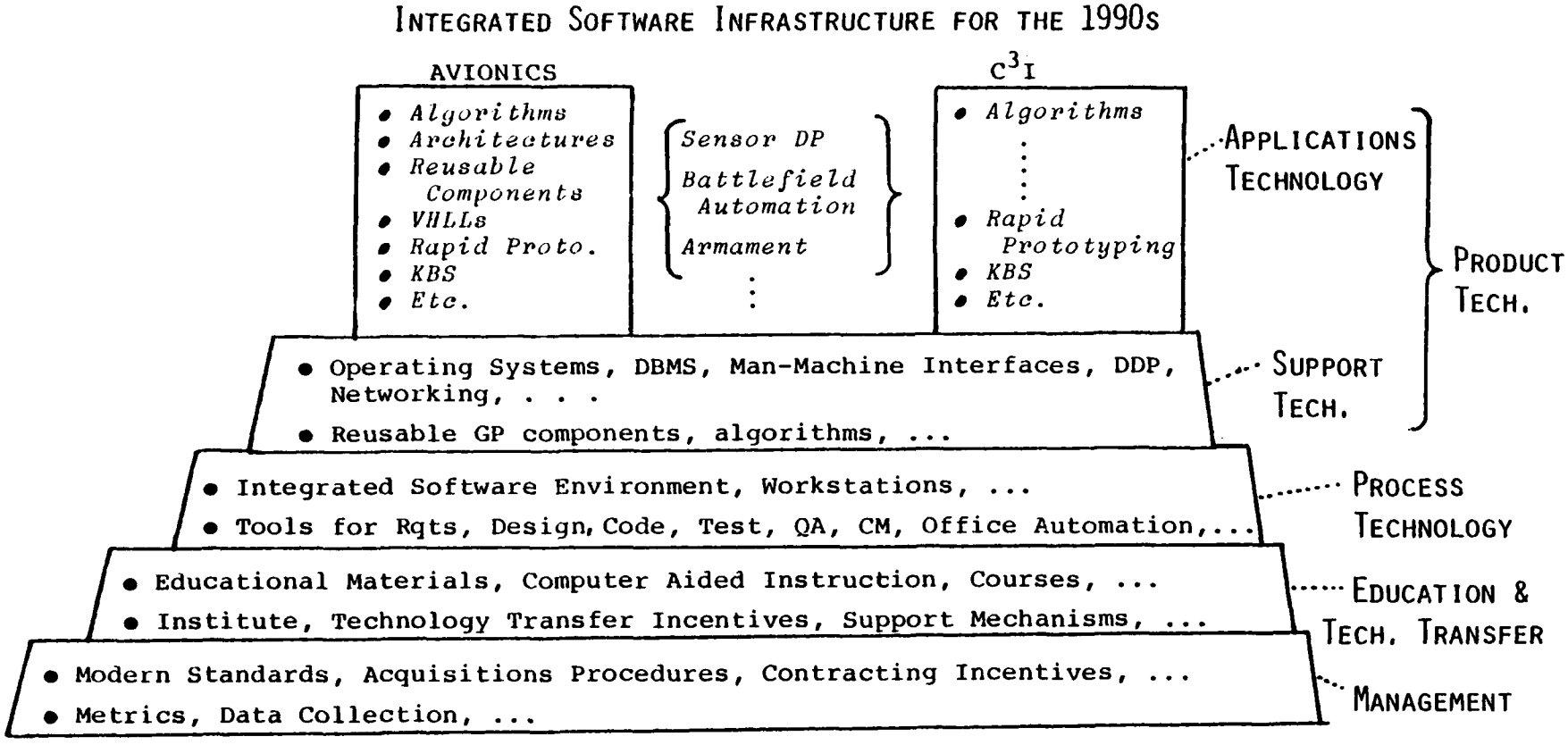

INTEGRATED SOFTWARE INFRASTRUCTURE FOR THE 1990s
AVIONICS
• Algorithms
• Architectures
• Reusable Components
• VHLLs
• Rapid Proto.
• KBS
• Etc.
Sensor DP
Battlefield Automation
Armament
C3I
• Algorithms
• Rapid Prototyping
• KBS
• Etc.
APPLICATIONS TECHNOLOGY
PRODUCT TECH.
• Operating Systems, DBMS, Man-Machine Interfaces, DDP, Networking, . . .
• Reusable GP components, algorithms, ...
SUPPORT TECH.
• Integrated Software Environment, Workstations, ...
• Tools for Rqts, Design, Code, Test, QA, CM, Office Automation, ...
PROCESS TECHNOLOGY
• Educational Materials, Computer Aided Instruction, Courses, ...
• Institute, Technology Transfer Incentives, Support Mechanisms, ...
EDUCATION & TECH. TRANSFER
• Modern Standards, Acquisitions Procedures, Contracting Incentives, ...
• Metrics, Data Collection, ...
MANAGEMENT

FIGURE 5

In fact, nearly all of the components of the environment illustrated in Figure 5 have some sort of implementation today. The main problems with these existing components are:

1. They are largely immature and incompletely developed. Examples are rapid prototyping capabilities, requirements aids, APSE master data bases, portable KAPSEs, etc.

2. They are poorly integrated with each other. Examples are tools supporting different phases of the life cycle; obsolete acquisition regulations, specifications, and standards; environments requiring the mastery of 12 distinct tool command languages to do one's job; people not trained or qualified to use advanced capabilities, etc.

If these shortfalls in the existing DoD and support contractor environment could be corrected, most of DoD's current software problems could be eliminated. Furthermore, as the analysis in the previous section has suggested, DoD could reap an estimated software life cycle productivity gain of a factor of 4 if it could achieve the fully integrated environment depicted in Figure 5.

Thus, a key element in our vision of the state of software practices in the 1990s is to achieve the fully-populated, fully-integrated software environment whose structure is illustrated in Figure 5.

Certainly some of the effort of the current decade must be devoted to developing new techniques and capabilities, in areas such as rapid prototyping, requirements aids, reusable software component libraries, etc. But the major effort must be a carefully planned and coordinated program of enhancing and refining existing technical, personnel, managerial, and institutional capabilities to fit within and perform as a unified environmental structure.

The key payoff activities we need to undertake are therefore ones of defining and integrating complete support environments and mastering their transfer into widespread support use.

coordinated program of enhancing and refining existing technical, personnel, managerial, and institutional capabilities to fit within and perform as a unified environmental structure.

The key payoff activities we need to undertake are therefore ones of defining and integrating complete support environments and mastering their transfer into widespread support use.

Before we describe further elements of our vision of the 1990s having to do with the role we envisage for the Ada program and the role a Software Engineering Institute could play, we need to make some observations about forces in the computing marketplace. It is essential to understand the nature and evolution of these forces in order to master the technology transfer process as indicated above.

## The Computer Industry Environment: Stages of Growth

The dependence of significant productivity gains on a high level of shared support infrastructure has been demonstrated for a number of industrial advances in the past; see, for example, [Graham 1982].

The computer industry has gone through several stages of growth, in which higher levels of shared infrastructure led to increases in productivity.

Achieving each new level requires a successful resolution of the conflict between two opposing forces in the computing marketplace:

- An anti-sharing, anti-standardization force motivated by the desire to retain one's existing customer base;
- A pro-sharing, pro-standardization force motivated by the desire and need to improve users' productivity in order to keep them as customers.

To progress from one level to the next requires two primary ingredients:

- o A technology which is sufficiently mature to support standardization and improved productivity;
- o A set of catalysts to get the improvement process far enough along to overcome the anti-sharing forces.

Table 1 presents six stages of growth in the computer industry each roughly corresponding to a decade of growth. These are characterized in terms of the shared infrastructure available to computers users at each stage and the primary value-added standardization vs. installation-uniqueness issues being resolved by the industry at the time.

Thus, for example, around 1950, there was no real shared infrastructure, but the technology available was beginning to raise such issues as:

- o Should punch-card formats be standardized in the interests of data sharing, or should multiple (80-column, 90-column, etc.) formats continue and proliferate further:
- o Given that Lockheed, Douglas, and North American are competitors, should they share mathematical routines and system utilities to promote common productivity?

By 1960, these issues were largely resolved in favor of shared infrastructure and improved productivity. By then, though, similar new issues had arisen over such items as use of HOL's and I/O standards allowing users to choose between several options for peripheral devices.

Table I. Stages of Growth in the Computer Industry

| Stage | Shared Infrastructure | Value-Added Issues |
|---|---|---|
| I (1950) | None | shared utilities<br>media standards |
| II (1960) | shared utilities<br>media standards<br>algorithms | MOL/HOL<br>mixed peripherals<br>(I/O stds.) |
| III (1970) | HOL's<br>vendor utilities<br>I/O standards<br>plus previous base | software unbundling<br>stable O/S interface |
| IV (1980) | vendor-line OS, utilities<br>plug-compatible mainframes<br>commercial S/W packages<br>basic S/W environments<br>plus previous base | portable OS, environment, networking standards |
| V (1990) | Ada<br>portable OS, utilities<br>portable environments<br>networking standards<br>some mainframe standards<br>-Nebula, 1750 | application standards<br>mainframe standards |
| VI (2000) | Knowledge-based application standards, program generators, component libraries for some areas | application standards for more complex knowledge domains |

Right now, the computer industry is roughly at Stage IV in its evolution. Its shared infrastructure now includes some vendor-line operating systems and utilities allowing the choice of plug-compatible mainframes and commercial software packages, and the beginnings of portable operating systems and environments such as those of Unix and CP/M.

Some of the major sharing vs. installation-unique issues prevalent today are:

- Should mainframe vendors support portable operating systems and software environments providing users access to more productivity options, at the risk of losing users tied to vendor-specific operating systems, utilities, and software environments?
- In particular, should those be based on Ada?
- Should software houses developing their own software environments (for competitive advantages or software product sales potential) merge their products with common-use standard environments?
- Should the industry establish at least interim standards for local area network interfaces?

There are many strong, conservative, anti-sharing forces which could easily cause the balance on these issues to tip toward installation-specific proliferation and reduced value-added productivity potential in the U.S. This is a matter of grave national concern, as it is likely that in Japan the balance will tip toward shared use, higher productivity, and competitive superiority in the computer industry.

The proposed DoD Software Initiative represents the U.S.'s major opportunity to tip the balance on the software-related issues above toward a solution providing a higher level of shared infrastructure and productivity. It can provide the necessary technology base via maturation of Ada-based operating systems, utilities, and software environments, and the set of catalysts (R and D contracts, technology transfer activities, and system contract incentives) to ensure that the technology base evolves into the necessary shared infrastructure.

We have now laid the groundwork that motivates our presentation of the elements of our vision of the future. We are

clearly aiming at achieving a state of practice in the 1990s that embraces use of complete software life cycle environments that improve productivity and, at the same time, improve our capacity to produce software that is reliable and adaptable. To get from here to there, we will need to adopt increased levels of shared infrastructure, we will have to integrate much of what is possible to do in isolation today, and we will have to achieve effective technology insertion of the resulting environment.

## 5.0 COMPONENTS OF OUR VISION FOR THE 1990'S

In this section, we sketch briefly some components of our vision. By putting these components together into a coordinated whole, the entire vision is achieved. For the purpose of this paper, let's think of the Software Initiative (SI) as the umbrella under which all of what follows happens, and let's consider the Ada Program and the Software Engineering Institute (SEI) to be parts of it.

### 5.1 Software Productivity Policies

While software producers have ample incentive for attempting to do something on their own about software productivity (for among other reasons, to attempt to maintain a competitive posture in the industry), some national leadership in this matter may greatly help. In fact, the maintenance of the national technological lead in software may depend critically on national leadership on matters of productivity, and failure to address this issue may mean forfeiting that lead to other nations.

We envisage that the Software Initiative would take some action to achieve the property of "affordability" of software in the 1990s. As we have outlined, we envisage that vigorous pursuit of productivity improvement policies could achieve this aim.

Guidelines for how to set up an in-house productivity project could be issued by DoD. These would be in the spirit of the current DoD CAD/CAM initiative to develop guidelines and support capabilities to improve productivity of DoD equipment manufacturers. For software, an initial set of such guidelines have already appeared in the open literature [Boehm, 1981b]. The SI could initiate an activity to draft productivity project guidelines, called say, PRODUCTIONMAN, or some such. The guidelines could be circulated to the community at large and could be tuned in the fashion of STEELMAN, STONEMAN, METHODMAN, and EDUCATIONMAN. Appendices could provide a few case histories, and the Software Engineering Institute could be the showcase for providing a working model free for all to adopt. While several corporations, such as IBM Santa Teresa and TRW, currently have such projects, an explicit model in the public domain invented to have the right characteristics for effective technology insertion could provide important advantages.

We envisage that DoD could formulate and adopt an effective means to spread software productivity policies nationally by: (1) producing guidelines in the form of a PRODUCTIONMAN under a SI activity, (2) making the Software Engineering Institute a living showcase for PRODUCTIONMAN and ensuring that an easy, effective technology insertion path for it is provided, and (3) seeing to it that DoD procurement policies require the adoption of effective software productivity plans by contractors.

In our vision of the software world of the 1990s, we forsee that a result of effective leadership from the SI will be to accomplish something equivalent to our scenario that 70% of the software producers will have quadrupled their productivity by 1990. By making it an important goal and providing the guidance and

incentives for how to accomplish it, the SI could make it happen. Without focus, leadership, and the will to do it, it may not happen.

## 5.2 Software Procurement Policies

We envisage that the SI will study procurement policies and will draft, tune, and cause to be adopted some procurement policies that are to be used in connection with Ada military software. Some elements of such policies are already under discussion with respect to the requirement to use certified Ada compilers on Ada programming projects.

As new techniques mature, procurement policies may have to be revised to take advantage of associated benefits. For example, if rapid prototyping techniques emerge to enable contractors to buy information that reduces risk in the early stages of a software project, some revision of procurement policies might be advisable to enable prototyping to be incorporated into the development cycle (without the contractor's being accused of "coding before writing his B5 specs."). The current initial DoD use of competitive concept definition contracts is a good step in this direction, but is not yet well-supported by DoD policies and procedures. As another example, if techniques are developed enabling reuse of reliable software components, some incentives may need to be built into the procurement process to encourage contractors to reuse components rather than rebuilding their own on a cost-plus basis.

As outlined in the previous section, incentives and guidelines to encourage the adoption of productivity improvements may be critical, and changes in the procurement process may be essential to catalyze the needed changes in some segments of the industry.

## 5.3 The Role of the Ada Program

The important thing about Ada, in our opinion, is not so much the product as the process. The process involves getting a wide community of sometimes competing interest groups to come together, formulate an agreement about sharing a way of doing business, and then living with it and reaping its benefits. The product, Ada, is a medium of exchange in which future sharing and cooperation can take place. Ada may represent a critical new element of shared infrastructure enabling value addition commerce to take place.

In our vision of the software world of the 1990s, we envisage that the Ada process could be used repeatedly in the decade of the 80s to achieve shared agreements and provide shared infrastructure that enhances productivity. Some possible examples are: (1) portable, standard programming environments --- mature APSEs with complete life cycle support toolsets, (2) an Ada software reuse library and retrieval system, (3) a standard "a-code", or Ada machine code, which could be cheaply reproduced on new target machines (in, say, a man-month of labor), and which, like Pascal's p-code, could provide a machine-independent way for supporting Ada software but which could also be micro-coded or implemented in silicon for great performance, (4) APSE virtual workstation standards and local network protocol standards to enable simple porting of Ada application software without having to change interface and network communication protocols.

We envisage that the Software Engineering Institute could spearhead this effort by pulling together pieces of the Ada program such as these, actively soliciting community input and managing the evolution of related proposed standards and shared agreements, providing a showcase of how it all works, and providing the means to do technology insertion into receiving organizations.

## 5.4 The Software Engineering Institute

We envisage the founding of a new, national-level Institute to carry out improvements critical to attaining the state-of-practice needed in the 1990s.

We believe existing institutions can't do the job. What is needed is something that can provide a bridge between existing organizations in government, industry, and academia --- something capable of bringing these existing institutions into cooperative endeavors. We doubt this can be accomplished from within.

An Institute is needed to play several key roles: (1) collecting and integrating existing tools into common, unified software life cycle support frameworks; (2) acting as a catalyst by energetically soliciting community opinion and helping the community to achieve a consensus on critical new aspects of shared infrastructure for the 1990s; and (3) furnishing effective institutional support for the technology insertion process, which, if not carefully planned and managed and if not vigorously supported, cannot accomplish an essential part of the overall job.

A key feature of technology transfer is to have people from DoD, industry, and academia rotating through the Institute. Such rotation would have the additional benefit of keeping the Institute fresh and vital over time. It would thus be a magnet for top talent without being a talent sink.

We envisage the Software Engineering Institute as a showcase for the SI. First, let's discuss our vision of its generic hardware/software tools environment. Then, let's discuss our vision of its mission and function.

### 5.4.1 The SEI's Programming Environment

Bauer's Second Law says, "The Talent Goes Where the Action Is." To give the SEI some glamor and prominence, to help it attract great people and to have prestige, we think it important that the hardware/software programming environment at the SEI should be right at the cutting edge of technology. Here are some ingredients in our vision of what this could mean:

#### The SAW and the Dynabook

People who work at the Institute should have Super APSE Workstations (SAWs) and portable Dynabooks to carry with them. The SAW should be based on bit-mapped display technology (probably in color, since if Mitsubishi and Hitachi already have it for under $5000 complete with a personal computer, that may be a coming thing). The SAW should contain a reasonably powerful chip computer (such as those in the M68000 or Dandelion class), it should have a reasonable amount of memory (such as that contained in, say, a Star), it should have packet-voice capabilities, and it should be linked over a local-area network (such as a 10Mbit Ethernet) to various shared services (such as laser printers, file servers, and gateways to external wide-area networks). The SAW should have standard virtual interface properties, thus setting an Ada community standard, and it should be a-coded so any manufacturer could produce a computer for a SAW with the a-code either micro-coded or programmed in software.

It is quite important that the SAW be powerful enough to support a complete electronic office and that all documents in the SEI be in machine form. Since there are two cases where functioning in an entirely electronic world is inconvenient these should be addressed: (1) There should be a graceful transition from the world

of electronics to the world of paper --- a laser-printer gets us from bit-mapped displays to paper, but to go back from paper to electronics, we need a high-resolution TV device that can photograph a document and produce a bit-mapped electronic image. A character recognition program needs to be added to this to recognize characters in many fonts and to convert from the bit-mapped character image to an internal binary computer code. (2) Since the SAW is not envisaged to be portable, we need a portable computer terminal to take home or to take with us when we travel. Alan Kay's notion of the dynabook fits in nicely here. A dynabook is a notebook sized object with a flat-panel display, a keyboard and a computer inside it that can connect to a network. The Grid Computer, with an electroluminescent display panel and autodial capability is here today for on the order of $8000, and could serve as a satisfactory initial dynabook. It may be possible to do better than this with fine resolution LCD technology, but the Grid Computer proves that the goal is clearly achievable.

## The Standard APSE

We envisage that the SEI would help define, use, and spread a Standard Ada Programming Support Environment (APSE) to accomplish its work. This standard APSE is envisaged to incorporate a broad range of life cycle support tools, including word-processing, management support (budgeting, scheduling, critical-path management, wo k breakdown structures), software project support (requirements documents, design documents, system architecture documents, code, test sets, unit development folders), an electronic office (appointments, reminders, memo-preparation and distribution, electronic mail, etc.), and, of course, Ada programming tools (compilers, optimizers, editors, linker/loaders, etc.). Complete life cycle support and a high degree of integration are the aims.

Standard Ada Methodology

For model defense application projects, a candidate standard methodology for Ada software development would be tried out at the Institute as a matter of policy. The Standard APSE would provide tool support for each phase or method in the candidate Standard Ada Methodology, and we would envisage that the SAW and the computer network to which it is attached would be capable of representing, displaying, and manipulating every work-product required in the Standard Methodology. The SAW network with its attached database computers would thus be the medium for providing life cycle integration and complete life cycle support in the service of (what we hope will be) a highly integrated Standard Ada Methodology. For emphasis, we think computer supported management practices and software disciplines will be a key factor in the success of the Institute.

### 5.4.2 The SEI's Mission and Function

The Software Engineering Institute has, in our vision of the 1990s, several key missions and functions. It may take a few years to get the SAWs, the APSE, and the Standard Methodology in place and working, but once this has been accomplished (and some standards have been shaken down and tested out in the process), it is time for the Institute to serve as a demonstration showcase and a technology insertion agent.

Technology insertion happens gradually and may be the single biggest obstacle to overcome in achieving the success of the SI. Organizations will not adopt new technology unless: (1) they hear about it and have the opportunity to see it work, (2) they know it is within their economic means to acquire it, (3) it is demonstrated to their satisfaction (either by in-house pilot projects or credible

demonstrations elsewhere) that it yields good cost-benefit improvements, and (4) it can be readily learned. While some technology insertion happens without any explicit leadership or management, we envisage that the SI will stand a better chance of success if its technology insertion components are explicitly planned, managed, and funded. Here the Software Engineering Institute can play a big role, and our 1990s vision holds that it will do so.

The SEI should select one or more key application areas, and apply its advanced Ada methods to the actual construction of Ada software systems for real target computers. In the process, it should demonstrate vastly improved productivity (compared to current baselines), it should write the software for adaptability and reliability (and spearhead advances in our knowledge for how to produce Ada software with these characteristics), it should derive from its created application a Library of Ada Software Components (to provide a basis for software reuse), and it should support technology insertion by: (a) publicizing its work and methods and giving demos, (b) making sure its methods and equipment are affordable for others to acquire and paving the way for acquisition, (c) giving statistics on its projects that demonstrate great cost-benefit results, (d) providing course materials to educate people in the use of Ada methodologies and APSE toolsets, and (e) engaging in actual technology insertion activities using rotating assignments of personnel on a two-way street from the Institute to the organization receiving the technology. Thus, the most important mission and function of the Institute, after shaking down Ada methods, is its technology transfer of those methods into appropriate receiving organizations.

We wish to stress that it will be essential to have an affordable Ada-based method for rehosting APSEs. If it costs a half

a man-millennium to rehost a standard Kernel APSE (KAPSE) which is to be used as a platform for rehosting Ada software on a new host computer, or if the Standard KAPSE is too inefficient (similar to the difficulty not yet overcome by the National Software Works), it will be impractical to rehost APSE developments and technology transfer cannot take place. Perhaps cheaply reproducible Ada run-time kernels (based on a-codes, or some such) may provide an alternative if Standard KAPSEs that are both efficient and cheaply rehostable do not emerge. Our vision of the future isn't going to work if we lack an effective means to rehost Ada software and if vigorous Institutional support for technology insertion cannot be counted on.

## 6.0 PUTTING IT ALL TOGETHER

Suppose the components of the vision we have sketched become a reality and are in place and operational in the 1990s. How will the world be different? Let's list some things we believe will differ dramatically from the world of today.

First, software manufacturing in this country would be vastly more productive than it is now. In fact, it would be over 300% more productive by 1990. And this could be the case, not at the sacrifice of software quality, but rather with the simultaneous achievement of improved adaptability and reliability --- in short, we would have achieved PAR, and the software demand gap would be closed.

Second, we would have in place (or at least in the initial stages of technology insertion, to be honest) a set of shared agreements that would form the backbone and infrastructure for a flourishing commerce and exchange of computer software. The Ada language would be the basic medium of exchange, but a-codes, SAWs, Standard APSEs, and Libraries of Ada Software components would set the stage for an improved economic system of value-added production,

in which software components could be acquired in the marketplace, assembled into new products, and then sold in the marketplace after value had been added.

Third, a Software Engineering Institute would be a showcase and a technology insertion agent for embedded systems in key application areas. It would illustrate the new methods and techniques, quantify their benefits, construct credible cases that the new methods and technologies in fact work well, assist in technology insertion into receiving institutions, and spearhead educational efforts to spread the new way of doing business.

Fourth, improved procurement policies would provide guidance and incentives for software reuse, the incorporation of prototyping into contracts to buy information reducing risk, and for encouraging the spread of productivity improvements in the contractor community.

Fifth, educational materials would be available to train practitioners in the use of good tools and effective practices.

Sixth, and finally, all of these capabilities would be integrated into the cohesive framework of the "smokestack diagram" given in Figure 5. The easy, consistent, mutually reinforcing use of these capabilities across the entire software life cycle will improve the productivity, adaptability, and reliability of the software maintenance process and products at least as significantly as it will impact software development.

In sum, this world of the 1990s does not require an unknown technological breakthrough* to achieve its significant PAR gains. It does require a lot of thoughtful, carefully-coordinated hard work to improve all the components of our existing software process and

---

* On the other hand, we do believe DoD investments to search for potential technological breakthroughs in software technology are certainly worthwhile and could produce even more significant PAR gains.

environment in ways that makes them more individually effective and mutually reinforcing.

## 7.0 REFERENCES


1. Alford, M. W., et al [1981] Distributed Computer Design Study, TRW Report to USA-BMDATC, (August 1981)
2. Boehm, B. W. [1981a] Software Engineering Economics, Prentice-Hall, Englewood Cliffs, N.J., (1981)
3. Boehm, B. W. [1981b] Improving Software Productivity, Proc. IEEE COMPCON Fall '81, Washington, D.C. (September 1981)
4. Boehm, B. W., et al [1982] The TRW Software Productivity System, to appear, Proc. 6th International Conf. on Software Engineering, Tokyo, Japan, (September 1982).
5. Graham, A.K., [1982] Software Design: Breaking the Bottleneck, IEEE Spectrum, (March 1982), 43-50.

## APPENDIX III

## PART B

### SOFTWARE TECHNOLOGY IN THE 1990's USING A NEW PARADIGM

### 1.0 INTRODUCTION

One direction that DoD's software initiative can take is to attempt incremental improvement in each portion of the existing software life cycle. This is a conservative, evolutionary approach, with a high probability of short-term payoff. It definitely deserves a major role in DOD's initiative.

However, because it is based on the existing software paradigm, the evolutionary approach is ultimately limited by any weakness of that paradigm. Since this paradigm arose in an era of little or no computer support of the software life cycle, it is important to examine the continued suitability of this paradigm for the future. The crucial change is that, whereas in the past computers were expensive relative to people, that position has radically shifted today towards expensive people. This shift will continue to widen in the future as hardware costs plummet. Not only are people the expensive commodity, there is an inadequate supply of those who are adequately trained. This problem is particularly acute for the military. Furthermore, several studies have shown that major military systems are becoming increasingly reliant on software. The speed with which it can be produced, the functional complexity it can embody, and the reliability it can attain become major determinants of these properties in the overall system.

Thus, there is a clear need to investigate a software paradigm based on automation which augments the effectiveness of a limited

supply of expensive people in producing and maintaining software. Unfortunately, the existing software paradigm is not an appropriate candidate because it contains two fundamental flaws which exacerbate the maintenance problem. First, the process of software development (the conversion of a specification into an implementation) is informal and largely undocumented. It is just this information, and the rationale behind each step, that is crucial, but unavailable, for maintenance. Second, maintenance is performed on source code (i.e. the implementation). All of the programmer's skill and knowledge has already been applied in optimizing this form (the source code). These optimizations spread information (take advantage of what is known elsewhere). Thus, they increase the dependencies among the parts. This clearly exacerbates the maintenance problem, especially since these dependencies are implicit.

With these two fundamental flaws, plus the fact that we assign our most junior people to this onerous task, it is no wonder that maintenance is such a major problem within the existing software paradigm.

Thus, we must look elsewhere for an appropriate automation-based software paradigm. This search, and the technology needed to support it, represents a viable and important alternate direction for DOD's software initiative. As will be explained below, the technology needed to support an automation-based software paradigm does not yet exist. This is why the current paradigm persists, and why incremental improvements to it are an essential short-term component of DOD's software initiative.

The rest of this Appendix is devoted to examining the longer term, higher payoff, and higher risk task of creating a new automation-based software paradigm.

## 2.0 DISCUSSION AND RATIONALE

### 2.1 Characterizing the Automation-based Software Paradigm

The nature and structure of this new software paradigm becomes evident as we examine the objectives it must satisfy. First and foremost among those objectives is maintainability. As important as maintenance is today (80 to 90% of total effort), its importance will increase in direct proportion to our ability to handle it. There is a tremendous backlog of pent-up user needs for existing systems which remain unanswered because of delay, expense, and limited human resources.

To avoid the two fundamental flaws in the existing paradigm (optimization spreading information and unrecorded implementation processes) which impede maintenance, consider a paradigm in which maintenance is performed on the specification rather than the implementation (source program) and the revised specification is then re-implemented.

By performing maintenance on the specification, we would be modifying the form which is least complex and most localized. At this level (before optimization issues have been integrated) modifications are almost always simple, if not trival. We are constantly reminded of this insight by end-users and managers who only understand systems at this (unoptimized) specification level and who have no trouble integrating new or revised capabilities in their model. It is only at the implementation level that those "simple" changes can have massive effects, both structural and textual.

Thus, by directly maintaining the specification we would drastically simplify the maintenance problem. In fact, since end-users understand this specification-level and generate the new/revised capability requirements, they should be able, with suit-

able specification languages, to perform this specification maintenance themselves. Such an occurrence fundamentally improves the software paradigm by employing a user created and maintained specification as the interface between those users and the implementors.

For this to become a reality, the end-users must be provided with a means of ensuring that the specified system matches their intent. This feedback is the second objective of the automation-based software paradigm. Today, this task is informally handled by system analysts who, by reason of their experience, are expected to be able to predict the behavior of the unimplemented system and determine whether it matches the users' needs. This is an awesome responsibility and becomes more and more unrealistic as the complexity of systems increase and as they become increasingly specialized to various user domains.

Suppose, instead, that the specification were "operational". That is, it had a formal semantics and could be executed as a program, albeit slowly. Then its behavior would (except for speed) be identical to a valid implementation. Hence, the specification itself could be used as a prototype of the system. Users could experiment with this prototype to determine whether or not it matched their requirements. In fact, experience has shown that the existence of such prototypes normally leads to improved perceptions of their needs. Thus, more responsive systems would result.

Thus far, we've been able to characterize the new automation-based software paradigm as one in which formal specifications are created and maintained by end-users who use those (revised) specifications as prototypes of the desired system to improve the responsiveness of those systems to real needs.

But where is the automation in the automation-based software paradigm? It is in the implementation process which we haven't yet considered. In this paradigm the specification is re-implemented after each revision. This brings us to the final three objectives of our paradigm--that the implementation process be fast, reliable, and inexpensive. If it is to be repeated after each modification it must obviously be fast and inexpensive. Most importantly, it must be highly reliable so that the entire testing process need not be re-performed on each iteration. Fully automatic implementation (i.e., with a compiler) clearly fulfills these objectives. Unfortunately, such a solution is not feasible because of the wide gap between high-level specification languages and implementations. The next section describes an alternative to full automation.

## 2.2 Automated Implementation Support

The main reasons that a fully automatic compiler cannot be created to translate from high-level specifications to efficient imlementations are that very little is known about how local optimizations can be combined to form global optimizations, and that the space of possible implementations is too large to search. These problems are avoided in existing compilers because their input is at a low enough level that purely local optimizations are sufficient. Strategic questions about algorithm choice, control structure, representation selection, caching intermediate results, buffering, etc., have been already made by the programmer. In our new software paradigm, such implementation issues have been specifically suppressed in the specification. They must be addressed as part of the implementation process. Since these decisions cannot be automated, they must be supplied by a person.

However, once these decisions have been made by a person, they can be automatically carried out. This defines a new division of labor for the implementation process. Programmers would still

make decisions about which optimizations to employ, but all of the program manipulations needed to realize these optimizations would be performed by the system.

Four important benefits would result from such a division. First, because all of the manipulations which convert the specification into an efficient implementation are being performed by the system, they could be performed without clerical error and could be cnecked for any special applicability criteria. Thus, the validity of the resulting implementation could be guaranteed by the process by which it was derived rather than by testing or proving correctness.

Second, by automating the manipulation of the program, the programmer would be freed of this duty (and the corresponding need to maintain consistency which occupies the vast majority of his/her current effort), and could spend more time considering the optimization issues, and hence, do a better job. In fact, implementation could become cheap enough that programmers would experiment with alternative implementations, thereby gaining experience and insight leading to better optimized designs.

Third, since the programmer's optimization decisions are being communicated to the system, they can be recorded and used as documentation of the implementation process for any later maintenance. When maintenance is performed (by modifying the specification), the revised specification needs to be reimplemented. Some modifications will not affect the set of optimizations to be employed. For these, the previously recorded optimization decisions (called the development) can be "replayed" to produce a new implementation. Usually, however, modifications to the specification will make some previous optimization decisions inappropriate and/or cause the set of decisions to be augmented. After the development has been suitably modified, it can then be "replayed" to produce a new imple-

mentation (and used as documentation of the current implementation for the next round of modification).

Finally, since maintenance has been simplified (by maintaining the specification and then re-implementing it) and the implementation process has been recorded (as the development) and is modifiable, the elusive goal of reusable software can be attained. Software libraries (with the exception of mathematical subroutine libraries) have failed because we have put the wrong things in those libraries. We've populated them with implementations which necessarily contain many arbitrary decisions. The chance that such an implementation is precisely right for some later application is exceedingly small. Instead, we should be placing specifications and their recorded development in libraries. Then, when one is to be "reused" the specification can be modified appropriately ("maintained") and its development changed accordingly. Thus maintenance of both the specification and its development becomes the basis of reuse, rather than a fortuitous exact match.

## 2.3 Sociological Effects of the Automation-Based Software Paradigm

Besides the obvious and dramatic effects on the reliability and cost of producing software that would result from such a paradigm, there are several which may be just as profound.

First, systems will evolve much more than currently. Today, maintenance tends to destroy structure, thus increasing the difficulty of further maintenance. Maintaining the specification should eliminate this problem. The pace of maintenance will be limited by the user's ability to generate and assimilate changes rather than by technological constraints.

Second, because systems are more evolvable, they will evolve more and, hence, be longer lived. They will become larger and will

integrate more capabilities. Evolution, rather than creation, will be the dominant programming mode.

Third, because systems are easier to modify, standard packages, which dominate today's market because of cost and personnel limitations will disappear. Users may start from a "standard" application, but they will personalize and augment it for their own purposes.

Fourth, portability will disappear as a problem because the machine on which software runs is just one of the decisions which gets made during the implementation process and which can be changed like any of the others.

Fifth, because the programmer's responsibility has been limited to just decision making, even very large systems will be implemented and maintained by a single programmer.

Finally, end users will be much more involved in the specification process and will use prototyping as the means for debugging specifications before implementation. This will open up "programming" to a much wider population and many systems will be created and successfully used without ever proceeding beyond the prototype stage (i.e. no human programmer will be employed to produce an "efficient" implementation).

## 3.0 USING A KNOWLEDGE-BASED APPROACH

One approach to achieving the automation-based software paradigm is by use of knowledge-based techniques. In this section we discuss a knowledge-based programming system characterized by a high degree of integration. It is an extrapolation from existing knowledge-based programming systems.

A single knowledge-base management system is used to store all program specifications, programming methods, synthesis rules, requirements, project management information, etc. The system is based upon one language. It is a very-wide-spectrum language ranging

from requirements to code, and can express all concepts for program development such as editing and help requests. The user interacts with the system in that one language or in multiple formats that are translated into that language.

The formats of information presented to both man and machine can be formal, graphical, example-based, trace-based, or even english-like. The computer will use an abstract version of its wide-spectrum language for all knowledge, although the detailed representation structures will vary and the representations used for different types of reasoning might vary.

A user's interaction with the system can range from simple commands to "programs" in a VHLL. The more complex requests will themselves require synthesis of programs for answering the requests. The requests will be answered by inference using the knowledge base. Examples would be requests for:

- bug reports, changes,
- general schemas or methods for solving domain specific problems (e.g., all clustering methods),
- programs or rules of interest, such as all rules that select data structures to implement ordered sets,
- explanations of requirements, specifications, code, program derivations, etc.,
- help in debugging, such as locating an inconsistency in a large rule base or a large program specification that has been modified,
- finding some part of a program specification or requirement that will be impacted by a proposed change,
- finding all relevant test programs that might be impacted by a change in a specification so that a re-implementation can be re-tested,

- monitoring of a program execution such as monitoring a derivation to find all uses of an optimization rule that combines loops during a derivation,

- analysis of alternative implementation paths to predict their likely cost,

- analysis of the execution of some program to see what operations are using up all the space or time,

- analysis of two non-integrated programs to help in interfacing them.

Examples of some of the kinds of tools or capabilities that would be available are as follows:

- a monitor that measures performance, frequency of execution of program statements, data set sizes, etc. This information helps to decide when pieces of a program should be recompiled. They also guide the knowledge-based compiler by providing data to help in selecting the appropriate new implementations,

- an efficiency estimator that predicts which proposed implementation will be best, where the bottlenecks are, or which pieces of the program should be made into modules,

- a help system that includes summarization and natural explanation capabilities for specification or whatever,

- a tutor system to help maintenance personnel familiarize themselves with pre-existing specifications. Newcomers to the system would use the help, explanation and program analysis tools to learn about specs and requirements to be able to modify them as needed,

- an editor that will incorporate both syntactic, semantic and domain-specific knowledge to assist in its editing task,
  (The boundaries between classical modules tend to break down as higher level capabilities are available, i.e., the distinction between an editor, consistency checker, and knowledge-base manager, is blurred as all capabilites are used in processing particular requests. For example, as editors incorporate more semantic and domain specific knowledge they blend with consistency testers. This argues against building non-integrated environments.)

- o a deductive inference and simplification system to help in optimizing and analyzing programs,

- o a project management and communication system that sends out notices or reminders of bugs, changes, tasks, schedules, etc. It would have sets of procedures for handling common management situations,

- o a requirements definition assistant that helps with requirements consistency, simplification, explanations, walkthrus, tradeoffs, etc., during requirements acquisition and modification,

- o a verification tool to help in verifying the correctness or consistency of a new program optimization rule (since only transformations need be verified).

## 4.0 TECHNOLOGY NEEDED FOR KNOWLEDGE-BASED APPROACH

The technology necessary for utilizing a knowledge-based approach to the new software paradigm consists of a wide variety of programming knowledge, along with appropriate ways to represent, utilize, acquire, and explain that knowledge.

The representation of this knowledge needs languages for requirements, specifications, refinements, transformations, annotations, constraints, assertions, etc. Such languages may eventually be integrated into very-wide-spectrum programming languages. Necessary work on these languages will include design, definitions, semantics and abstract representations.

In order to be able to utilize this knowledge, it should reside in suitably designed knowledge bases having suitable knowledge-base management tools.

The kinds of knowledge will be far-ranging, running from generic to application-specific. The more generic includes knowledge of data and control structures, common algorithms and programming techniques, optimization methods, loop combining, subroutine finding, program simplifications, problem reformulation, etc. The knowledge can also

be viewed by generic application area such as numerical computation, distributed computing, memory management, graph techniques, etc. More domain- or application-specific knowledge will be needed for higher level application support. Examples could be drawn from command and control, avionics, surveillance, logistics, etc.

Some of this knowledge will be in the form of appropriate inference and problem-solving techniques. These will also range from generic methods for deductive inference, inductive inference, constraint maintainance and simplification to software-specific methods including dependency analysis, efficiency estimation, specific decision procedures, etc.

This knowledge, including the appropriate inference and problem-solving techniques, drives the various tools for synthesis, debugging, management, maintenance, editing, modification, testing, validation, and acquisition.

Also important are the issues of communication, input-output, knowledge acquisition, human-machine interfaces, program specifications, explanations, etc. Suitable methods are needed for acquiring, debugging, validating and explaining the general and specific software knowledge, as well as program specifications, requirements, implementation decisions, domain models, etc., As systems grow, tools are needed to maintain self-consistency and consistency with domain models.

Specification (input) and explanation (output) formats include not only formal languages but traces, examples, graphics, and natural language where possible. Both the input and output of information in the various formats requires inductive and deductive inference. Methods of summarization and abstraction are also needed in such areas as explaining complex refinements or derivations.

Intelligent help systems will be of use in handling the apparent complexity of large knowledge-based systems.

Tools for project management will need taxonomies and models of users, communication, project structures, schedules, configurations, bugs, reporting procedures, etc. If project management tools are to help with particular programming methodologies, they need formal models thereof.

# APPENDIX IV

## SUMMARIES OF THE 1981 DEFENSE SCIENCE BOARD STUDIES AND RELATED PANEL RECOMMENDATIONS

## APPENDIX IV

## A BRIEF SUMMARY OF SOFTWARE RELATED DSB RECOMMENDATIONS

During the past year, a number of Study Panels sponsored by the Defense Science Board (DSB) and a USDRE Independent Review Committee have made recommendations related to the conduct of DoD-sponsored software R&D. Among these are: 1) the 1981 Defense Science Board Summer Study Panel on Technology Base, 2) the Defense Science Board Task Force on University Responsiveness to National Security Requirements, 3) the Defense Science Board 1981 Summer Study Panel on Operational Readiness with High Performance Systems and 6) the Defense Science Board Task Force on Embedded Computer Resources (ECR) Acquisition and Mangement. Important recommendations of these Board and Committees as they relate to DoD software R&D are summarized below.

### 1.0 THE DSB 1981 SUMMER STUDY PANEL ON TECHNOLOGY BASE [1]

The Defense Science Board 1981 Summer Study Panel on Technology Base has identified 17 technologies which can be expected to make "an order of magnitude" difference in DoD's deployable operational capability. The Panel considered Advanced Software/Algorithm development technology to be among the top 3 technologies which are most likely to lead to dramatic improvements in future weapon system capabilities. Two specific goals for software set forth by the Panel are:

1) An order of magnitude impact improvement in programmer productivity within 3 to 5 years, and

2) A noticeable shift away from the 90% of systems cost attributable to software.

The Panel concluded that cost effective software technology is crucial for:

1) Assuring software portability (including operating systems),

2) Fast software design, assembly, testing and maintenance.

3) Growth over time (expansion, adaptation).

4) Computational robustness and fault tolerance.

5) Automated programming.

The DSB Study Panel on Technology Base recommended that the DoD increase Advanced Software Technology R&D funding from the approximately $7 million now being invested per year to $30 million per year. The Panel further recommended that a vertically integrated R&D effort would be an appropriate structure for the program.

## 2.0 DSB TASK FORCE ON VHSIC PROGRAM [2]

The Defense Science Board Task Force on the Very High Speed Integrted Circuit (VHSIC) Program reviewed this Tri-Service Program to assess its organizational structure, its technical progress and the likelihood of the VHSIC program meeting its stated goals and objectives. The importance of the VHSIC program stems from the relationship of its technical objectives to future DoD weapons systems. In addition, the VHSIC program serves as a potential paradigm for future DoD coordianted, Tri-Service R&D programs.

The findings of the DSB Task Force on VHSIC concluded that the VHSIC technical goals are well conceived and that the program has a good probability of achieving them. The program, with its six prime contractors and numerous subcontractors, include industry leaders whose performance on the program has been excellent. In sumamry, the DSB Task Force concluded that the VHSIC Program is proceeding as planned on a technology program of great significance to the security of the United States.

The DSB Task Force on the VHSIC Program, in identifying future research topics highlighted the following areas which relate closely to software technology:

1) Fail-soft VLSI architectures for systems reliability,

2) A "silicon compiler", and

3) User-friendly design automation.

## 3.0 DSB TASK FORCE ON UNIVERSITY RESPONSIVENESS TO NATIONAL SECURITY REQUIREMENTS [3]

The Defense Science Board Task Force on Unviersity Responsiveness to National Security Requirements addressed important issues pertaining to the relationship of DoD to the University Community. Issues considered included national security manpower training and basic research, DoD/University contracting procedures, technology export controls and the relationship of DoD to the Unviersity Community. Issues considered included national security manpower training and basic research, DoD/University contracting procedures, technology export controls and the relationship of DoD with other University research sponsors, such as NSF.

The Task Force found that there is a pronounced manpower shortage today in computer science specialities where some areas now have 10 jobs for every computer science graduate with an advanced degree. While there has been a recent upturn in DoD-sponsored university research, the level of funding appears inadequate in some areas. For example, the DSB Task Force found that there is a shortage of DoD funds available to universities to purchase expensive modern computing equipment.

The DSB Task Force, in software technology-related areas, recommended that:

1) Emphasis be given to increasing the number of computer science graduates,

2) DoD funding of basic computer science research be continued, and

3) Provisions be made for increased DoD expenditures for University computing equipment.

## 4.0 USDR&E INDEPENDENT REVIEW OF DOD LABORATORIES [4]

The USDR&E initiated an Independent Review of DoD Laboratories with the objective of identifying where action needs to be taken to insure the long-term health of the DoD Laboratory structure. Particular attention was given to Laboratory operational procedures, personnel practices, and procurement practices.

The study team found that while there is a strong need for DoD Laboratories, a discontinuity does exist between the Laboratories and the operating forces which hinders technology transition to the field. This appears to be especially true in area of software technology. The committee, while recognizing the proliferation of breakthrough in microelectronics, expressed concern that there have not been parallel breakthroughs in the ability to develop, test and maintian necessary software. In the area of automated and fault-tolerant systems, the committee observed that both the Government and the private sector feel very much behind in their ability to support the software aspects of the problem.

To begin to meet these needs, the Independent Review Committee recommended that a new institution consisting primarily of non-government personnel be formed to provide the best environment for recruiting and retaining the personnel skills necessary to create a center of excellance in microelectronics and computer science.

## 5.0 1981 SUMMER STUDY PANEL ON OPERATIONAL READINESS WITH HIGH PERFORMANCE SYSTEMS [5]

The DSB 1981 Summer Study Panel on Operational Readiness with High Performance Systems emphasized that all phases of the software life cycle are important to high performance DoD systems. Specifically, the Summer Study Panel pointed out that "...software represents the major development risk for command and control systems." The Summer Study Panel surfaced several global concerns related to software design, production, testing and validation of DoD software. Among these are:

1) The design and production of operational software tends to lag behind the associated hardware and the software is often presumed to make up for system and hardware design deficiencies because it is assumed that software is relatively easy to design and modify.

2) Software costs dominate for some systems. The magnitude of this trend is not well recognized thorughout the DoD. The Electronic Industries Association (EIA) estimates that by 1990, 60% of the DoD electronics budget, some $46 billion, will be computer-related and that 80% of that or $37 billion will be for software and services with $9 billion for hardware.

3) Software for support training and maintenance systems is often far down the development priority ladder and, therefore, tends to lag even further behind than the operational software with attendent negative impacts in system availability.

4) Not only are the number of applications of software growing, but the complexity of the tasks being levied on the software is growing.

5) Software is always a development item and must be managed like one. We must recognize the cost/performance uncertainties, manage it in ways analogous to development hardware, prototype early, redesign and recost.

6) Significantly better tools and techniques are needed to design, build and check out sophisticated software.

7) The U.S. currently has some 300,000 full-time programmers. The current shortage is estimated to be over 150,000. As

non-DoD applications of computers increase, DoD faces a severe shortage of programmers and computer scientists.

To meet these concerns, the Task Force made the following recommendations:

1) Take planning and funding actions to minimize the rapidly increasing impact of software on readiness.

2) In the acquisition directives, reemphasize the need to manage software as a development item.

3) Plan and budget for continuing development and evolution of the software throughout the acquisition and operational phases of each program.

4) Institute a government funded 2 year training course in the software field with an accompanying 4 year military commitment.

5) Modify IR&D procedures to encourage contractors to make software related proposals.

6) Exempt software from the requirements for small business set-asides.

7) Ensure that the DSB software study includes readiness considerations.

## 6.0 DSB TASK FORCE ON EMBEDDED COMPUTER RESOURCES (ECR) ACQUISITION AND MANAGEMENT [6]

The Defense Science Board Task Force on Embedded Computer Resources (ECR) Acquisition and Management identified the following major issues realted to software:

1) Should DoD's proposed policy to standardize on a small set of Instruction Set Architectures (ISAs) be issued?

2) Should the DoD's program to provide a common language for broad use across the Department -- the Ada Program -- be modified?

3) Are changes desirable to management of DoD's computer acquisition and management policies?

In brief, the Task Force found that:

1) Arguments in favor of proceeding with the standardization of Instruction Set Architectures are compelling. This action is important if the DoD is to improve its position with respect to software development costs.

2) High Order Language standardization has had a positive effect. Implementation of the policy has, however, been spotty. The DoD's Ada Program is well based and is proceeding well. The Ada Language System (ALS) should be strengthened in view of its criticality to the MCF program and to efforts of the other services.

3) There is no consistent management process across the OSD staff and the military departments as its relates to computer (automation) technology. OSD has been relying on a committee management approach since 1976. The scope of automation in systems and in their support has far outstripped this approach.

The following software-related recommendations for action OUSDRE were made:

1) The proposed DoD Instruction 5000.5x, related to the adoption of a small number of ISAs for broad use within DoD should be issued. OSD should actively manage implementation of the policy and control both the waiver process and changes to the approved ISA list.

2) The Under Secretary should assure adequate and continuing support for the Ada Joint Program Office.

3) DoD Directive 5000.29 should be revised and re-isued expeditiously. It should emphasize the "software first" approach to system design and development and should encourage use of rapid software prototyping and competitive concept definition.

In summary the observations and recommendations of the DSB Panels and USDRE Committee reported above, highlight the need for a carefully coordinated DoD software initiative.

## 7.0 REFERENCES

1. Report of the Defense Science Board 1981 Summer Study Panel on Technology Base, November 1981, Office of the Under Secretary of Defense for Research and Engineering, Washington, D. C.

2. Report of the Task Force on Very High Speed Integrated Circuits (VHSIC) Program, February 1982, Office of the Under Secretary of Defense, Research and Engineering, Washington, D. C.

3. Report of the Defense Science Board Task Force on University Responsiveness to National Security Requirements, January 1982, Office of the Under Secretary of Defense, Research and Engineering, Washington, D. C.

4. USDRE Independent Review of DoD Laboratories, February 1982, Office of Under Secretary of Defense, Research and Engineering Washington, D. C.

5. Report of the Defense Science Board 1981 Summer Study Panel on Operational Readiness With High Performance Systems, April 1982, Office of the Under Secretary of Defense, Research and Engineering, Washington, D. C.

6. Report of the Defense Science Board Task Force on Embedded Computer Resources (ECR) Acquisition and Management, February 1982, Office of the Under Secretary of Defense, Research and Engineering, Washington, D. C.

# A P P E N D I X V

## SOFTWARE RELATED FOREIGN TECHNOLOGY INITIATIVES

## APPENDIX V

## SOFTWARE-RELATED FOREIGN TECHNOLOGY INITIATIVES SUMMARY

### Japan
### Japanese Fifth Generation Computer Initiative

The Japanese government, as a matter of economic policy, is actively promoting the development of knowledge-intensive industries. A specific objective of the Japanese in the 1980's is to leapfrog U.S. computer technology and become the world's leading supplier of advanced computer systems. Following two years of study and research, the Japanese have produced a body of ideas and plans for an initiative which they believe will result in "Fifth-Generation Computer Systems" in 1990.

The Japanese have recognized that the key to fifth-generation computers is software, the proliferation of which has led to a situation which they refer to as the "software crisis". To overcome this crisis, the Japanese believe that fifth-generation computer systems must have a wide variety of sophisticated functions to solve the problems which todays computers have.

The Japanese fifth-generation computer systems goals are:

1. To give Japan a leading role worldwide in the field of computer technology development.
2. To make Japan a better, richer society by:
   a. Improving the economic position of Japan;
   b. Improving the use of energy and other resources;
   c. Coping with the problems of an Aged Society.
3. Increase productivity in low-productivity areas, particularly the office, engineering design, agriculture and fisheries, medicine, education, public service, government.

4. Meet international competition.

5. Assist in saving energy and resources.

6. Cope with an Aged Society.

The Japanese fifth-generation system requirements are:

1. Increased intelligence and ease of use:

    a. It should be able to process speech, image, and hardcopy document inputs.

    b. It should be able to understand natural languages.

    c. It should be able to make inferences and learn from its experiences.

2. Lessen the burden of software generation:

    a. Automatic generation of software from specifications.

    b. Use a language in which programs can be automatically verified.

    c. Provide an appropriate computer architecture that can process such an ultra-high level language efficiently.

3. Improved performance to meet social needs:

    a. Better cost/performance.

    b. Light, low-power, small computers.

    c. High-speed, large-capacity computers.

    d. Reliable computers.

    e. Effective systems for protecting secrets.

Seven broad categories of fifth-generation computer research have been identified. These include: 1) basic application systems, 2) basic software systems, 3) new advanced architecture, 4) distributed-function architecture, 5) very large scale integrated

technology, 6) systematization technology and 7) development support technology.

The Japanese expect to advance the fifth-generation computers through a government coordinated program encompassing the research areas summarized above. It is estimated that the Japanese plan to invest approximately $500 million in their fifth-generation computer systems research and development between now and 1990.

## References


1. "Proceedings of International Conference on Fifth Generation Computer Systems," Tokyo, Japan, 19-22, October 1981.

2. "Japanese Map Computer Domination," Electronics, 17 November 1981.

3. "International Competition in Advanced Industrial Sectors: Trade and Development in the Semiconductor Industry," A study prepared for the Joint Economic Committee of the Congress of the United States, 18 February 1982.

4. "Japan's Strategy for the 80's," Business Week, 14 December 1981.

5. "A Newsletter on Japan Electronics," published by Advanced American Technology, Inc., Los Angeles, CA., 1 June 1982.

Japanese Themes in Research and Development

1. Basic Application Systems
   1.1 Machine Translation
   1.2 Question-Answering
   1.3 Speech Understanding
   1.4 Picture and Image Understanding
2. Basic Software Systems
   2.1 Knowledge base management system
   2.2 Problem solving and inference system
   2.3 Intelligent interface system
3. New Advanced Architecture
   3.1 Logic Programming Machine
   3.2 Functional Machine
   3.3 Relational Algebra Machine
   3.4 Abstract Data Type support Machine
   3.5 Data Flow Machine
   3.6 Innovative von Neumann Machine
4. Distributed function Architecture
   4.1 Distributed function architecture
   4.2 Network Architecture
   4.3 Data base machine
   4.4 High-speed numerical computation machine
   4.5 High-level man-machine communication system
5. VLSI Technology
   5.1 VLSI Architecture
   5.2 Intelligent VLSI CAD system
6. Systemization Technology
   6.1 Intelligent Programming Machine
   6.2 Knowledge Base Design System
   6.3 Systemization Technology for Computer Architecture
   6.4 Data Base & Distributed Data Base System
7. Development Support Technology
   7.1 Development Support System

## United Kingdom

Multiple organizations within the United Kingdom are presently planning software technology research and development thrusts. These planning efforts, which appear to be early in their evolution, are in obvious competition with each other for a limited amount of total funding and, therefore, detailed technical information is not gen-

erally available. The following summarizes what is known at present about two of the competing efforts in the United Kingdom.

The Science and Engineering Research Council (SERC) of Great Britian, corresponds in purpose and operation to the National Science Foundation (NSF) in the United States. As with NSF, the SERC is a major funding source for university research in the United Kingdom. In this capacity, the SERC has received proposals from various universities in the United Kingdom to undertake a technically-focused effort in software technology research.

One university has proposed to promote the unification of software technology research -- first among the United Kingdom research universities, and later industry-wide in the United Kingdom. While specific details are not known, it is believed that the recommended program is founded upon the promotion of software standards. In particular, standards have been proposed in operating systems, a high order programming language, computer hardware, local communications and remote communications protocols. UNIX, Pascal, Perq computers, the Cambridge ring and X.25 communications appear to be the leading contenders in each of these respective areas. A basic premise of this proposed effort is that maximum advantage be taken of existing, leading software and hardware technologies. This strategy, coupled with hoped-for adoption of standards among the heretofore disjoint university research efforts in the United Kingdom, will result in a "concentration of force" in software technology research.

The level of effort of the program proposed to SERC is on the order of 20 million over a five year period. This money will be invested in equipment and research contracts.

The other major contender for a United Kingdom software technology initiative is the Ministry of Defense (MoD). The Ministry of Defense in the United Kingdom corresponds to the DoD in the United

States. Like the software technology initiative proposed to SERC, the MoD's program is not yet totally defined and has not been funded.

The initiative that is evolving within the MoD appears to be similar to the Ada Programming Support Environment effort in the United States. The MoD software technology research effort is characterized as a "starting from scratch and doing it right" approach. Emphasis would be on software tools.

The proposed level of effort is estimated to be on the order to $10-15 million per year. MoD laboratories involved include the Royal Signals and Radar Establishment (RSRE), Malvern, United Kingdom.

In summary, the United Kingdom's software technology initiative is not totally defined at present. A sorting-out process is obviously underway, however. The two leading constituents are the Science and Engineering Research Council and the Ministry of Defense. The proposed programs differ in approach with one emphasizing adoption of existing technologies as initial standards (SERC), and the other developing its own software technology which will later become a standard (MoD).

## References

1. Industrial Research and Development, May 1982.

2. Computer World, May 24, 1982.

## France

### World Center for Computer Science and Human Resources

Early in 1982, President Francois Mitterand of France established an institution called the "World Center for Computer Science and Human Resources." The mission of this Center is to "unite the social sciences with computer technologies at a rate of development which exceeds that of automation." The World Center is charged with "accelerating the design of concepts and inspiring construction of a truly personal computer with which any person can interface in a fashion comparable to human-to-human conversation for purposes of human training."

The World Computer Science Center is built upon three concepts which the French government believes are fundamental:

1. A close link exists between technological progress and social progress;

2. The incapacity of the world economic system during the 1970's to arrest the aggravated social crises created by the explosive scientific development of automated systems which increasingly displace the work of human beings;

3. The irreversible emergence around the few economically advantaged countries of the more than three billion people of the third world by 1990 in search of the means of survival.

The individuals chosen by President Mitterand to head this Center are outstanding members of the world social and scientific research communities. These include: Jean Jacques Servan-Schreiber, Chairman; Dr. Samual Pisar, Vice Chairman; Professor Nicholas Negroponte, Executive Director; Professor Ray Reddy, Associate Director; and Professor Edward Ayensu, Associate Director. In addition, the Center's Board Members include Nobel Prize winner from Pakistan, Dr.

Adbul Salam and other scientists from France, Kuwait and Japan as well as nine French Cabinet Ministers.

Detailed technical plans as to how the French will proceed are presently being formulated. It is reported that the French are investing approximately $20 million in the World Center.

## References


1. "Hearings on the International Transfer of Technology before the Committee on Science and Technology Subcommittee on Investigations and Oversight and Subcommitte on Science, Research and Technology," Congress of the United States, 19 May 1982.

2. "French World CPU Science Center Stirs House Panel Concern," Electronic News, Electronic News, 7 June 1982.

# A P P E N D I X VI

## JOINT SERVICE TASK FORCE PROBLEM SUMMARY

Chapter 2 of the Report of the DoD Joint Service Task Force on Software Problems, Dept. of Defense, July 30, 1982, pp. 10-27.

## 2. Fundamental Difficulties with Software

Software is a critical element in DoD weapon systems and is essential to the DoD mission. It presents enormous opportunities, but the DoD faces a broad range of significant difficulties associated with the development, support, and management of software. There is no single description of a software problem that best characterizes the situation because the difficulties are numerous and interrelated. For example, poor design decisions are often related to management and engineering inexperience as well as inadequate modeling tools to evaluate the design; these bad design decisions may, in turn, result in software that is costly to develop and support; costly support breeds other problems such as a drain on development resource. The multiplicity of interrelated problems makes improvement in software development difficult unless many of the problems are addressed by a coordinated effort. Simply improving the tools of software production will not have the same impact as parallel improvements in the associated software engineering disciplines, and in the skills of the professional staff.

Appendix A contains a description of many of the difficulties DoD experiences with software. The appendix is not intended as a complete taxonomy but rather as a strong indication of the extent of the difficulties. The following sections parallel the taxonomy in Appendix A, summarizing the problems presented in it and illustrating the impact on the military mission with factual examples, where possible. Many of the examples are excerpts from the case studies presented in Appendix C.

Although there are many ways to organize a problem taxonomy, the task force chose four major categories: the life cycle of software, the software production environment, the software product, and software technical and management professionals. These categories represent different and overlapping views of software problems. Several different views are necessary to describe the complex state of DoD software.

### 2.1 The Life Cycle of Software

The life cycle of software is the set of activities associated with the development, in-service support, and operation of the software from its conception to its retirement from use. The part of the life cycle pertaining to development and in-service support is usually presented in terms of a process model. There are many models of the life cycle process which

differ primarily in the definition of activities and in the relations among these activities. No specific model is presented in this report.

The following sections present a discussion of life cycle problems categorized as requirements, management, acquisition, product assurance, and transition.

### 2.1.1 Requirements Definition and Analysis

The requirements definition phase of the system life cycle is the least formalized phase in the sense of software engineering disciplines. During this phase system requirements are stated, defended, and negotiated among the users, the development or support agent, and the acquisition agent. This negotiation process relates the estimate of resources needed for the development project to the severity of the threat and the scope of the needs to which the system is directed. Understanding among users, acquirers, and contractors is extremely important. One problem in understanding stems from the lack of precision of English prose. Another is the process of relating the system requirements to lower level software and hardware subsystems. The inclusion of interoperations with other equipment adds to the difficulty of achieving and maintaining understanding among the participants.

Requirements analysis and definition is one of the most critical yet troublesome activities in the software life cycle. The software requirements are often ill-defined and, more importantly, they are characterized by continual change. This is true more for software than other parts of the weapon system because software is the most adaptable to change. Therefore, many weapon system requirements changes are manifested in software changes. The mechanisms to manage this change are inadequate in many projects. In particular, it is often difficult to estimate the impact of a requirements change, adding to the unpredictablilty of software development resources and to frequent cost or schedule overruns.

Not only are some embedded software requirements difficult to define and highly volatile, but they have proven to be difficult to validate. This results in deployed systems that do not meet the full need of the user. The COMPREP System experience is an example of this situation (See Appendix B.8). This message processing system was abandoned after development when

it failed to pass its operational evaluation because it was deemed inappropriate to the operational needs of its user. Military units had to operate without a needed message processing system; this impact to DoD perhaps was more significant than the lost dollars.

Too little is known about how to assess the impact of a requirement, or a change in requirements, on software size and complexity. This problem exists not only for the mission system but also for its support systems. For example, small changes in aircraft performance requirements may lead to large changes in the air crew training device software.

On the positive side, properly designed software can be advantageous in handling changing system requirements. The addition of relatively small amounts of software can have a large effect on system success. An excellent example of this is found in NASA's Viking program, which was the mission to Mars. Originally, the ground support software was to receive and process simultaneously, the data from both Mars orbiters. The development contractor (after development was well along) recommended that, since each orbiter had on-board data storage, alternating (rather than simultaneous) transmission and processing be permitted. This change saved an estimated 30% of the already high projected costs of software without degrading the mission. Another issue on Viking was the ability to transmit new computer programs to the space vehicles. This requirement was added at relatively low cost and proved critical to achieving the launch date and extending the mission life of the spacecraft.

### 2.1.2 Management of Life Cycle Activities

Management of software life cycle activities exhibits many associated difficulties. Planning is often inadequate. This is due in part to the difficulty of estimating development resources accurately and of evaluating the impact of changes in requirements. These planning and control difficulties place management in a position of reacting to situations after they occur rather than anticipating them in time to prevent their occurrence or minimize their impact. The result is both added cost and extension of the development schedule.

Managers possessing little understanding of embedded computer software are often thrust into positions where their decisions directly affect the software process. This can be

manifested in many ways including: inappropriate contracts, lack of strong systems engineering in software development or modification, lack of attention to disciplines that are appropriate to software development, lack of effective plans for software activities, inappropriate methods and tools for tracking software projects, inappropriate assignment of responsibility, and inappropriate prediction, establishment, and monitoring of schedules and budgets.

Part of the difficulty in improving our ability to manage software life cycle activities is the lack of suitable measurement data or metrics. Without this data, management will continue to be hampered in planning, controlling, and evaluating software efforts. Measurement data on past projects are a prerequisite to adequate estimation of new projects. Measurements throughout the life cycle are necessary to give management the information required for control. Measurements are needed to allow management to evaluate new techniques. The current state-of-the-art does not permit software professionals to quantify the benefits of techniques because little empirical data exists.

Acquisition and engineering managers lacking familiarity with embedded computer software are often thrust into positions of responsibility. Some managers ignore the software development and support aspects of their jobs. Officers with little acquisition and embedded computer software experience are often responsible for projects involving millions of dollars. Fixed price contracts have been used inappropriately for high risk development causing financial strain to the contractor and cut corners on the developed software.

There is a lack of appreciation for discipline and good communication among systems, specialty, and software engineers. There is a need for measurable milestones and planning, assessment of quality, and planning for integration and test of software and systems.

There are no generally accepted models that can predict development size, complexity, staffing, schedule and budget for software. Unrealistic schedules, difficulty in monitoring development, and poor planning for the life cycle remain problems for some service elements. There is often a lack of good management discipline in all life cycle phases.

Foreign Military Sales (FMS) aggravate issues concerning release since software can contain intelligence, technology, performance and vulnerability information. Lack of understanding at the top government levels leads to inconsistencies in guidance and split responsibilities. Vague and incomplete disclosure letters leave the services in the awkward position of trying to satisfy conflicting DoD and State Department guidance. FMS also aggravate the design and skill shortage problems in the software area.

### 2.1.3 Software Acquisition

There is a lack of uniform methodologies for handling the acquisition of software. The detailed methodologies for hardware contracts are often inappropriate to software but are often used. Interfaces are poorly maintained with weak configuration control when different parts of a system are developed by different contractors because focus is normally on hardware with too little attention to software. Documentation is often of little value in understanding project status, because it is either absent or late. Procedures for change control during the procurement process are often time-consuming, inconsistent and ineffective. The procurement process is so lengthy that changes to requirements occur during the process. This causes further delay and, if uncontrolled, these changes can cause unnecessary cost escalation.

There is a great reliance within DoD on software contractors for creating the software. Contractors often convince project managers to permit use of non-standard techniques, tools and hardware which do not always offer an overall life cycle benefit in that they focus on the development cycle and totally ignore or pay only lip service to the in-service support.

A frequent error for embedded computer software is the belief that software and hardware can be developed and supported by different companies or agencies without a strong systems engineering authority to allocate requirements, coordinate design, resolve conflicts, and perform the system integration function. One particular $C^2$ system, started in 1973, attempted acquisition with separate hardware and software contractors. The software development cost went from $10M to $40M due to requirements changes and configuration differences. No contractor was responsible for overall system delivery. The program was cancelled in 1978 because the software would not work properly with the hardware.

### 2.1.4 Software Product Assurance

Software product assurance includes all the life cycle activities associated with demonstrating the correctness of the software product (e.g. requirements, design, code) and its quality. This is done through analysis, reviews, and testing techniques.

Product assurance of software through testing should be a continuing process throughout the system life cycle. This requires allocation of adequate resources for test planning and cooperation among the acquisition agent, development contractor, test agent, support agent and user. Independent verification and validation (IV&V) implemented early in the system life cycle can help achieve better quality software.

Too little is known about different testing techniques and how much testing is enough. Secure systems (not just operating systems), artificial intelligence systems, and distributed systems are applications where there are no good criteria for various applications to determine how much testing is optimal at each step in the software development process. In a recent digital flight control system development, during software integration, there were so many hard-to-find errors (due to insufficient module testing) that the integration was curtailed and more extensive module testing resumed. This caused a substantial slip in schedule.

Test environments are different from the real environment - a factor that influences the amount of test and the software reliability. Too often, testing is governed by the amount of time that is available. Optimal test requirements for different embedded computer software applications need far better definition.

### 2.1.5 Transitions in the Life Cycle

There are several points in the life cycle of software where problems occur in the transition to succeeding activities. Two of the more difficult transitions are from exploratory research to engineering development and from development to support. The former is characterized by new and rapidly changing technology, while the latter requires a stable technology base.

The very rapid change in computer technology has a destabilizing effect on acquisition and support engineering. For instance, firmware is widely used in today's weapon systems yet

acquisition familiarity, policy and support concepts have not kept pace with the technology. Contractual requirements intended for software have been disputed by contractors as not being applicable to firmware. For engineering development, the economies of hardware and software are shifting, software becoming the dominant cost item, but engineering tradeoffs are still imposed which favor hardware optimization. The DoD Acquisition Improvement Program has provided guidelines toward deciding whether the most recent technology should be incorporated in weapon systems or whether state-of-the-art research and development should be followed with pre-planned product improvement; and how technology transfer should take place using contractor incentives; but it is not clear how these guidelines are to be meaningfully applied to embedded systems software.

The transition from development to in-service support (sometimes referred to as "maintenance") is another such point where difficulties occur. These difficulties are usually characterized by differences in software support environments, standards, and methods between the development group and the in-service support group. There are several cases where this incompatibility has cost DoD millions of dollars.

## 2.2 The Support Environment

The software support environment is defined as the methods, tools, and facilities used for software development or in-service support. Problems in the support environment are categorized by the lack of disciplined methods, labor intensive process as caused by inadequate tools, reinvention of software, and insufficient capital investment. These four categories are discussed in the following sections.

### 2.2.1 Disciplined Methods

Software has been developed and supported by contractors (and in-house personnel) who are not required to practice adequate engineering discipline in software activities. This lack of discipline varies from those who recognize the necessity for discipline but fail to apply it on a particular program, to those who fail to recognize software as requiring discipline. Some of the shortcomings include: lack of adherance to good

programming standards, lack of adequate configuration management practices, inadequate baselining and documentation, and lack of adequate system engineering practices applied to a program. Major causes of this lack of discipline are: inadequate development standards, ineffective product assurance programs, inadequate subcontractor management, and deficiencies in contract requirements.

The problem of an inadequate software engineering discipline is characterized by the laissez faire attitude in the selection of techniques used from one project to another, antiquated programming methods, ad hoc requirements analysis and design, and the informal methods of testing complex systems. This situation has an adverse effect on development costs but can have a more significant effect on in-service support costs. Varied styles of documentation and coding, coupled with inadequate quality due to ad hoc techniques, can significantly increase the cost of maintaining and enhancing deployed software.

### 2.2.2 Development and Support Tools

Software engineering is, by its very nature, labor intensive. However, this is aggravated by the fact that techniques and tools are out-of-date, insufficient, inefficient, and limited. Automation of mechanical processes is occurring but more is needed to improve productivity. Automated tools could increase productivity but there are no accepted productivity measures for tools or tool sets.

The approach used in developing the F-15 software support facility illustrates the level of success possible in achieving tool uniformity. The F-15 Avionics Integration Support Facility (AISF) was developed by WR-ALC/MMEC, beginning in late 1976. The approach taken for development was to be as common as possible with the existing F-111 facility. This approach was intended to serve two purposes: (1) software would be transportable between facilities; and (2) by keeping the design concept as similar as possible, a model would be available for scoping, pricing, sizing, etc.

The design approach was built around a common support concept with common processors and peripherals, common language (where possible), and other common hardware. Due to the vast difference in weapon system mission and avionics system architecture, only a minimal amount of software was directly trans-

portable. However, much of the previously developed F-111 support software approaches and algorithms were reused. It was the opinion of personnel closely associated with the background of the F-15 facility planning that resource estimates would have been grossly underestimated had they not decided to use the F-111 facility as a model.

Proliferation of languages and instruction set architectures (ISA's) increases development support costs such as training, documentation, and tool reinvention. Personnel are unaware of tools and models that could save time. Tools are either unpublicized, hard to understand, or inefficient. Often, support tools are not useful because they are non-reusable due to unique language/ISA, have poor quality and poor documentation, or consist of a set of tools which do not interface and are not user-friendly. In addition, there is difficulty in making the transition from development to contractor and support personnel of non-weapon system specific tools.

Automated support of the software life cycle process activities is heavily oriented to the single activity of code generation which can represent a relatively small part of the total development effort. Computer-based tools to support requirements analysis, design, verification and validation, and management are inadequate. What tools do exist in these areas are not designed to work with tools supporting other activities. The consequence is that the development process remains heavily labor intensive and prone to errors. With the rising cost of software professionals, continuation of the current situation will become more and more costly to DoD.

### 2.2.3 Reinvention in Software

A concern in software development today is the inability to make use of functionally similar software developed for other systems. Reuse of software has been limited because of the difficulty of determining if functionally similar software exists, and because the large majority of software is not designed with reuse in mind. The result is a high degree of re-invention in software development with the impact of higher costs.[9]

---

[9]The Air Force does not perceive this concern with equal priority.

There may be abundant opportunities for reusable software components within such areas as realtime executives, file management subsystems, display software and report generators, which, though developed independently, represent very similar functional capabilities.

The elimination of reinvention may represent a large opportunity for cost avoidance for DoD, potentially measuring in the hundreds of millions of dollars. This opportunity is illustrated by the examples presented in Appendix C.3. One of the examples presented, the Navy Command and Control System (Ashore) software, represents an opportunity cost of over $10 million. Although the cost savings potential is significant if reinvention can be eliminated, there are other important payoffs. Namely, development lead time can be reduced and software reliability can be improved.

### 2.2.4 Capital Investment

In order to solve many of the software problems facing DoD, capital investment is required. To date, software-related capital investment has not been sufficient. The Army's Post-Deployment Software Support (PDSS) Centers illustrate this point. The funds required for capital investment and labor (R&D, OMA, MPA, and MCA funds) for nine support centers is $404 million for the period 1982-1987. The funds available are $135 million or a shortfall of $264 million. This funding shortfall will seriously impact the effective deployment of Army systems.

The area of software support environments represents both a great opportunity and a necessary burden for DoD. Better support environments are a prerequisite for better software engineering and the elimination of reinvention. The investment necessary to attain this (equipment, education, etc.) is large, and there has been an unwillingness to make the necessary capital commitment. What money is allocated to software support is often reprogrammed to other weapon system needs. Software is viewed as a low priority item by management when it comes to a funding crunch. If DoD is to solve its current software problems which pose a serious threat to its military mission, software must be given the proper priority in funding.

A subject related to capital investment is government-funded independent research and development (IR&D) by industry. There are too few good ideas for software research projects

being funded in the IR&D arena. The task force has the perception, which it was unable to validate, that software IR&D projects get low scores because evaluators are not software oriented, and results are not as tangible or quantifiable as the competing projects.

## 2.3 The Software Product

The software product is defined as the operational embedded computer software and any material required for adequate life cycle support - requirements specifications, design specifications, source code, test data, system generation data, unique support tools, etc. The problems associated with the software product are categorized as not meeting the need, difficult to measure, design attributes, and documentation. These categories are discussed in the following sections.

### 2.3.1 Software Utility

Deployed software often fails to meet the full need of the intended users. This can occur when the original requirements were incorrectly stated or when requirements were incorrectly implemented. Ambiguous, unclear, and incomplete communication between the user and implementor causes undetected omissions or internal contradictions. In some cases, it is difficult to assess the real user need without first placing a system in the field for the purpose of use and evaluation. Since completion and performance criteria are often inappropriate, incorrect, or impractical, product evaluation is difficult. In summary, the right requirements may not be stated and, even after the embedded computer software has been developed, it may be difficult to decide if the software meets its stated requirements.

An example of requirements uncertainty or conflict is provided by a recent fault isolation concept for a weapon system. The initial concept for fault isolation was to provide a rigid step-by-step guide to maintenance personnel through a video display. Although this was appropriate for inexperienced maintenance personnel to use, there was some concern that experienced maintenance personnel might want to avoid many of the pre-planned steps to get quickly to the tests that would exercise the components suspected to be faulty. An early hands-on evaluation of the system indicated that both approaches were necessary; hence, the software had to be modified to accommodate the additional requirement for ready access to specific tasks.

### 2.3.2 Software Metrics

Good analytic models and hard empirical data on software are lacking. Without these it is difficult to estimate development resources, evaluate future cost and mission impact of embedded computer-related decisions, or evaluate the positive benefit of new software engineering techniques. Existing software cost estimating models are inappropriate because they require data not available until the project is underway. Experience with these models indicates that they do not provide accurate estimates. To create better models and compute better estimates, a comprehensive data base of measurements is needed. Such a data base is not available today. This is due in part to the reluctance of companies to share data. The lack of a standard set of metrics also hampers collection and sharing.

There are no validated models of life cycle costs and productivities in embedded computer software development and support. For example, difficulty with using current models can be seen in these figures taken from a guidebook on software cost estimating:

| Project | Forecast Total MM | Actual Total MM | Ratio of Forecast/Actual |
|---|---|---|---|
| A | 419.7 | 71.0 | 5.9 |
| B | 2288.5 | 991.7** | 2.3 |
| C | 51.5 | 43.8 | 1.2 |
| D | 3298.7 | 514.8** | 6.4 |
| E | 7.9 | 7.3 | 1.1 |

(** Contains some estimate-to-complete data, along with actuals.)

One reason that metrics are not well-defined is that there are so many different approaches to software development and support. Until uniform software engineering approaches are defined, it will be difficult to define standards or guidelines for data collection. Such standardization is essential to the automation of the collection and data reduction activities which will make measurement cost effective and practical.

### 2.3.3 Design Attributes

The design of a system strongly influences its eventual total cost, how long it takes to develop it, and how amenable it is to change during development and support. The concepts and constraints for embedded computer software design are sensitive to technology--especially the architecture and capacity of the computational system (computers, memories, data buses etc.) and the role embedded computer software must play in the weapon system. Embedded computer software design demands a thorough knowledge of many fields--the weapon system, engineering, computers, and software technology. Good design provides many benefits to a system while poor design causes many difficulties.

Many of the problems associated with the software product find their source in design. Poor choices at the system design level can place constraints on the software that result in cost escalation. Software design which does not properly consider future changes increases the life cycle cost and limits the ability to react to new threats through software modification.

In addition, poor human engineering or lack of robustness of the software may make a system operationally unacceptable to the user.

Software's remarkable flexibility for change (when compared to hardware) often masks a serious problem--most embedded computer software cannot be changed as easily as it should because of its inflexible design. This is illustrated by systems where relatively small requirements changes (measured from the point of view of function) impact many aspects of the software and force a disproportionate amount of redesign and code generation. The cost of this inflexibility is significant. More disturbing is the inefficient use of scarce professional resources. They could be applied to the implementation of further enhancements which, otherwise, have to be delayed or cancelled.

The software implications of design decisions made at the weapon system or embedded computer system level are often ignored or misunderstood with disastrous results, as illustrated by the following example. An aircraft engine control system was based on an eight-bit microprocessor architecture. This

choice required the use of double precision in the software to achieve the necessary accuracy. The use of double precision resulted in timing and memory problems. To solve these problems, algorithms were revised and a more complex software design was necessary, severely complicating verification and testing. The project was delayed when errors became increasingly difficult to find and correct. Some might blame the software for this failure when, in fact, the root of the problem was poor decisions at the system design level because personnel did not understand the application.

### 2.3.4 Documentation

Documentation plays many roles in the development, operation, and support of embedded computer systems. Its focus can be anywhere from the broad concepts of a system design to minute details. The primary role of documentation is to convey information. During development, documentation captures the status of the software system at its current stage of development, including requirements, design, design trade-offs, implementation status, etc. This documentation allows the personnel of a large project to communicate to each other the information necessary to develop and manage a complex system. When the software is deployed, this documentation is used by the in-service support group to make modifications to the software. Modification of large software systems can be very costly without adequate documentation. During operation, documentation on the operation of the software is necessary for effective utilization by the user.

Documentation for embedded computer software is often inadequate for cost effective in-service support. There are a number of reasons for this. Documentation of a complex software system can be voluminous and costly to produce. If the development agent is under cost and schedule pressure, which it usually is, documentation is one of the first items to be cut or deferred. Until software documentation is an integral by-product of the software engineering process, supported by automation, and maintained in an electronic form, poor documentation will be a continuing problem.

When documentation is produced, it very often lacks the important attribute of traceability -- relation of a requirement to the design and code components that implement it. This makes it difficult to evaluate the impact of requirements changes. It also impedes the updating of documentation to reflect the continued changes to the software.

## 2.4 Embedded Computer Software Professionals

People are the most important resource in any software development or support effort. Design and redesign remains an intensive intellectual activity even after the best of automated tools is applied. Not all software professionals are so adept as others. The variation in effort among different individuals doing the same job can be as much as 25:1. With this degree of variance, it is no wonder successes and failures alike in software can be traced to the skills and experience of the professionals involved, both technical and management. Many rate the capability of the development team as the most important productivity factor.[10]

The problems relating to personnel are categorized as skills, personnel supply and proper incentives. These categories are discussed in the following sections.

### 2.4.1 Embedded Computer Software Professional Requirements

The key skill required by embedded computer software personnel is the ability to communicate across traditional engineering, computer science, and management disciplines to evolve coherent software requirements, designs, and modifications. Lack of currency for both technical and management personnel impedes the low risk transfer of new technologies to weapon systems. The government needs mature and adequate technical and management skills to prepare requirements, monitor developments, and support future changes. Acquisition and support skills for embedded computer systems are not taught in universities.

Qualified managers and professionals must have a wide range of skills and experience. The example in Appendix C.3 illustrates the difficulties that can be caused by inexperienced management. The management decisions concerning the signal processing computer and proper reserves for immature GFE/GFI caused a three year delay and millions of dollars in cost overruns. The ultimate impact was a three year delay in the deployment of a major surveillance system.

---

[10] See Software Engineering Economics by Barry W. Boehm, Prentice Hall, Inc., 1981.

The personnel problem is exacerbated by the limitation of most entry level and middle technical/management civil service positions to the Engineering (GS-800) series in the commands that acquire ECS. This excludes computer science and other related degree fields from pursuing careers or shifting to careers involving ECS acquisition. It should be noted that Civil Service regulations currently prohibit advertising a position as interdisciplinary when one of the disciplines is a "Professional" series (as is the GS-800 Engineering Series).

### 2.4.2 Availability of Qualified Professionals

The demand for software engineers and computer scientists exceeds the supply. There are no data on which to determine the true need, particularly for software acquisition personnel. The Service policies of rotating officers through a variety of jobs reduces and disrupts the supply of qualified personnel. For all Services there is a lack of adequately trained civilian and military personnel to satisfy post-deployment software support requirements. The shortage has caused the Services to become heavily dependent on contractors. This reliance creates problems with personnel turnovers, long learning curves, and less skilled personnel.

The Navy FCDSSAs are a good example of the projected impact of personnel shortages. By 1985, the projected shortfall in professional software personnel (as compared to demand at the FCDSSA in San Diego) will be over 200 people, and by 1990 it will be over 600 which means less than 50% of the demand will be met by the available resource. (See Appendix C.2)

The Army Post-Deployment Software Support Centers estimate a current shortage of 300 civilian and military personnel required for the manning of nine centers. This shortfall reflects a spending shortfall for the centers, but even if the funding were available, it is doubtful that the required personnel could be obtained.

The competition for qualified personnel will intensify with the increasing shortage in computer professionals. DoD will have difficulty competing with industry. For instance, the COMTEC 2000 study (Appendix B) shows that the second term retention rate of enlisted personnel with certain computer resource skills is only 50% because of the attraction of industry jobs.

The Electronic Warfare Support Directorate at Warner-Robins Air Logistic Command is another example of the people problem. They (WR-ALC/MMR) are only 60% staffed against their engineering slots. This shortage is aggravated by Foreign Military Sales of the Electronic Warfare systems they support. Since November, 1981, there has been a 30% increase in such sales (FMS EW cases). This number is expected to grow since there are more sales being negotiated. Though this activity is strongly driven by the policies of the present U.S. administration, it results in the requirement for long term support. Staffing the FMS work is difficult at best and tends to drain skilled people from U.S. embedded computer software work.

The problem of shortages in qualified software professionals will not be solved easily. This problem is not new, but it has not been perceived to be a pressing issue by DoD. As more demand is made of the current limited resource, corners will be cut, gradually reducing the operational capability and reliability of our weapon systems until failures become painfully more frequent.

### 2.4.3 Incentives

Present incentives favor migration of skilled personnel from job to job. Government software talent moves to other government jobs or to industry for promotion, better working conditions, and educational opportunities. Perceived salary shortcomings reduce the government ability to attract highly qualified people. There is no formal, effective career management program with classification/qualification standard adjustments, continuing professional education, challenging assignments, and identification/communication of career paths.

The following career pattern for entry level software system engineers has been observed at one Navy field activity. Entry level personnel are difficult to obtain. For those engineers that enter at GS-7 level, promotion to GS-9 occurs in one to two years. At this point, the software engineer is faced with a career decision: to market his valuable skill to the private sector or remain within the civil service. Approximately 60% opt for the private sector. Those electing to remain in the civil service generally reach GS-12 level in two to three more years. Then even the best are faced with a career stop causing another decision at which 40%, usually the top achievers, turn to the private sector. This loss of mature journeyman software engineers is the most difficult for the organization to absorb.

There needs to be a "software engineering" career field for both military and civilians which recognizes the skills needed are a combination of engineering and computer science. The career path should encourage the development of managerial and technical skills and experience pertinent to embedded computer software applications.

# A P P E N D I X  V I I

## SOFTWARE TECHNOLOGY INITIATIVE

## QUESTIONNAIRE SUMMARY

Main Body (Chapters 1 through 3) of
Eric D. Siegel, Summary of Responses
to the Software Technology Initiative
Questionnaire, MITRE, McLean, Virginia
(MTR-82W00085), May 1982 p. 1-21

## 1.0 INTRODUCTION

This document summarizes responses to the Software Technology Initiative (STI) questionnaire. The STI is part of a comprehensive Department of Defense software R&D effort to achieve order-of-magnitude improvements in software productivity, reliability, and maintainability for military systems. The goal is the development of a standard, coherent set of capabilities to support and enhance software development and maintenance.

The first phase of the STI included the creation and distribution of the report Candidate R&D Thrusts for the Software Technology Initiative (DTIC/NTIS # AD-A102 180) (nicknamed the "green book"). Forty recommended thrust areas were described in the report along with thirty-seven other possible thrust areas, a discussion of software problem areas, and a summary of some prior studies. The thrusts were organized solely according to the time when their principal benefits could be expected (under four years, four through seven years, over seven years).

Each copy of the book included a questionnaire for reader feedback on software problem areas and proposed thrusts. The goal of the questionnaire was to provide a simple means for respondents to:

- Rank problem areas and thrusts,
- Focus attention on work being done in suggested thrust areas,
- Discuss additional problem areas and thrust candidates,
- Suggest possible thrust groupings, and
- Suggest overall directions for the STI.

As our primary goal was to encourage comments from individual technical professionals and organizations, not to conduct a formal attitude survey, formal statistical sampling techniques were not used.

## 1.1 Questionnaire Distribution and Respondent Profile

The green book was widely distributed to obtain as much feedback from the military and civilian communities as possible. Based on address lists compiled by the R&D Technology Panel of the Management Steering Committee for Embedded Computer Resources (MSC-ECR), 225 copies of the book were distributed throughout DoD. (See Table I.) Each copy was accompanied by a cover letter from Mr. Joseph Batz of the Office of the Under Secretary of Defense for Research and Engineering (Electronics and Physical Sciences), and by a cover letter from the recipient's Service or Agency requesting that the accompanying questionnaire be filled out and returned to Mr. Batz.

Industry and academia were informed of the STI in late 1980, when announcements appeared in Commerce Business Daily and Communications of the Association for Computing Machinery asking for qualification statements. Over 120 statements were received, and this list of respondents formed the nucleus of the address list used to distribute the green book to DoD contractors. Ninety copies were distributed to industry and forty to academia, each accompanied by a questionnaire and a cover letter from Mr. Batz.

Over 355 copies of the report were eventually distributed directly, and additional copies were distributed indirectly by the Defense Technical Information Center (DTIC) and the National Technical Information Service (NTIS). Some replies were received from people who obtained questionnaires from the DTIC/NTIS copies.

To encourage a larger response, a follow-up letter was sent to recipients of the directly-mailed copies asking for return of the completed questionnaire. An interim tally was made of the first 124 questionnaires, and these findings were presented to the R&D Technology Panel of the MSC-ECR on 10 November 1981. By January 1982, 146 questionnaires had been received, and preparation of this report

TABLE I

DISTRIBUTION OF BOOKS AND RESPONSES

| | Books Distributed | Responses Received |
|---|---|---|
| Department of Defense | 225 | 108 |
| Air Force | 67 | 41 |
| Army | 55 | 31 |
| Navy | 73 | 21 |
| Miscellaneous DoD* | 30 | 15 |
| Universities | 40 | 11 |
| Industry | 90 | 27 |
| TOTAL | 355 | 146 |

*Miscellaneous DoD includes Agencies and offices of the Joint Chiefs of Staff and the Secretary of Defense.

began. It should be noted that the number of questionnaires does not indicate the number of people involved in preparing responses, because, as we had anticipated, many responses were assembled from the inputs of a group of people. Ten percent of the questionnaires were marked "official response of your organization," and many others show the signs of group effort. We have counted each questionnaire once, regardless of the number of people who may have worked on it.

Detailed information on the respondents was not directly collected, but the quality and quantity of the comments received, the respondents' job titles, and results of a small telephone survey imply that the great majority of the respondents are high-level managers both inside and outside of DoD who have substantial experience in computer project management.

## 1.2 Outline of the Questionnaire

The questionnaire is divided roughly into three parts: problem areas, proposed thrusts, and detailed comments about proposed thrusts. Each area has sections for formatted responses (e.g., ranking possible thrusts on a scale of one through five) and for unformatted responses (e.g., comments and additional problem areas). A copy of the questionnaire is included as Appendix A.

In the part dealing with problem areas, respondents were asked to indicate the seriousness of the various problem areas discussed in Appendix C of the green book and to add new problem areas as needed. In the part dealing with proposed thrusts, respondents were asked to indicate the overall worth of each of the 40 recommended thrusts and to recommend disposition of each thrust: accept, reject, modify, or combine with another thrust. There was also a provision for suggesting new thrusts and for including thrusts from the "Other Ideas" section of the green book. The third part of the questionnaire deals with specific proposed thrusts in detail. Respondents having special

knowledge of specific thrust areas were asked to complete a separate "Specific Candidate Questionnaire" for each such thrust. More than 200 Specific Candidate Questionnaire responses were received.

## 1.3 Outline of this Report

The questionnaire response summaries are presented in the body of this report. The next section of the report presents a summary of the part of the questionnaire dealing with software problem areas. It is followed by a section dealing with the proposed thrusts. Five appendices follow: (A) the questionnaire, (B) detailed presentation of additional problem areas suggested by respondents, (C) summary sheets for the Specific Candidate Questionnaires, (D) summaries of miscellaneous comments from all of the questionnaires and letters, and (E) a list of respondents.

## 2.0 SUMMARY OF PROBLEM AREA RESPONSES

The first section of the questionnaire deals with the software problem areas outlined in Appendix C of Candidate R&D Thrusts for the Software Technology Initiative. Summaries of the formatted and unformatted reponses follow a description of the questions.

### 2.1 Summary of Questionnaire

As can be seen from Appendix A, the problem section is divided into two parts: first, the respondent is asked to rate the relative seriousness of the 19 problem areas discussed in Appendix C of the green book by giving each a value from 1 (least serious) to 5 (most serious); these are the formatted responses. Second, the respondent is asked to add any problem areas that have been omitted and to rank them; these are the unformatted responses.

### 2.2 Formatted Responses

Results of computer analysis of the formatted responses are shown in Table II. Problem areas are listed in the order of importance given by respondents working for the Department of Defense; the rank and problem seriousness scores given by other respondent groups are also provided.

Although the data gathering method was not designed to create a statistically accurate sample, a few points are nevertheless evident:

- The bottom of the list is only slightly below average in terms of seriousness, and the variance for a typical problem area is 1.0. Therefore, all listed problem areas were considered important by the respondents.

- "Problems finding and keeping qualified personnel" (3.1) and "poor definition of goals and measures" (i.e., requirements) (1.3) are two of the top three problems for all columns.

- "Weak project leadership and coordination" (2.1), "inferior testing methodology" (1.7), and "ambiguous, unclear, out-

## TABLE II

## RANKING OF PROBLEM AREAS

| | Problem Area | All DoD | | Non DoD | | All | | Air Force | | Army | | Navy | | Misc. DoD | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 3.1 | Problems finding and keeping qualified personnel | 1 | 4.2 | 3 | 4.0 | 2 | 4.1 | 1 | 4.3 | 1 | 4.0 | 1 | 4.3 | 3 | 4.0 |
| 1.3 | Poor definition of goals and measures | 2 | 4.1 | 1 | 4.3 | 1 | 4.1 | 2 | 4.0 | 2 | 3.9 | 2 | 4.1 | 1 | 4.4 |
| 2.1 | Weak project leadership and coordination | 3 | 3.8 | 6 | 3.5 | 4 | 3.7 | 4 | 3.6 | 3 | 3.9 | 3 | 4.1 | 6 | 3.7 |
| 4.1 | Ambiguous, unclear, incomplete communication | 4 | 3.8 | 2 | 4.1 | 3 | 3.8 | 7 | 3.5 | 4 | 3.8 | 5 | 3.9 | 2 | 4.1 |
| 2.4 | Flawed methodology for the acquisition process | 5 | 3.7 | 11 | 3.0 | 7 | 3.5 | 3 | 3.9 | 7 | 3.4 | 9 | 3.6 | 5 | 3.9 |
| 1.7 | Inferior testing methodology | 6 | 3.7 | 7 | 3.5 | 6 | 3.6 | 6 | 3.5 | 6 | 3.5 | 4 | 4.1 | 4 | 4.0 |
| 1.4 | Faulty design | 7 | 3.6 | 4 | 3.9 | 5 | 3.7 | 5 | 3.6 | 5 | 3.7 | 10 | 3.6 | 10 | 3.3 |
| 2.2 | Poor monitoring and prediction of schedules and budgets | 8 | 3.5 | 5 | 3.6 | 8 | 3.5 | 9 | 3.3 | 10 | 3.2 | 6 | 3.9 | 8 | 3.7 |
| 2.3 | Unsatisfactory project control | 9 | 3.4 | 9 | 3.2 | 9 | 3.3 | 10 | 3.3 | 12 | 3.1 | 8 | 3.7 | 9 | 3.6 |
| 1.1 | Flawed and conflicting standards | 10 | 3.3 | 15 | 2.9 | 11 | 3.2 | 14 | 3.1 | 8 | 3.3 | 7 | 3.8 | 11 | 3.2 |
| 1.8 | Unsatisfactory product evaluation and follow-up | 11 | 3.3 | 13 | 3.0 | 12 | 3.2 | 8 | 3.4 | 14 | 3.0 | 13 | 3.2 | 7 | 3.7 |
| 4.4 | Poor phase-to-phase continuity | 12 | 3.2 | 8 | 3.4 | 10 | 3.2 | 15 | 3.1 | 9 | 3.3 | 12 | 3.3 | 16 | 2.9 |
| 3.3 | Poor exploitation of personnel | 13 | 3.1 | 12 | 3.0 | 14 | 3.1 | 11 | 3.3 | 16 | 2.9 | 15 | 3.0 | 12 | 3.2 |
| 1.6 | Poor use of implementation tools | 14 | 3.1 | 10 | 3.1 | 13 | 3.1 | 13 | 3.2 | 15 | 2.9 | 14 | 3.1 | 15 | 3.1 |
| 4.3 | Lack of project history | 15 | 3.0 | 16 | 2.8 | 15 | 3.0 | 16 | 3.0 | 17 | 2.9 | 11 | 3.3 | 14 | 3.1 |
| 1.2 | Inappropriate external constraints | 16 | 2.9 | 14 | 2.9 | 16 | 2.9 | 18 | 2.9 | 13 | 3.1 | 17 | 2.6 | 13 | 3.1 |
| 4.2 | Slow, outdated communication | 17 | 2.9 | 17 | 2.6 | 17 | 2.8 | 17 | 2.9 | 11 | 3.2 | 16 | 2.7 | 18 | 2.9 |
| 1.5 | Incorrect selection and use of languages and packaged software | 18 | 2.9 | 18 | 2.6 | 18 | 2.8 | 12 | 3.3 | 18 | 2.8 | 18 | 2.5 | 19 | 2.3 |
| 3.2 | Unsuitable competence measures | 19 | 2.7 | 19 | 2.4 | 19 | 2.6 | 19 | 2.7 | 19 | 2.6 | 19 | 2.5 | 17 | 2.9 |

dated communication" (4.1) appear in the top six for all columns.

- o With only a few minor exceptions, the managerial-related problems (2.x) always appear in the top half of the problem list for all columns.

- o "Finding and keeping qualified personnel" (3.1) is perceived as much more of a problem than measuring their competence (3.2) or making best use of them (3.3).

- o "Unsuitable competence measures" for personnel (3.2) and "incorrect selection and use of languages and packaged software" (1.5) invariably fall at the bottom of the problem list for all columns except the Air Force, where (1.5) is distinctly higher.

## 2.3 Unformatted Responses

A large number of respondents (45%) chose to write in one or more additional problem areas. In all, 142 problem areas were suggested. They have been clustered into the 26 groups shown in Table III. As is evident from the table, lack of proper management training, poor definition and evaluation of requirements and specifications, and lack of appropriate software and management tools and techniques were considered to be the problems most worthy of special mention. A more detailed tabulation of results is presented in Appendix B.

## 2.4 Summary

Problems finding qualified personnel and problems in defining the goals of a project and the appropriate measures of success were key problem areas, along with weak project leadership and coordination. These problem areas are strong indicators of the poor state of the art in software engineering. Because it is extremely difficult to define and manage a project by use of standard, accepted definitions, methodologies, and measures of success, DoD software project

## TABLE III

## ADDITIONAL PROBLEM AREAS

| Problem Area | # of times suggested | avg. score |
|---|---|---|
| **TECHNICAL** | | |
| Definition and evaluation of requirements and specifications | 13 | 4.85 |
| Standards and modularization | 10 | 4.70 |
| Software tools and languages | 9 | 4.56 |
| Maintenance and modification | 8 | 4.75 |
| Documentation | 6 | 4.17 |
| User involvement in design | 5 | 4.50 |
| Hardware / software tradeoffs | 5 | 3.75 |
| Inclusive methodology for design, construction, & maintenance | 4 | 4.50 |
| Quality assurance | 4 | 4.50 |
| Computer security | 4 | 4.00 |
| Tool evaluations | 3 | 5.00 |
| Software metrics | 3 | 4.67 |
| Software reliability | 1 | 4.00 |
| Use of computer graphics | 1 | 3.00 |
| **MANAGERIAL** | | |
| Budgeting and estimating tools | 11 | 4.91 |
| Management having software expertise | 11 | 4.70 |
| Budget and contract management | 7 | 4.29 |
| Changing specifications during initial development | 5 | 5.00 |
| Maintenance of standards of excellence | 5 | 5.00 |
| Configuration management | 5 | 4.20 |
| DoD computer policies and broad goals | 5 | 3.80 |
| Planning of the software development effort | 2 | 4.00 |
| Access to computing facilities | 1 | 5.00 |
| Lead time to acquire replacement hardware | 1 | 5.00 |
| **PERSONNEL** | | |
| Training and technology insertion | 10 | 4.30 |
| Availability and selection of qualified personnel | 3 | 4.67 |

managers have had to rely on experts with a successful track record in the art of creating a successful software package or system. Unfortunately, the techniques used by these experts have not been quantified or standardized, their turnover is high, and they are in extremely short supply. The responses also appear to indicate that improving small areas of the software development process, while helpful, is not sufficient. The entire development process needs major improvement both technically and managerially to increase development productivity and the rate of success. The problem area responses rank managerial considerations high, while the proposed thrust responses rank technical considerations high.

## 3.0 SUMMARY OF RECOMMENDED THRUST AREA RESPONSES

Analyses of the questionnaire's second portion (Candidate Thrusts) and the third portion (Specific Candidate Questionnaire) are combined in this section.

### 3.1 Summary of Questionnaire

Respondents were first asked to rate all 40 of the tentatively accepted thrust candidates on a 1 (low) to 5 (high) scale in terms of the thrust's contribution towards an order-of-magnitude improvement in the software development process. They were also asked to suggest a disposition for each candidate: accept, reject, accept with modifications, or combine with another candidate. Space was provided for the respondent to indicate that he or she would fill in a "Specific Candidate Questionnaire" for certain thrusts. These Specific Candidate Questionnaires asked detailed questions about thrust applicability to DoD, possible payoffs, R&D success probability, ease of technology insertion, and ongoing research. Finally, the respondents were asked to provide comments and to name tentatively rejected candidates that should be reconsidered.

### 3.2 Thrust Candidate Responses

Results of computer analysis of the thrust worth rankings are shown in Table IV. The thrusts are listed in the order given by respondents working for the Department of Defense; the rank and thrust worth scores given by other groups of respondents are also provided. The rank and worth scores of a thrust are enclosed in a box if its rank in a respondent group differs from its rank in the "all DoD" group by ten or more places. The "All" group is excluded from the boxing procedure as its inclusion would not add any information. Ranks were used instead of thrust worth scores because of the different ranges between the maximum and minimum scores among the

## TABLE IV

## RANKING OF TENTATIVELY ACCEPTED THRUSTS

| | Thrust | All DoD | | Non DoD | | All | | Air Force | | Army | | Navy | | Misc. DoD | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A.1.1.1 | Integrated Software Support Environment | 1 | 4.1 | 1 | 4.5 | 1 | 4.2 | 1 | 3.9 | 1 | 4.1 | 1 | 4.7 | 6 | 4.0 |
| A.1.1.4 | Set(s) of Tools Covering Entire Lifecycle | 2 | 3.9 | 2 | 4.2 | 2 | 3.9 | 2 | 3.8 | 12 | 3.6 | 2 | 4.1 | 5 | 4.0 |
| A.1.1.9 | Earliest Possible Error Detection | 3 | 3.8 | 5 | 3.9 | 3 | 3.8 | 6 | 3.6 | 4 | 3.8 | 4 | 4.0 | 13 | 3.8 |
| A.1.4.5 | Distributed Functions and Resources | 4 | 3.7 | 4 | 3.9 | 4 | 3.7 | 10 | 3.4 | 6 | 3.7 | 3 | 4.1 | 10 | 3.9 |
| A.1.6.1 | High-Confidence Software Testing | 5 | 3.7 | 6 | 3.7 | 5 | 3.7 | 5 | 3.6 | 13 | 3.6 | 6 | 3.7 | 1 | 4.1 |
| A.2.3 | Technology Insertion in the Software Area | 6 | 3.7 | 10 | 3.6 | 7 | 3.6 | 12 | 3.4 | 2 | 4.0 | 7 | 3.7 | 15 | 3.7 |
| A.1.7.2 | Impact Analysis of Proposed Change | 7 | 3.6 | 22 | 3.1 | 10 | 3.5 | 8 | 3.6 | 14 | 3.5 | 5 | 3.8 | 11 | 3.9 |
| A.1.1.6 | Programmer Workstation | 8 | 3.6 | 8 | 3.7 | 8 | 3.6 | 18 | 3.2 | 5 | 3.8 | 10 | 3.7 | 4 | 4.0 |
| A.1.4.6 | Suitable Communication Interconnection | 9 | 3.5 | 19 | 3.2 | 13 | 3.5 | 20 | 3.1 | 9 | 3.7 | 8 | 3.7 | 2 | 4.0 |
| A.1.1.2 | Ada Package Sets for Common Usage Areas | 10 | 3.5 | 13 | 3.4 | 11 | 3.5 | 16 | 3.2 | 3 | 3.9 | 18 | 3.2 | 12 | 3.9 |
| A.1.3.6 | User-Oriented Requirements Interface | 11 | 3.5 | 3 | 4.1 | 6 | 3.6 | 19 | 3.2 | 16 | 3.5 | 9 | 3.7 | 7 | 3.9 |
| A.1.3.1 | Rapid Prototyping | 12 | 3.5 | 7 | 3.7 | 9 | 3.5 | 11 | 3.4 | 11 | 3.6 | 17 | 3.3 | 16 | 3.6 |
| A.1.1.7 | Useful Measures of Software Quality | 13 | 3.5 | 12 | 3.5 | 12 | 3.5 | 7 | 3.6 | 26 | 3.0 | 11 | 3.6 | 14 | 3.7 |
| A.3.2 | Intensive Advanced Programmer Training | 14 | 3.4 | 18 | 3.2 | 15 | 3.4 | 21 | 3.1 | 15 | 3.5 | 13 | 3.5 | 9 | 3.9 |
| A.4.2 | Built-In Training and Documentation | 15 | 3.4 | 14 | 3.3 | 16 | 3.4 | 23 | 2.9 | 7 | 3.7 | 15 | 3.4 | 8 | 3.9 |
| A.2.1 | Acquisition Manager's Support System | 16 | 3.3 | 34 | 2.7 | 21 | 3.2 | 3 | 3.7 | 22 | 3.2 | 16 | 3.4 | 31 | 2.9 |
| A.1.1.10 | Configuration Independence | 17 | 3.3 | 30 | 2.8 | 17 | 3.2 | 9 | 3.4 | 10 | 3.7 | 32 | 2.7 | 18 | 3.4 |
| A.1.5.2 | Formal Verification of Large Systems | 18 | 3.3 | 29 | 2.8 | 20 | 3.2 | 4 | 3.7 | 21 | 3.2 | 23 | 3.0 | 20 | 3.4 |
| A.1.7.1 | Facilitating System Evolution | 19 | 3.3 | 9 | 3.6 | 14 | 3.4 | 14 | 3.3 | 20 | 3.2 | 14 | 3.5 | 21 | 3.3 |
| A.3.5 | Improved Education About Software | 20 | 3.3 | 25 | 3.0 | 19 | 3.2 | 13 | 3.4 | 8 | 3.7 | 27 | 2.8 | 28 | 3.0 |
| A.1.3.4 | Built-In Testing | 21 | 3.2 | 20 | 3.2 | 18 | 3.2 | 15 | 3.3 | 27 | 3.0 | 26 | 2.9 | 3 | 4.0 |
| A.1.2.1 | Rapid Simulation | 22 | 3.2 | 24 | 3.1 | 22 | 3.2 | 22 | 3.1 | 24 | 3.1 | 12 | 3.5 | 22 | 3.3 |
| A.1.3.5 | Forgiving Systems | 23 | 3.1 | 17 | 3.2 | 23 | 3.1 | 30 | 2.8 | 17 | 3.4 | 22 | 3.0 | 24 | 3.2 |
| A.2.2 | Software Technology-Compatible Acquisition | 24 | 3.1 | 28 | 2.9 | 24 | 3.0 | 17 | 3.2 | 19 | 3.2 | 33 | 2.7 | 26 | 3.1 |
| A.1.1.3 | System Dictionary/Directory | 25 | 2.9 | 21 | 3.2 | 25 | 3.0 | 28 | 2.8 | 29 | 3.0 | 25 | 2.9 | 23 | 3.3 |
| A.3.4 | Personnel Independence | 26 | 2.9 | 37 | 2.6 | 30 | 2.8 | 27 | 2.9 | 30 | 2.9 | 19 | 3.2 | 33 | 2.8 |
| A.3.6 | User Programming | 27 | 2.8 | 16 | 3.3 | 27 | 2.9 | 24 | 2.9 | 33 | 2.8 | 37 | 2.4 | 19 | 3.4 |
| A.1.4.4 | Exception Handling | 28 | 2.8 | 31 | 2.8 | 31 | 2.8 | 35 | 2.6 | 32 | 2.8 | 20 | 3.0 | 17 | 3.4 |
| A.1.3.2 | Application Domain Expertise | 29 | 2.8 | 33 | 2.7 | 32 | 2.8 | 26 | 2.9 | 23 | 3.1 | 30 | 2.7 | 38 | 2.4 |
| A.1.3.7 | Complex Knowledge-Based Systems | 30 | 2.8 | 15 | 3.3 | 28 | 2.9 | 25 | 2.9 | 31 | 2.8 | 24 | 2.9 | 36 | 2.4 |
| A.1.4.1 | Data Flow Approach | 31 | 2.8 | 23 | 3.1 | 29 | 2.9 | 33 | 2.6 | 25 | 3.1 | 29 | 2.7 | 34 | 2.8 |
| A.3.1 | Superperformer Competencies | 32 | 2.8 | 39 | 2.3 | 36 | 2.7 | 34 | 2.6 | 18 | 3.2 | 28 | 2.8 | 39 | 2.3 |
| A.1.1.5 | Software Engineer's Support System | 33 | 2.8 | 11 | 3.5 | 26 | 3.0 | 29 | 2.8 | 28 | 3.0 | 36 | 2.4 | 35 | 2.8 |
| A.1.4.3 | Predicate Approach | 34 | 2.7 | 32 | 2.7 | 34 | 2.7 | 36 | 2.5 | 35 | 2.7 | 21 | 3.0 | 30 | 2.9 |
| A.1.4.2 | Self-Interfacing Software | 35 | 2.7 | 38 | 2.5 | 37 | 2.6 | 32 | 2.6 | 36 | 2.6 | 34 | 2.6 | 25 | 3.2 |
| A.3.3 | Programmer Laboratory | 36 | 2.6 | 27 | 2.9 | 35 | 2.7 | 37 | 2.4 | 34 | 2.7 | 31 | 2.7 | 32 | 2.9 |
| A.1.3.3 | Data Validation | 37 | 2.6 | 26 | 3.0 | 33 | 2.7 | 31 | 2.7 | 37 | 2.6 | 38 | 2.2 | 27 | 3.1 |
| A.1.5.1 | Transform Software to Improve Quality | 38 | 2.4 | 36 | 2.6 | 38 | 2.4 | 39 | 2.3 | 40 | 2.4 | 39 | 2.0 | 29 | 3.0 |
| A.4.1 | Voice Replaces Text | 39 | 2.3 | 40 | 2.1 | 40 | 2.3 | 40 | 2.1 | 38 | 2.5 | 35 | 2.4 | 37 | 2.4 |
| A.1.1.8 | Multiple Representations of Software | 40 | 2.3 | 35 | 2.6 | 39 | 2.4 | 38 | 2.4 | 39 | 2.5 | 40 | 2.0 | 40 | 1.8 |

various respondent groups and because of the compression of scores within each group.

Most respondents also submitted one or more Specific Candidate Questionnaires when they had special knowledge. More than 200 such questionnaires were received, and the results for each thrust are summarized in Appendix C.

A few observations may be made by looking at Table IV, although, as mentioned before, the data gathering was not designed to create a statistically accurate sample:

- "Integrated Software Support Environment" (1.1.1) is clearly the top choice, and it is followed by "Set(s) of Tools Covering Lifecycle" (1.1.4). The "miscellaneous DoD" respondent grouping has significant score compression at the top of their rankings, so their ranking of these thrusts in positions six and five is not particularly significant. Only the Army is markedly different, ranking "Set(s) of Tools..." in position twelve.

- "Earliest Possible Error Detection" (1.1.9) and "high-confidence software testing" (1.6.1) are consistently ranked very highly.

- "Distributed Functions and Resources" (1.4.5) is also ranked highly, reflecting the emergence of distributed systems and of firmware implementation of functions.

- "Transform Software to Improve Quality" (1.5.1), "Voice Replaces Text" (4.1), and "Multiple Representations of Software" (1.1.8) are at or near the bottom of the rankings in all respondent groups.

- Reflecting the needs of DoD, Defense Department personnel ranked a number of thrusts at a distinctly higher level than did non-Defense personnel. These thrusts were "Impact Analysis of Proposed Changes" (1.7.2), "Suitable Communication Interconnection" (1.4.6) [Contractor desire to avoid premature standardization may play a part in this discrepancy.], "Acquisition Manager's Support System" (2.1), "Configuration Independence" (1.1.10), and "Formal Verification of Large Systems" (1.5.2).

- On the other hand, thrusts that were ranked markedly higher by non-DoD personnel than by DoD include "Facilitating System Evolution" (1.7.1), "Complex Knowledge-Based Systems" (1.3.7), "Software Engineer's Support System" (1.1.5), and "Data Validation" (1.3.3).

In addition to evaluating the worth of the various thrusts, the respondents were asked to suggest dispositions for them. After examining the data, it was decided to focus only on the disposition of "combine with another candidate," because the dispositions of accept and reject could almost always be inferred from the thrust worth score, and the "modify" disposition was rarely used. (There were a few cases where highly-ranked thrusts were rejected because the respondent believed the STI was not the appropriate vehicle for them.) An attempt to cluster the thrust areas into groups through correlation and factor analysis failed to produce useful results, since the thrust worth scores are probably correlated with a goodness scale more directly than with each other. The respondents suggested 417 thrust pairings, and these were used for the cluster analysis with far more success. All pairings that were suggested more than five times were used, producing the clusters shown in Figure 1. Note that the figure shows the number of times each pair was suggested. Table V shows the ranking of the clusters as if they were inserted into the Table IV rankings of thrust worth. The clusters are ranked according to the highest-ranking thrust contained within them. The average of the component thrust worth scores is also given for each cluster, but the average was not used for ranking because it was assumed that ranking was more sensitive to the cluster's highest-ranking thrust than to the average of all contained thrusts. A few observations may be made:

- Cluster #1, "Integrated Software Support Environment and Software Packages," is by far the favorite. Not only is it the most frequently mentioned, but it also contains a highest ranked thrust in all respondent groups except for

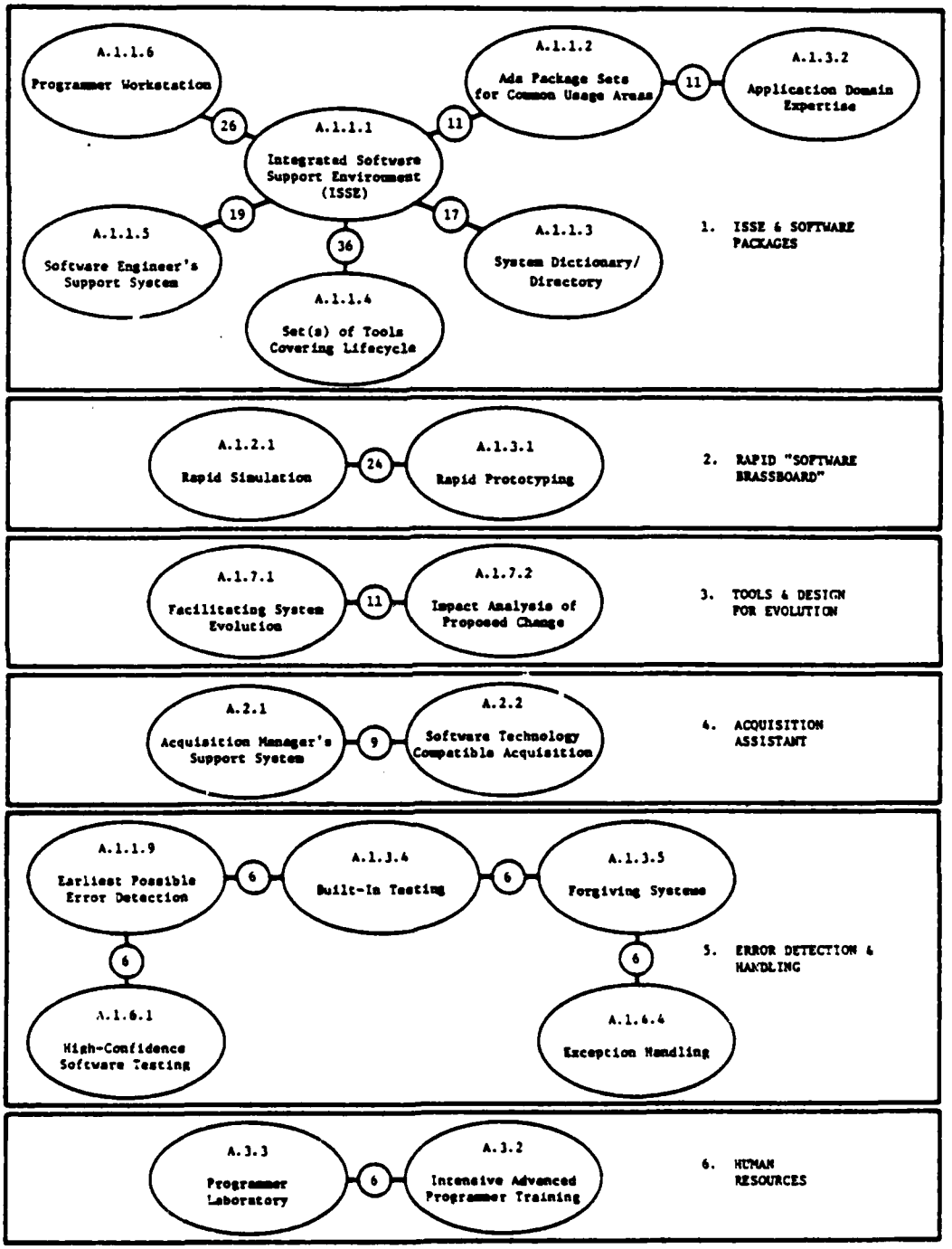



**FIGURE 1**
**THRUST CLUSTERS**

TABLE V

RANKING OF THRUST CLUSTERS

| | Cluster | All DoD | | Non DoD | | All | | Air Force | | Army | | Navy | | Misc DoD | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1. | ISSE & Software Packages | 1 | 3.4 | 1 | 3.6 | 1 | 3.4 | 1 | 3.2 | 1 | 3.5 | 1 | 3.5 | 6 | 3.5 |
| 5. | Error Detection & Handling | 3 | 3.3 | 5 | 3.6 | 3 | 3.3 | 5 | 3.2 | 4 | 3.3 | 4 | 3.3 | 1 | 3.7 |
| 3. | Tools & Design for Evolution | 7 | 3.5 | 9 | 3.4 | 10 | 3.5 | 8 | 3.4 | 14 | 3.4 | 5 | 3.6 | 11 | 3.6 |
| 2. | Rapid Software Brassboard | 12 | 3.3 | 7 | 3.4 | 9 | 3.3 | 11 | 3.2 | 11 | 3.4 | 12 | 3.4 | 16 | 3.4 |
| 6. | Human Resources | 14 | 3.0 | 18 | 3.1 | 15 | 3.0 | 21 | 2.8 | 15 | 3.1 | 13 | 2.9 | 9 | 3.6 |
| 4. | Acquisition Assistant | 16 | 3.2 | 28 | 2.8 | 21 | 3.1 | 3 | 3.4 | 19 | 3.2 | 16 | 3.0 | 26 | 3.0 |



Notes: - Cluster #1 was suggested the most often, cluster #6 the least often, etc.

- Each cluster is ranked by the highest-ranking thrust within that cluster.

- Each cluster score is the average of the scores of its constituent thrusts.

- ISSE = Integrated Software Support Environment

"miscellaneous DoD," where the top thrusts have virtually no difference in worth scores.

- Most clusters have approximately the same position in all respondent groups except for cluster #4, "Acquisition Assistant," for which the need is felt most keenly by DoD (as opposed to non-DoD) personnel, and, especially, by the Air Force.

Respondents were also asked to read the "Other Ideas" section of the green book and then to recommend candidates from the section for inclusion in the list of recommended thrusts. The number of recommendations for each candidate and for some candidate areas are shown in Table VI. In addition, a number of respondents suggested totally new thrusts. These new thrusts, grouped into the categories used in the green book, are presented in Table VII. It is interesting to note the emphasis on management- and personnel-related thrusts.

### 3.3 Summary

By far the "Integrated Software Support Environment and Software Packages" cluster (cluster #1) is the most popular. It contains the original ISSE thrust (1.1.1) along with the supporting thrusts "Ada* Package Sets for Common Usage Areas" (1.1.2), "System Dictionary Directory" (1.1.3), "Set(s) of Tools Covering Entire Lifecycle" (1.1.4), "Software Engineer's Support System" (1.1.5), and "Programmer Workstation" (1.1.6). The "Ada Package Sets..." thrusts also has "Application Domain Expertise" (1.3.2) associated with it as a support thrust. In addition, a relatively large number of respondents suggested the addition of associated thrusts from the Other Ideas section; i.e., "Publication of Standard Designs" (B.1.3.5), "Reusable Software" (B.1.4.7), and "Very High Level Languages" (B.1.3.2).

---

*Ada is a Registered Trademark of the Ada Joint Program Office -- U.S. Government.

## TABLE VI

## SUGGESTED THRUSTS FROM OTHER IDEAS SECTION

| Thrust Area | # of times suggested |
|---|---|
| B.1 Technical | |
| B.1.1 General | 1 |
| B.1.1.1 Presentation and Manipulation | 6 |
| B.1.1.2 Rigorous Documentation | 10 |
| B.1.1.3 Conflict Recognition Among Representations | 4 |
| B.1.1.4 Exploratory Systems Applications of VHSIC | 4 |
| B.1.1.5 Military Information Utility | 3 |
| B.1.1.6 Multiple Classes of Service | 1 |
| B.1.1.7 Standard Real-Time Operating System | 7 |
| B.1.2 Requirements | 3 |
| B.1.2.1 Rapid Derivation of Requirements | 4 |
| B.1.2.2 Transform Informal to Formal Requirements | 5 |
| B.1.2.3 Requirements Languages Translation | 2 |
| B.1.2.4 Weakest Possible Requirements Description | 3 |
| B.1.3 Design | |
| B.1.3.1 Derivation of Software from Specifications | 7 |
| B.1.3.2 Very High Level Languages | 14 |
| B.1.3.3 Component Tailoring and Interfacing | |
| B.1.3.4 Publication of Standard Designs | 11 |
| B.1.3.5 Data Structure and Abstraction | 8 |
| B.1.4 Programming | |
| B.1.4.1 Code Skeletons | 3 |
| B.1.4.2 Graph-Oriented Language | 4 |
| B.1.4.3 Generating Assertions from Requirements | 2 |
| B.1.4.4 Transform to Satisfy Physical Constraints | 1 |
| B.1.4.5 Man-Machine Quality Improvement Team | |
| B.1.4.6 Application Generators | 4 |
| B.1.4.7 Reusable Software | 10 |
| B.1.4.8 Actor Languages | 3 |
| B.1.5 Testing | |
| B.1.5.1 Static Analysis of Software | 7 |
| B.1.5.2 Generating Test Data from Requirements | 10 |
| B.1.5.3 Generating Test Data to Violate Assertions | 2 |
| B.1.5.4 Testbed Facilities | 5 |
| B.1.6 Operations | |
| B.1.6.1 Construction for Future Evolution | 3 |
| B.1.6.2 Modification of Large Systems | 4 |
| B.2 Managerial | |
| B.2.1 General | |
| B.2.1.1 Model Contracts for Buying Software | 2 |
| B.2.1.2 Maximizing DoD Rights to Software | 4 |
| B.2.1.3 Multiplying Expert Effectiveness | 1 |
| B.2.2 Conception/Feasibility | |
| B.2.2.1 Quick Look Feasibility/Evaluation | |
| B.3 Continuity-Related | |
| B.3.1 General | |
| B.3.1.1 Completely Captured Software | |
| B.3.1.2 Multi-person Machine Mediated Programming | |
| B.3.1.3 Totally Visible Software | 1 |
| B.3.1.4 Systems that Never Forget | |

TABLE VII

THRUSTS SUGGESTED BY RESPONDENTS

TECHNICAL

Software/Hardware synergy: support of advanced software structures; data base machines; software test; network protocol interfacing

Impact of VLSI on software

Firmware engineering tools

Data base technology: large, efficient relational data bases; distributed data base partitioning and management; other thrust areas from DARPA Very Large Data Base work

Distributed data processing instrumentation and test techniques

Comparison of different languages

Empirical investigation of methods for man:machine communications

Computer security

MANAGEMENT

Top-down redesign of entire software engineering process

Methodology for contracting for and enforcing software quality

Establishment of government-controlled test-beds to accredit software packages, maintain configuration control, and test and verify changes

Advanced technology test beds

Re-evaluation of needs for standards and the best ways of applying them

Methods for encouraging technology insertion

Contractor incentives toward technology utilization and evaluation

Methodology for estimating resources required for software development

Design of a software management information system

Software cost estimation

TABLE VII (Concluded)

THRUSTS SUGGESTED BY RESPONDENTS

Techniques for assessing the value of new or changed applications

Decision support system research

Development of methodology for system requirements definition

PERSONNEL-RELATED

Improved understanding of programmer psychology/training/motivation

Human factors: software project organization and group dynamics; motivational factors influencing productivity

Personnel management techniques

Improvement of "people environment" — better training, better career paths, less incentive for personnel to leave, in-depth mid-career or renewal/sabattical program

Improve Office of Personnel Management policy, procedures, and standards for GS-334 computer personnel

Provide a career path for officers in ADP area

Improved techniques for technology insertion

Standardized approach to training — users and technicians share common training foundation and terminology

Management training about computers

Cross-training of software and weapon system specialists

CONTINUITY-RELATED

Automated technical writing and editing assistance

Program testing was also an area of high interest. The cluster thrust "Error Detection and Handling" (#5) did very well, as did its components, especially "Earliest Possible Error Detection" (1.1.9) and "High-Confidence Software Testing" (1.6.1). Again, a number of thrusts from the Other Ideas section were suggested in this area (B.1.5).

A number of supporting areas also appeared. Very high interest was shown in the area of "Distributed Functions and Resources" (1.4.5), "Suitable Communication Interconnection" (1.4.6), and "Useful Measures of Software Quality" (1.1.7).

Finally, many respondents saw the need for thrusts in the areas of personnel development, project management, and technology insertion. "Technology Insertion in Software Area" (2.3) was ranked highly, as was "Intensive Advanced Programmer Training" (3.2). Many respondents suggested new thrusts in these areas. A number of thrusts in personnel management and the human programming environment were suggested, along with recommendations for the development of better management tools.

# APPENDIX VIII

## ESTIMATED SOFTWARE PRODUCTIVITY GAINS

# APPENDIX VIII

## ESTIMATED SOFTWARE PRODUCTIVITY GAINS

### 1.0 SUMMARY

This Appendix uses cost estimating relationships (CER's) and DoD software project data from the COCOMO model and data base (Reference 1) to estimate the productivity benefits which would result from the development and use of the automated integrated software environment described in this report. Two methods of calculating these benefits are used:

1. An estimate of software process efficiency gains based on improvements in DoD's software cost driver attributes such as the use of software tools, interactive software development, and reduced hardware constraints.

2. An estimate of software process efficiency gains based on an analysis of the effort savings for each phase and activity within the software life-cycle.

When these process-efficiency gains are combined with complementary effort-avoidance gains due to re-tooling avoidance, reduced requirements volatility, and re-use of existing software, each estimation method results in an overall estimated productivity gain of roughly a factor of 4 by 1990. (4.34 via the cost-driver analysis; 3.93 via the activity-savings analysis). These estimates may even be conservative, as they do not include some of the more subjectively-based productivity factors such as staffing and personal motivation.

The sections below describe the COCOMO model, estimate the process efficiency gains based on the cost-driver method and the activity analysis method, and estimate the overall productivity gains achievable by 1990 via the use of the integrated software environment.

## 2.0 THE CONSTRUCTIVE COST MODEL (COCOMO)

The Constructive Cost Model (COCOMO) is a new model for software cost estimation based on a carefully screened sample of 63 projects representing business, industry, government, and commercial software-house organizations. It estimates the cost of a proposed software product in the following way:

1. A nominal development effort is estimated as a function of the product's size in Delivered Source Instructions in thousands (KDSI).

2. A set of effort multipliers are determined from the product's ratings on a set of 15 cost driver attributes.

3. The estimated development effort is obtained by multiplying the nominal effort estimate by all of the product's effort multipliers.

4. Additional factors can be used to determine dollar costs, computer costs, annual maintenance costs, and other cost elements from the development effort estimate.

### Nominal Effort Estimation

The COCOMO nominal effort estimator for DoD embedded software development takes the form

$$MM = 2.8\ (KDSI)^{1.20}$$

where MM is the nominal number of man-months required to go from a reasonably-complete software requirements specification to a software acceptance test. Other formulas with lower exponents cover less-constrained forms of software such as business data processing.

### Effort Multipliers

Each of the cost driver attributes in COCOMO has a rating scale and a set of effort multipliers which indicate by how much the nominal effort estimate must be multiplied to account for the project's having to work at its rating level for the attribute.

For example, Figure 1 shows the rating scale for the "Use of Software Tools" attribute. A project with a Very Low tools rating will require 1.24 times the nominal project effort to complete; a project with a Very High rating will require only 0.83 times the nominal project effort to complete.

These rating levels and effort multipliers can then be used to estimate the reduction in effort resulting from an investment in software tools. For example, if DoD were able to improve its average Tools rating from Nominal to Very High, it could reduce its average software effort by a factor of 0.83.

COCOMO Model Fidelity

These cost driver attribute ratings and effort multipliers explain a great deal of the variation in productivity encountered on software projects. Figure 2 shows the degree of agreement between the COCOMO estimates and the actual man-months on the 63-project sample. More details on the COCOMO model are given in Reference 1.

Productivity Ranges

A useful indicator of the relative productivity leverage of a software cost driver is its Productivity Range: the ratio of the highest to lowest effort multipliers for that cost driver in the COCOMO model. For example, the Productivity Range for the Tools factor is 1.24/0.83 = 1.49. These Productivity Ranges provide us a means of identifying the high-payoff areas to emphasize in a software productivity improvement activity. For example, we can see that the level of Personnel/Team Capability far outweighs any of the other factors influencing software productivity, while Language Experience provides only a small amount of productivity leverage.

Effort Distribution by Phase and Activity

The COCOMO model also includes CER's for a software project's distribution of effort by phase and activity. These indicate, for example, that roughly 31.5% of a typical DoD project effort is

# SOFTWARE TOOL LEVELS

**VERY LOW (BASIC MICRO)** - Effort Multiplier EM = 1.24

- Assembler
- Basic Linker
- Basic Monitor
- Batch Debug Aids

**LOW (BASIC MINI)**

- HOL Compiler
- Macro Assembler
- Simple Overlay Linker
- Language-Independent Monitor
- Batch Source Editor
- Basic Library Aids
- Basic Data Base Aids

**NOMINAL (STRONG MINI/BASIC MAXI)** - EM = 1.10

- Real-Time/Timeshare OS
- Data Base Management System
- Extended Overlay Linker
- Interactive Debug Aids
- Simple Programming Support Library
- Interactive Source Editor

**HIGH (STRONG MAXI/STONEMAN MAPSE)** - EM = 1.00

- Virtual Memory OS
- Data Base Design Aid
- Simple Program Design Language
- Performance Measurement/Analysis
- Programming Support Library — Configuration Management Aids
- Set/Use Analyzer
- Program Flow/Test-Case Analyzer
- Basic Text Editor/Manager

**VERY HIGH (ADVANCED MAXI/STONEMAN APSE)** - EM = 0.83

- Full Program Support Library/CM Aids
- Full, Integrated Documentation System
- Project Control System
- Reqts. Spec/Analysis Tools
- Automated Verification System
- Special Purpose Tools: Crosscompilers, Instruction Set Simulators, Display Formatters, Comm. Processing Tools, Data Entry Control Tools, Conversion Aids, etc.

FIGURE 1. USE OF SOFTWARE TOOLS

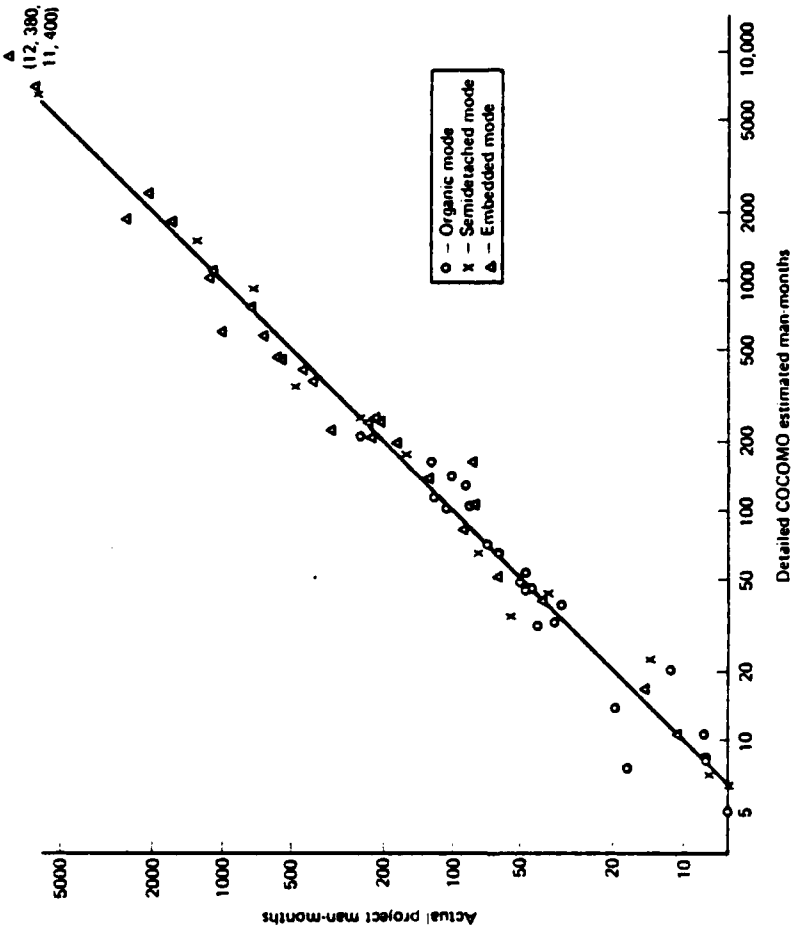



FIGURE 2. COCOMO MODEL ESTIMATES VS PROJECT ACTUALS

# COMPARATIVE SOFTWARE PRODUCTIVITY RANGES

## (BASED ON ANALYSIS OF 63 SOFTWARE PROJECTS)

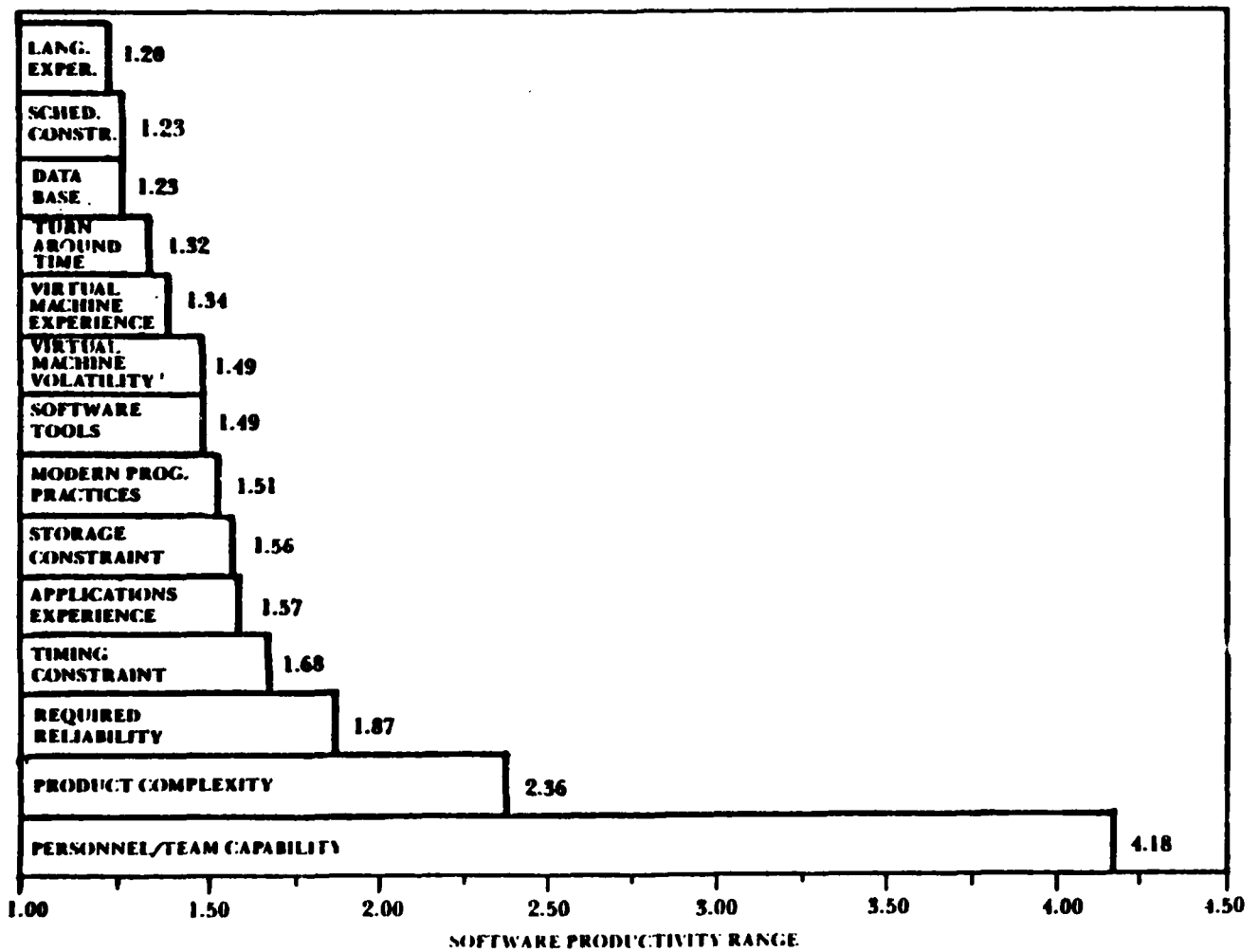



FIGURE 3. COCOMO SOFTWARE COST DRIVER PRODUCTIVITY RANGES

expended during the Integration and Test phase, and that roughly 48% of that effort is devoted to Programming activities (performing integration, fixing residual errors, etc.). If we estimate that the software initiative integrated software environment can save 60% of this effort, we would have an estimated savings for this phase and activity of

(31.5%) (0.48) (0.60) = 9.1%

of the project's total effort. By performing these estimates and calculations for the other phases and activities, and summing the results, we can obtain an overall estimate of the effort reduction or process efficiency gain for a typical DoD project by 1990.

## 3.0 EFFICIENCY GAINS: COST-DRIVER ANALYSIS

### Effort Multiplier Calculations

Table 1 shows the software process efficiency gains estimated with the use of The COCOMO cost drivers and the data on past DoD software projects from the COCOMO data base. Only the cost drivers providing the most significant and objectively-rated process efficiency gains have been used here: use of software tools, modern programming practices, programming language and software environment (virtual machine) experience, hardware speed and storage constraints, and use of interactive software development.

The column headed "1976" in Table 1 presents the average effort multipliers observed for 18 DoD software projects in the COCOMO data base (Ref. 1, Chapter 29), for each of the cost drivers above: the average development date for these projects was 1976. Thus, the average tools effort multiplier for these projects was 1.05, roughly halfway between the Low and Nominal ratings in Figure 1.

The column haded "1982" in Table 1 estimates the corresponding effort multipliers for an average DoD project starting in 1982. With respect to tools, the reduction from 1.05 to 1.02 results from the

TABLE 1. PROCESS EFFICIENCY GAINS: EFFORT MULTIPLIER ANALYSIS

| COST DRIVER | DoD AVERAGE EFFORT MULTIPLIERS | | |
|---|---|---|---|
| | 1976 | 1982 | 1990 |
| • Use of Software Tools | 1.05 | 1.02 | 0.85 |
| • Use of Modern Programming Practices | 0.98 | 0.95 | 0.85 |
| • Programming Language Experience | 1.03 | 1.02 | 0.98 |
| • Software Environment (Virtual Machine) Experience | 1.045 | 1.03 | 0.95 |
| • Computer Execution Time Constraint | 1.25 | 1.18 | 1.11 |
| • Computer Storage Constraint | 1.22 | 1.15 | 1.06 |
| • Computer Turnaround Time | 1.03 | 1.01 | 0.90 |
| Overall Multiplier | 1.74 | 1.40 | 0.71 |
| Process Efficiency Gain | | 1.24 | 1.97 |

estimated net effect of two trends: a general improvement in software tools, and increase in the amount of software developed on minicomputers and microprocessors with weaker tool support.

The column headed "1990" estimates the corresponding effort multipliers for an average DoD project starting in 1990, using a mature integrated software environment resulting from the STI. With respect to tools, the 0.85 effort multiplier is somewhat conservative compared to the 0.83 achievable for a very high tools rating.

Sources of Process Efficiency Gains

A short explanation of the sources of process efficiency gains contributed by each of the cost drivers is given below. Detailed explanations for each cost driver are given in Reference 1.

- Use of Software Tools. The major savings come from the elimination of tedious, repetitive, manual tasks which currently consume a great deal of the software worker's time.

- Use of Modern Programming Practices. Major savings result from eliminating or reducting the cost of expensive software fixes later in the life cycle by performing the earlier phases in thorough, well-structured ways. Both the software initiative tools effort (including Ada) and educational efforts reinforce this.

- Programming Language and Software Environment Experience. The current proliferation of programming languages (including MOL's) and environments is the source of considerable lost effort due to retraining, misunderstandings, and lack of awareness of effort-saving features. A strong, stable, consistent, well-instituted Ada/APSE environment would save much of this lost effort.

- Computer Time and Storage Constraints. Even in the 1990's a good deal of software will be developed for capacity-limited embedded microprocessors. The development of cross-compilers and a source-machine/target machine mode of operation for the Ada/APSE environment will mean that software development and maintenance for these applications can be carried out on a high-capacity source machine, unconstrained by the limitations of the target machine.

- o Computer Turnaround Time. A great deal of DoD software is still developed in batch mode. Going interactive not only eliminates wait time and provides more computer shots per workday; it also stimulates more creative and intensive design, experimental prototyping, debugging, electronic mail-based group coordination, and software modification.

Process Efficiency Gains

The overall multiplier determined for each column is obtained by multiplying together the individual effort multiphase in the column. It is an index of the relative efficiency of the DoD software process in 1976, in 1982, and with a mature software initiative environment in 1990: i.e., it is proportional to the number of man-months that the COCOMO model would estimate to complete a given software project in the 1976, 1982, and 1990 environments.

The calculations indicate that the process efficiency gain between 1976 and 1982 was 1.74/1.40 = 1.24, for an average gain of about 4% per year over 6 years, largely due to reduced hardware constraints. Between 1982 and 1990, the estimated process efficiency gain achievable by developing and using the STI integrated environment is 1.40/0.71 = 1.97. As we shall see in Section 5.0, this process efficiency gain, when combined with several sources of software effort avoidance, will yield an overall estimated productivity gain of a factor of 4 due to the software initiative environment.

## 4.0 EFFICIENCY GAINS: ACTIVITY ANALYSIS

Table 2 shows the software process efficiency gains estimated with the use of the COCOMO model's distributions of software effort by phase and activity. The Phase Effort ($P_i$) row shows the percent of the overall development effort consumed during phase i.

For each phase, the AE column shows the COCOMO effort distribution by activity, including an activity distribution for the maintenance phase. For example, Table 2 indicates that 7.4% of the

TABLE 2. ACTIVITY - ORIENTED ESTIMATES OF EFFORT REDUCTION

| | PLANS AND REQUIREMENTS | PRODUCT DESIGN | PROGRAMMING | INTEGRATION AND TEST | DEVEL-OPMENT TOTAL | MAINTENANCE |
|---|---|---|---|---|---|---|
| Phase Effort Pi (%) | 7.4 | 16.7 | 44.4 | 31.5 | | 100.0 |
| Activity | AE, AS, ES | AE, AS, ES | AE, AS, ES | AE, AS, ES | ES | AE, AS, ES |
| Requirements Analysis | 0.42 0.25 0.8 | 0.10 0.50 0.8 | 0.03 0.50 0.07 | 0.02 0.50 0.3 | 2.6 | 0.05 0.40 2.0 |
| Product Design | 0.16 0.15 0.2 | 0.42 0.30 2.1 | 0.06 0.50 1.3 | 0.04 0.50 0.6 | 4.2 | 0.11 0.30 3.3 |
| Programming | 0.10 0.10 0.1 | 0.14 0.50 1.2 | 0.55 0.25 6.1 | 0.48 0.80 9.1 | 16.5 | 0.41 0.50 20.5 |
| Test Planning | 0.06 0.25 0.1 | 0.08 0.30 0.4 | 0.08 0.30 1.1 | 0.05 0.25 0.4 | 2.0 | 0.06 0.25 0.1 |
| V & V | 0.10 0.20 0.1 | 0.10 0.40 0.7 | 0.12 0.50 2.7 | 0.20 0.55 3.5 | 7.0 | 0.14 0.55 0 |
| Project Office | 0.08 0.30 0.2 | 0.07 025 0.3 | 0.05 0.20 0.4 | 0.6 0.20 0.4 | 1.3 | 0.06 0.20 1.5 |
| CM/QA | 0.03 0.36 0.1 | 0.02 0.30 0.1 | 0.06 0.55 1.4 | 0.08 0.55 1.4 | 3.1 | 0.06 0.50 3.0 |
| Manuals | 0.05 0.55 0.2 | 0.07 0.45 0.5 | 0.05 0.45 1.0 | 0.07 0.45 1.0 | 2.7 | 0.11 0.55 6.0 |
| TOTAL | 1.8 | 6.1 | 14.8 | 16.7 | 39.4 | 46.1 |

development effort is devoted to the Plans and Requirements phase. Within this 7.4%, Requirements Analysis activities consume 42%, Product Design activities consume 16% of the effort, etc.

The AS (Activity Savings) column shows the results of a Delphi exercise to estimate the percentage of effort for each phase-activity combination that would be saved as a result of using the environment. For example, 25% of the Requirements Analysis effort was estimated to be saved during the Plans and Requirements phase, etc.

The ES column shows the resulting Effort Savings for each activity j within phase i:

$$ES_{ij} = (P_i)(EAE_{ij})(AS_{ij})$$

When summed over all phases and activities, the overall results show a development savings of 39%, and a maintenance savings of 46%, exclusive of any savings due to software results. These savings are not as great as those estimated by the cost-driver approach, but they are reasonably comparable and quite significant.

## 5.0 OVERALL ESTIMATED PRODUCTIVITY GAINS

Table 3 combines the process efficiency gains estimated by the cost-driver analysis and the activity-savings analysis with estimates of three additional sources of software savings due to effort avoidance, to produce overall estimates of DoD software productivity gains achievable via the STI integrated software environment. The three sources of effort-avoidance savings are:

1. Reduced Requirements Volatility: the amount of added effort are to changes in requirements during the development or upgrade period. The 18 DoD projects in the COCOMO data base had an average effort multiplier of 1.17 due to requirements volatility, corresponding to the year 1976. Table 3 indicates that the situation has not changed much in 1982, but that better techniques of rapid prototyping, requirements validation, and incremental development could reduce the effort multiplier from 1.15 to 1.00 between 1982 and 1990.

TABLE 3. ESTIMATED STI PRODUCTIVITY GAINS

| FACTOR | DoD AVERAGE RELATIVE EFFORT | | | |
|---|---|---|---|---|
| | COST-DRIVER ANALYSIS | | ACTIVITY-SAVINES ANALYSIS | |
| | 1982 | 1990 | 1982 | 1990 |
| o Process Efficiency | 1.40 | 0.71 | 1.00 | 0.56 |
| o Requirements Volatility | 1.15 | 1.00 | 1.15 | 1.00 |
| o Retooling | 1.06 | 1.00 | 1.06 | 1.00 |
| o Software Re-use | 0.90 | 0.50 | 0.90 | 0.50 |
| Relative Effort | 1.54 | 0.355 | 1.10 | 0.28 |
| Productivity Gain | | 4.34 | | 3.93 |

2. Retooling Avoidance. The lack of a generally usable, portable tool set usually means that a DoD project will currently spend an extra 6% effort to rehost or rebuild tools just to get back to the level of previous projects.

3. Software Re-Use. The 1976-average effort reduction due to software re-use for the 18 DoD projects in the COCOMO data base was a factor of 0.93. With the reusable Ada applications and support packages resulting from the software initiative, it is estimated that the re-use effort reduction will be improved from 0.90 in 1982 to 0.50 in 1990.

The potential gain for software maintenance may be overstated somewhat, as these three factors apply less to maintenance than to development. On the other hand, the lower COCOMO maintenance multipliers for modern programming practices partly compensate for this effect.

The 0.56 multiplier for Process Efficiency in the activity-savings analysis is derived as follows: the life-cycle effort savings is a weighted average in the ratio 30:70 of the development savings and the maintenance savings, or

$$0.30\ (39\%) + 0.70\ (46\%) = 44\%.$$

This 44% savings corresponds to a relative effort multiplier of 0.56.

As seen in Table 3, both methods estimate an overall productivity gain of roughly a factor of 4:

- Cost-Driver Analysis: $1.54/0.355 = 4.34$
- Activity-Savings Analysis: $1.10/0.28 = 3.93$.

## 6.0 REFERENCES

1. Boehm, Barry W., Software Engineering Economics, Prentice-Hall, 1981.