03 53

BRMC-84-5114

ADA 159374

USE OF A SOFTWARE DEVELOPMENT AND SUPPORT ENVIRONMENT AS GOVERNMENT-FURNISHED EQUIPMENT (GFE)

Technion International, Inc.
201 Webster Building, Concord Plaza
3411 Siverside Road
P.O. Box 417
Wilmington, Dealware 19899

28 June 1985

Final Report for Period Covering 1 October 1984 - 28 June 1985
Contract No. F33615-84-C-5114

Prepared for
AIR FORCE BUSINESS RESEARCH MANAGEMENT CENTER
Wright-Patterson AFB Ohio 45433

ACKNOWLEDGEMENTS

The authors are indebted to a number of people for their help on this study. Ken Nidiffer of AFSC was a constant source of knowledge and inspiration. In a different sense, Ron Deep, Jim Weber and Ed Mitchell of AFBRMC made it all possible through their support and funding.

At SASC, the help of Ed Barr and Joe Clema was indispensable during the early stages. The graphics department, particularly Laura Geissler, Vicki Hodges, and Joan Sargies, met our many challenges for artwork that would help clarify intangible concepts.

Dick Werling and Al Chande have special admiration for Ray Houghton's depth of professional knowledge, and for his skill in charting the course through the intricacies of software tools.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ILLUSTRATIONS, continued

LIST OF TABLES

GLOSSARY OF ACRONYMS AND TECHNICAL TERMS

Ada

A standard programming language used in new DoD systems (ANSI/MIL-STD-1815A). Ada is a registered trademark of the U. S. Government, Ada Joint Program Office.

ALS

The U. S. Army's Ada Language System.

APSE

Ada Programming Support Environment.

Contractor

Developer of systems, including the software required for their operation. Often, contractors also enhance and maintain systems and software after the initial development and through the systems' in-service (post-deployment) life.

Environment

A framework for integrating sets of methods, proedures, and computer programs (computerized software tools), to support the entire software life cycle.

GFE/Environment

As used in this report, a standard software development and support environment made available to contractors as government furnished equipment.

HAPSE

Hypothetical Ada Programming Support Environment, postulated during this project. The HAPSE is conceptualized as an "ideal type" to be used for investigating productivity across the life cycle and for comparison with actual environments as they are developed. The HAPSE definition includes four levels of capability.

The first level contains a basic set of software tools. The second set adds to the basic set those tools judged most necessary by software development managers. The third level contains those plus additional tools judged valuable. Finally, the fourth level contains all the above plus additonal tools judged useful but not critical.

Life Cycle

As generally used, "the period of time from the perception of need for a software system to its retirement. In this report a distinction is made between the life cycle of defense systems and the life cycle of the software components for them. (For details of this new insight, see chapter 2, section B2).

GLOSSARY, continued

MCCR

Mission-Critical Computer Resources.

Methodology

As used in this report, a general collection of rules, methods, and philosophies supporting software life cycle activities.

Method

As used in this report, a set of specific rules, guidelines and techniques supporting software life cycle activities.

Post-Deployment Support

Support of software after its initial deployment. During the total life cycle of a system containing software, most so-called "maintenance" is done to enhance performance of the system in which the software is embedded, by meeting new requirements or adapting to changes in other system components.

Productivity

As used here, the average number of delivered source instructions per staff work-month. Includes both freshly written and reusable code components.

Reliability

The probability that software will not cause the failure of a system for a specified time under specified conditions.

Reusable Code

Standard proven fragments of code that can be reused, and hence need not be rewritten. Reusable code provides both improved reliability and maintainability and increased productivity.

System Life Cycle

As used in this report, the Life Cycle of a defense system that contains MCCR. Includes hardware, software, facilities, policy and human elements. For contrast, see Software.

Software Life Cycle

As used in this report, the life cycle of the software components of a defense system. Includes both initial development and post-deployment support of MCCR software. For contrast, see System.

UNIX

An operating system, developed by Bell Laboratories. It also has many software tools.

EXECUTIVE SUMMARY

A. BACKGROUND

1. Purpose of Research

The research was begun to answer this question:

> Can and should the United States Air Force (USAF) build and supply to contractors as Government Furnished Equipment (GFE) a Standard Government-Owned Environment (GFE/Environment) for their use in developing Ada*-based Computer Software for Mission-Critical Systems?

2. Methodology

The research was conducted as three tasks, which built upon one another. The tasks were to identify, define and document:

Task 1. what an integrated automated software development/support environment would consist of; [see chapter 3]

Task 2. what tools and methods are available and what needs to be developed; [see chapter 3]

Task 3. what the pros and cons are of developing a standard environment to be provided as Government Furnished Property (GFE). [see chapters 4 and 5].

3. Results

Results of the research make it clear that:

1. The annual costs of maintaining the existing massive inventory of non-standard computer programs (projected by JLC to be $5 billion for FY 1990) may justify heroic measures, such as imposing a standard environment as GFE [JLC84].

2. The USAF can build an integrated, automated environment with today's technology.

* Ada is a registered trademark of the U. S. Government, Ada Joint Program Office (AJPO).

3. An extensive amount of research has been completed and more than 400 software tools have been identified. At least one software tool (usually written in the FORTRAN, not in the Ada, language) exists to support each function needed during the life cycle of software for mission-critical systems.

4. Use of a standard development/support environment would improve the post-deployment maintenance and enhancement of mission-critical system software.

5. The standard environment must be designed to accommodate future changes in software modules, user interface, and methodology. That is, the functional capabilities built into the environment must be designed for frequent change during the next 20 years, or throughout the entire life cycle of the environment.

6. The pre-software development work and the post-deployment support phases demand the most resources and time, and exert powerful effects on software development and support throughout the system life cycle. Not only must the environment support the pre- and post-coding work, defense project planning should be conducted with the extended definitions in mind.

4. Sources of Benefits to USAF

Based on our research, we believe that use of standard integrated software support environments would bring USAF these benefits:

1. Reduction in the number of unique environments now supported by the Air Force. Each weapon system contractor supplies a unique environment tailored to the system supported. Currently, the Air Force is supporting about 400 different languages and dialects, and several dozen unique environments [Ichb84]. Other benefits include reduction of training costs and costs of contractor lock-ins.

2. Increases in productivity of staff who actually use the standard environment [Boeh84]. Concentration on a few environments per unit provide opportunities for focused technology investments, which lead to greater productivity. (For details, see chapter 2).

3. Increased reliability and maintainability of the software produced by the standard environment, which lead to lower maintenance costs. (See chapter 4).

B. ROOTS OF THE CURRENT PROBLEM

How did the Air Force accumulate its massive inventory of non-standard software? Although there are many sources, they can be traced to two roots.

1. Exponential growth in capabilities of technology

The first is analogous to aiming at a target that moves rapidly and takes protean evasive actions, such as literally changing its shape. Computer hardware and software have changed their technology, roughly every five years, throughout the past three decades [Werl83].

The incessant change in technology has required managerial responses to technologies that constantly evolved. This led to many decisions to optimize development of individual mission-critical subsystems, each of which depended heavily on technologies that were improving exponentially. During the past 25 years the performance of new computer-based hardware products has doubled every four years and the cost has decreased by the same amount (i.e., at the combined rate of about 3Ø percent every year) [Werl83]. This means that each of today's hardware items, which may have been near state of the art when it was developed, was soon overtaken by technological evolution. A comparable but less dramatic pattern occurred for software [Phis79].

In short, with the increasing likelihood that software performs tasks formerly done by hardware [Fox82], the support technology has slipped farther and farther behind the technological and economic state of the art.

2. Incomplete view of life cycle

The second root is found in the incomplete view of the software life cycle that prevailed during most of the past 25 years. This view, reinforced by required contracting practices, saw completion of the initial development of software as an end in itself. Subsequent changes and enhancements to the delivered software were seen as beyond the scope of the initial development process. This conception might have been valid for products bought "off the shelf", but was misleading for development of software components of mission-critical systems. Software products were not only custom built, but evolved throughout the initial development stage and then throughout the subsequent years of the system life cycle.

A 1984 study [IDA84a] showed that it takes an average of 14 years to develop an innovative system, such as those on which defense systems are based. Because support software is developed early in that period, when the system is delivered the support software is old with respect to state-of-the-art technology. The application software usually needs to be updated within a short time after delivery of the defense system. Other factors that complicate the post-deployment phase of the life cycle also include poor documentation and low functional capability of the delivered support system.

The USAF World Wide Military Command and Control System (WWMCCS) Improvement Program (WIS) is attempting to correct this situation. The support software is being designed to accommodate updates in the post-deployment support phase. The support software is being designed to be portable among various hardware systems. A large portable WIS support environment will be standard with respect to all its sites. It is not clear that the WIS acquisition strategy should not be applied throughout USAF. The system engineering function for the WIS C^3I mission area may be different from other mission areas such as mission simulators, missiles, space, and avionics. Therefore, it may be necessary to have a small number of versions of the standard environments.

C. POTENTIAL SOLUTIONS

How have other organizations addressed problems associated with support of software? It is not, after all, a condition unique to the USAF.

1. Standardization

One key strategy used by industry has been to standardize, first on hardware and then on the software needed to use the hardware optimally during the developmental and later during the support phases. A corollary industrial strategy, made feasible by the practice of standardization within vendors' product lines, is to increase computing capacity by upgrading equipment more frequently than is possible for Federal agencies [GAO81a]. This is possible because vendors design new hardware so that software uses the same "instruction set architecture" and so is "upward compatible". Software designed for the S/360 computers can be run on the later S/370 and 303X models, although the reverse may not be true.

2. Increasing Productivity

A second strategy is to increase productivity of software development and support people, by providing them with better tools. Figure S-1, "Programming Productivity Increases Exponentially", demonstrates that productivity increased for both "small" and "large" projects producing software of the sort embedded in mission-critical weapon systems. Some of the technological changes are indicated on the chart, in roughly the time periods in which they became (or will become) effective. (For details of the data base and results, see Chapter 4).

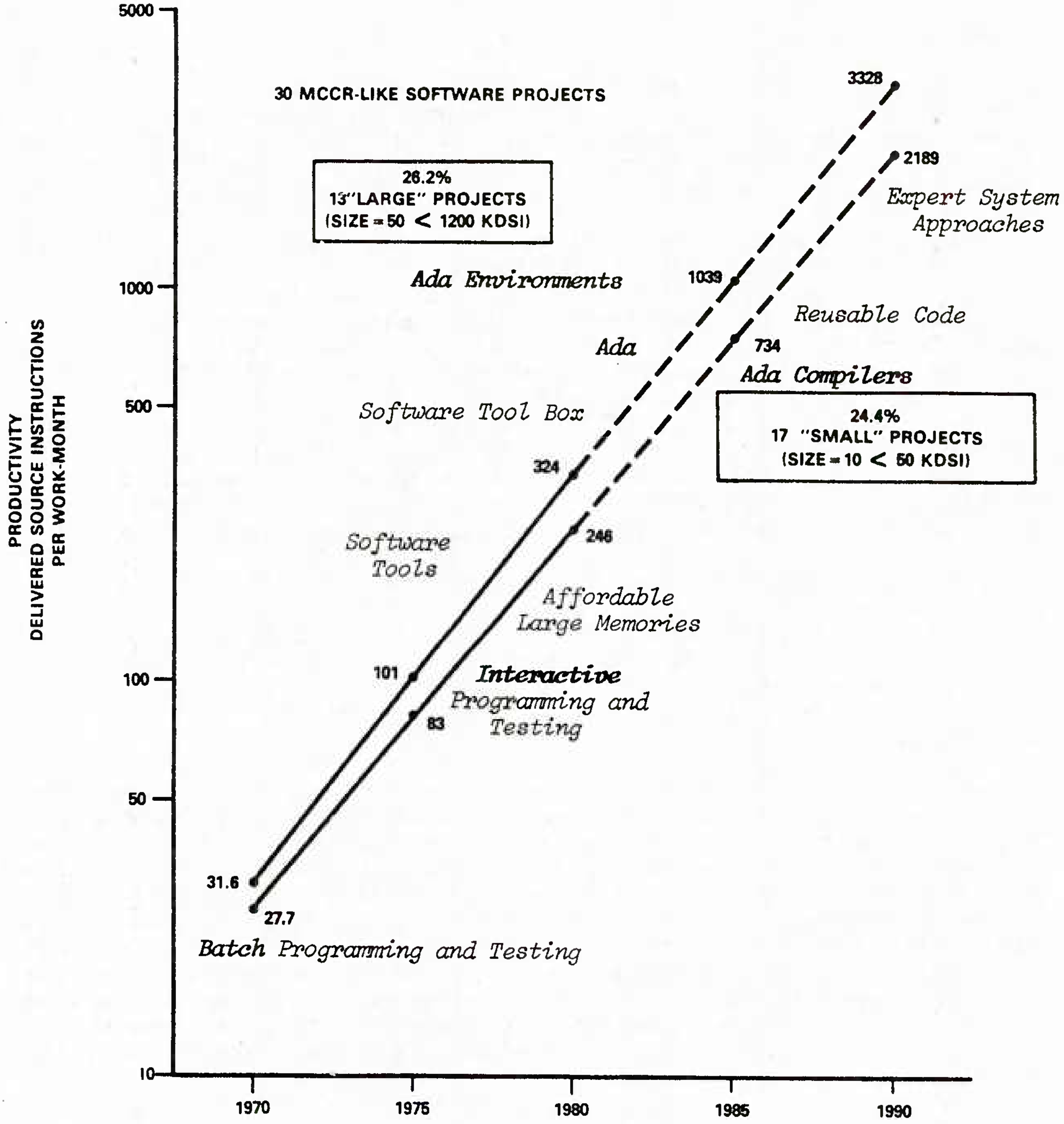


Figure S-1. Software Productivity Increases Exponentially.

The reader is advised that technical agreement has not been reached on metrics to measure productivity. These curves are "least squares semi-log" lines, derived from statistical analysis of the "COCOMO" project database of "well-managed" projects. The curves show delivered source instructions (DSI) per work-month (W-M), or DSI/W-M. (For additional detail, see chapter 4).

3. Benefits and Costs Extend Over Decades

The benefits cited above are substantial, but they are not free. In this research, we have identified both the general "pros" and "cons" of furnishing a standard environment as GFE, and identified the categories of costs that would be incurred in obtaining the benefits cited above. (See chapters 4 and 5). The need to estimate benefits and costs more accurately, for both quantitative and non-quantitative cost categories, led us to design a simple econometric model of the software development and support process (See chapter 4).

To identify these costs, we first defined a Hypothetical Ada Programming Support Environment ("HAPSE"). It includes four increasingly essential groups of software tools. The costs that must be considered extend beyond the costs of acquiring and supporting a HAPSE-like standard environment, and include costs incurred by variations in implementation in different services and with different control strategies.

D. SUMMARY OF RECOMMENDATIONS

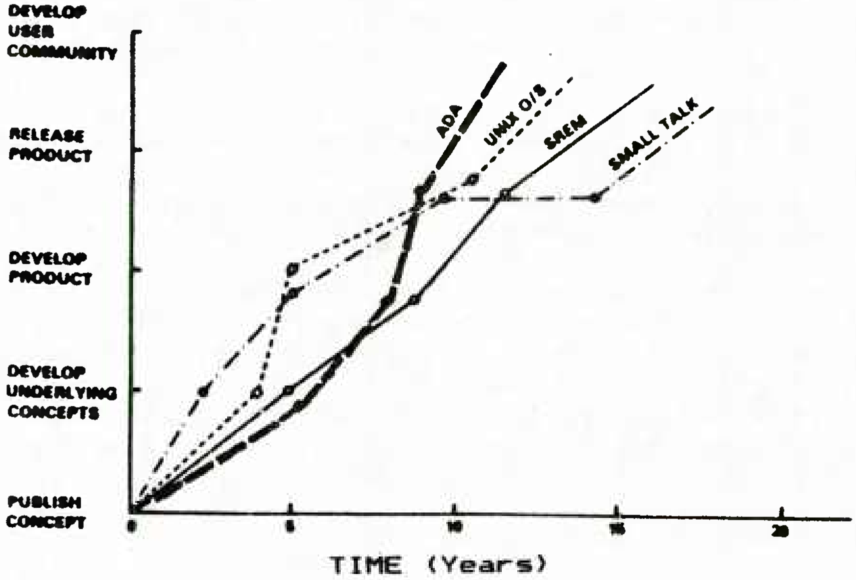
Six recommendations have emerged from this research.

1. Continue Ada and STARS programs

The USAF should continue its present actions to increase productivity of software development and post-deployment support for mission critical computer systems. The Ada language program addresses the DoD concern with excessive resources needed to cope with the more than 400 incompatible programming languages and dialects. The Software Technology for Adaptable and Reliable Systems (STARS) Program primarily addresses automated Software Engineering Environments (SEE). Each program will require several years of effort before yielding the massive potential benefits of which they are capable. However, progress is already visible.

Ada Acceptance "on schedule". Figure S-2, "Diffusion of Software Systems Takes Decades," shows that the progress of the Ada standard language program, though begun a decade ago, is comparable with similar software innovations. Such new software technologies typically require about 14 years to progress from concept to fielded status [IDA84a]. Ada compilers and support environments are beginning to appear, and promising signs of successful institutionalization of Ada are visible [Elec85].

Productivity Increasing. Software productivity (measured in lines of delivered source code per work-month) is growing dramatically for programming and unit-level testing, although the benefits of this improvement have not been fully reflected in improved costs to DoD. For example, this improvement has not yet been extended to the activities that precede and follow programming. (For possible reasons, see chapter 2).



SOURCE: IDA84a

Figure S-2. Diffusion of Software Systems Takes Decades.

Suboptimization caused by incomplete reporting of costs. Present federal accounting and reporting systems generally do not report all costs or permit determination of all benefits that bear on this type of issue. For example Federal accounting records, which do not depreciate investments in computer hardware or software, make it difficult to show the true net costs of one policy as compared with others [Werl83].

2. Pursue standard software development/support environment

The Air Force should continue planning for ways to increase software quality and improve the effectiveness of software production. One promising way is to integrate in one "environment" several automated software tools that aid in developing and supporting mission-critical software, then supplying such productivity enhancing tools to contractors, maintainers, and users as GFE. Such a GFE/ Environment is technically possible today. At least one software tool has been built to serve each function needed in the development and lifelong support of software.

The proposed strategy involves use of many existing public domain (or Air Force owned) software tools, and developing an "overlying HAPSE environment" that would integrate the tcols, use a common set of commands, and provide an effective user interface. (For details, see chapter 3). Two important potential benefits to USAF are improved: (1) portability of tools from one hardware system to another; and (2) interoperability, which allows transported tools to be integrated in the transported environment.

3. Quantify net benefits of GFE/Environment strategy

The Air Force should quantify the net benefit (in terms of software quality and effectiveness of software development and support) from a GFE/Environment strategy. This is a necessary and complex undertaking, with ramifications affecting policies, costs, and organizational configurations.

Econometric model. Overcoming this complexity requires preparation of a primitive "econometric model" to describe the costs and benefits available by using productivity-enhancing tools in the different Services, in different settings, and by a variety of users. (For details of this model, which, though primitive, is quite complex, see chapters 4 and 5).

Non-quantitative elements of econometric model. Critical elements in the situation to be modelled are not only technical in nature. They also involve matters of ownership, control, and historically-based ways of doing things. (These forces are economic, organizational, and political in nature. For a discussion of similar forces that affected Federal ADP managers during the years 1958-1983, see [Werl83]). As one example, "benefits" from the GFE/Environment strategy accrue to one group of people (taxpayers, DoD, and the individual Services), while costs are borne by others (program managers, contractors, and software developers, maintainers, and users).

Offsetting the obvious benefits are strong forces that oppose use of standard languages and a standard environment [Werl83]. In part as a resultant of these forces, every software-intensive defense system delivered to the government now has a unique environment supplied by the system's developer.

General form of econometric model. For an assumed software workload (i.e., programs to be developed, enhanced, or repaired), the model compares the economic variables listed below.

Net Benefit = (Software cost, current)

- (Software cost, with GFE/Environment)

- (Pro rate share of cost for developing, supporting, and implementing GFE/Environment).

Total cost to the government will be the annual number of software units required (perhaps in millions of delivered source instructions per year) multiplied by the unit cost per delivered source instruction. For each assumed level of software workload, the econometric model will include the following variables:

Software cost, current
Software cost to Government, using current practices and support tools.

Software cost, GFE/Environment
Software cost to Government, using the GFE Environment.

Costs, for both current methods and GFE/ Environment, will include variables such as these:

Cost of <u>producing</u> software, using current practices or GFE/Environment method

- Cost of personnel who use current methods or the GFE/Environment (salary, benefits, equipment, space, etc.)
- Productivity of personnel, in measures such as source code instructions per work-year (quality, reliability, and maintainability being held constant)
- Cost of software reliability, a post-deployment support cost
- Cost of software maintainability, a post-deployment support cost

Cost of developing, producing, supporting, and implementing a GFE/Environment

- cost of initial design and development
- cost for production and continuing support
- cost of continuing development, to assure that benefits of evolving technology are available to the government for the next decade
- Cost of organizations and procedures used to control software development/ support practices
- Cost of training users (both gov't and contractor) in effective use of the GFE/Environment

4. Control software by improving control of defense system

Because "software inherits system problems" [Fox82], the Air Force should extend its efforts to improve conduct of projects in the requirements and specification phases of system development. This would precede development of software, and cover several years during which system "requirements" are hammered out of the operational needs and technological capabilities, then fitted into budgetary realities. We recognize that implementation of this recommendation, which would require action at high levels in each DoD organization, is clearly beyond the scope of the present study. Nevertheless, because the phenomenon is only beginning to be understood, it is necessary to reinforce the idea now, so that the benefits from improving software development and support can be obtained eventually.

System variables drive software productivity. Software productivity is driven by system variables like these:

- Volatility of requirements, (partially imposed by changes in the defense system of which the software is an integral part).
- Reliability required from software.
- Complexity of defense system.

These "software" development variables are functions of the total defense system more than simply of software development.

5. Design GFE/Environment for continuing development

The standard GFE/Environment must be designed to accommodate future changes in system requirements, as well as in software modules, user interfaces, and methodologies used for software development and support. Expressed another way, the environment must be designed for frequent change during the next 20 years, or throughout the entire life cycle of the environment. (See chapters 2 and 5).

6. Extend environment functions to include pre- and post-coding

Finally, definitions of the "life cycles" of defense systems to be served by the environment must include the work done before software development is started. Not only must the GFE/Environment support the pre- and post-coding work on defense systems, project planning should be conducted with the extended definitions in mind. This is again because "Software Inherits System Problems". (See chapter 2).

E. UNDERLYING CONCEPT -- INTERCHANGEABILITY

Except for the promise of Ada, today there is no single integrating standard language, methodology, and support technology that addresses the problems of the USAF's multi-billion dollar software inventory. Although everyone is familiar with hardware composed of standard interchangeable parts, which can be repaired easily and inexpensively, the concept of standard interchangeable parts for software has not been accepted for the computer programs that have become so vital in weapon systems.

In this report, we describe briefly the results of research indicating the potential benefit the Air Force can get by increasing its use of interchangeable software [Boeh84]. In addition to integrated collections ("toolboxes") of software tools, the GFE/Environment accesses libraries of "reusable code fragments".

F. IMPLICATIONS OF COMPLETED WORK

It is clear that today's technology is adequate to build environments for software development/support during the entire system life cycle. Many examples already exist in laboratories, and a few (such as UNIX) have become indispensable in industrial applications.

1. "Managing" productivity drivers

In the present contract, we have successfully identified software project variables on which USAF can focus environmental efforts. These are the 15 independent project characteristics used in the "COCOMO" model for forecasting software costs. USAF can manage: programmers' experience with the programming language used on a project (LEXP); reliability needed in the resulting product (RELY), use of modern programming practices (MODP); and use of software tools (TOOL). Drivers at the system level include program complexity (CPLX), volatility of requirements (RVOL), and schedule constraints (SCED). (For details, see chapter 4).

2. Next steps in research

To support the Air Force business decision on developing one standard environment (perhaps having several compatible versions) and requiring its use as a standard GFE, we now need a different kind of research. Table S-1 summarizes what we have learned ("What we know") and, in the second column shows what we still need to learn.

Table S-1. RESULTS OF GFE/ENVIRONMENT RESEARCH TO DATE

WE KNOW:	WE NEED TO KNOW:
- Principle technical elements of an Ada-based environment needed to support each of the functions required for software support throughout the system life-cycle.	
- At least one software tool exists to support each activity of the software life cycle.	
- These tools can be integrated into support environments.	- How to overcome proliferation of environments that are non-compatible.
- How environments relate to the variety of methodologies now available for software development and post-deployment support.	
- To build a software engineering environment in the near term, an existing environment or operating system (such as UNIX) must be used as the basis.	
- Exponential improvement for the design, programming, and test activities for years 1970-79.	- Rates of exponential growth in productivity for requirements specification, design, and post-deployment support.
- Improvement can be projected by using software parameters in pricing models such as "COCOMO".	- How to avoid technological stagnation from eventual obsolescence of GFE environments.
- Many constraints and barriers to implementation. For example: vendors/market structure; economic and contractual factors; personal and organizational inertia; and political factors.	- Effect of imposing a single mandatory standard environment on vendors of the many mission-critical weapons system projects.
	- Effects on benefits to different Federal organizations using different control strategies.

CHAPTER ONE

OBJECTIVES, HYPOTHESIS, AND METHODOLOGY

A. OBJECTIVES

The objectives of this contract were to "identify, define and document what an integrated automated software development/support environment would consist of, what tools and methods are available, what needs to be developed to build this environment, and what the pros and cons are of developing a standard environment to be provided as Government Furnished Equipment (GFE)".

B. HYPOTHESIS

In conducting the study, we began by developing and stating the research hypothesis.

> DoD should develop, maintain, and provide as GFE on Government Contracts, a Single Integrated Ada-based Software Support Environment (which can be tailored to meet unique needs). For this research, we postulate a Hypothetical Ada Programming Support Environment (HAPSE).

In classic scientific tradition, we then directed the study effort toward testing the hypothesis. This approach let us concentrate our forces in a relatively narrow channel.

C. METHODOLOGY

In October 1984, AFBRMC/RDCB, WPAFB, OH, engaged Technion International, Inc. to evaluate the feasibility of developing integrated, automated software development/support environments to be provided as GFE to contractors, software support organizations, users, and trainers responsible for managing software-intensive mission-critical systems.

1. Three Tasks

The work was structured as three tasks:

Task 1: Identify and define the principal [sic] technical elements of a single hypothetical integrated automated software development/support environment (HAPSE) required to support the complete software life cycle for embedded systems. (For details, see chapter 3).

We used public sources, such as IEEE and ACM journal publications and interviews, for this task.

Task 2: Outline a development plan leading to an implementable HAPSE. (See chapter 4).

We found that at least one software tool exists today that can support each activity of the software life cycle. We reviewed current laboratory and operational software tools, as well as programming environments. We investigated the state of the art in software support environments, then defined a Hypothetical Programming Support Environment ("HAPSE") that would meet USAF needs for a GFE environment. Finally, we verified the current technical feasibility of building a HAPSE. (Chapter 3).

Task 3: Investigate the pros and cons of imposing HAPSE as a standard GFE for contractors, users, support organizations, and trainers, dealing with mission-critical software. (See chapters 4 and 5).

With technical feasibility confirmed by Tasks 1 and 2, in Task 3 we looked at the pros and cons to USAF of implementing the HAPSE/GFE strategy. We found that this area contains more constraints on successful implementation -- in the form of organizational, control, and economic variables -- than do the purely technical concerns. Success in achieving the potential improvements in software reliability, maintainability, and productivity will depend on negotiations among the many parties who are motivated to conduct business as usual.

In Task 3, we performed statistical analyses of 34 software projects from the COCOMO project database [Boeh81]. This, with material from journal articles, enabled us to determine which project dependent variables help most in improving performance of software development and support activities. (See chapters 3 and 4).

2. System/Software Engineering

It became necessary to differentiate sharply between two quite different complex entities, each of which is usually referred to as "...the system". We made this differentiation:

A Defense System contains hardware (e. g., sensors, launchers, communications, weapons, computers, etc.) people, and computer software.

A Software System is a subset of the defense system of which it forms a vital part. The software function, analogous to the nervous system in an organism, coordinates and controls all the varied components of the Defense System. The software subsystem itself comprises a complex collection of computer programs, support and test equipment, and trained people. But it is only a subset, not the complete Defense System.

It is widely believed that the proper intent is to optimize performance, reliability, and maintainability of the defense system with respect to its mission objectives, as opposed to optimizing performance, maintainability, or reliability of any subsystem (such as software). While we used both system engineering and software engineering frameworks in the research, we viewed defense software projects as subsets of, and interdependent with, the defense systems of which they are parts.

Complex defense systems require support during the research, development and post-deployment operational phases of their life cycles. This is particularly true of the software components of defense systems, because they are often required to compensate for incompatibilities among hardware and human resource components.

The HAPSE will need to assist many different software people working over many years, in the initial development as well as in the later operational enhancement and support activities for defense systems. As a hypothetical example, let us take the initial development of software for a small defense system. It is done by contractors and takes, say, five years to complete. Subsequent enhancement during the operational life of the defense system may be done by the same contractor but is performed by different people and -- for a successful mission-critical system -- may be needed for more than 20 years after the initial delivery. It is usually the case, in both industrial and government, that development is conducted much differently than is maintenance and enhancement [Fox82, Pari84, and others].

Much of the software developed, and the literature describing software engineering, refer to administrative and process control programs. These are clearly different in nature from the software contained in defense systems. For the purpose of this study, we differentiated mission-critical software from administrative data processing systems by the characteristics in Table I-1.

TABLE I-1

CHARACTERISTICS OF MISSION-CRITICAL SOFTWARE COMPARED TO ADMINISTRATIVE SOFTWARE

Mission-Critical Software	Administrative Software
Perform complex mathematical calculations such as solution of simultaneous differential equations in real-time.	Perform simple calculations involving data replacement, arithmetic operations, or symbol manipulations in batch mode.
Stringent real-time processing requirements with limited run-time constraints and complex interrupt-processing patterns.	Less stringent run-time constraints, and simple interrupt processing.
Relatively small run-time memory, sometimes no secondary storage.	Large run-time memory and secondary storage.
Cost of failure measured in terms of possible loss of human lives.	Failure is costly only in terms of dollar costs for recovery.
Life cycle often includes up to ten years for system containing software; software development often exceeds ten years.	Life cycle for development seldom more than 3-5 years.
Post-deployment support of software is usually very active; changed requirements often added after system has been deployed.	Post-deployment support is relatively inactive.

SOURCE: Adapted from Redwine, et. al., Institute of Defense Analysis Report P-1788, pp. 48-49 [IDA84a].

3. Technical Elements of Environments

At the outset, we studied current environments and those features that will probably be available in the next seven years. The need to provide support throughout the system life cycle dictates use of a "General" environment approach, which will support more than one programming language and does not require users to follow one specific methodology.

As detailed in chapter 3, the most effective approach, given the schedule restriction of seven years, is to build the "HAPSE" as a layer "on top of" an existing operating system. This permits the "HAPSE" overlay to present a uniform set of commands to the user, and compensates for differences in the individual software tools called by the user, which may use different command structures internally.

The existing operating system (the "underlying" system) permits use of existing technical capabilities such as: (a) Ability to build databases; (b) File system; (c) Editor; and (d) Ada translator and interpreter. Several experimental environments have been built in this manner (e.g., "on top of" the UNIX operating system).

Using this approach, programmers will see the HAPSE, with its commands and set of integrated software tools. Four sets of tools are described in chapter 3. Toolsets begin with a bare minimum set of four tools, and add a set of seven additional tools "required" for effective support. An additional six, less critical but still "important", tools comprises the third set. The fourth adds five tools judged "useful" but not required.

4. Modelling the Effects of this Technology Push Strategy

Since we began work on this contract, Technion International, Inc. and our subcontractor, Systems and Applied Sciences, have concluded that standard software support environments can be built using current technology. We verify (in chapters 4 and 5) the technical soundness of the Air Force's proposed GFE approach and describe potential savings of substantial magnitude.

The criteria used for design of the software support environment included: (a) improving the productivity of software development and support activities; (b) improving reliability of the software products; (c) producing operational portability of software products [to different projects, hardware and organizations]; and (d) factors such as effectiveness, user-friendliness and product usability that are essential to achieving the other objectives.

The research and analyses performed yield the conclusion that the HAPSE is technically feasible. For the HAPSE to become a reality in the competitive market place -- and to ensure that it escapes the fate of quick obsolescence -- the Air Force also needs to consider economic and organizational issues. These are especially important with regard to different plausible levels of standardization to be imposed. Chapter 4 sketches a primitive econometric model that depicts the complexity of implementing the strategy. Additional research is proposed to develop this model further and possibly to parameterize it dynamically.

It is foolhardy to estimate potential savings in a field subject to so many uncertainties. However, a "rule of thumb" in the management consulting profession is that a target of ten percent savings can be realized in nearly any operation. Attainment of such savings requires changes in the ways of doing things. While this conceivably may not be true for the software field, the leverage is enormous. Because of the large annual DoD costs for software development and support (estimated at up to $1Ø billion annu- ally), an improvement of only one percent would make about $1ØØ million available for other purposes each year. A ten percent savings would release $1 billion annually.

CHAPTER TWO

RELEVANT LITERATURE

A. INTRODUCTION

In this chapter we first describe the nature of the problem addressed by the proposal to develop a single standard GFE/Environment, then review some factors that affect feasible solutions. Reviewing literature of this field is like a journey in a time machine. Most of the material published only a few years ago seems out of date. Precious little written before 1975 describes the realities of 1985. Our survey has five parts.

In part B, "View from the Top", we find a conception of software development and support that is available from no other source. We learn what it looks like to those responsible for success of software development and support projects. Our guides are two of the rare authors who have actually managed development of software systems, and who help us learn from their mistakes.

Next, we look briefly at the nature of the programmer's task, and describe the recent revolution in the way designers and programmers construct and support software. This is important, because the nature of this work has changed dramatically during the past 15 years and it will probably continue to evolve during the foreseeable future.

In part D, we take a bird's eye view of some landmark developments that have led both to the present problems and to the proposed solutions.

In part E, we review the situation that led to the proposal to develop a single standard GFE/Environment for contractors, maintainers, and users to use in developing and supporting DoD software.

Finally, in part F we look at the organizational settings in which the DoD conducts software development and support, with a view to identifying barriers to successful transition. We review past developments which have had unforeseen results and ask how those results can be avoided in implementing the GFE/Environment proposal.

B. VIEW FROM THE TOP

In this section we discuss two publications that changed the way we look at the symptoms of software problems, thereby affecting where we search for solutions. Each book changed conventional thinking by adding insights born of actual experience. In its time, each represented a fresh look in managerial thinking about software systems. Each author writes in language different from that of the theoretician, the academician, or the engineer, and presents ideas far in advance of conventional thinking when first published. Most importantly, each brings valuable insights for the GFE/Environment implementation effort.

1. Frederick P. Brooks, Jr.

Wider known of the two is Frederick P. Brooks, Jr., with his now-classic The Mythical Man-Month [Broo75]. Brooks was manager of IBM's development of the OS/360 operating system software project during the 1960s. His best known axiom is "Brooks's Law":

> Adding manpower to a late software project makes it later.

He explains,

> More software projects have gone awry for lack of calendar time than for all other causes combined (pp. 25,26).

Ten years have passed and Brooks's Law still seems to be valid. Other comments are equally valid, but less well known:

a. The general tendency is to over-design the second system, using all the ideas and frills that were cautiously sidetracked on the first one.

[Another second system effect is] . . . the tendency to refine techniques whose very existence has been made obsolete by changes in basic system assumptions. . . OS/360 has many examples of this. (pp. 55-58).

b. Brooks was the first to apply to software engineering projects the understanding that communication and coordination among team members is a necessary, even a major, part of the work on large projects:

> If there are n workers on a project, there are $(n^2-n)/2$ interfaces across which there may be communication, and there are potentially almost 2^n teams within which coordination must occur (p. 78).

c. He advised that we should ". . . plan to throw [the first] software system away; you will anyhow." Following up on this observation, he stressed the needs to "plan the system for change", and to "Plan the organization for change". (pp. 116-118).

d. Viewing program maintenance over the years (the term "Life Cycle" had not yet become prominent in 1975) he found:

> All repairs tend to destroy the structure, to increase the entropy and disorder of the system. Less and less effort is spent on fixing original design flaws; more and more is spent on fixing flaws introduced by earlier fixes. (p. 122).

2. Joseph M. Fox

Even more important for the GFE/Environment project is Joseph M. Fox's Software and Its Development [Fox82]. In this book, Fox describes software development at a later period, as seen by a top level manager (he headed IBM's Federal Systems Division during part of the 196Øs and 197Øs). He emphasizes the "hands-on" aspects of managing system development and support projects. His insights on development and support cover the complete range of software products, from very small to very large ones similar to defense systems. Most valuable to the GFE/Environment project are these:

a. [The Continued Development Phase of the Life Cycle] . . . is often the most ignored piece of the life cycle, left to be taken care of by some new and often unnamed team. One of the key ideas we will stress is that this piece of the cycle must be taken into consideration from the very beginning of the development effort. (p. 45). (Our emphasis).

Building on his description of the "Continued Development Phase of the Life Cycle", he subtly revolutionizes life cycle thinking about software by distinguishing among (1) initial development (which is not really complete when software is accepted, in part because the software contains "latent" or unfound shortcomings, and in part due to the "abandon function" phenomenon seen as scheduled completion time approaches); (2) [operational] use; and (3) continued development (i.e. addition of new software capabilities, as well as correcting shortcomings when they are found). (pp. 35-46).

b. To implement the concept of the "Continued Development Phase of the Life Cycle", he advises:

> The first requirement of a large system of software is that it be built so that it is easy to change.

> The first job of the manager of a large software effort is that he or she budget for many releases of the software.
>
> . . . the lack of clear requirements is the single most difficult problem in developing large . . . systems. The project manager does not know where he or she is going. (p. 75).

Why does he state this so strongly? Fox has observed that:

c. "Software Inherits the System Problems".

He explains,

> In the case of large . . . systems, as the other pieces of the system solidify, the last piece that can be modified is the software. What do we mean, 'solidify'? In large . . . systems there are often many elements that are under development, [and/or] being improved. The communications/display/radar/sonar/IR/telemetry/missile/satellite/propulsion/control/whatever -- some of these will be the newest, most advanced in the world when they work in our system, or they will be in new connections. Therefore, they will surprise us in the way they work, and we will have to adapt to reality.
>
> The burden for adapting to the differences falls on two pieces of our system -- the software and the human operators. We try to push as much into the software as we can, and then let the rest fall onto the operators. The software is "soft," if we designed it right and controlled it right. If we documented it and modularized it. Then indeed, it is soft. If we did not, it can be a block of solid concrete. (p. 73).

d. Particularly appropriate for the GFE/Environment project is Fox's citation of:

> A Department of Defense study . . . conducted in August, 1977 of nine major automated systems. Most were communications systems. . . The study summarized:
>
> - All had unstable and changing requirements; the bigger the system, the worse the rate of change.
> - Most lacked any formal mechanism to track/ manage requirements.
> - Some did not even perceive the need to validate requirements.
> - Most were plagued by "wish lists".
>
> The study accurately describes [the author's] experience in the commercial realm of computers. (p. 1Ø4).

e. Adapting to practice the insights just described, he explains a ubiquitous phenomenon. Most software engineers and programmers have observed that requirements on their projects are seldom static. It remained for Fox to point out that unceasing change is not an isolated anomaly or a sign of "mismanagement," but in fact is a fundamental characteristic of the role of software in complex systems. His succinct statement, "Requirements Definition is a Continuous Task," (p. 1Ø7) has rich implications for all who work with defense systems:

f. On progressing toward the software equivalent of "interchangeable parts", he compares it to the historical evolution of hardware production. He begins by describing manufacturing practices before the "industrial revolution" (when individual items were literally handmade), and follows the evolution to wide use of machinery and of standard interchangeable parts. The results of the evolution included greater productivity in manufacture, lower costs to consumers, and longer life resulting from easier repair.

> Software development is still in the early phases of its industrial revolution. . . . some specialization of labor has occurred, and some automation, but we do not yet have . . . the interchangeable part. It is on its way; it is inevitable, even with software, but it is not

> here yet. We are still learning how to organize to "produce" software. We are developing the tools and technologies simultaneously. We are proceeding at a great rate, faster probably than most imagine. (p. 288).

The significance to USAF of Fox's concepts has only begun to be recognized since about 1982 [Fox82, JLC84].

C. REVOLUTION IN SOFTWARE CONSTRUCTION

The view of the system life cycle, completed by Fox and JLC, has been accompanied by an outmoded view of programming. In 1979 Winograd noted that an obsolete view of software construction still dominated thinking about software practices, years after the true nature had changed dramatically [Wino79]. In the earliest days, the software developer's image was that of a mathematician/artist. Ideally, the software product was the equivalent in elegance of the "Mona Lisa". While the artistic image remained, capabilities of the artist's tools (hardware and software) grew enormously, and the nature of the developers' work changed. Today a more appropriate image might be that of an engineer developing and supporting a telephone system or an electric power utility [Arth83, Myer85, Silv85].

Throughout the development of numerous programming languages and systems, the artist/mathematician's implicit objective was to design an algorithm that could be written down in a precise and exhaustive number of instructions. But the development of high order languages (HOL) and compilers quietly revolutionized the work actually done by software people. They began to use the basic HOL building blocks (instructions which represent logical algorithmic structures in both the control and data domains). In addition to increasing productivity, this meant that programmability was definable at a level closer to the system than to the level of the machine. But the older self-image remained; programmers still think of themselves as artists.

Stating this restricted view of programming and programming languages, Winograd points out trends in the technology of large complex systems that happen to contain software. First, the computer is no longer just a computational machine, but now forms a prime component in an integrated hardware-software system of high complexity. Such "embedded" systems are proliferating as the microcomputer revolution continues, making feasible applications like message processing on telephone networks or satellite TV networks. Embedded computer software exhibits different characteristics than administrative software. It is more fault tolerant, responds in real time, and interacts with a more advanced display input/output (I/O) front end.

Second, the building blocks are larger. No longer HOL instructions, now they tend to be subsystems or packages which are themselves collections of integrated data structures, programs and protocols. Winograd basically addresses the difficulty currently being faced in integrating many independent components which are not within the same computer hardware.

Third, most programming activity (more than 60 percent) is now concentrated on integration and modification of existing software instead of generation of new independent programs [Wino79). This is confirmed by a General Accounting Office survey conducted in 1983 [GAO83]. The survey showed that 61 percent of programs surveyed were modified during the year because requirements had changed. Only 17 percent were modified because of software defects. Finally, nearly four fifths of programs surveyed were maintained during the year. Only 22 percent of programs had no maintenance during the survey period.

Reasons for software maintenance	Percent
Enhancing program beyond original objectives	21%
Upgrading hardware or software	16
Keeping tables/codes current	14
Changing legislation/regulations	10
Total due to changing requirements	61%
Removing defects in software	15
Other	2
Total defects	17
Programs not maintained during year	22
Total programs in survey	100%

The GAO survey supports Winograd's finding. Other reports show that maintenance is generally not done systematically, either in industry or in Federal departments [GAO81, GAO83, Mart83, Wien84].

The main reason for the software evolution demonstrated by these findings seems to be that systems containing software components are required to evolve over many years. In the DoD environment, this means that the software components, called on to satisfy new and changed requirements throughout a system's life cycle, permit the system to remain useful in supporting DoD missions.

D. LANDMARK DEVELOPMENTS IN SOFTWARE ENGINEERING

Landmarks along the road include FORTRAN, JOVIAL, COBOL, optimizing compilers, systematic use of software tools, and reusable code. Together with other technical developments, the result was the exponential growth in productivity shown in Figure S-1, "Software productivity increases exponentially". If history is a guide, the journey will probably continue along this exponential path. The new landmarks will include increasingly high level languages and the tools that support them, "reusable code", "support environments", use of expert systems and other artificial intelligence techniques and, most importantly, developments that are still unknown.

A rough map of the journey is found in a paper by Herb Hecht and Ray Houghton [Hech82]. NBS reported that, in 198Ø, we were still not using software tools widely [Hech81]. Another NBS report, on the May 198Ø workshop on Programming Environments [Bran81], described the state of the art and proposed environment-related research for the next five years.

Reports from the advance scouts include descriptions of the Boeing experimental software environment, ARGUS [Stuc83], and Japanese practices [Taj84]. Other key reports are Boehm's article on the TRW Software productivity system (SPS) [Boeh84], which suggests that the way to go is an n-fold path; and the Zelkowitz survey of industrial software practices in use in the U.S. and Japan in 1983 [Zelk84]. Additional details are given in Chapter 3.

The difficulties encountered in building and modifying large systems are numerous and not easy to overcome. Solutions probably lie not in rigorous academic discipline but in more adequate tools. Let us consider the Ada experience as an example.

1. Standardization of Languages

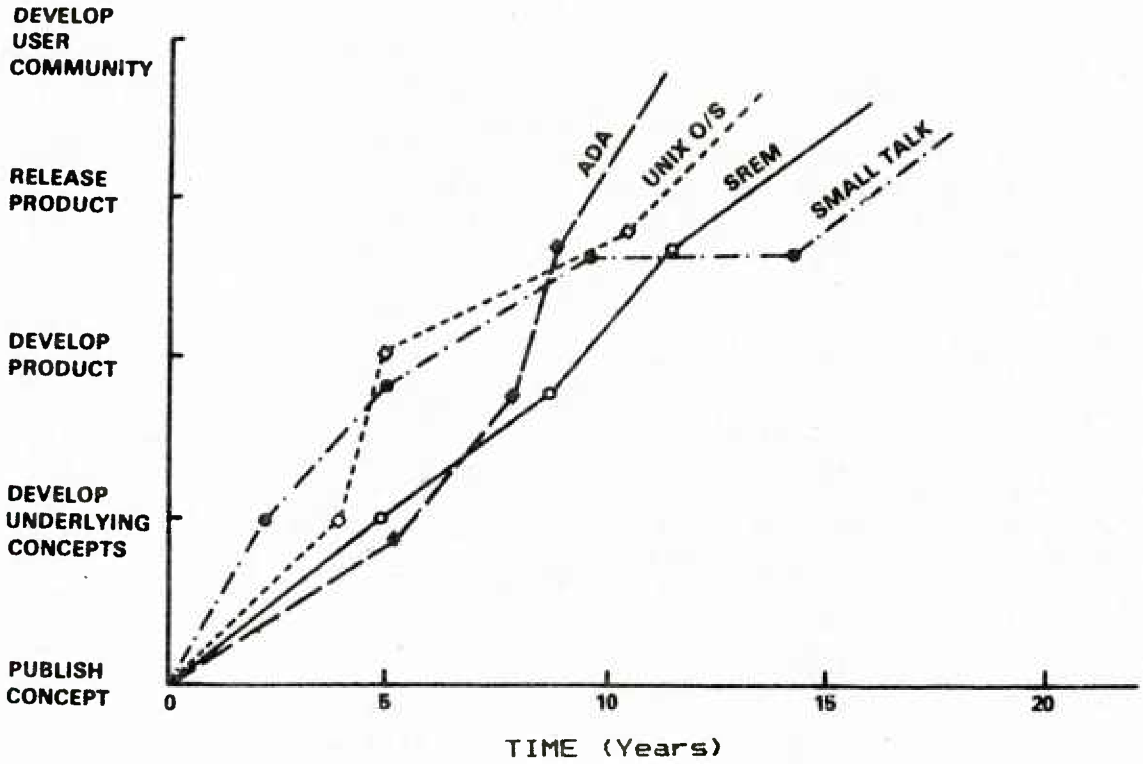
Agreed on standards and conventions are required to prevent misconceptions and miscommunications among the many different people involved in the development of any complex system. During the 197Øs, DoD had some success in standardizing the COBOL and Jovial languages. These DoD efforts addressed the proliferation of languages and dialects by standardizing languages as a means to attain independence from a single vendor's computing hardware. In an attempt to limit language proliferation, DoD and USAF later limited the number of standard HOLs to be used on new defense systems [USAF76]. DoD subsequently extended the effort by sponsoring development of the Ada language, which began around 1975. It was intended to standardize on a single programming language. This would help in DoD's battle to control escalating software costs and create software that could run on computers built by many vendors.

In July 1980, a draft of the Ada language was proposed, and in 1983 the ANSI standard for Ada was accepted [Ichb84]. From 1980 to 1983 Ichbiah's Ada development team had tested and evaluated the Ada language and incorporated 7000 modifications that came from 15 different countries. The entire process involved more than 1000 people from all over the world [Ichb84]. But the road to an accepted standard is not smooth. The Association for Computing Machinery (ACM) standards committee's position on Ada did not oppose the principle of standardization, but objected to the Ada specification proposed as of February 1982. The ACM position was given in [Skel82].

> "Although the ACM is in opposition to the present proposal, two points should be emphasized:
>
> - The ACM is not opposed to national and international standardization of the Programming Language Ada, but views the present specification as inappropriate for such adaptation;
>
> - The raising of this issue at the national standards level has had the extremely beneficial effect of focussing attention on the specification, and thus eliciting pertinent comments which might otherwise not have been available."

Some potential Ada users still see problems with adapting to the language [Buxt85, Ledg82, Wild83].

Figure II-2, Diffusion of Software Systems Takes Decades, shows that progress of the Ada standard language program, though begun a decade ago, is not slower than other comparable innovations have been. New systems typically require about 14 years toprogress from concept to fielded status) [IDA84a]. Ada compilers and support environments are beginning to appear, and some promising signs of successful institutionalization of Ada are visible [Elec85].



SOURCE: IDA84a

TIME (Years)

Figure 2-2. Diffusion of Software Systems Takes Decades.

2. Software Engineering of Tools and Environments

Software Tools. Hundreds of individual software tools have been developed over the years, and some are used widely by both commercial and government customers [Data84, FSTC83a]. To show the wide usage, we provide a sample, drawn from 17 commercial "program development aids" packages studied by Datamation in 1984. Because of the nature of Datamation's study, all 17 packages had substantial numbers of users. We obtained reliable estimates of the number of users were available for 9 of the packages reported. After each package, the number of users is shown in parentheses.

CONDOR (400+);
MANTIS (1700);
CPG (530);
ADR/VOLLIE (1600);
QUOTA II (550);
SPEED I (4000);
INTERTEST (1400);
O-W-L (400);
DATAMACS (950+).

Such wide use shows that standard software packages meet real needs in the industrial market. These packages provide various portions of the capabilities needed for the HAPSE, but none has the complete range. Further, none is usable for USAF's purposes. All reported packages are designed primarily for computers and languages widely available to commercial users, but not used in MCCR.

Studies of Environments. In November 1981, the Institute for Computer Science and Technology (ICST) at the National Bureau of Standards published an overview of software tools usage, with results of a survey and an interpretation of findings [Hech81]. It also gave the requirements for future tools usage. In September 1982, the same group published another document on the introduction of software tools [Hech82]. This document details the levels of tool usage envisioned in various types of user organizations,. outlines user tool needs, and enumerates event sequences for tool development.

During February, 1984 ICST put out a study plan for comparing software development schemes for Ada [Houg84b]. The study details typical developmental phases and maintenance phases.

Concurrent with this standardization activity, industry and academia have been making headway in software engineering by research in software environments. Many different experiments are under way [Bars84, Wass81]. In this report we focus on the experience of commercial firms which have described their work with integrated software support environments.

3. Commercial Firms' Experience

Perhaps a dozen separate environments have been, or are being, developed [IDA84a]. Several of these, particularly relevant to the GFE/Environment proposal, are discussed below.

ARGUS. Boeing Computer Services developed the ARGUS environment, which combined CAD/CAM-like functions for producing software products [Stuc83]. The aims of ARGUS are to increase productivity throughout the software life cycle, improve software quality, provide MIS control capabilities, and establish an integrated software environment. ARGUS contains four separate "toolkits", which have been built on the UNIX operating system.

The ARGUS makers have three observations on distributed software engineering environments.

- First, in the development of micro-based workstations, a bigger environment may not necessarily be better; in fact smaller environments built on top of the UNIX operating system have significant software capability.
- Second, net computing costs are lower with automated tools because increased productivity uses relatively low cost computer time to increase the effectiveness of high cost employees' work hours. The human component is now more expensive than the hardware.
- Third, the environment must have timely infusion of project information that directly bears on project success parameters [i.e., project management data].

TRW'S "Software Productivity System (SPS)." At TRW, Barry W. Boehm's group developed an automated software environment primarily to boost productivity and decrease maintenance cost [Boeh84]. TRW's Software Productivity System (SPS) is a conglomerate of strategies; it includes a work environment, evaluation and procurement of hardware equipment, provision of immediate access to computing resources through area networks, integration of software developmental tools and transfer of new technology. The motivating factors for construction of the SPS were increasing demand for software, limited supply of software engineers, increasing support expectations and reduced hardware costs. The broad guideline specifications laid down by TRW management for the SPS support environment included some adapted from the DoD Ada Stoneman requirements [Buxt8Ø], and added these:

a. provide multiple-programming language capability

b. support mixed target-machine complexes

c. integrate existing programs and data

d. support of classified projects

e. facilitate non-programming activities, such as documentation.

The SPS architecture currently supports a broadband local area network to perform high speed terminal-to-computer communications. The SPS uses the UNIX operating system as the base, the "underlying environment".

The conclusions of the earlier TRW study that lead to the SPS experiment are given below [Boeh84]. Emphasis is added to show Technion's estimate that seven of those conclusions most important to the GFE/Environment effort:

i. The integrated approach and immediate access to an excellent set of tools has the highest payoff.

ii. Office automation with project support capabilities is a must.

iii. A master project data base with software development artifacts is worthwhile.

iv. Adherence to user interface standards is a must for preserving the capability to evolve.

v. User acceptance of development environments needs careful fostering.

vi. [Local Area Networks] . . . allowing interconnection of user terminals are a strong support to distributed work environments.

vii. Privacy of [programmers'] offices improves productivity.

TRW management believes that "The SPS is a long term ambitious project that we can learn from as it evolves over the years." [Boeh84]. That evolutionary learning can also be of value to AFSC.

Computer Integrated Manufacturing (CIM). At McDonnell Douglas in St. Louis, corporate managers have embarked on an ambitious Computer Integrated Manufacturing (CIM) task that virtually amalgamates the company's software tools into one system, which forms a global resource for a fully automated manufacturing facility. Tools involved in CIM are those of Computer-Assisted Design (CAD), Computer-Aided Manufacturing (CAM), Computer-Aided Engineering (CAE), Management Information Systems (MIS), and Decision Support Systems (DSS).

DoD Experience. There is some experience with incorporating automated software environments into DoD software production practices. The life cycle management analysis and development schemes described by Stuebing [Stue84] in his Systems Engineering Environment (SEE) for weapon systems software is the first step in this direction. The SEE approach brings to light many pertinent parameters that need to be examined for successful development of automated environments. The FASP (Facility for Automated Software Production) at the U.S. Naval Air Development Center (which Stuebing describes) has now been rewritten, using the UNIX operating system on a VAX 11/780 [Stue84]. During the survey, the Technion researchers noted that the UNIX operating system cited by Stuebing is now being used by a majority of the U. S. automated software environment developers.

4. Limited Use in Software Production

Yet these experimental developments have not entered the workplaces in which software is developed and supported. In a recent survey of current software engineering practices in the U.S. and Japan, data from 20 organizations (including IBM and five Japanese firms) were collected and analyzed by a group at the University of Maryland [Zelk84]. They found that:

i. Every company had its own guidelines for software development that were either written or unwritten.

ii. There was a disparity between techniques used in industry and the current software engineering literature.

iii. Use of software engineering practices was quite rare.

iv. None of the firms used tools to support software engineering practices in any significant way.

The analysis also makes observations of existing organizational structures in the simplistic sense, to determine tool usage and data collection for the development of software environments. Their survey evidently was restricted to the initial development of software, using the traditional phases [requirements and specification, design, code and unit test, and integration test].

Are tools investments or expenses? Zelkowitz, et. al. point out that software developers in the United States are primarily oriented to individual projects or applications. In contrast, for the the Japanese firms centralized development of software tools and centralization of software resources is prominent. The Japanese tend to invest in software tools, seeing the costs as part of their firms' capital investment base, rather than charging tool costs as unique expenses to separate projects. The Japanese also incorporate a post-mortem analysis of error data to track related failures [Zelk84, Mats81, Taj84].

E. HOW DOD GOT TO THE GFE/ENVIRONMENT

DoD's attention to the symptoms of software problems is not new. The topic has been studied frequently and in some depth for the past two decades, during which time the field has undergone several complete transformations. Technion has noted that recognition of the inherently evolving nature of software, and of the magnitude of required post-deployment support were not widely recognized until about 1982. For example, well done key reports dated 1970, 1975, 1978, and 1983 mention post-deployment support, but do not seem to have understood the revolution in software development and support that they implied. Examples of these studies include:

- U.S.A.F. Select Committee on Computer Technology Potential, "An Air Force study of Air Force Organizational Ability to Exploit and Manage Computer Technology", 1970 [USAF70].
- A. Asch, D. W. Kelliher, J. P. Locher III, and T. Connors, "DOD Weapon Systems Software Acquisition and Management Study", MITRE Corporation, May 1975. [Mitr75].
- U. S. Office of Management and Budget, "President's Federal ADP Reorganization Study Reports," 1978 [OMB78].
- Booz, Allen and Hamilton, U.S.A.F. Acquisition Improvement Project Reports, 1981 [Booz81].

1. A Single DoD Programming Language (Ada)

In the early seventies, the U. S. Department of Defense began laying the foundation for a single high order computer language for new DoD embedded computer systems. It was clear that even DoD could not continue supporting as many as 400 different computer languages and dialects. The projected software development and maintenance costs had become astronomical. In 1975, DoD initiated the U. S. Department of Defense Common High Order Language program (which later produced the Ada language). The preliminary design for the Ada language was completed in 1979, and 16 Ada compilers had been validated by 1984. At present, there are no approved Ada dialects. Further, there are no plans for future approval of dialects for Ada.

2. Software Engineering Approaches: APSE and STARS

With the Ada development under way, by 198Ø DoD's attention was directed toward solving the next part of the problem, the Ada Programming Support Environment (APSE). The "Stoneman" document [Buxt8Ø] included this summary in its description of requirements for Ada programming support environments:

> It was recognized from the beginning that the major benefits to DoD from a common language would be *economic* and would derive from Ada's appropriateness to military operations, from the portability that comes with a machine independent language, from the availability of software resulting from acceptance of the language for nonmilitary applications, and most importantly from the use of Ada *as a mechanism* for introducing and distributing effective software development and support environments to those developing and evolving military systems. [Buxt8Ø]

The progress of this massive and complex effort can be seen in other key statements issued since that time:

- "Final Report of the Software Acquisition and Development Working Group", July, 198Ø [DoD8Ø].
- "Report of the DOD Joint Service Task Force on Software Problems," July 3Ø, 1982 [DoD82]
- "Software Technology for Adaptable, Reliable Systems (STARS) Joint Task Force Report," 15 March 1983 [DoD83a]
- "Department of Defense Computer Technology: A Report to Congress", August 1983 [DoD83b]
- "Plan of Action and Milestones for Definition and Preliminary Design of a Joint Services Software Engineering Environment (JSSEE)", January 1984 [DoD84a]

3. Implementing Standard Environments

The rest of this report analyzes the issue of combining the power of the standard Ada language with that of a standard programming support environment, designed to improve adaptability and reliability of DoD software. The report defines a "HAPSE," and suggests an adaptive model for the technology transition.

E. BARRIERS TO SUCCESSFUL TRANSITION

In this section we mention some of the special conditions faced by DoD and its components in this large scale standardization effort. The effects of various proposals for overcoming these conditions are to be determined using in the econometric model discussed in chapter 4. Three barriers with special relevance for the GFE/ Environment strategy are:

1. Present DoD and Service organizational settings, which are designed to optimize organizational interests rather than successful development and operation of mission-critical systems. Development efforts for these systems are thus characterized by fragmentation of systems development work among many organizations, each having its own unique incentives and reward system [OMB78, DoD8Ø, Werl83];

2. Staff assignments. Governed by Service needs for staff development that are only weakly related to success of mission-critical systems, staff assignments are typically of shorter duration than the life cycles of systems containing software. [Luttwak, Edward N., The Pentagon and the Art of War, 1984, New York, Institute for Contemporary Studies/Simon and Schuster, esp. pp. 89-91, 166-182, and 218-219; and OMB78].

3. Responsibility for systems is shifted among organizations on many occasions during the typical system's life cycle. Each shift has wrenching effects, often accompanied by delays and changes in requirements. Glowing exceptions have included such programs as the Lockheed "Skunk Works" which developed the U-2 and SR-71. Other examples include the Navy's Polaris program, and Adm. Rickover's mission-oriented Navy, in which responsibility was maintained in one organization throughout development, during deployment, and in post-deployment enhancement and maintenance.

These are among the issues discussed in Chapter 4, "Pros and Cons", and Chapter 5, "Planning for Implementation".

REFERENCES CITED IN CHAPTER TWO

Arth83 Arthur, Lowell Jay. Programmer Productivity: Myths, Methods and Murphology. New York: John Wiley, 1983.

Bars84 Barstow, David R., Shrobe, Howard E., and Sandewall, Erik. Interactive Programming Environments. New York: McGraw-Hill, 1984.

Basi84 Basili, V. R. and Perricone, B. T. "Software Errors and Complexity: An Empirical Investigation." Communications of the ACM, Vol. 27, No. 1, Jan 1984, pp. 43-52.

Boeh81 Boehm, Barry W. Software Engineering Economics. Englewood Cliffs: Prentice-Hall, 1981.

Boeh84 __________, et. al. "A Software Development Environment for Improving Software Productivity." Computer. June 1984, pp. 30-42.

Booz81 Booz, Allen and Hamilton, Inc. "Final Report: Defense Acquisition Study." Washington, D.C., 1981. This is the official document describing the USAF "Acquisition Improvement Project" headed by Col. Don Sawyer.

Bran81 Branstad, Martha A., and Adrion, W. Richards, (Eds.). "NBS Programming Environment Workshop Report." National Bureau of Standards, NBS Special Publication 500-78, June 1981.

Bray83 Bray, G. "Implementation Implications of Ada Generics." ACM Ada Letters, Vol. III, No. 2, Sept/Oct 1983.

Broo75 Brooks, Frederick P., Jr. The Mythical Man-Month: Essays on Software Engineering. Reading Massachusetts: Addison-Wesley Publishing Company, 1975.

Buxt80 Buxton, J. "Requirements for Ada Programming Support Environment: STONEMAN." U. S. Department of Defense, Washington, D.C., February 1980.

Buxt85 ___________. "Keynote Address", Proceedings of the ACM AdaTEC "Future Ada Environment Workshop," Santa Barbara, California, 17-20 Sept. 1984. Reprinted in Ada Letters, Vol. IV, Number 5, March, April 1985, pp. IV.5-40/44.

Buzz85 Buzzard, G. D., and Mudge, T. N. "Object-Based Computing and the Ada Programming Language." Computer, March 1985, pp. 11-19.

Cast84 Castor, Virginia L., et. al. "Evaluation and Validation (E & V) Team Public Report, Vol. I, Interim Technical Report for Period 1 October 1983 - 30 September 1984." AFWAL TR 85-1016. Wright-Patterson AFB, Ohio, 45433-6543.

Data84 "Annual Survey of Commercial Software Packages." Datamation, December 1983, esp. pp. 106-114 and 128-134.

DoD80 Software Acquisition and Development Working Group. "Final Report of the Software Acquisition and Development Working Group." (Chairman, Mr. Victor E. Jones), July 1980.

DoD82 "Report of the DOD Joint Service Task Force on Software Problems." July 30, 1982.

DoD83a "Software Technology for Adaptable, Reliable Systems (STARS) Joint Task Force Report." 15 March 1983.

DoD83b "Department of Defense Computer Technology: A Report to Congress." August 1983.

DoD84 "Plan of Action and Milestones for Definition and Preliminary Design of a Joint Services Software Engineering Environment (JSSEE)." January 1984.

Elec85 Wolfe, Alexander. "Critical Mass Builds for Ada", Electronics Week, January 14, 1985, pp. 18-19.

FIPS99 "Guideline: A Framework for the Evaluation and Comparison of Software Development Tools." FIPS PUB 99. National Bureau of Standards, March 1983.

Fox82 Joseph M. Fox. Software and Its Development. Englewood Cliffs: Prentice-Hall, 1982.

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

GAO81 U. S. General Accounting Office. Federal Agencies' Maintenance of Computer Programs: Expensive and Undermanaged. AFMD-81-25. February 26, 1981.

GAO81a __________. Non-Federal Computer Acquisition Practices Provide Useful Information for Streamlining Federal Methods. AFMD-81-104. October 2, 1981.

GAO83 __________. Greater Emphasis On Testing Needed to Make Computer Software More Reliable And Less Costly. GAO/IMTEC-84-2. October 27, 1983.

Hech81 Hecht, H. Final Report: A survey of software tools usage. National Bureau of Standards. Special Publication 500-82, November 1981, p. 53.

Hech82 Hecht, H., and Houghton, R. "The Current Status of Software Tool Usage." Proceedings of COMPSAC 82, November 1982.

Hech82a Hecht, H. The Introduction of Software Tools. NBS Special Publicaton 500-91, September 1982, p. 35.

Houg82a Houghton, Raymond C., Jr. "Software Development Tools." NBS Special Publication 500-88. March 1982. See esp. Appendix A, Tools by General Classification. Data in this data base are maintained by RADC (but not according to the NBS taxonomy of software functions).

Houg82b __________. "A Taxonomy of Tool Features for the Ada Programming Support Environment (APSE)." National Bureau of Standards. NBSIR 82-2625, December 1982.

Houg83 __________. "Software Development Tools: A Profile." Computer. May 1983, pp. 63-70.

Houg84 __________. "Comparing Software Development Methodologies for Ada: A Study Plan." National Bureau of Standards. NBSIR 84-2827.

Hunk81 Hunke, Horst, ed. Software Engineering Environments. North-Holland, 1981.

Ichb84 Ichbiah, J. "Ada: Past, Present, Future." Interview, in Communications of the ACM. Vol. 27, Number 10, October 1984, pp. 991-997.

IDA84a Redwine, Samuel T., Jr., Becker, Louise Giovane, Marmor-Squires, Ann B., Martin, R. J., Nash, Sarah H., and Riddle, William E. DoD Related Software Technology Requirements, Practices, and Prospects for the Future. IDA Paper P-1788. Washington: Institute for Defense Analyses. June 1984.

IDA84b DeMillo, Richard A, Marmor-Squires, Ann B., Redwine, Samuel T., Jr., and Riddle, William E. Software Engineering Environments for Mission Critical Applications -- STARS Alternative Programmatic Approaches, IDA Paper P-1789. Washington: Institute for Defense Analyses. August 1984.

JLC84 Joint Logistics Commanders' Workshop. "Final Report of the Joint Logistics Commanders' Workshop on Post Deployment Software Support (PDSS) for Mission-Critical Computer Software, Vol. I - Executive Summary." June 1984.

Kern84 Kernighan, Brian W. and Pike, Rob. The UNIX Programming Environment. Englewood Cliffs, N. J.:Prentice-Hall. 1984.

Klum85 Klumpp, A. R. "Space Station Flight Software: HAL/S or Ada." Computer. March 1985, pp. 20-28.

Ledg82 Ledgrad, M. F., and Singer, A. "Scaling Down Ada." Communications of the ACM. Vol. 25, Number 2, February 1982, pp. 121-125.

Lutt84 Luttwak, Edward N. The Pentagon and the Art of War. New York: Institute for Contemporary Studies/Simon and Schuster. 1984. See esp. pp. 89-91, 166-182, and 218-219.

Mart83 Martin, Roger J., and Osborne, Wilma M. "Guideline on Software Maintenance." National Bureau of Standards. NBS Special Publication 500-106. December 1983.

Mats81 Matsumoto, Y., et. al. "SWB System: A Software Factory." In Software Engineering Environments. H. Hunke, Editor. Amsterdam: North-Holland. 1981.

Mitr75 Asch, A., Kelliher, D. W., Locher, J. P., III, and Connors, T. "DOD Weapon Systems Software Acquisition and Management Study." Washington: MITRE Corporation. May 1975.

Myer85 Myers, W. "An Assessment of the Competitiveness of the United States Software Industry." Computer. March, 1985, pp. 81-92.

OMB78 U. S. President's Reorganization Project, Federal Data Processing Reorganization Study. Ten separate reports were issued; the "National Security Team Report" was directly relevant to this subject.

Pari84 Parikh, Girish. Programmer Productivity: Achieving an Urgent Pricrity. Reston, Virginia: Reston Publishing Company. 1984.

Phis79 Phister, Montgomery, Jr., Data Processing Technology and Economics. 2nd ed. Santa Monica, Calif.: Digital Press. 1979.

Silv85 Silverman, Barry G. "Software Cost and Productivity Improvements: An Analogical View." Computer. May 1985, pp. 86-96.

Skel82 Skelly, P. G. "The ACM Position on Standardization of the Ada Language." Communications of the ACM, Vol. 25, Number 2, February 1982, pp. 118-120.

Stuc83 Stucki, L. G. "What about CAD/CAM for software? The ARGUS concept." IEEE order no. 83CH1919-0. July 1983.

Stue84 Stuebing, H. G. "A Software Engineering Environment (SEE) for Weapon System Software." IEEE Transactions on Software Engineering. Vol. SE-10, No. 4, July 1984, pp. 384-397.

Taj84 Tajima, Denji, and Matsubara, Tomoo. "Inside the Japanese Software Industry." Computer. March 1984, pp. 34-43.

USAF70 USAF Select Committee on Computer Technology Potential. "An Air Force study of Air Force Organizational Ability to Exploit and Manage Computer Technology." 1970.

USAF76 USAF Regulation 300-10, December 1976. Section 4, Policy, specified that "an Air Force standard high order programming language will be exmployed in all future Air Force systems [4.h]. These were designated as standard high order programming languages: (1) COBOL; (2) FORTRAN [ANSI X3.9-1966]; (3) JOVIAL (J3) [MIL-STD-1588 (USAF)]; (4) JOVIAL (J73/I) [MIL-STD-1589 (USAF)]; and (5) PL/I [ANSI X3.53-1976].

USAF83 USAF Scientific Advisory Board, "Report of the USAF Scientific Advisory Board Ad Hoc Committee on The High Cost and Risk of Mission-Critical Software." December 1983.

USAF85 USAF, AF Regulation 700-9, Vol. I, Attachment 4, 15 March 1985. Section A, 2-4.b specified that certain Air Force standard programming languages will be used. They are: (a) Ada [ANSI/MIL-STD-1815]; (b) COBOL [ANSI X3.23-1974]; (c) FORTRAN [ANSI X3.9-1978, and DOD supplement MIL-STD-1753]; and (d) JOVIAL (J73) [MIL-STD-1589, JOVIAL (J73) (USAF)].

Wass81 Wasserman, Anthony I. (ed.). Tutorial: Software Development Environments. New York: Institute of Electrical and Electronics Engineers, Inc. IEEE Order No. EHO 187-5), 1981.

Werl83 Werling, P. R. Alternative Models of Organizational Reality: The Case of Public Law 89-3Ø6 [the Brooks Act]. D.P.A. Dissertation submitted to the University of Southern California School of Public Administration. 1983.

Wien84 Wiener-Ehrlich, W. K., et. al. "Modelling Software Behavior in Terms of a Formal Life Cycle Curve: Implications for Software Maintenance." IEEE Transactions on Software Engineering. Vol. SE-1Ø, No. 4, July 84, pp. 376-383.

Wild83 Wilder, W. L. "Minimal Host for the KAPSE." ACM Ada Letters. Vol. III, No. 2, Sept/Oct 1983.

Wino79 Winograd, T. "Beyond Programming Languages." In Communications of the ACM. 22:7 (July 1979), pp. 391-4Ø1.

Zelk84 Zelkowitz, Marvin V., et. al. "Software Engineering Practices in the US and Japan." Computer. June 1984, pp. 57-65.

CHAPTER 3

WHAT WOULD A HAPSE LOOK LIKE? -- RESEARCH RESULTS

A. INTRODUCTION

In this chapter we describe the Hypothetical Ada Programming Support Environment (HAPSE) which we defined in this research project. We also present the intermediate results of this work: (1) we first identified the principle technical elements of a single integrated Ada-based software support environment; then, (2) verified that it is now technically feasible to integrate them into a "HAPSE".

The chapter contains five sections. In B, we refresh our memory of the goals and objectives for which the GFE/HAPSE was proposed. In C, Software Tool Technology, we describe the functions served by software tools required during the various phases of the complete MCCR life cycle. We focus on tool functions, using the taxonomy of FIPS 99. This simplifies the task of comprehending the significance of hundreds of individual software tools now available.

In section D, Environments, we identify four discrete types of software support environments (each containing several separate software tools) and indicate the type environment that is now most suitable for a HAPSE. In section E we describe how we selected and prioritized the tool capabilities to be included in the HAPSE. Finally, in section F we describe Environment Technology and the HAPSE itself.

B. GOALS AND OBJECTIVES

The driving force for this work is the compelling need for control of total life cycle cost for systems in which software is embedded.

The major targets of research with respect to a single integrated automated environment were improvements in: (1) productivity of the work force required to develop and support mission-critical software; (2) reliability of the software produced (and, where appropriate of the system in which the software is embedded); and (3) maintainability of the software produced across the entire operational life cycle.

In this document the terms "productivity", "reliability", and "maintainability" are used in conventional engineering senses.

For example, the term "productivity" is defined as "unit output" divided by "unit input", or "output/input". The output is delivered source instructions (DSI) of quality software produced. The input is taken as employee work-months (W-M) required to produce that output. The resulting measure, defined as "productivity" and useful for planning and budgeting, is:

$$\frac{\text{Delivered source instructions}}{\text{work-month}} \quad \text{or} \quad \frac{\text{DSI}}{\text{W-M}}$$

Improvements in productivity can take the form of shorter development times, fewer resource inputs, or of improved product reliability and quality levels produced with the same resource inputs. The integrated automated programming support environment is considered as a vital tool for reaching all three targets.

C. SOFTWARE TOOL TECHNOLOGY

In terms of the technology of software tools, there is no tool gap. We found that ample software tools exist. The sample of tools listed in Table 3-1 (drawn from NBS Special Publication 500-88) have features that are applicable and useful for every stage of the software life cycle. Figure 3-1 shows the distribution of these tool functions across the software life cycle.

1. Existing tools cover the entire software life cycle

Today it is possible to use existing tools for initial development and post-deployment support through the complete life cycle. In terms of the HAPSE, this means that today it is feasible technically to build a HAPSE that will support DoD software throughout the full life cycle.

We do not mean that today's tools, which are written in many different languages, for different computers, and with many different command languages, can be integrated easily. Later in this report (in sections E and F, and in chapter 4) we address technology for integrating the tools needed for a GFE/Environment.

TABLE 3-1

A SAMPLE OF SOFTWARE TOOLS AVAILABLE

54 Requirements/Design Specification and Analysis Tools.

- ADF
- AFFIRM
- ARTS
- AUTO-DBO
- AUTOIDEFO
- CADSAT
- CARA
- CBLSHORT
- COBOL/SP
- CONFIGURATOR
- CRISPFLOW
- CS4
- DARTS
- DATA DESIGNER
- DECA
- DQM
- FAME
- FOSTRA
- IORL
- IPDS
- ISDS
- LOGICFLOW
- MED-SYS
- MEDL-R
- MEDL-D
- MSL
- MTR
- NETWORK PLANNER
- PBASIC
- PDL
- PDS
- PERCAM
- PIDGIN-FASP
- PSL/PSA
- RA
- RTT
- SARA
- SCG
- SCG/DOM
- SCHEMACODE
- SCOPE
- SDDL
- SDL
- SDP/MAYDA
- SIGS
- SPECLE/DARS
- SREM
- SREP
- SRIMP
- STAG/TEMS
- STRUCTURE(S)
- SYDIM
- TRANSFOR
- XAS8

13 Software Modeling and Simulation Tools

- AISIM
- ASRP
- BEST/1 (TM)
- CRYSTAL (TM)
- DAS
- DDRP
- DPAD
- HARDWARE
- SIMULAMEDL-P
- POD
- SALSIM
- SCERT
- SDVS

36 Program Construction and Generation Tools

- ADA-ATOM
- ADA COMPILER
- CHILL TRANS
- COBOL/SPP
- COGENT
- COPE (TM)
- CSPP
- DI-3ØØØ
- FOCUS
- GRAFMAKER
- IFTRAN (TM)
- INFORM
- JOCIT
- MAGLE
- MARK IV (TM)
- MEFIA
- METRAN
- MODULE ORDERER
- PERLUETTE
- PROGRAM GENERAT
- QUIKCODE
- RATCODER
- RATFOR
- S-FORTRAN
- SCOBOL (TM)
- SFORT-1
- SFTRAN3
- SMAL/8Ø
- SMMA
- SRTRAN.BASELINE
- STRUC1/STRUC2
- STRUCTURIZER
- SURGE 72
- SYSTEM-8Ø
- TAB4Ø
- UCSD P-SYSTEM
- YACC

SOURCE: National Bureau of Standards Special Publication 5ØØ-88, 1982. Listings shown are from Appendix A. Tool Abstracts for all tools listed are given in Appendix N.

TABLE 3-1, continued

12 Software Support System/Programming Environment Tools

ADA ENVIR'T	ARGUS/MICRO	ASSET	COBOL/ADE
FASP	LILITH	MSEF	PWB FOR VAX/VMS
SEF	SOFTOOL 8Ø (TM)	TOOLPACK	VIRTUAL OS

126 Source Program Analysis and Testing Tools

ADS	AMPIC	ASSIST-I	ATA-FASP
ATA-SAI	ATDG	ATTEST	AUDIT
AUDITOR	BSC	CA	CADA
CASEGEN	CAVS	CCA	CCREF
CCS	CENSUS	CGJA	CICS DUMP ANALY
COBOL/DV	COBOL STRUCT	COBOL TRACING	COBOL/QDM
COBOL OPTIMIZ	COBOL TESTING	COBOL/CP	COMMAP
COMSCAN	CORE	COTUNE II	CPA-ADR
CQD	DAVE	DRIVER	DYNA
EAVS	ECA AUTOMATION	EFFIGY	ENFORCE
EVP	EXPEDITER	FACES	FADEBUG-I
FAST	FAVS	FCA	FORAN
FORTRAN TRACING	FORTRAN TESTING	FORTRAN OPTIMIZ	FTN-77 ANALYZER
FTN ANALYZER	FTNXREF	GENTESTS	GENTEXTS
GOTO-ANALYZER	HAWKEYE (TM)	INSTRU	INTERFACE DOCUM
ITB	JAVS	JIGSAW	JOVIAL TCA
JOVIAL/J3SC	JOVIAL/VS	JOYCE	LOGIC
MENTOR	MONITOR	NASA-VATS	NODAL
NUMBER	OPTIMUS	OPTIMIZER II	OSCYBR
PACE	PACE-C	PET	PPE
PREF HDR GEN	PROGLOOK	RADC/FCA	REALIGNMENT SYS
REFLECT II	REFORM	REFTRAN (TM)	REL MEAS MODEL
RISOS TOOLS	RXVP8Ø (TM)	SADAT	SAP
SAP/H	SARA-U	SARA-H	SARA-III
SARA-IV	SCAN/37Ø	SELECT	SPTRAN
SSA	STAT ENT & EVAL	STRUCTURING ENG	STRUCT
SUBCRS	SURVAYOR	SUS	SYDOC
SYMCRS	SYSTEM MONITOR	SYSXREF	TAFIRM
TATTLE	TCAT	TDEM	TEST PREDICTOR
TEVERE-1	TFA	THE ENGINE	TIMECS
TIMER	TPT	TRAILBLAZER	TSA/PPE
UCA	XPEDITER		

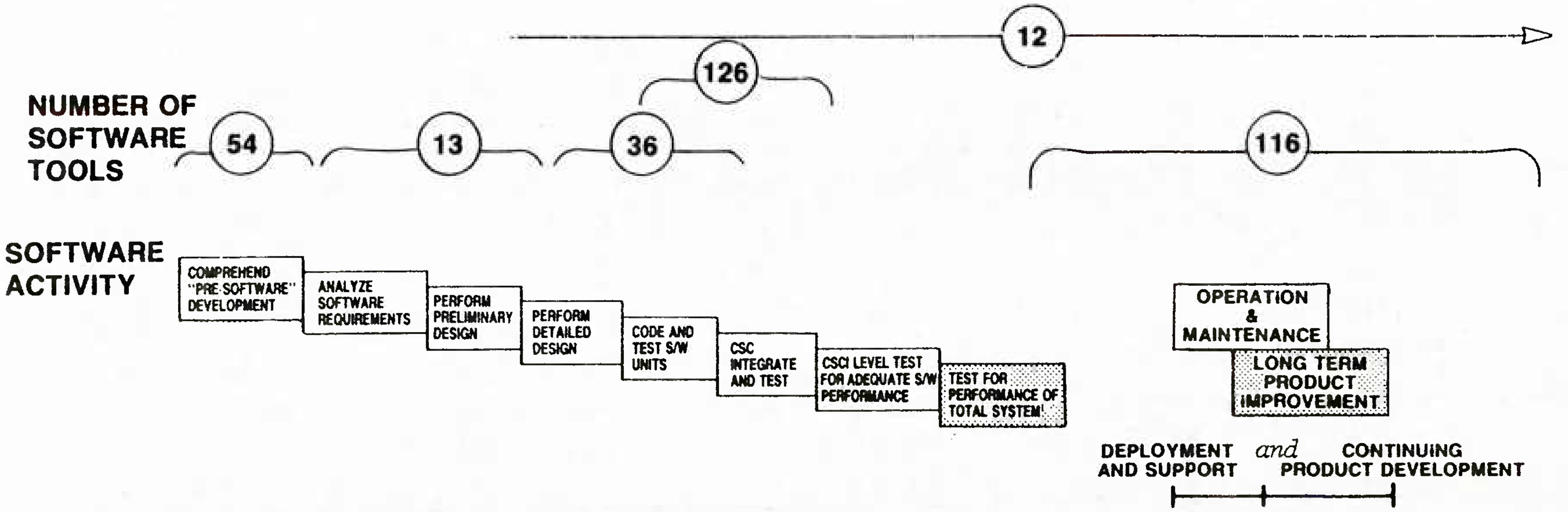
SOURCE: National Bureau of Standards Special Publication 5ØØ-88, 1982. Listings shown are from Appendix A. Tool Abstracts for all tools listed are given in Appendix N.

TABLE 3-1, concluded.

116 Software Management, Control, and Maintenance Tools

ABS	ACT/1	ADS/CERL	AFS
ALIAS	ASA-PMS	ASC	ASEQ
AUTDOC	AUTOCOM	AUTOFLOW/TRW	AUTOFLOW(TM)*
AUTOMATIC DOC	AUTORETEST	BLKGEN-BDD	BLKGEN/SPECPN
BUDGET VS ACT	CADMUS	CALLREF	CAPTURE/MVS(TM)
CHECKSUM	CONDIM	COMGEN	COMGEN/TRW
COMLIST/TRW	COMLIST	COMPARE	COMSORT
COMSTAR	CONFIG	CPA	CPAL
CROREF	CTC	CUE	DA
DATAMACS	DCD	DECKBOY COMPAR	DEPCHT
DICTANL/LOCA	DIFFS (TM)	DIRCOM	DOCGEN
DOCU/TEXT	DOCUMENTER	DOCUMENTER A	DOCUMENT
DOCUMENTOR	DOSSIER	DPNDCY	EASYTROL
ESAP	FLOBOL	FLODIA	FLOWGEN
FORMAN	FORREF	FORTREF	FTNCODER
GADTR AID	GIM/GIM II	GIRAFF	HARP
INFORM/REFORM	INSERT	ISUS	JET
JSDD	LANG INSTRUCTOR	LAYOUT	LEXICON
LIBRARIAN	LIBREF	LOGIFLOW	LOGOS
LOOK	MEDL-X	MEMORY MNG LIB	MPS
N55ØØ	N-SQUARED	NUMBER/DEC	ONLINE ASSIST
PAC II	PDS FLOW	PDSS	PFORT
PFS	PMCS	PMS IV	PPP
PROG COMP ANAL	PRONET	PSL	QCM
QCRT	QUICK-DRAW	REFER	RENAME
SDP	SLIB	SLIM	SMS
SMT	SNOOP	SPC	SPEAR
SPECTRUM-1	SPELL	SPREAD	SPRINT
TAPS/AM	TDBCOMP	TIDY	TOOLS DATABASE

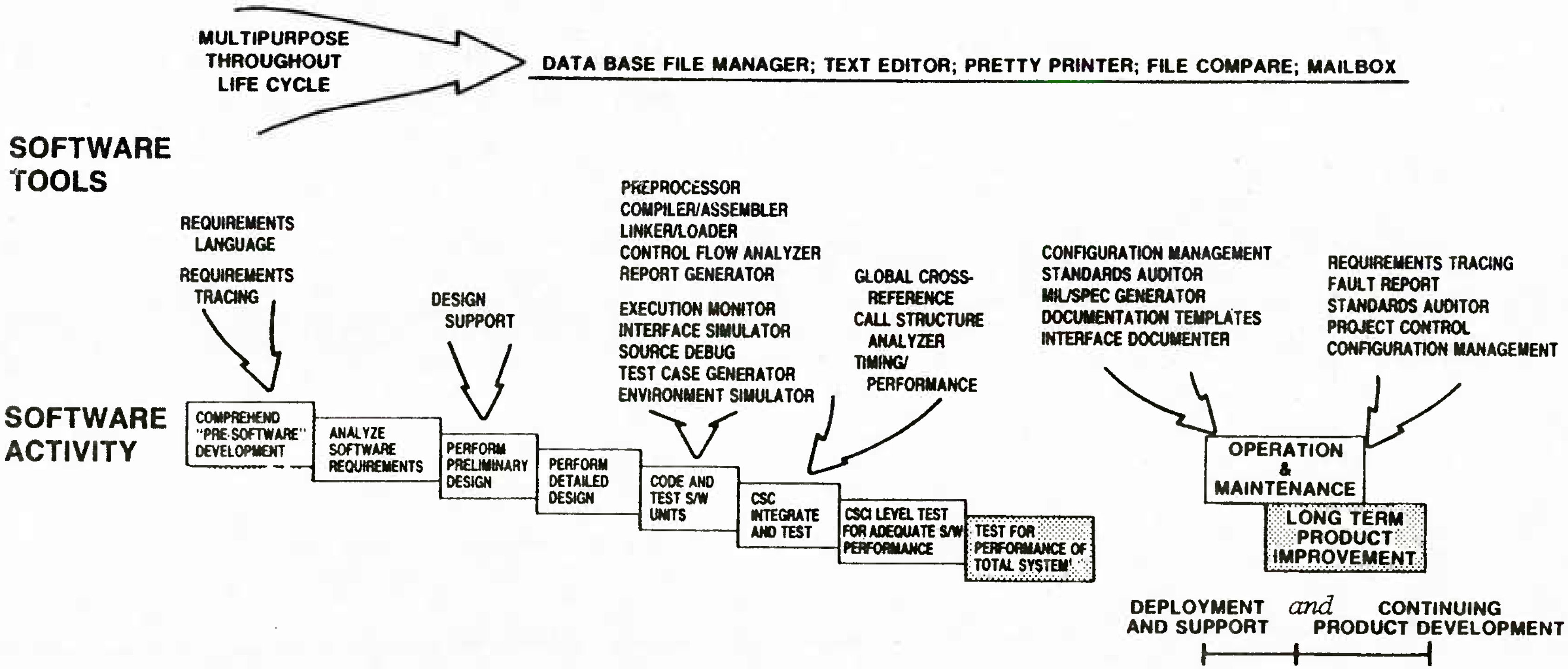
SOURCE: National Bureau of Standards Special Publication 5ØØ-88, 1982. Listings shown are from Appendix A. Tool Abstracts for all tools listed are given in Appendix N.


12
126
NUMBER OF SOFTWARE TOOLS
54
13
36
116
SOFTWARE ACTIVITY
COMPREHEND "PRE-SOFTWARE" DEVELOPMENT
ANALYZE SOFTWARE REQUIREMENTS
PERFORM PRELIMINARY DESIGN
PERFORM DETAILED DESIGN
CODE AND TEST S/W UNITS
CSC INTEGRATE AND TEST
CSCI LEVEL TEST FOR ADEQUATE S/W PERFORMANCE
TEST FOR PERFORMANCE OF TOTAL SYSTEM
OPERATION & MAINTENANCE
LONG TERM PRODUCT IMPROVEMENT
DEPLOYMENT AND SUPPORT
and
CONTINUING PRODUCT DEVELOPMENT

SOURCES: NBS Special Publication 500-88, Apps A-J
J.M. Fox, Software and Its Development; Final Report of the Joint Logistics Commanders' Workshop on Post Deployment Software Support (PDSS) for Mission-Critical Computer Software, Vol. I, June 1984, p. 1-3; Report of the USAF Scientific Advisory Board ad hoc Committee on the High Cost and Risk of Mission-Critical Software; and Dr. Richard Warling, Technion International.

Figure 3-1. Distribution of Sample Tools across Life Cycle.

Table 3-1 listed software tools, and Figure 3-1 showed the distribution of these tool functions across the software life cycle. In contrast, Figure 3-2 identifies some specific tool capabilities along with the life cycle phase in which they are most used. Table 3-2 gives definitions of tool capabilities indicated in Figure 3-2.



SOURCES: J.M. Fox, Softwara and Its Development; Final Report of the Joint Logistics Commanders' Workshop on Post Deployment Software Support (PDSS) for Mission-Critical Computer Softwara, Vol. I, June 1984, p. 1-3; Report of the USAF Scientific Advisory Board ad hoc Committee on the High Cost and Risk of Mission-Critical Software; and Dr. Richard Werling, Technion International.

Figure 3-2. Software Tool Capabilities, by Life Cycle Phase.

TABLE 3-2

DEFINITIONS OF SOFTWARE TOOLS

Requirements Phase

Requirements Language

A formal language, which may be graphical and/or textual in nature. A requirements analyzer can check the requirements as expressed in the requirements language, for syntactical errors in the requirements specifications and then produce a useful analysis of the relationships between system inputs, outputs, processes, and data. Logical inconsistencies or ambiguities in the specifications can also be identified by the requirements analyzer.

Requirements Tracing

Requirements tracing provides a means of verifying that the software of a system addresses each major requirement of that system and that the testing of the software produces adequate and appropriate responses to those requirements.

Design Phase

Design Support

Software tools for design support aid in the synthesis, analysis, modeling, or documentation of a software design. Examples include simulators, analytic aids, design representation processors, and documentation generators.

Development Phase

Preprocessor

A computer program that preprocesses source code, part of which may be unacceptable to another program, to generate equivalent code that is acceptable to the program. An example is a preprocessor which converts structured FORTRAN to ANSI-standard FORTRAN.

TABLE 3-2, continued

Ada Translator	A program that transforms a sequence of Ada language statements into object code.
Ada Interpreter	A program (which may be software, hardware, or "firmware") that translates and executes each Ada source language statement of a computer program before translating and executing the next statement.
Ada Compiler/ Assembler	A computer program used to translate from Ada language or assembler language statements into machine executable form (object code).
Linker/Loader	A computer program used to create one load module from one or more independent modules by resolving cross-references among the modules, and reads the load module into main storage prior to its execution.
Control Flow Analyzer	A computer program that analyzes the flow of control through another computer program.
Report Generator	A computer program that generates computer instructions for extracting data from a data base and preparing reports from the data. The program prepares computer code, using instructions from a few descriptive statements, rather than from source language statements.
Execution Monitor	A program that monitors execution of another program, instruction by instruction, as the program runs.
Interface Simulator	A device or computer program that represents certain features of a hardware, software, or data base with which a system or system component must interface.

TABLE 3-2, continued

Source Debugger	Interactive test aids, which are used in the process of locating, analyzing, and correcting suspected faults in computer programs. Debuggers are used to assist in identifying and isolating program errors. Tools allow the user to: ● suspend program execution at any point to examine program status, ● interactively dump the values of selected variables and memory locations ● modify the computation state of an executing program, ● trace the control flow of an executing program.
<u>Test Phase</u>	
Test Case Generator	A software tool that accepts as input a computer program and test criteria, and generates test input data that meet the criteria. A fully automated test generator may also determine the results of running the test data.
Environment Simulator	A device or computer program that represents certain features of the environment in which a computer system will function (e.g., temperature, vibration, atmospheric pressure, g-loads, etc.).
Test Tools	Computer programs used in the process of exercising or evaluating a system or system component to verify that it satisfies specified requirements or to identify differences between expected and actual results.
Global Cross-reference generator	Computer programs which produce lists of data names and labels showing all of the places they are used in a program.

TABLE 3-2, continued

Call Structure Analyzer	Computer program which analyzes control structure of programs. Analyzes process by which one program invokes another, passes parameters to it, and receives results back. Analyzer helps detect some types of improper subprogram usage and violation of control flow standards. Also identifies control branches and paths used by test coverage analyzers.
Statement Coverage Analyzer	Special case of test coverage. Test coverage analyzers monitor the execution of a program during program testing in order to measure the completeness of a set of program tests. Completeness is measured in terms of the branches, statements or other elementary program constructs which are used during the execution of the program over the tests.
Performance Evaluation Tools	Computer programs that aid in technical assessment of a system or system component to determine how effectively the operating objectives have been achieved.
Timer/Performance Analyzer	A software tool that estimates or measures the execution time of a computer program or portions of a computer program either by summing the execution times of the instructions in each path, or by inserting probes at specific points in the program and measuring the execution time between probes.
<u>Management Function</u>	
Configuration Management	The process of identifying and defining the configuration items in a system, controlling the release and change of these items throughout the system life cycle, recording and reporting the status of configuration items and change requests, and verifying the completeness and correctness of configuration items.

TABLE 3-2, continued

Standards Auditor	A computer program used to examine source code, which automatically determines whether prescribed programming standards and practices have been followed.
Data Dictionary	A collection of the names of all data items used in a software system, together with relevant properties of those items; for example, length of data item, representation, etc. Useful as a standardization tool.
MIL/SPEC Generator	A computer program that verifies a program's compliance with specified MIL/SPECs.
Library Management	Management of a software library, i.e., a controlled collection of software and related documentation designed to aid in software development, use, or maintenance. Types include software development library, master library, production library, program library, and software depository.
Project Control	Control of project to ensure completion of specified product on schedule and within budget.
Documentation Management	Management and control of technical data or information, including computer listings and printouts, in human-readable form, that describe or specify the design or details, explain the capabilities, or provide operating instructions for using the software to obtain desired results from a software system.
Documentation Template	A pattern used to simplify and speed the preparation of documentation; data entry may be limited to "filling in the blanks".

TABLE 3-2, concluded

Interface Documentor	A computer program that helps document and describe characteristics of the hardware, software, or data base elements with which the system or system component must interface.
Data Base File Manager	A computer program that facilitates storage and retrieval of sets of data fundamental to a system.
<u>Documentation</u>	
Text Editor	A computer program that permits selective revision of computer-stored data.
Editor, Syntax-Directed	Interactive text editor, adapted to a specific programming language so that it verifies correctness of syntax required by that language as the programmer writes his instructions.
Graphics Generator	Computer program that generates graphic representations from instructions provided by operator.
Text Formatter	Program which takes text material and modifies it into prespecified format.
Typesetter	Program which takes written material and prepares instruction code for use by typesetting equipment.
Speller	Program which checks spelling of words in text, to verify correctness.
On-Line Help	Explanations or instructions, which the programmer can call up with a keystroke, and which then appear on the screen for study. May include explanations, reminders, and suggestions appropriate to the function for which help is requested.
Menus	Lists of options available to programmer, which appear on the screen. Programmer makes selection from the options listed.

2. Analysis using Tool Taxonomy of FIPS 99

In concluding that it is feasible today to build a HAPSE that can support DoD software throughout the full software life cycle, we have made a strong statement. We support it in Figures 3-3 to 3-6 (pages III-17 to III-21), in which we which summarize the results of our research on tool technology according to the taxonomy of tools used by the National Bureau of Standards [FIPS 99; Houg82a; Houg82b]. The NBS taxonomy describes software tools in terms of four tool functions:

- transformation;
- static analysis;
- dynamic analysis; and
- project management.

Each of the four functions is shown on a separate chart, (Figures 3-3 to 3-6) with specific features identified. For example, Figure 3-4, "Tool Technology--Static Analysis", is subdivided to show these features of software tools: Auditing; Comparison; Measures of Completeness; Completeness Checking; Consistency Checking; Cross-Reference Checking; Data Flow Analysis; Error Checking; Interface Analysis; Scanning; Statistical Analysis; Structure Checking; Type Analysis; Units Analysis; and Input/Output Specification Analysis.

Across the top of figures 3-3 to 3-6 we show ten life cycle activities. These activities are:

- SYSTEM Management	Management of Defense System (which contains software components)
- SYSTEM Requirements	Determine requirements for defense system
- SOFTWARE Requirements	Determine requirements for software in defense system
- Software Design	Allocate functions to software modules, develop strategies for coding and testing
- Code and Compile Software Modules	Write, compile, and debug individual modules
- Test Software Modules	Test Modules separately before integration

- Software Integration, Software System Test	Test software modules together
- Post-deployment Evolutionary Enhancement	Longest phase of system life; Operational use and enhancement
- Documentation	To aid in operational use and subsequent enhancement [This placement of Documentation is for convenience only. It should not be taken as implying that documentation is delayed until the end of the development cycle].
- Software Configuration Management	To control software as it changes during its many successive versions

Tool Technology-"Transformation". Software tools take text and data, then transform them in some way. Transformation tool functions change text, code, or data from one form to another. The left column of Figure 3-3 shows the most important of these transformations. Some sort of tool function is needed in every one of the life-cycle activities. In Figure 3-3, for example, the most common transformations, editing and formatting, occur in nearly every activity in the life cycle. Transformation tools are used most in the four activities of software design; code and compile; software integration and systems test; and post-deployment continuing development.

Tool Technology-"Static Analysis". Static Analysis tools are used to audit, compare, and check for completeness, consistency, and accuracy. They are needed for verifying program structure, data flow and interfaces. These tools are indispensable aids in analyzing code, requirements languages, design languages, and other fixed formats such as graphics. Their outputs are error reports, diagnostics, and other forms of documentation.

The Figure shows 15 functions of software tools. They are applied most often in the activities involving system management, determination of requirements, software design, code and compile, and in post-deployment continuing development.

Tool Technology-"Dynamic Analysis". Figure 3-5 shows dynamic analysis tools, which are used to analyze the behavior of specification languages and program code during and after execution. As system requirements and software technology have become more complex, these tools have become absolutely indispensable. Figure 3-5 includes the tools of greatest value to programmers and those assuring quality of the resulting software.

LIFE CYCLE ACTIVITIES / TOOL FUNCTION	SYSTEM MANAGEMENT	REQUIREMENTS		SOFTWARE DESIGN	CODE AND COMPILE	TEST MODULES	SOFTWARE INTEGRATION AND SYSTEMS TEST	POST DEPLOYMENT CONTINUING DEVELOPMENT	DOCUMENTATION	SOFTWARE CONFIGURATION MANAGEMENT
		SYSTEM	SOFTWARE							
TRANSFORMATION	●	●	●	●	●	●	●	●	●	●
EDITING	●	●	●	●	●			●	●	●
FORMATTING	●	●	●	●	●			●	●	●
INSTRUMENTATION						●	●			
OPTIMIZATION					●		●			
RESTRUCTURING				●	●		●			
TRANSLATION					●		●	●		●
ASSEMBLY					●		●	●		●
COMPILATION					●		●	●		●
MACRO EXAMINE				●	●					
STRUCTURE PREPARING				●						
SYNTHESIS				●	●					

SOURCE: TECHNION INTERNATIONAL, INC.

Figure 3-3. Tool Technology -- "Transformation"

LIFE CYCLE ACTIVITIES / TOOL FUNCTION	SYSTEM MANAGEMENT	REQUIREMENTS		SOFTWARE DESIGN	CODE AND COMPILE	TEST MODULES	SOFTWARE INTEGRATION AND SYSTEMS TEST	POST-DEPLOYMENT CONTINUING DEVELOPMENT	DOCUMENTATION	SOFTWARE CONFIGURATION MANAGEMENT
		SYSTEM	SOFTWARE							
STATIC ANALYSIS	●	●	●	●	●	●	●	●	●	●
AUDITING	●	●	●	●	●	●	●	●	●	●
COMPARISON	●	●	●	●	●	●	●	●	●	●
COMPLETE MEAS	●			●	●			●	●	
COMPLETENESS CHKING	●	●	●	●	●			●	●	
CONSISTENCY CHK	●	●	●	●	●			●	●	
CROSS REFERENCE	●	●	●	●	●			●	●	
DATA FLOW ANALYSIS	●			●	●			●	●	
ERROR CHECKING	●	●	●	●	●			●	●	
INTERFACE ANALYSIS	●			●	●			●	●	
SCANNING	●	●	●	●	●			●	●	
STATISTICAL ANALYSIS	●				●			●	●	
STRUCTURE CHECKING	●			●	●			●	●	
TYPE ANALYSIS	●				●			●	●	
UNITS ANALYSIS	●			●	●			●	●	
I/O SPEC ANALYSIS	●					●	●			

SOURCE: TECHNION INTERNATIONAL, INC.

Figure 3-4. Tool Technology -- "Static Analysis"

LIFE CYCLE ACTIVITIES / TOOL FUNCTION	SYSTEM MANAGEMENT	REQUIREMENTS		SOFTWARE DESIGN	CODE AND COMPILE	TEST MODULES	SOFTWARE INTEGRATION AND SYSTEMS TEST	POST DEPLOYMENT CONTINUING DEVELOPMENT	DOCUMENTATION	SOFTWARE CONFIGURATION MANAGEMENT
		SYSTEM	SOFTWARE							
DYNAMIC ANALYSIS	●	●	●	●	●	●	●	●	●	●
ASSERTION CHECKING	●					●	●	●	●	
CONSTRAINT EVALUATION	●					●	●	●	●	
COVERAGE ANALYSIS	●						●	●	●	
RESOURCE UTILIZATION	●									
SIMULATION	●	●	●	●					●	
SYMBOLIC EXECUTION	●			●	●	●	●	●	●	
TIMING	●					●	●	●	●	
TRACING	●				●	●	●	●	●	
BREAKPOINT CONTROL	●				●	●	●	●	●	
DATA FLOW TIMING	●				●	●	●	●	●	
PATH FOW TIMING	●				●	●	●	●	●	
TUNING	●				●	●	●	●	●	
REGRESSION TESTING	●					●	●	●	●	●

SOURCE: TECHNION INTERNATIONAL, INC.

Figure 3-5. Tool Technology -- "Dynamic Analysis"

LIFE CYCLE ACTIVITIES / TOOL FUNCTION	SYSTEM MANAGEMENT	REQUIREMENTS		SOFTWARE OESIGN	CODE AND COMPILE	TEST MODULES	SOFTWARE INTEGRATION ANO SYSTEMS TEST	POST OEPLOYMENT CONTINUING OEVELOPMENT	OOCUMENTATION	SOFTWARE CONFIGURATION MANAGEMENT
		SYSTEM	SOFTWARE							
MANAGEMENT	●	●	●	●	●	●	●	●	●	●
CONFIGURATION CONTROL	●	●	●	●	●	●	●	●	●	●
INFORMATION MANAGEMENT	●	●	●	●	●	●	●	●	●	●
OATA OICTIONARY	●	●	●	●	●	●	●	●	●	●
OOCUMENTATION MGT	●	●	●	●	●	●	●	●	●	●
FILE MGT	●	●	●	●	●	●	●	●	●	●
TEST OATA MGT	●	●	●	●	●	●	●	●	●	●
PROJECT MGT	●	●	●	●	●	●	●	●	●	●
COST ESTIMATION	●								●	
RESOURCE ESTIMATION	●								●	
SCHEOULING	●								●	
TRACKING	●	●	●	●	●	●	●	●	●	●

SOURCE: TECHNION INTERNATIONAL, INC.

Figure 3-6. Tool Technology -- "Management"

Tool Technology-"Management". The tools in Figure 3-6 aid various levels of management in managing and controlling software projects in development and post-deployment support. They include tools for managing quality and consistency, as well as tools for managing costs and schedules. For example, they include the customary cost estimating, scheduling and tracking tools. This category also includes tools used for configuration management, documentation management, and data dictionaries.

3. Tool Functions Found in Typical Environments

Figure 3-6A shows tool capabilities found in four existing environments, as well as those planned for the DoD APSE. Highlights of the display are:

- None of the five environments has tools for the software integration and systems test phase (usually done with simulation, and instrumented hardware/software "test beds"), or for software configuration management.
- The Boeing "ARGUS" and TRW "SPS" environments have capabilities in all but software integration and systems test phases. We note that one could argue that SPS (and even ARGUS) supports these phases.
- The UNIX environment is well supplied with tools for the programming ("code and compile"), documentation, and continuing development phases.
- The ALS and DoD APSE plan for capabilities in each life cycle phase in which the ARGUS and SPS systems now have capability.

4. Prioritizing Tool Capabilities

How can we use the information presented in such detail in figures 3-3 to 3-6? A survey of software professionals [Houg82a, appendix] helped to assign priorities for the important software tool capabilities. The respondents ranked tool functions according to four priorities: minimal; required; important; and useful. These responses, which we used in defining the HAPSE, are indicated in Table 3-3, "Priorities for Tool Capabilities."

LIFE CYCLE FUNCTIONS ENVIRONMENT	SYSTEM MANAGEMENT					REQUIREMENTS		SOFTWARE DESIGN
						SYSTEM	SOFTWARE	
	CONFIGURATION MANAGEMENT	PROJECT CONTROL	FAULT REPORT	STANDAROS AUDITOR	GRAPHICS GENERATOR	REOUIREMENTS LANGUAGE	REDUIREMENTS TRACING	DESIGN SUPPDRT
DoO APSE	●							
ALS	●							
BOEING "ARGUS"		●		●	●			●
TRW "SPS"	●	●				●	●	●
UNIX								

LIFE CYCLE FUNCTIONS ENVIRONMENT	CODE AND COMPILE						TEST MOOULES						
	CDMPILER/ASSEMBLER	LINKER/LOAOER	REPORT GENERATOR	PREPROCESSOR GENERATOR	CONOITIONAL COMPILATION	CONTRDL FLOW ANALYZER	STATEMENT EXECUTION MONITOR	INTERFACE SIMULATOR	SDURCE DEBUG	FORMAL VERIFICATION SYSTEM	TEST CASE GENERATOR	INSTRUCTION LEVEL SIMULATOR	ENVIRONMENT SIMULATDR
DoO APSE	●	●					●	●	●				
ALS	●	●											
BOEING "ARGUS"	●	●	●	●									
TRW "SPS"	●	●	●	●					●				
UNIX	●	●		●	●				●				

LIFE CYCLE FUNCTIONS ENVIRONMENT	SOFTWARE INTEGRATION ANO SYSTEMS TEST		POST OEPLOYMENT CONT'NUING DEVELOPMENT					
	SIMULATIDN	INBTRUMENTED TEBT BED	GLOBAL CROSS-REFERENCE	DISASSEMBLER	CALL STRUCTURE ANALYZER	TIMING/PERFORMANCE ANALYSIS	OPTIMIZATION/TUNING ANALYSIS	REUSABLE CDOE LIBRARY
DoO APSE			●			●		
ALS								
BOEING "ARGUS"								
TRW "SPS"	●				●			
UNIX			●			●		

LIFE CYCLE FUNCTIONS ENVIRONMENT	OOCUMENTATION							SOFTWARE CONFIGURATION MANAGEMENT				
	MIL-SPEC GENERATOR	WORO PROCESSING	TYPESETTER	TEXT PRIMITIVES	SPELLER	DOCUMENTATION TEMPLATES	INTERFACE DOCUMENTS	AOA LIBRARY·REUSABLE CODE	APPLICATION·VERSION CONTRDL	LIBRARY	OATA OICTIONARY	SPECIFICATIDN MANAGEMENT
DoO APSE	●											
ALS		●										
BOEING "ARGUS"	●	●		●		●	●					
TRW "SPS"		●			●				●			
UNIX		●	●	●	●							

LIFE CYCLE FUNCTIONS ENVIRONMENT	MULTIPURPOSE TOOLS				
	DATA BASE FILE MANAGER	EDITDR	PRETTY PRINTER	FILE CDMPARE	MAIL BOX
DoO APSE	●	●	●	●	
ALS	●	●			
BOEING "ARGUS"	●	●			●
TRW "SPS"	●	●		●	●
UNIX	●	●	●	●	●

BOURCE: TECHNION INTERNATIONAL, INC.

Figure 3-6A. Tool Functions Found in Typical Environments.

TABLE 3-3

Priorities for Tool Capabilities.

Tool Features	Minimal	Required	Important	Useful
Transformation	X			
Formatting	X			
Optimization		X		
Compilation	X			
Instrumentation			X	
Editing	X			
Syntax direction			X	
Input-Output		X		
On-Line Assistance		X		
Error Assistance		X		
On-Line Tutor				x
Definition Assistance				x
Menu Assistance			X	
Static Analysis		X		
Type Analysis		X		
Interface Analysis		X		
Statistical Profiling				x
Cross-Reference		X		
Auditing			X	
Complexity Measurement				x
Completeness Checking				x
Reference Analysis				x
Dynamic Analysis		X		
Timing Analysis			X	
Tuning Analysis			X	
Tracing/Debugging		X		
Regression Testing			X	
Assertion Checking				x
Coverage Analysis			X	
Management		X		
Configuration Control		X		
Information Management		X		
Ada Library Management		X		
Specification Management			X	
Data Dictionary Management			X	
Ada Package Management				x
Test Management			X	
Project Management			X	
Cost Estimation			X	
Scheduling			X	
Tracking			X	

Source:
Houghton, Ada/APSE Taxonomy, NBSIR-2625,"A Taxonomy of Tool Features for the Ada* Programming Support Environment (APSE)," 1983.

D. FOUR TYPES OF ENVIRONMENTS

In the previous section, we focused on software tool technology, using the taxonomy of FIPS Publication 99. This taxonomy is useful for undertanding the wide variety of functions that software tools must satisfy in the software life cycle.

1. Descriptions of Environments.

We now report on our survey of programming support environments. We found that contemporary environments can be categorized into four definite types. We begin by defining the four types and providing examples of environments that fit into each type.

a. "PROGRAMMING" environments support the programming and testing phases of the software life cycle. They usually support only one programming language. Examples include: ALS [Army's Ada language system]; Arcturus; and Smalltalk [Technion International].

b. "FRAMING" environments concentrate on the earliest tasks in developing defense systems and software, the activities of systems definition and software definition that occur before programming can begin. Framing environments usually support only one specific methodology. Examples of these include Riddle's DREAM [Ridd81] and Wasserman's USE [Wass83]. Framing environments are typically used by research organizations and, for the most part, have not migrated out of these organizations.

c. "GENERAL" environments support all phases of the software life cycle. They sometimes include specific software tools for some life cycle activities. They usually support more than one programming language and do not require users to follow one specific methodology. They tend to include a "toolbox" from which users can choose a software tool to support specific activities. Examples include: TRW's Software Productivity System (SPS) [Boeh84], Boeing's ARGUS [Stuc83], UNIX, and the French "Platine" system [Hunk81].

d. "METHODOLOGY-SPECIFIC" life cycle environments. This type of environment would provide the greatest control, in terms of conformity to a prescribed methodology for development and support of software throughout the life cycle. Such environments would be designed around, and dependent on, a specific methodology for developing and supporting software. No environment of this type exists beyond the conceptual stage. In their "Methodman I" report, Freeman and Wasserman surveyed 24 software development methodologies. They found that six of the 24 support the entire life cycle, but found no methodologies that have automated support for the entire life cycle [Free82]. This situation remains true in early 1985. The AJPO's "Methodman" work, still in the conceptual stage, is one effort directed toward defining a methodology that could lead to development of a "METHODOLOGY-SPECIFIC" life cycle environment.

2. "General" Environment Design is Required

Only "General" environments are within the current [1985] state of the art to support the entire software life cycle. While a substantial amount of development is in process, we believe it is improbable that a methodology-specific environment can be developed sufficiently to be produced in quantity and supplied as GFE by 1992. We conclude, therefore, that the HAPSE must be a "General" environment. This means that, as in the past, we will need to rely on management controls and military standards to insure conformance to whatever software development/support methodology is being used.

E TOOL FUNCTIONS

We concluded in section D that the GFE/Environment must be a "General" type environment, because only "General" environments are within the state of the art. In this section, we look at the specific tool functions that are to be integrated into the GFE/Environment.

1. Tool functions for GFE/HAPSE

What tools should be integrated in the GFE/Environment to support software throughout the life cycle? We looked first at the most important environments that exist today.

Figure 3-7 shows the tool selection in each of six contemporary environments. Each of the six includes a different combination of software tools. The environments are:

AIE	Ada Integrated Environment
APS	Full DoD APSE
ALS	Army's Ada Language System
UNX	UNIX
ARG	Stucki's ARGUS (Boeing)
SPS	TRW's Software Productivity System

The figure shows 23 types of software functions supported in one or more of the six environments. An "X" indicates a function supported by a specific software tool within the environment. Shaded functions represent the minimum needed for the complete life cycle.

	AIE	APS	ALS	UNX	ARG	SPS
Requirements						
- Requirements Tracing		X				X
- Requlrements Language						X
Deslgn						
- Deslgn Support		X			X	X
Implementatlon						
- Compller/Assembler	X	X	X	X	X	X
- Linker/Loader		X	X	X	X	X
Checkout						
- Statement Coverage Analyzer		X	X	X	X	X
- Debugger		X	X	X	X	X
- Cross-Reference		X	X	X	X	X
- Call Structure Analyzer		X	X			
- Timer/Performance Analyzer		X	X	X	X	X
Management						
- Conflguration Manager		X		X	X	X
- Library Management		X	X	X	X	X
- Project Control		X			X	X
- Standards Auditor		X				
Documentation						
- Graphics Generator					X	
- MIL/SPEC Generator		X				
- Text Formatter		X		X	X	X
- Typesetter				X	X	X
- Speller				X	X	X
- Editor		X	X	X	X	X
User Interface						
- On-line Help		X	X		X	X
- Menus					X	
- On-line Documentation				X	X	X

Figure 3-7. Types of Tools Included in an Environment Vary Significantly.

2. Minimal tool functions required

The four groups of shaded tools indicate the minimum tools necessary to support software development. The minimal groups of software tools, shaded on the figure, are:

Compiler/Assembler;
Linker/Loader

Text Formatter

Editor

Definitions of these and other Software Functions and Tool Capabilities are given in Table 3-2.

3. Tool capabilities for subsequent HAPSE versions

The survey of software professionals (Table 3-3) helped to assign priorities for the important software tool capabilities [Houg82a]. These priorities, which we used in defining the HAPSE, are indicated in Figure 3-8. Tool capabilities are added systematically in evolutionary developments of the HAPSE. HAPSE versions II to V, as discussed in chapter IV, contain the the four minimal tools plus the following tool capabilities.

TOOL CAPABILITY	HAPSE II	HAPSE III	HAPSE IV	HAPSE V
Requirements Tracing	X	X	X	X
Debugger	X	X	X	X
Cross-reference analyzer	X	X	X	X
Call structure analyzer	X	X	X	X
Configuration management	X	X	X	X
Standards auditor	X	X	X	X
On-line help	X	X	X	X
Design support		X	X	X
Statement Coverage Analyzer		X	X	X
Timer/Performance Analyzer		X	X	X
Project control		X	X	X
Syntax-directed editor		X	X	X
Menus		X	X	X
Requirements language			X	X
Graphics generator			X	X
MIL/SPEC generator			X	X
Typesetter			X	X
On-line documentation			X	X
Locked security controls				X

	Prioritization*			
Tool Capabilities	Minimal	Required	Important	Useful
Requirements				
Requirements Tracing		b		
Requirements Language				b
Design				
Design Support			b	
Implementation				
Compiler/Assembler	X			
Linker/Loader	X			
Checkout				
Statement Coverage Analyzer			X	
Debugger		X		
Cross-Reference		X		
Call Structure Analyzer		X		
Timer/Performance Analyzer			X	
Management				
Configuration Management		X		
Library Management				
Project Control			X	
Standards Auditor		@		
Documentation				
Graphics Generator				b
MIL/SPEC Generator				b
Text Formatter	X			
Typesetter				b
Speller				
Editor	X			
-syntax-directed			X	
User Interface				
On-Line Help		X		
Menus			X	
On-Line Documentaion				x

*Priorities based on comments received from reveiwers of NBSIR-2625.

@ Item not covered in NBSIR-2625; estimate supplied by Technion International.

b Item not covered in NBSIR-2625; estimate of future usefulness to DoD supplied by Technion International.

Figure 3-8. Priorities for Tool Capabilities.

F. ENVIRONMENT TECHNOLOGY

In order to provide an Ada-based Environment as GFE in the near future, the HAPSE must have four characteristics. It must:

1. Be portable, to different projects, hardware, and organizations. Otherwise, HAPSE could not be expected to be successful as GFE.

2. Support the entire system/software life cycle.

3. Be based on proven technology, so that all tools can be integrated in HAPSE within seven years. There is time for development, but not for new basic research.

4. Provide for common interfaces, to accommodate expansion of functions and addition of software tools that have not yet been developed. For example, tools incorporating "expert systems" or "artificial intelligence" techniques, though not likely to be perfected within seven years, are now being discussed.

We concluded that the constraints of time and cost mean that the HAPSE must be built "on top of" an existing environment. We found that others have reached the same conclusion:

1. Today's production environments are built "on top of" existing environments. For example, Stucki's ARGUS and TRW's SPS are built on top of a UNIX operating system. The French "Platine" is built on top of the VAX VMS operating system [Hunk81].

2. Many research environments are built on UNIX. Examples: Wasserman's USE, Riddle's Joseph, and TOOLPACK [Hunk81].

3. Programming environments are built "on top of" lower-level environments. For example, the ALS environment is built on top of the VAX/VMS or KAPSE.

1. Requirements for the Existing ("Underlying") Operating System

Some capabilities must be provided to the HAPSE by the underlying operating system, the system on top of which the HAPSE will be built. These capabilities include:

1. Database-Building Capability -- to track documentation, programs, and different versions of software.

2. Ada Interpreter -- for debugging.

3. Editor Generator ("Syntax-Directed", or "Orientable" Editor) -- for Ada code and DoD-specific documentation.

4. File System -- for security and library handling.

5. Government Ownership -- to minimize the constraints of time and cost for developing and building the HAPSE.

As indicated above, many operating systems are capable of supplying these capabilities. UNIX and the VAX VMS are particularly popular at this time. The UNIX software is of special interest because, along with more than one hundred low level software tools, it is in the public domain.

2. Overlying Requirements for the HAPSE

The requirements that must be visible to programmers are those shown in Table 3-3 and Figure 3-8, Priorities for Tool Capabilities." They are summarized in Figures 9A and 9B, using the FIPS 99 taxonomy used in section B.

3. Defining the GFE/HAPSE

In this section we bring together into one definition the material described in prior sections. Following the logic described earlier in this chapter, we defined the GFE/HAPSE to have the features and the priorities shown in Figures 3-9A and 3-9B. To minimize resource requirements for the HAPSE, we defined it as being built on top of the UNIX operating system (with immediate access to the many public domain software tools in the UNIX environment).

	REQUIRED	IMPORTANT	USEFUL
DYNAMIC ANALYSIS	X	—	—
TIMING ANALYSIS	—	X	—
TUNING ANALYSIS	—	X	—
TRACING/DEBUGGING	X	—	—
REGRESSION TESTING	—	X	—
ASSERTION CHECKING	—	—	X
COVERAGE ANALYSIS	—	X	—
TRANSFORMATION	X	—	—
FORMATTING	X	—	—
OPTIMIZATION	X	—	—
COMPILATION	—	X	—
INSTRUMENTATION	—	X	—
EDITING	X	—	—
SYNTAX DIRECTION	—	X	—
INPUT/OUTPUT	X	—	—
ON-LINE ASSISTANCE	X	—	—
COMMAND ASSISTANCE	X	—	—
ERROR ASSISTANCE	X	—	—
ON-LINE TUTOR	—	—	X
DEFINITION ASSISTANCE	—	—	X
MENU ASSISTANCE	—	X	—

Figure 3-9A. Overlying Requirements for HAPSE

	REQUIRED	IMPORTANT	USEFUL
MANAGEMENT	X	—	—
CONFIGURATION CTRL	X	—	—
INFORMATION MGT	X	—	—
ADA LIBRARY MGT	X	—	—
SPECIFICATION MGT	—	X	—
DATA DICTIONARY MGT	—	X	—
ADA PACKAGE MGT	—	—	X
TEST MGT	—	X	—
PROJECT MGT	—	X	—
COST ESTIMATION	—	X	—
SCHEDULING	—	X	—
TRACKING	—	X	—
STATIC ANALYSIS	X	—	—
TYPE ANALYSIS	X	—	—
INTERFACE ANALYSIS	X	—	—
STATISTICAL PROFILING	—	—	X
CROSS REFERENCE	X	—	—
AUDITING	—	X	—
COMPLEXITY MEASUREMENT	—	—	X
COMPLETENESS CHECKING	—	—	X
CONSISTENCY CHECKING	—	—	X
STRUCTURE CHECKING	—	X	—
REFERENCE ANALYSIS	—	X	—

Figure 3-9B. Overlying Requirements for HAPSE

The various interfaces are shown in Figure 3-10, "UNIX as a MAPSE". Programmers and others using the HAPSE would see only the HAPSE and its software development and support capabilities. We believe that this is important, to attain the DoD objective of emphasizing use of the standard Ada language. The GFE/HAPSE may need to be designed so that users cannot defeat the requirement for exclusive use of the Ada language by programming in the "C" language rather than in Ada. This topic is one of several advantages and disadvantages discussed in the next section.

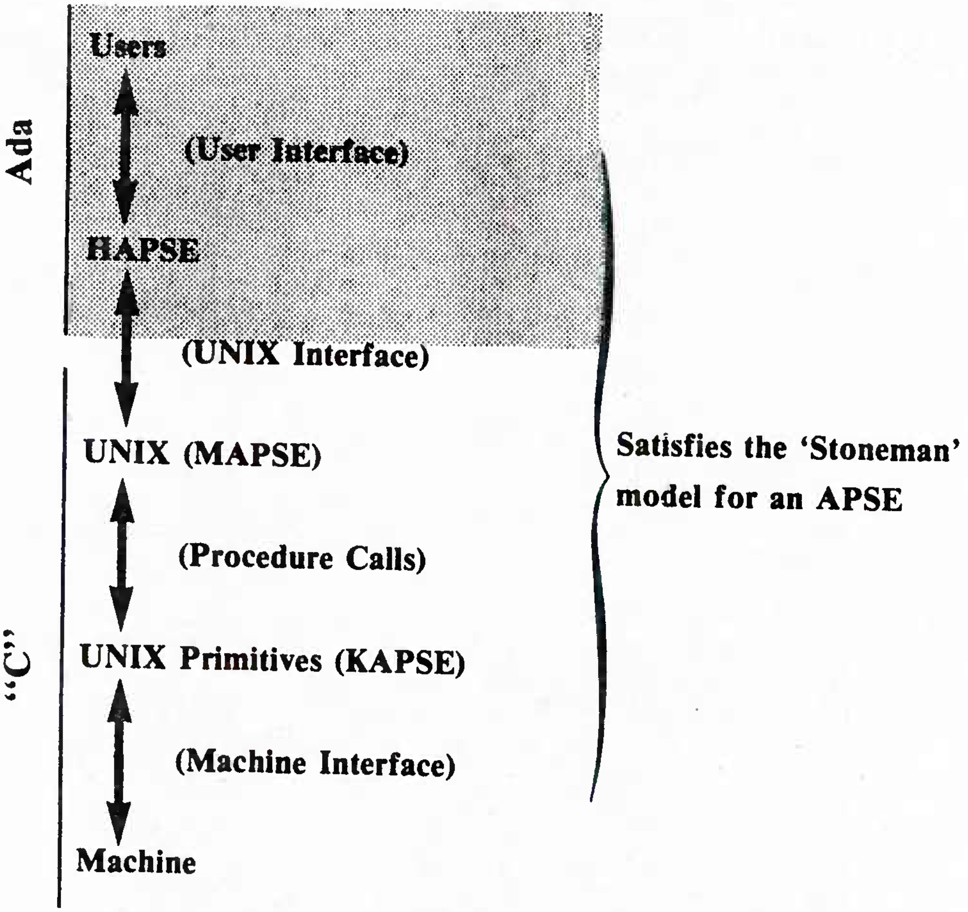


Figure 3-10. UNIX as a MAPSE.

Figure 3-10 shows the host computer (the "bare machine") as the lowest level. Interface with the machine is provided by the UNIX "primitives" ("write", "read to device", "interrupt signal", etc.) functioning as a Kernel APSE ("KAPSE") at the next level. The next interface, from KAPSE to Minimal Ada Programming Support Environment ("MAPSE"), is made by procedure calls. Above this point, users see only the HAPSE and the Ada language. The HAPSE interface with the UNIX MAPSE, though written in "C" (the UNIX language), is hidden from users.

4. Advantages and Disadvantages of UNIX as a MAPSE

Technion researchers identified several advantages to using the approach described above, as well as some disadvantages. Both sets of factors must be reviewed by concerned USAF staff.

Advantages. The principle advantages of using UNIX as the MAPSE are the low risk and low requirements for resources and time. More than 1ØØ UNIX low level software tools are established and in the public domain [Kern84]. The UNIX file system can be used to meet the HAPSE underlying requirements for database capabilities. These and other important advantages are highlighted below:

- UNIX "primitives" are small in number. This makes UNIX portable, and would make the HAPSE portable.
- UNIX tools are large in number (over 1ØØ). These tools can be used as building blocks for the HAPSE, reducing development costs by a significant amount.
- UNIX interface. The UNIX "shell" is a programmable command language. It features directed input and output. These features make it possible to use tools as "objects", in the same manner as a programmer uses "variables". The result is a powerful tool for the expert UNIX-user, but a tool that is terse and "unfriendly".
- UNIX File System is an advantage. UNIX files are defined as character strings. The file system itself is hierarchical. These characteristics make it possible to use the UNIX file system as an underlying database for the HAPSE. The database is important for storing and keeping track of the different products (and versions) produced.
- Low Risk, because it has been done before. TRW's "SPS" is the best example, since it functions in a true production role.

Disadvantages. The disadvantages of using UNIX are highlighted below. They stem from the same characteristics that make UNIX advantageous for a GFE/HAPSE. UNIX is widely known, with many users. Its strengths include use of "shell programming." These features become disadvantages when considered in the light of requirements for security.

- UNIX has poor security. However, it may be possible to overcome this by blocking access to the UNIX shell.

- UNIX is primarily supported by programs written in the "C" language, and "C" is not "Ada". This characteristic may be acceptable for early versions of the HAPSE, useful for prototypes. If required, the underlying environment could be reprogrammed in Ada for later HAPSE versions.

- Performance will be slower than UNIX, since the overlying environment (HAPSE) will require additional "processing overhead." However, the performance penalty may not be significant; only minor performance degradation has been reported in the ARGUS and TRW "SPS" systems built this way. Also, all of the UNIX features may not be needed for the HAPSE; eliminating some features might reduce the performance penalty.

- Size may not be adequate immediately for use on very large system projects. The size range of projects supported by UNIX has traditionally been small to medium.

In Chapter IV we present a feasible schedule for implementation of the HAPSE. The schedule was designed with the above advantages and disadvantages in mind, and allows time for review of requirements that may be impacted by them.

CHAPTER FOUR

PROS AND CONS OF FURNISHING A STANDARD ENVIRONMENT AS GFE

In this chapter, we work with the HAPSE as defined in chapter 3, and identify the "pros" and "cons" of developing a standard environment to be provided as GFE. We describe the research done on Task 3 of this contract. In Task 1, we identified what an integrated automated software development/support environment would consist of. In Task 2 we identified the tools and methods now available, looked at what tools need to be developed, and defined the HAPSE. Results of Tasks 1 and 2 were described in chapter 3.

Chapter 4 has three sections. In section A we describe arguments for and against imposing the HAPSE as a standard GFE environment for contractors to use in developing mission-critical software.

Section B presents a plan for developing a GFE/HAPSE, using a conventional development acquisition strategy. Finally, in section C we discuss a primitive econometric model required to compare the costs and benefits of implementing alternative environments. This is required because of the complex socio-technical situations in which the GFE/HAPSE will be used. It is simply not possible to quantify -- at least accurately enough to make a $100 million decision -- without such a model.

A. PROS AND CONS

In this section, we summarize the arguments for and against a USAF decision to provide a standard environment to contractors as GFE.

1. Pro-HAPSE Arguments

a. The HAPSE will help the Air Force obtain large savings in its developments of new software. Our estimate (discussed in chapter 5) is that potential savings of 66 percent can be obtained on new software and 82 percent for maintenance of existing software. (See Figure 5-1, "'HAPSE' tools to enhance life-cycle productivity").

b. The HAPSE will increase the reliability and maintainability of the software it produces because of the standards that it will contain and enforce.

c. The HAPSE will provide continuing savings, resulting from standardization of data and "reusable software fragments."

d. The HAPSE will improve the Air Force's ability to manage its development and support projects, by providing increased visibility of work as it is performed.

e. The HAPSE will provide a vehicle for increasing productivity for development and support of software systems. This could result in quicker response to changed requirements and reduced costs for the same levels of software quality.

2. Arguments Against

a. The visible ccst to the Government of developing, building, and constantly upgrading the HAPSE. This is really more a matter of obtaining congressional commitments for keeping the HAPSE at the state of the art level over a period of ten or 15 years. Past experience indicates that the Federal government's computer hardware and software become obsolete rapidly, in part because of congressional reluctance to continue funding at required levels over an extended period [OMB78; Booz81; Wer183].

b. If the environment's productivity lags behind the state of the art available from contractors by more than a year or so -- at the annual productivity increase rate of 2Ø percent -- much of the benefit of the HAPSE will be negated.

c. The many points, highlighted in Figures 4-2 and 4-3, at which unforeseen costs can be expected.

d. Contractors' policies for bidding on development projects may change if they believe they cannot count on exclusive maintenance contracts for systems they develop.

e. The lengthy duration of development, if the "conventional" acquisition strategy is followed. As shown in the next section, the basic HAPSE (HAPSE I) would require 54 months after contract startup to be available for release as GFE. HAPSE II, III, IV, and V -- each adding more tool capabilities -- would be developed in parallel and would follow at intervals of six to nine months. HAPSE V, with locked security controls and the complete set of HAPSE tool capabilities, would be ready for use as GFE in about 93 months (nearly eight years) after contract startup.

B. PLAN FOR DEVELOPING AND IMPLEMENTING A GFE/ENVIRONMENT

This section describes the process and suggests a timetable for developing and implementing a HAPSE, using an evolutionary approach and a conventional acquisition and development strategy. The plan involves the activities shown in Figure 4-1 and summarized in Table 4-1. Each of these activities is discussed briefly in this section.

1. Evolutionary development of HAPSE in five versions

The plan is evolutionary, with five successively more capable releases of the HAPSE. They have the following characteristics:

HAPSE I	UNIX plus Minimal tool capabilities
HAPSE II	HAPSE I, plus "Required" tool capabilities
HAPSE III	HAPSE II, plus "Important" tool capabilities
HAPSE IV	HAPSE III, plus "Useful" tool capabilities
HAPSE V	HAPSE IV, plus locked security controls

Figure 4-1 shows the development plan and approximate schedule for all five versions.

2. Development schedule

Work on phase I (boxes 1.1, 1.2, and 1.3), would be completed at about 12 months after contract startup. Development of HAPSE I would be begun at that time. The work of Phase II (boxes 2.1, 2.2, 2.3, and 2.4) would be conducted in parallel with the development of HAPSE I. With this "head start", HAPSE I would be ready for limited release (Beta test, box 5.1) in 30 months after contract startup. After evaluation and update (box 6.1) in about 40 months, HAPSE I would be ready for general release as GFE (box 7.1) in 54 months.

After completion of Phase II work (boxes 2.3 and 2.4) at about 30 months after startup, the development of HAPSE II could be started (box 3.2) at about 39 months after contract startup.

HAPSE II, integrating the "Required" tool capabilities, would be in Beta test (box 5.2) at 60 months, and ready for general release as GFE (box 7.2) at about 72 months.

HAPSE III, integrating with HAPSE I and HAPSE II the "Important" tool capabilities, would start (box 3.3) at 54 months, be in Beta test (box 5.3) at 66 months, and ready for release (box 7.3) at 78 months.

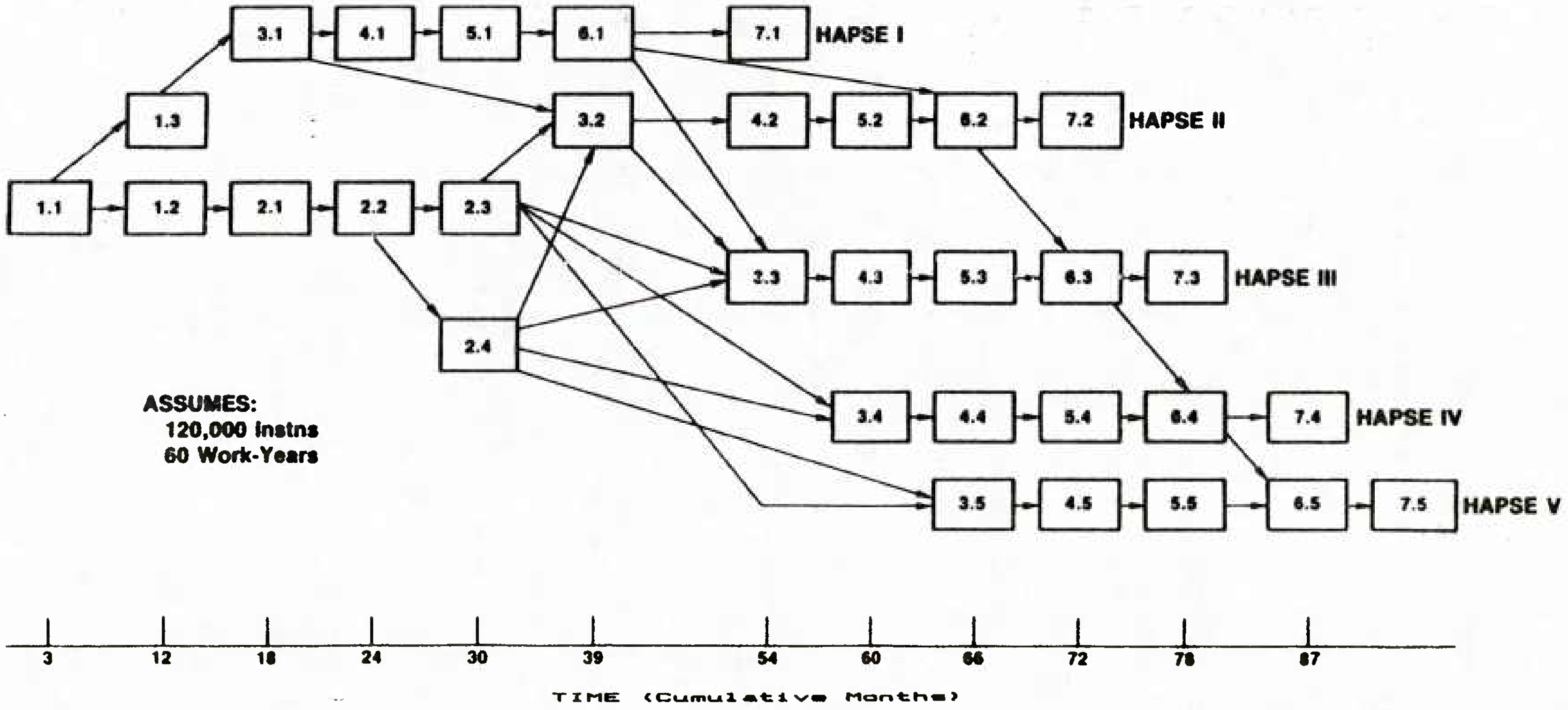


Figure 4-1. HAPSE Implementation Schedule
(Cumulative Months After Startup)

Similarly, HAPSE IV, integrating the "Useful" tool capabilities, would start (box 3.4) at 60 months, be in Beta test (box 5.4) at 72 months, and ready for release (box 7.4) at 87 months after contract startup.

Most important for highly secure applications, HAPSE V, which would add locked security controls for all tool capabilities of HAPSE IV, would start (box 3.5) at 66 months, be in Beta test at 78 months, and ready for release as GFE in about 96 months after contract startup.

The development plan in Figure 4-1 involves seven life cycle phases.

1. Phase I Analysis and Evaluation

   1.1 Contractor analyzes HAPSE Requirements; gets up to speed, resolves outstanding issues

   1.2 Develop and evaluate prototype user interface; develop several prototype user interfaces, evaluate them under simulated use conditions, and modify as appropriate using feedback from users.

   1.3 Evaluate existing software tools; determine which tools may be selected from the public domain UNIX tools, which need enhancement to meet Ada requirements, and which, if any, need to be developed.

2. Phase II Specification

   2.1 Specify user interface; specify HAPSE-to-user interface, based on results of 1.2.

   2.2 Specify UNIX/HAPSE tool interface; Specify HAPSE-to-UNIX interface.

   2.3 Specify database interface; specify "underlying" UNIX database.

   2.4 Specify tool functions; specify tools required by HAPSE.

3. Phase III. Development of HAPSE Release.

4. Phase IV. Integration and Testing

5. Phase V. Limited Release (Beta Test)

6. Phase VI. Feedback, evaluation, update

7. Phase VII. Release, Installation, Training, etc.

Table 4-1. SUMMARY OF HAPSE DEVELOPMENT SCHEDULE
(Showing Fig 4-1 box number, and months after startup)

HAPSE Version	Development	Integration, Test	Beta Test	Release as GFE
Phase I, Analysis & Eval.	(1.2, 1.3) 12 months			
Phase II, Specification	(2.1-2.4) 30 months			
HAPSE I	(3.1) 18 months	(4.1) 24 mos	(5.1) 30 mos	(7.1) 54 months
HAPSE II	(3.2) 39 mos	(4.2) 54 mos	(5.2) 60 mos	(7.2) 72 mos
HAPSE III	(3.3) 54 mos	(4.3) 60 mos	(5.3) 66 mos	(7.3) 78 mos
HAPSE IV	(3.4) 60 mos	(4.4) 66 mos	(5.4) 72 mos	(7.4) 87 mos
HAPSE V	(3.5) 66 mos	(4.5) 72 mos	(5.5) 78 mos	(7.5) 93 mos

C. ECONOMETRIC MODEL

The work described in previous chapters has convinced us that the USAF may benefit substantially (conceivably by as much as $100 million annually by 1990, more in out years) if it changes the existing policy of unique environments for each defense system. Projected savings accrue from: (a) reduction in the number of unique environments supported by the Air Force; (b) increases in the productivity of staff who actually use the standard environment; and (c) increased reliability and maintainability of the software produced by the standard environment. Quantifying benefits in a meaningful manner requires substantially more rigor than is possible without a model that considers such non-quantitative factors as control strategy and organizational structure. Chapter 5, Planning for Implementation, discusses similar phenomena from the viewpoint of optimal implementation.

1. More quantification needed

The econometric model described below is intended to compare the effectiveness and benefit/cost performance of several standard Ada programming support environments ("environments"). These alternative environments, which may be imposed as GFE or mandatory standards for mission-critical software-intensive projects, are part of a "Technology Push" program to improve performance of mission-critical systems. The environments include, but are not limited to, JSSEE (Joint Services Software Engineering Environment), ALS (the Army Ada Language System), the WIS (WWMCCS Information System environment being developed by GTE), the HAPSE (Hypothetical Ada Programming Support Environment) defined in chapter 3, and continuation of the present policy of non-standard environments or BAU (Business as Usual).

The primitive econometric model discussed below uses both minimum time and cost required to provide quantitative support for the FY 1986 Air Force decision on software support environments. Work to be done using the model includes these activities:

a. Prepare and validate an econometric model for implementing an Ada-based software support environment in contractor and DoD organizations. Quantitative portions (sub-model), to be adapted from the existing foundation of the "COCOMO" cost estimating model, will be augmented by non-quantitative submodels that describe organizational, control strategy, and policy behavior. Most of this work will consist of searches of public literature.

b. Assuming typical software development/support workloads, estimate effects of alternative control strategies by exercising the model with various "control strategies" and "constraints on successful implementation".

c. Assuming typical software development/support workloads, estimate effects of providing a standard environment to contractors, by exercising the model for differences in contracting strategy and contractor motivations.

d. Based on a sensitivity analysis of the work above, select strategies and prepare cost/benefit tables for implementing environments under varied conditions. Alternatives will include specific combinations of organizational structure, control strategies, and projected effects of software tools on productivity. To the extent possible, these tables should be consistent with present accounting data. Accuracies should be estimated, using standard statistical inferential techniques.

2. Background

This work is an important part of the STARS [Software Technology for Adaptable, Reliable Systems] program. The STARS initiative is a systematic response to the DoD/USAF business management problems that stem from their massive investments in mission-critical computer software. Much of the software inventory is obsolete and nearly all of it is difficult and expensive to support. One estimate puts the 1990 cost for Post-Deployment Software Support at $5 to $7 billion annually by 1990 [JLC84].

Figure 4-2 depicts the process proposed to develop a parameter-driven econometric model that can provide quantitative data to Air Force decision makers as they make the business decision regarding use of a standard software support environment as GFE. It is followed by several detailed sub-models (figures 4-2A to 4-2D).

Our recommendation is supported by statistical analyses of the COCOMO software project database. We focused our analysis on those project factors that demonstrably improve productivity in developing and maintaining software. In our statistical analysis we used correlation, multiple regression, and the important review of the cause-effect relationships involved. The results of the correlation analysis are given below, while the cause-effect relationships are discussed in chapter 5.

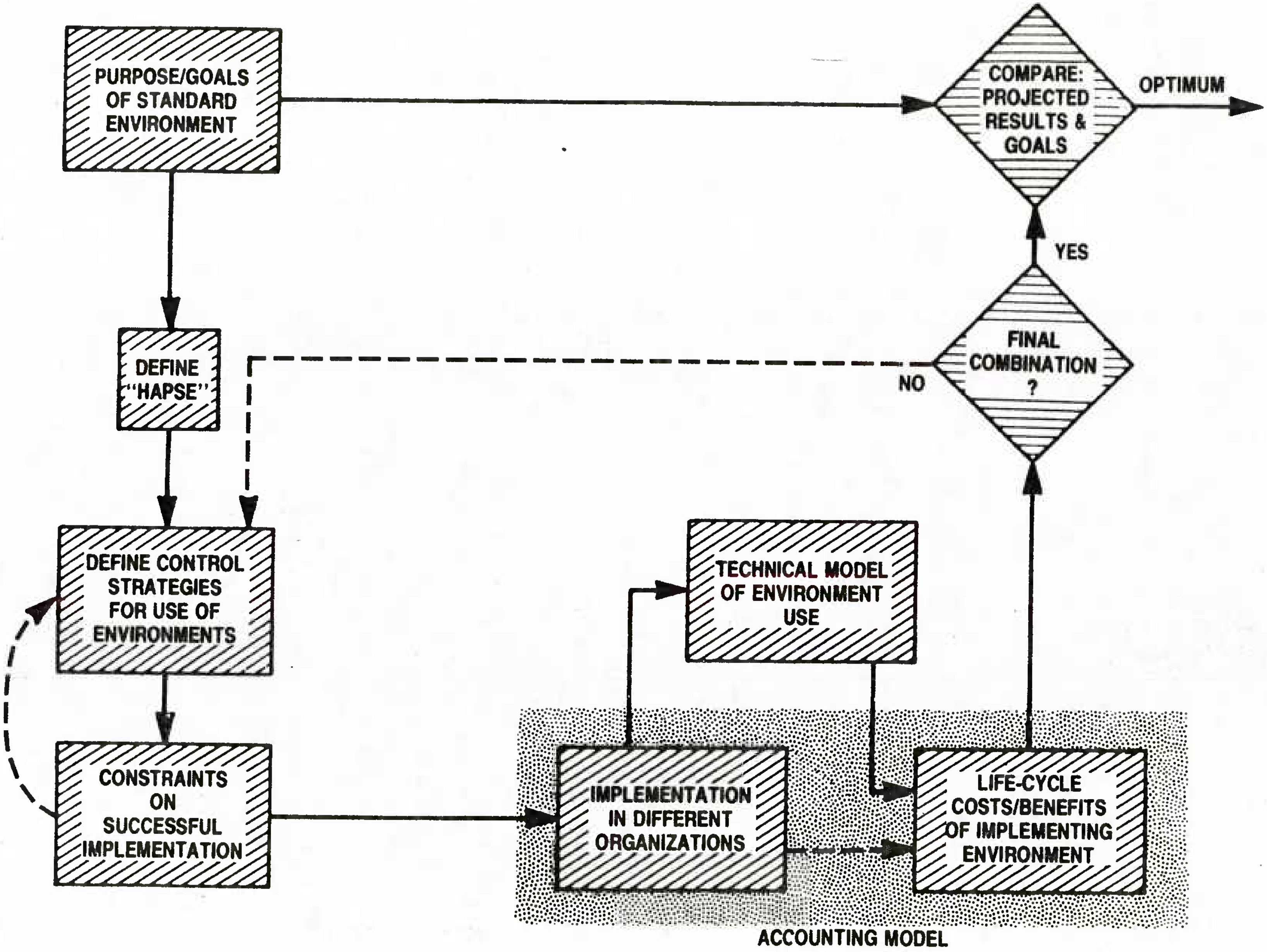

PURPOSE/GOALS OF STANDARD ENVIRONMENT
COMPARE: PROJECTED RESULTS & GOALS
OPTIMUM
YES
DEFINE "HAPSE"
FINAL COMBINATION ?
NO
DEFINE CONTROL STRATEGIES FOR USE OF ENVIRONMENTS
TECHNICAL MODEL OF ENVIRONMENT USE
CONSTRAINTS ON SUCCESSFUL IMPLEMENTATION
IMPLEMENTATION IN DIFFERENT ORGANIZATIONS
LIFE-CYCLE COSTS/BENEFITS OF IMPLEMENTING ENVIRONMENT
ACCOUNTING MODEL

Figure 4-2. Model of Environment Implementation

The work shown in Figure 4-2, which culminates in an econometric model capable of comparing performance of alternative environments, began with the "Purpose and Goals of a Standard Environment", shown at the top left. Both this and the next block, "Define HAPSE", have been completed in the current project.

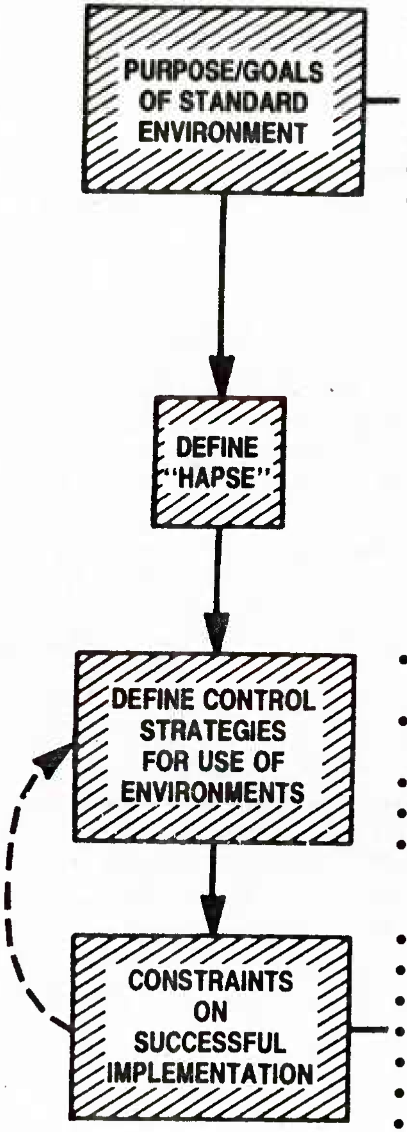
3. Non-Quantitative Parameters

Most important in terms of the model's credibility are the two following blocks, "Define Control Strategies for Use of Environments" and identify "Constraints on Successful Implementation". They translate non-quantitative factors (such as different control strategies and constraints stemming from different Services' policy choices) into inputs usable by the succeeding three blocks. Figure 4-2A includes examples of factors treated in these blocks. The resulting non-quantitative parameters are input to the "Technical Model of Environment Use".

4. Quantitative Parameters

This block, the most complex and detailed of the entire series shown in Figure 4-2, consists of "effort multipliers" and equations derived from the "COCOMO" software cost estimating program [Boeh81]. Its output is used by the two blocks below it; they translate its parametric output (work-months and elapsed time required for given software project parameters) into resource requirements. Their output, expressed in accounting terms, shows resource costs by category and year for the 2Ø years of a weapon system's life based on the parameters developed in the previous blocks. The relationship of these blocks is detailed in Figure 4-2B. An example of the output is shown in Figure 4-2C.

The diamond shaped decision processes shown at the top right side of Figure 4-2 represent decisions to continue iterating until an organizationally "optimal" [though not a rigorous mathematical optimum] solution is reached. The example at the top right of Figure 4-2B, and in Figure 4-2D, show the functions of these blocks.


PURPOSE/GOALS OF STANDARD ENVIRONMENT
DEFINE "HAPSE"
DEFINE CONTROL STRATEGIES FOR USE OF ENVIRONMENTS
CONSTRAINTS ON SUCCESSFUL IMPLEMENTATION

- Reduce Proliferation of Environments
- Reduce Life-cycle Support Costs of Software
- Reduce Training Costs by Standardization
- Reduce Costs for Maintaining Many Environments
- Provide Environment That is - and Will Remain - "State-of- Practice"
- Increase Productivity of Software Development/Support
- Facilitate Planning for PDSS Early in System Design Process

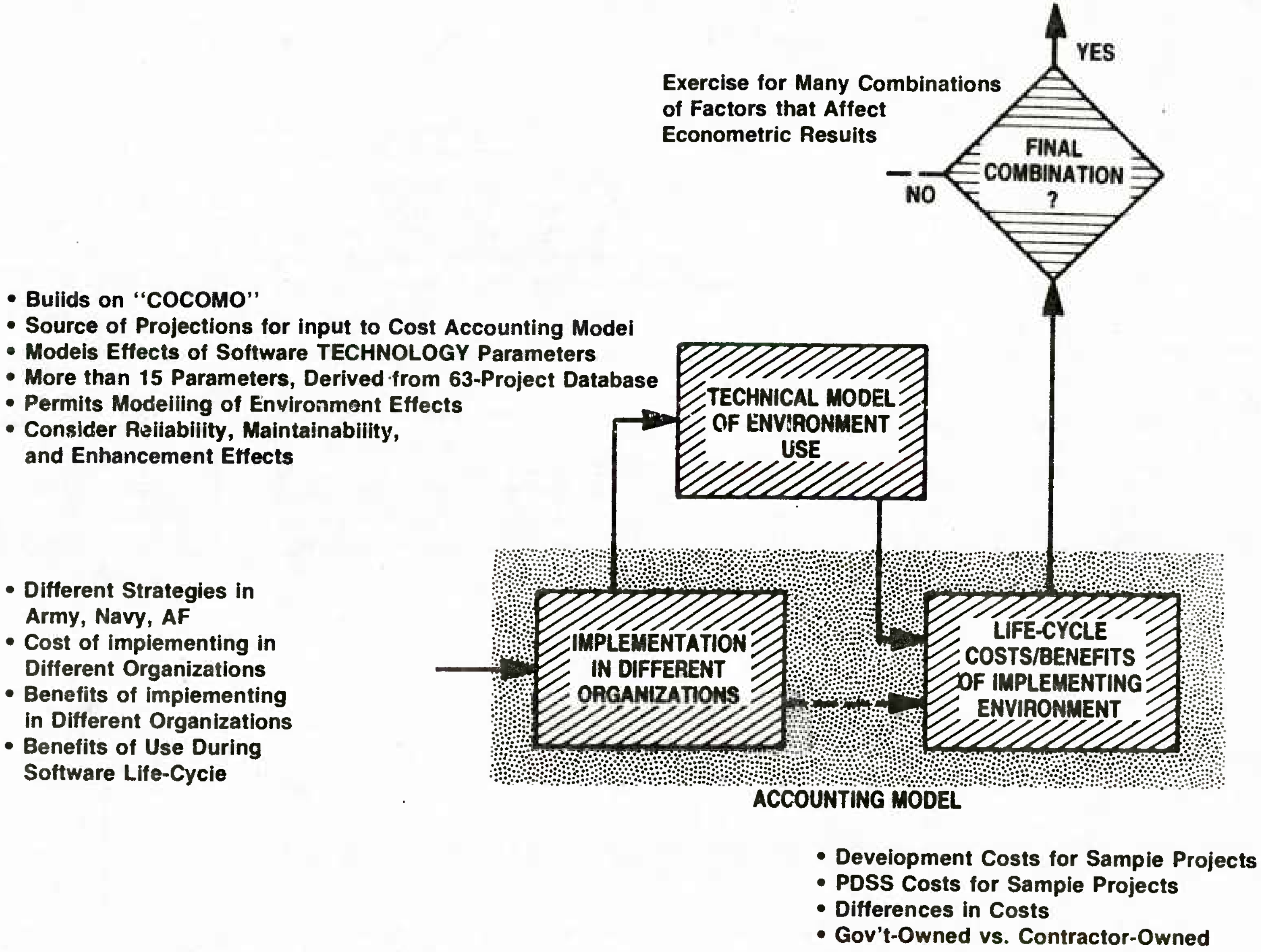
- Standardize on Environment for Software DEVELOPMENT?
- Standardize on Environment for POST-DEPLOYMENT SOFTWARE SUPPORT?
- Standardize on INTERFACES?
- Combinations of Above?
- None of Above?

STANDARD ENVIRONMENT

OPTION	DEVELOPMENT	PDSS	COMMENTS
1	NO	NO	STATUS QUO
2	YES	NO	JOVIAL
3	NO	YES	N/A
4	YES	YES	CMS-2, ALS
5	AVBL	AVBL	USE DICTATED BY COST/BENEFIT TO PROGRAM

- Sub-Optimization
- Different Needs of R&D/Support Communities
- MANY Different Languages Being Supported
- Different Organization of Services
- Organization Dynamics within DoD
- Political Dynamics Affecting DoD
- Technology Stagnation
- Vendor Self-Interest

Figure 4-2A. Detail of Non-Quantitative Parameters.


YES
FINAL COMBINATION ?
NO
TECHNICAL MODEL OF ENVIRONMENT USE
IMPLEMENTATION IN DIFFERENT ORGANIZATIONS
LIFE-CYCLE COSTS/BENEFITS OF IMPLEMENTING ENVIRONMENT
ACCOUNTING MODEL

Exercise for Many Combinations of Factors that Affect Econometric Results

- Builds on "COCOMO"
- Source of Projections for Input to Cost Accounting Model
- Models Effects of Software TECHNOLOGY Parameters
- More than 15 Parameters, Derived from 63-Project Database
- Permits Modeling of Environment Effects
- Consider Reliability, Maintainability, and Enhancement Effects

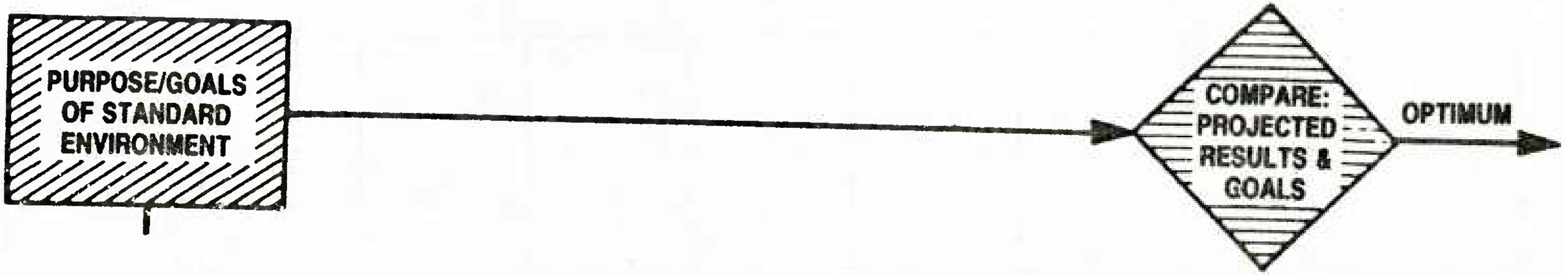
- Different Strategies in Army, Navy, AF
- Cost of Implementing in Different Organizations
- Benefits of Implementing in Different Organizations
- Benefits of Use During Software Life-Cycle

- Development Costs for Sample Projects
- PDSS Costs for Sample Projects
- Differences in Costs
- Gov't-Owned vs. Contractor-Owned

Figure 4-2B. Detail of Quantitative Parameter Treatment.

WORK-MONTHS OF EFFORT								
SOFTWARE PRODUCT CHARACT-ERISTICS	DEVELOPMENT PHASE				POST-DEPLOYMENT SUPPORT PHASE			
	JSSEE	ALS	HAPSE	WIS (GTE PHOENIX)	JSSEE	ALS	HAPSE	WIS (GTE PHOENIX)
a								
b								
c								
d								
e								
f								

Figure 4-2C. EXAMPLE: Comparison of Model Output


PURPOSE/GOALS OF STANDARD ENVIRONMENT
COMPARE: PROJECTED RESULTS & GOALS
OPTIMUM

- How Much Does the Strategy Reduce the NUMBER of DIFFERENT ENVIRONMENTS Needed?
- What Will Be the Net Savings for Support of MCCR Software (After Additional Costs for HAPSE?)
- How Well Will Environment Maintain State-of-Practice Over Next Decade?

Figure 4-2D. Detail of Result Comparison

5. Model builds on "COCOMO" research

The proposed model could be completed in a few months, because the underlying development work needed for this effort has already been done, much of it by Barry Boehm of TRW. In Boehm's Software Engineering Economics, he describes the derivation and presents equations that resulted from TRW's analysis of data from 63 "well managed" software development projects. Of these projects, 34 involved systems similar to those in mission-critical systems for which the HAPSE will be used. Technion International analyzed data for these 34 projects, using correlation, multiple regression, and other statistical techniques. The dependent variables (i.e., required work-months, task duration, and productivity) may be improved by optimizing the HAPSE design (for example, by using particular software tools or programming practices). The correlation coefficients between the dependent and independent variables are tabulated in the Correlation Matrix shown in Table 4-2.

Extension for Post-1979 History. It is necessary to add data for projects completed since 1979, and data from the answers to questions shown in figures 4-2A to 4-2D. The proposed work builds on the equations and underlying research of the public-domain "COCOMO" software estimating system. The "COCOMO" software project database, which includes 63 projects completed during the years 1964 to 1979, is a resource which provides consistent descriptions for projects done during a 15-year period characterized by rapid evolution in the practice of software development. However, it does not include any software programmed in Ada (since no Ada compilers were available during the time covered). The proposed work must add data on more recent projects (from the NASA software engineering lab files, RADC, and other sources), extend the existing COCOMO equations with this data, and improve their credibility for the purposes of comparing software environments. This task will permit us to project the effects of: (a) Ada- language environments and programming; (b) use of integrated automated software tools; and (c) use in subsequent post- deployment software support (PDSS) life cycle activities.

Validity of Approach. The proposed approach is feasible, as demonstrated by the results obtained from our analysis of the existing "COCOMO" project database. The significant results are highlighted in Table 4-2, "Correlation Matrix". The left hand column of that table shows the independent variables, available for each project in the COCOMO software project database, which exhibit statistically significant relationships with three dependent variables of vital interest in this study. "Independent variables can be manipulated to influence "dependent" variables, such as measures of programming output. The next three columns tabulate the "significance level" (i.e., the probability of a true correlation, at the customary .01 and .05 levels) of project variables with three output measures. The three dependent output measures are:

- ACTUAL work-months required for a project
- Productivity (Average number of Delivered source instructions per work-month required for the project
- ACTUAL duration of project, in months

The numbers shown identify relationships for which statistically significant correlation coefficients were found. For example, the entry ".01" shows that the odds against the relationship occurring by chance alone are one out of 100. Similarly, ".05" shows odds of five out of 100. Thus we can infer that the entries constitute genuine relationships that can credibly be used in developing an econometric model. Results show highly significant correlations (at better than the .01 level) between programmer productivity [expressed as delivered source instructions per work-month] and: (a) time [year a project was completed]; (b) use of "modern programming practices"; and (c) three characteristics (TIME, STOR, and DATA) of the "virtual machine" (the hardware and support software) that comprise the target computer. Both "time" and "use of modern programming practices" can be used to improve programming productivity. The "virtual machine" and "required reliability" are functions of the defense system, however, and cannot be as directly influenced by the productivity improvement effort.

Other significant correlations (at the .05 level) exist between programmer productivity and: (d) use of software tools"; (e) programmers' experience with the language used; and (f) volatility during the project of the "virtual machine" (hardware and software) used as the target computer. Finally, the analysis partially validates COCOMO's assumption of the multiplicative effects of the effort multipliers [PI, or the product of 15 effort multipliers for the project]. The Correlation Matrix shows significance levels for each of the "COCOMO" independent variables analyzed. [Boeh81, table on pp. 496-7].

Productivity increase. The significant correlation of YEAR with all three independent variables suggests exponential increases in productivity of the processes of software development and support during the years 1970-1979. During that decade, both hardware vendors and the independent software industry supplied programming support software packages, usually targeted toward the highest volume hardware systems (such as IBM 360/370/30XX).

Table 4-2. CORRELATION MATRIX
34 Selected COCOMO Embedded Software Projects

Project Variables		ACTUAL Work-Months Req'd	Delivered Source Instructions perWork-Month	Duration, ACTUAL Dev. Time,Months
ITEM	Definition			
		(Significance Level)		
YEAR	197Ø-1979	.Ø1	.Ø1	.Ø1
MODP	Modern Programming Practices Used	.Ø5	.Ø1	.Ø5
TIME	Time Constraint of target computer	-	.Ø1	-
STOR	Main Storage Constraint of target computer	-	.Ø1	-
DATA	Database Size, relative to program in target computer	.Ø1	-	.Ø1
TOOL	Software Tools Used	-	.Ø5	-
LEXP	Experience with programming language used	-	.Ø5	-
VIRT	Volatility of "virtual machine" used as target computer	-	.Ø5	-
RELY	Reliability required	-	.Ø5	-
CPLX	Complexity of project	-	.Ø5	-
RVOL	Volatility of Req'ts	.Ø5	.Ø5	-
TURN	Development Computer turnaround time	-	-	-
TYPE	Computer used in development (maxi = 3; mini= 1)	-	-	.Ø5
PCAP	Programmer capability	-	-	-
PI	Product of the 15 Effort Multipliers for the Project	-	.Ø1	-
PROJ	Project Type (Control, System, Scientific, Human-machine interaction, Support)	No correlations in sample data		
LANG	(Pascal,APL,PL/1; Fortran; Jovial; Machine Language)	No correlations in sample data		

During the time that they became more useful, programming productivity (in terms of delivered source instructions per work month) increased at rates of more than 20 percent annually. (We do not infer that this annual improvement was caused by software packages or programing environments alone. During these years, enormous improvements occurred in acceptance of higher order languages such as COBOL, in memory size, and displacement of batch programming and test operations with interactive programming techniques. Matsumoto et. al. reported a comparable improvement -- 14 percent annually -- for the years 1976-1980 in a Toshiba "soft- ware factory" that stressed reusability of code [Mats81].

Figure 4-3, "Programming Productivity Increases Exponentially", demonstrates that productivity grew for both "small" and "large" projects producing software of the sort embedded in mission-critical weapon systems. Some of the technological changes are indicated on the chart, in roughly the time periods in which they became effective. The points on Figure 4-3 through which the regression lines pass are:

Year	13 "Large" Projects		17 "Small" Projects	
1970	31.6	DSI/WM	27.7	DSI/WM
1971	39.9		34.5	
1972	50.3		42.9	
1973	63.5		53.3	
1974	80.5		66.3	
1975	101.2		82.5	
1976	127.7		102.7	
1977	161.1		127.7	
1978	203.3		158.9	
1979	256.6		197.6	
1980	323.8	(Projected)	245.9	(Projected)
1985	1039	(Projected)	734	(Projected)
1990	3328	(Projected)	2189	(Projected)

In summary, Technion's analysis has confirmed the steady exponential improvement over time [variable, YEAR] as well as productivity growth by use of improved programming practices [MODP]. For the years 1970-79 the correlations are statistically significant. We infer that the relationship with YEAR simply represents the effects of all the changes that took place during this time period, and has no inherent causative property. For example, the decade saw great improvements in hardware speed and memory size, in programming languages, and in development methodologies. We do, however, expect to see a continuation of

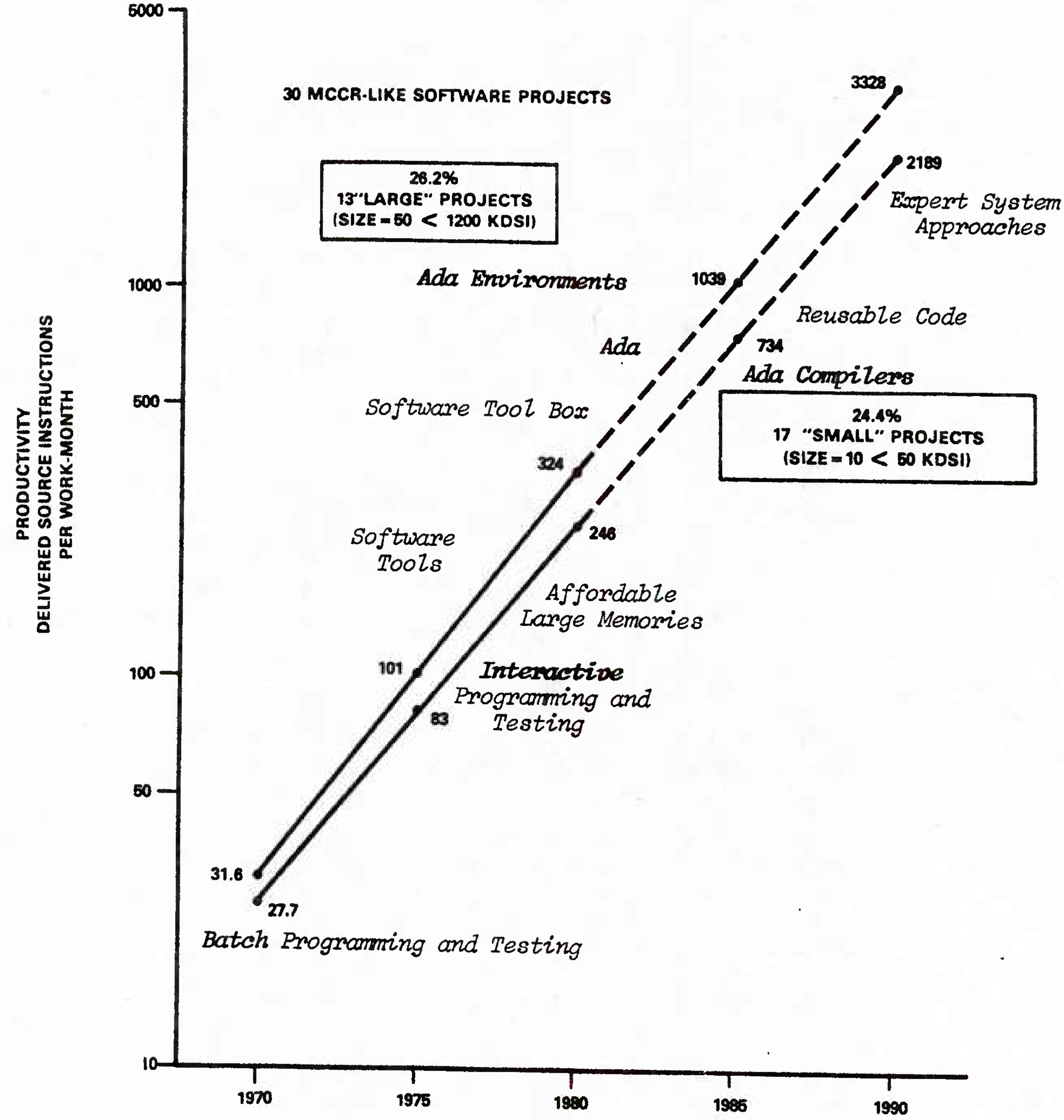


Figure 4-3. Software Productivity Increases Exponentially.

The reader is advised that technical agreement has not been reached on metrics to measure productivity. These curves are "least squares semi-log" lines, derived from statistical analysis of the "COCOMO" project database of "well-managed" projects. The curves show delivered source instructions (DSI) per work-month (W-M), or DSI/W-M.

the trend in these other technologies which will result in continued growth in productivity over the next decade. Therefore, to be realistic, the econometric model must include terms for annual *programmer cost* and *continuing increase in productivity*.

Required additions to project database. The project data base used for the statistical analysis was sufficient for Technion to *suggest* projections of improved productivity to the 1992 time frame. However, the data base must be enhanced to support credible inferences. A model based only on data for the years 1970 to 1979 would hardly be convincing for projections to 1992. Credibility of the model's results must be increased, by overcoming the incompleteness of data used in the analyses. Specific data deficiencies (that can be addressed by obtaining additional data that already exist) include:

a. Lack of data for any projects after 1979. This can be addressed by adding data available through the Software Engineering Laboratory at the NASA Goddard Space Flight Center, Rome Air Development Center, and other sources.

b. Lack of data to permit detailed analysis of enhancement and maintenance activities, or "post-deployment software support (PDSS)", which is especially important for the GFE decision.

c. Lack of Effort Multipliers applying to use of Ada (although some of the projects were coded in similar HOLs).

d. Lack of Effort Multipliers relating to specific "modern programming practices", use of "reusable code fragments" and specific "software tools".

e. Lack of an Effort Multiplier that applies to productivity improvement gained by using an integrated programming support environment. The environment is expected to improve productivity by multiplying the individual benefits of: (1) an integrated set of software tools; (2) modern programming practices; (3) higher order languages; (4) libraries of "reusable code fragments"; and (5) hardware that provides response time short enough and memory capacity large enough to avoid interfering with programmers' trains of thought.

With the model updated in this way, it will be possible to investigate effects of alternative control strategies by including the factors from study of the "control strategies" and "constraints on successful implementation" described in Figure 4-2A, and the "implementation in different organizations" from

Figure 4-2B. This would focus on comparing features of such environments as the Joint Services Systems Engineering Environment (JSSEE), the Army's Ada Language System (ALS), the environment to be developed by GTE for WIS, the "HAPSE" defined in this project, and the default policy, "Business As Usual". Some of the comparisons sought are indicated in Figure 4-2C, "Example: Comparison of Model Output".

Based on the resulting econometric model, it will be possible to estimate the costs and benefits of the different organization strategies for implementating environments, and to select alternatives. Using these alternatives, it will be possible to compute the cost/benefit table for the various environment and software applications. To help in calibrating the model, looking at net long-term cost/benefit results of various alternatives might be done most easily at Tinker AFB, using typical AFSC strategies and project workload mixes.

Similarly, it will be possible to investigate further the pros and cons of providing a HAPSE to contractors as GFE, with the focus on decreasing risks of the GFE approach that arise from differences in contracting strategy. The model could be used to address questions concerned with vendors' motivations, including both economic and cost issues and those relating to vendors' market shares and vested interests in their own technology. It would consider areas outside the COCOMO model, such as technical risk (the technological effects of not having a HAPSE as GFE, or of having a HAPSE with inadequate reliability), cause-effect relationships that impede vendors' ability to accept and use productively environments for which they are not reimbursed directly. It would address technology trends (the natural acceptance of technological change over time), the inter-relationships of modern programming practices, software tools, software development methodologies, and reusable code. Finally, and most important for DoD and the Air Force, the ability of a standard HAPSE to be continually improved at a rate adequate to maintain parity with the industry's increases in productivity (rather than to fall behind, with obsolescent hardware or methodologies, which has happened in the past).

6. Form of Econometric Model

The general form of the proposed model is presented below. Because such a model was not envisioned when Technion began its work, and thus was not fully funded, we present what is obviously only a preliminary sketch. The presentation probably omits terms that will be needed to present a picture complete enough for decision-making purposes. Determination of measures and units, and ways to build on the COCOMO effort multipliers, will be early tasks in the future preparation of the model.

$$\begin{aligned} \mathrm{NETBEN} = {} & \mathrm{SWCOST}_{\mathrm{Current}} - \mathrm{SWCOST}_{\mathrm{GFEEnv}} \\ & - \mathrm{ENVCOST} \\ & - \mathrm{DIFF}\ [\mathrm{CTLOHD};\ \mathrm{TRAINCOST};\ \mathrm{PERSCOST*PRODY}; \\ & \qquad \mathrm{RELCOST};\ \mathrm{MAINTCOST};\ \mathrm{REQVOL}] \end{aligned}$$

Where

Workload = Constant for each comparison, chosen to typify software workload ranges found in typical USAF defense systems or contractor "software factories" (expressed in lines of code per year, over a defense system life cycle of from ten to 15 years).

NETBEN = Net benefit to the Government from completing the assumed workload at lower cost (See figure 4-2D)

$\mathrm{SWCOST}_{\mathrm{Current}}$ = Software cost to Gov't using current practices

$\mathrm{SWCOST}_{\mathrm{GFEEnv}}$ = Software cost to Government using the GFE Environment (including a pro rata share of ENVCOST)

ENVCOST = costs incurred for initial development, plus annual costs for continuing development and support of the GFE/Environment

CTLOHD = Cost of organizations and procedures for control of software development/support practices (See figures 4-2A, 4-2B)

TRAINCOST = Cost of training users (both gov't and contractor) in effective use of the GFE/Environment (See figure 4-2A)

PERSCOST = Cost of personnel who use the GFE/Environment

PRODY = Productivity of personnel, in measures such as source code instructions per work-year (quality, reliability, and maintainability being held constant) (See figure 4-2B, and tables 4-2 and 5-1)

RELCOST = Cost for reliability of software produced (See figure 4-2B)

MAINTCOST = Cost for maintainability of software produced (See figure 4-2B)

REQFNS = Cost of adapting to requirements imposed on software by the defense system of which software is a component (See tables 4-2 and 5-1; and COCOMO variables VIRT, RELY, CPLX, and RVOL)

CHAPTER FIVE

PLANNING FOR IMPLEMENTATION

This chapter has three sections. Section A lists various functions performed during the software life cycle, and identifies functions for which software tools promise the greatest improvements in productivity.

Section B depicts the most important assumptions and the chain of cause-effect links that must be satisfied before successful implementation can be completed. Section C identifies those assumptions and cause-effect links for which additional data are required. Sections B and C overlap with the econometric model described in chapter 4, beginning on page IV-7.

A. FOCUSING ON HAPSE TOOLS TO ENHANCE LIFE-CYCLE PRODUCTIVITY

Selecting expanded sets of "tools" for the HAPSE can help USAF improve its effectiveness in development and support of defense systems, including those that happen to contain software.

1. Independent variables for software

Boehm's "COCOMO" model uses 15 different "Effort Multipliers" for estimating resource requirements for software projects. I n Figure 5-1, "Graphic Illustration of Relationships in Table 1," all 15 effort multipliers of these are depicted in schematic format.

Our statistical analysis identified the variables with the greatest effect on resource requirements and project duration. The effort multipliers for TOOL, MODP, and Libr promise the most short term improvement. Their effects yield the greatest reduction in resource requirement. Table 5-1, "HAPSE Tools To Enchance Life-Cycle Productivity", lists the effort multipliers for both new software development and for the "integration and test" phase, which is used here as a proxy for maintenance of existing software. Note that the effort multipliers are less than 1.00, indicating a reduction in resources required and leading to an improvement in performance. Page references in [Boeh81] explain each variable. As shown at the bottom of Table 5-1, under current conditions reductions are estimated to be 66 percent for NEW software and 82 percent for maintenance of existing software.

Table 5-1. "HAPSE" TOOLS TO ENHANCE LIFE-CYCLE PRODUCTIVITY

PROJECT VARIABLE		"COCOMO" Effort Multiplier		
		NEW Software	Maintenance	[Boeh81, pp.]
Computer Attributes				
STOR	Improved Main Storage, Target Computer	.87	.80	413+
TURN	Development Computer Response Time	.87	.90	417
Personnel Attributes				
VEXP	Experience with Target computer ["Virtual Machine"]	.90	.90	439
LEXP	Programmers' Experience with Language used	.95	.92	442
Project Attributes				
MODP	Modern Programming Practices Used	.91	.83	452
TOOL	Software Tools Used	.83	.70	459
Libr	Reusable Code Libraries [Boeh84, p. 33]	.70	.50	
PI	Product of Effort Multipliers*	.34	.18	498
Net Savings Projected for HAPSE [1.00 - PI]		66 percent		82 percent

Table 5-1, Continued

OTHER COST DRIVERS THAT MIGHT BE ENHANCED TO IMPROVE PRODUCTIVITY

PROBABLE Improvement

Libr Use of Existing Software Fragments, from Library. Use of such "reusable code fragments" in new development will have benefits throughout the software's life.

TIME Availability of Time on Target Machine. Ability to test software immediately after coding helps to improve programmers effectiveness. This variable can be addressed by making a target machine available early in development.

PCAP Programmer Capability. May be improved by adding software tools with abilities developed by the most capable programmers.

POSSIBLE Improvement

RELY Required Degree of Reliability of software in the defense system. For mission-critical systems, the needed reliability is usually quite high.

RVOL Volatility of Requirements that software must satisfy. Improvement in productivity would come with reduced frequency of change ("volatility") of requirements.

VIRT "Virtual Machine Volatility" (Target Machine). Reducing the "volatility", or amount of change to the target hardware or operating system, improves productivity of the software development and support.

ACAP Analyst Capability. May be improved by adding, in the form of tools, capabilities developed by the analysts.

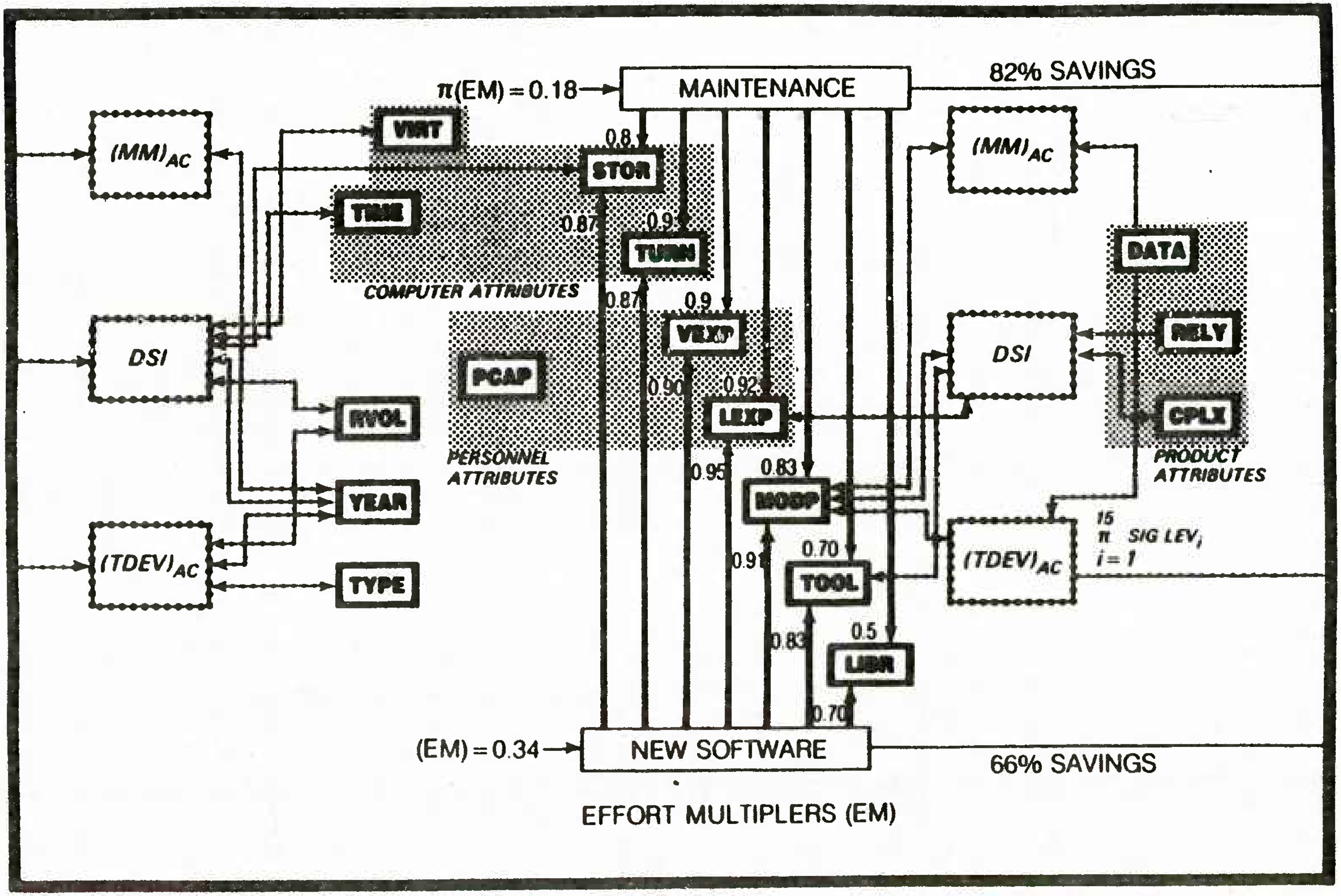
AEXP Analyst Experience. Added tools may help shorten analysts' learning time.

Function of Total Weapon System

SCED Required Schedule for Completion of software.

CPLX Complexity of Software Product.

DATA Size of run-time data base that must be accessed, relative to the size of programs processed at run-time by the target machine.



SOFTWARE FACTORY MODEL
Computer System Parameter Interdependencies

Figure 5-1. Graphic Illustration of Relationships in Table 5-1.

B. ASSUMPTIONS AND CAUSE-EFFECT CHAIN LEADING TO SUCCESSFUL IMPLEMENTATION OF THE GFE APPROACH

1. Introduction

Post-mortem reviews of complex programs often reveal that the assumptions made were incomplete or incorrect. For a program as important as the GFE/Environment we need to verify to the extent possible, the reasonableness of the assumptions and expected results for cause-effect transactions.
Projecting results in graphic form is a useful research tool for this purpose. It is particularly useful for analyzing complex programs that involve economic and organizational considerations as well as technical aspects.

The cause-effect chain -- a distant cousin of decision analysis -- is such a tool. The cause-effect chain is most used for analyzing projects that require successful negotiation of a series of steps before they can be completed. The chain's usefulness comes from its helpfulness for identifying and making explicit the assumptions made at each step, and the reasonableness of the results expected from each successive action in the chain.

2. Implementing the GFE/Environment

Figure 5-2 shows the chain of assumptions and cause-effect relationships for the GFE/Environment. It begins with "Generic standards are Valuable" and ends with "Lower Cost to DOD for New Software". The three rows of boxes crossing the figure show the assumptions and cause-effect relationships for: (1) use of a single standard programming language (top row); (2) use of a GFE/Environment (middle row); and (3) the resulting effect on a defense system project that contains hardware, software, facilities, data, and people (bottom row).

Quantifying Assumptions. The four shaded boxes at the left, and the one near the middle of the figure, represent basic assumptions that are made (sometimes implicitly) and seldom questioned. The three dashed boxes at the right represent results desired from the project. Throughout the figure, each pair of boxes connected by an arrow represents a cause-effect transaction that is assumed to be effective.

We show 27 numbered boxes on the figure. It is possible to quantify or measure at points associated with those 27 boxes, then to include the measurements in the equations of an econometric model. In Table 5-2 we indicate the box numbers, and indicate the type of information for those points that is required by an econometric model. We also show Technion International's estimates of expected ranges of data.

C. POINTS AT WHICH ADDITIONAL DATA ARE NEEDED

Keyed to Figures 5-2 and 5-3, and Table 5-2, this section details the data required, and gives a range of probable values. Figure 5-3 shows excerpts from the flow of Figure 5-2, with more emphasis on blocks affecting the implementation of a HAPSE-like environment. Table 5-2 shows specific data points and indicates the ranges of values expected.

Quantification is needed at these points in Figures 5-2 and 5-3:

1. Language

- What is the improvement in programmer productivity, both for development and for subsequent enhancement/maintenance of software, associated with use of one standard programming language? (Box 3)

- What is the incremental expense of training programmers in the one standard language, to the skill level at which they are able to implement the language's special features in their work? (Box 5)

- What are the net benefits to programmers of using the language? (Box 6)

- What are the net benefits to contractors from having their programmers use the language? (Box 7)

- To what degree are reliability and maintainability enhanced by having software written in the standard language? (Box 8)

2. Environment

- What are the benefits of a standard integrated automated environment [HAPSE], in terms of improved productivity in development, reduced time to complete testing, and improved productivity in subsequent enhancement/maintenance of software? (Box 12)

- What will be the cost of designing a HAPSE, building it, and providing it as GFE to contractors? (Box 13)

- What will be the cost of training programmers in use of the HAPSE? (Box 14)

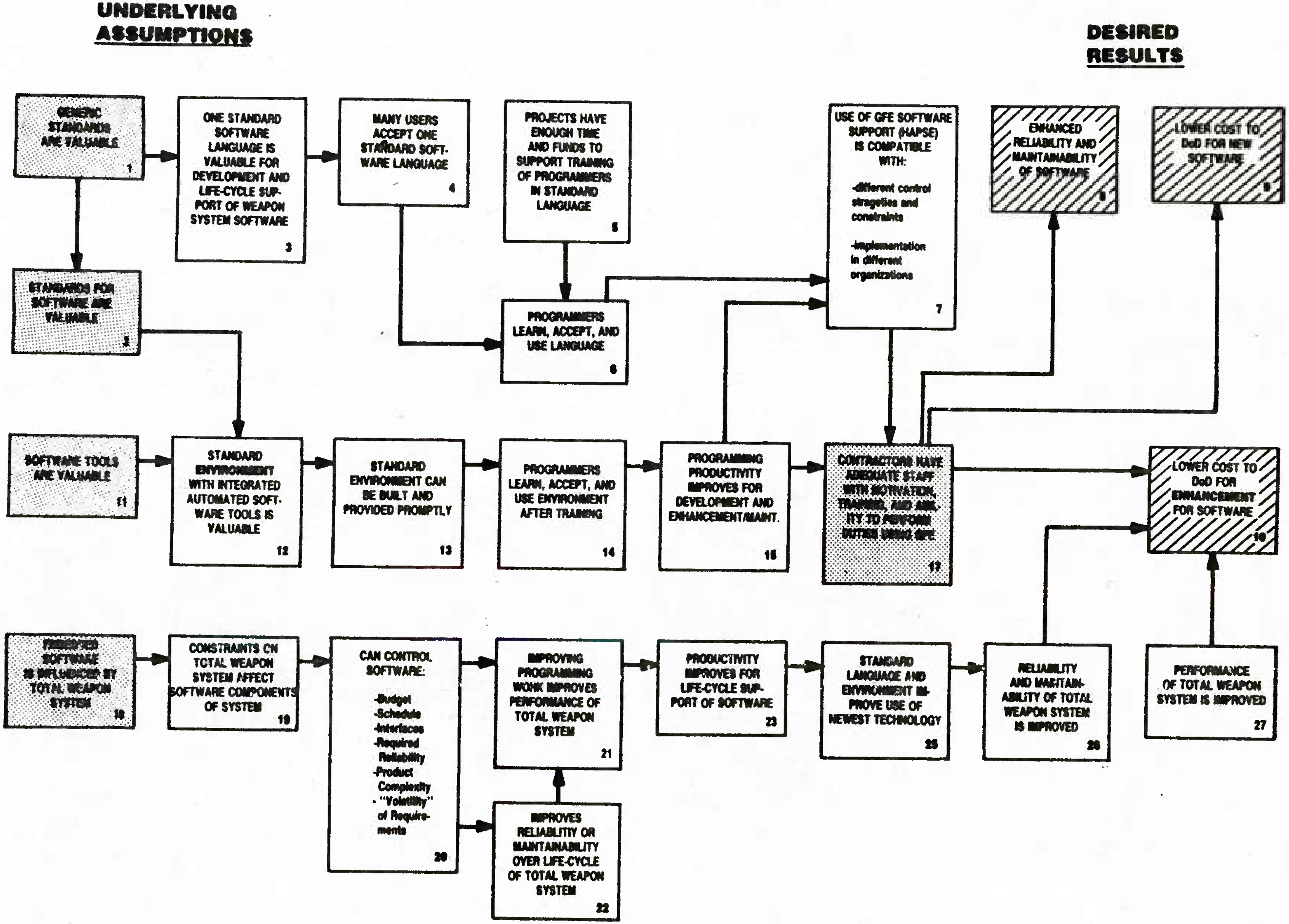

UNDERLYING ASSUMPTIONS
DESIRED RESULTS
GENERIC STANDARDS ARE VALUABLE 1
ONE STANDARD SOFTWARE LANGUAGE IS VALUABLE FOR DEVELOPMENT AND LIFE-CYCLE SUPPORT OF WEAPON SYSTEM SOFTWARE 3
MANY USERS ACCEPT ONE STANDARD SOFTWARE LANGUAGE 4
PROJECTS HAVE ENOUGH TIME AND FUNDS TO SUPPORT TRAINING OF PROGRAMMERS IN STANDARD LANGUAGE 5
USE OF GFE SOFTWARE SUPPORT (HAPSE) IS COMPATIBLE WITH:
-different control stragetics and constraints
-implementation in different organizations 7
ENHANCED RELIABILITY AND MAINTAINABILITY OF SOFTWARE 8
LOWER COST TO DoD FOR NEW SOFTWARE 9
STANDARDS FOR SOFTWARE ARE VALUABLE 2
PROGRAMMERS LEARN, ACCEPT, AND USE LANGUAGE 6
SOFTWARE TOOLS ARE VALUABLE 11
STANDARD ENVIRONMENT WITH INTEGRATED AUTOMATED SOFTWARE TOOLS IS VALUABLE 12
STANDARD ENVIRONMENT CAN BE BUILT AND PROVIDED PROMPTLY 13
PROGRAMMERS LEARN, ACCEPT, AND USE ENVIRONMENT AFTER TRAINING 14
PROGRAMMING PRODUCTIVITY IMPROVES FOR DEVELOPMENT AND ENHANCEMENT/MAINT. 15
CONTRACTORS HAVE ADEQUATE STAFF WITH MOTIVATION, TRAINING, AND ABILITY TO PERFORM DUTIES USING GFE 17
LOWER COST TO DoD FOR ENHANCEMENT FOR SOFTWARE 10
EMBEDDED SOFTWARE IS INFLUENCED BY TOTAL WEAPON SYSTEM 18
CONSTRAINTS ON TOTAL WEAPON SYSTEM AFFECT SOFTWARE COMPONENTS OF SYSTEM 19
CAN CONTROL SOFTWARE:
-Budget
-Schedule
-Interfaces
-Required Reliability
-Product Complexity
- "Volatility" of Requirements 20
IMPROVING PROGRAMMING WORK IMPROVES PERFORMANCE OF TOTAL WEAPON SYSTEM 21
IMPROVES RELIABLITIY OR MAINTAINABILITY OVER LIFE-CYCLE OF TOTAL WEAPON SYSTEM 22
PRODUCTIVITY IMPROVES FOR LIFE-CYCLE SUPPORT OF SOFTWARE 23
STANDARD LANGUAGE AND ENVIRONMENT IMPROVE USE OF NEWEST TECHNOLOGY 25
RELIABILITY AND MAINTAINABILITY OF TOTAL WEAPON SYSTEM IS IMPROVED 26
PERFORMANCE OF TOTAL WEAPON SYSTEM IS IMPROVED 27

Figure 5-2 - Assumptions and Causal Chain -- GFE/HAPSE

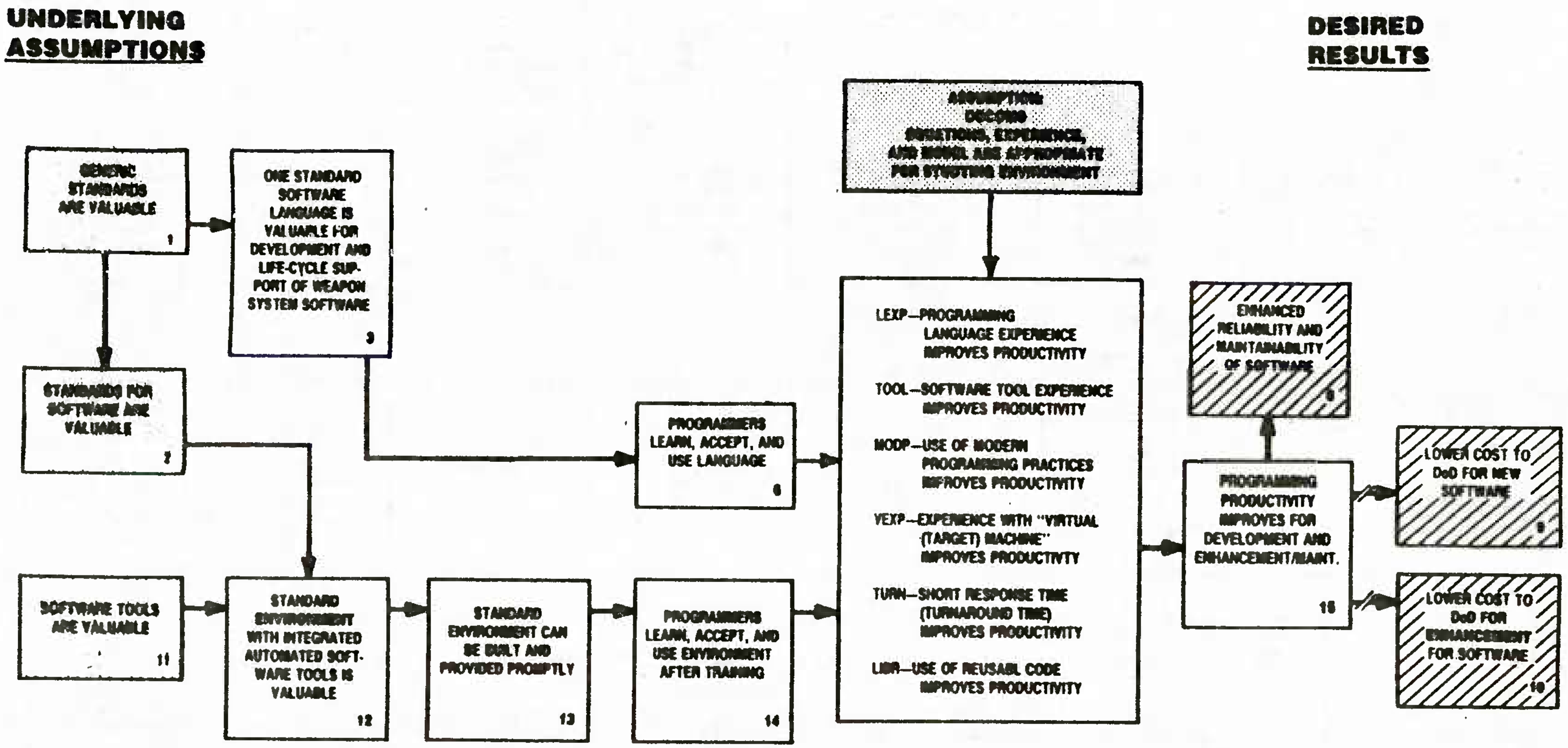

UNDERLYING ASSUMPTIONS
DESIRED RESULTS
GENERIC STANDARDS ARE VALUABLE 1
ONE STANDARD SOFTWARE LANGUAGE IS VALUABLE FOR DEVELOPMENT AND LIFE-CYCLE SUPPORT OF WEAPON SYSTEM SOFTWARE 3
STANDARDS FOR SOFTWARE ARE VALUABLE 2
SOFTWARE TOOLS ARE VALUABLE 11
STANDARD ENVIRONMENT WITH INTEGRATED AUTOMATED SOFTWARE TOOLS IS VALUABLE 12
STANDARD ENVIRONMENT CAN BE BUILT AND PROVIDED PROMPTLY 13
PROGRAMMERS LEARN, ACCEPT, AND USE ENVIRONMENT AFTER TRAINING 14
PROGRAMMERS LEARN, ACCEPT, AND USE LANGUAGE 6
ASSUMPTION: DSCOMO QUESTIONS, EXPERIENCE, AND MODEL ARE APPROPRIATE FOR STUDYING ENVIRONMENT
LEXP—PROGRAMMING LANGUAGE EXPERIENCE IMPROVES PRODUCTIVITY
TOOL—SOFTWARE TOOL EXPERIENCE IMPROVES PRODUCTIVITY
MODP—USE OF MODERN PROGRAMMING PRACTICES IMPROVES PRODUCTIVITY
VEXP—EXPERIENCE WITH "VIRTUAL (TARGET) MACHINE" IMPROVES PRODUCTIVTY
TURN—SHORT RESPONSE TIME (TURNAROUND TIME) IMPROVES PRODUCTIVITY
LIBR—USE OF REUSABL CODE IMPROVES PRODUCTIVITY
PROGRAMMING PRODUCTIVITY IMPROVES FOR DEVELOPMENT AND ENHANCEMENT/MAINT. 15
ENHANCED RELIABILITY AND MAINTAINABILITY OF SOFTWARE 8
LOWER COST TO DoD FOR NEW SOFTWARE 9
LOWER COST TO DoD FOR ENHANCEMENT FOR SOFTWARE 10

Figure 5-3. Causal Chain - Programming Support Environment.

- What are the incremental costs and benefits of having contractors use the GFE, instead of their customary software production facilities? (Box 15)

- How much lower will be the cost to DoD for development of NEW software? (Box 9)

- How much lower will be the cost to DoD for subsequent enhancement and continuing development of software? (Box 1Ø)

3. System Containing the Software

- What is the likelihood that characteristics of the weapon system (which drive the software development and maintenance efforts) are compatible with the design of the HAPSE and the environment in which it is used [e.g., system budget, schedule, required reliability, complexity, and volatility of requirements on software]? (Boxs 19 and 2Ø).

- What will be the effect of the resulting software on reliability and maintainability of the defense system over its life cycle? (Box 22).

- What will be the effect of the resulting software on the performance of the weapon system? (Box 27).

Table 5-2. DATA REQUIRED BY ECONOMETRIC MODEL

Figs 5-2,-3 box number:	Data Required	Range of Probable Values
(N/a)	GFE/HAPSE continues to be improved, and overcomes pressures toward obsolescence.	HAPSE continues to provide productivity improvements amounting to at least 27 percent per year after its deployment.
(5)	Additional Funding needed by contractors to train their people in using HAPSE.	$1000 - $5000 per programmer to be trained (one to five weeks each). "Learning Curve" effects: includes initial loss of productivity, followed by shift to more favorable GFE/HAPSE learning curve, with long-term gain.
(4); (12)	Productivity is raised, by language and software tools used in HAPSE.	Ada language produces 8-10 machine-language instructions per Ada language instruction.
(6); (14)	Programmers Learn, accept, and use HAPSE after training.	Set of integrated, automated software tools helps programmers produce better products and becomes part of their normal "tool kits".
(13)	Standard Environment can be built and provided promptly.	Yes, within 5 years. Development cost depends on development strategy. Additional annual direct support cost will also be significant.
(15)	Productivity improves for Initial Development as well as for post-deployment enhancement/maintenance of software.	Productivity increases by factor of two to four.

TABLE 5-2, concluded

(17)	Contractors have adequate staff, with motivation, training, and ability to perform duties using GFE/HAPSE.	From 25 to 5Ø percent of contractors' project staff have adequate motivation, training, and ability to use GFE/HAPSE.
(25)	GFE/HAPSE aids in transition and use of latest technology on AFSC MCCR projects.	Range of values, from plus 5Ø percent (net help) to minus 25 percent (net interference) with transition.
(26); (8)	GFE/HAPSE use gives improved reliability, maintainability, and effectiveness of weapon system.	Range of values, from zero to doubling of RMA with accompanying decreases in PDSS costs.
(9)	GFE/HAPSE lowers costs for developing new software.	Two-fourfold increase in productivity, with greater reliability and maintainability of new software products.
(1Ø)	GFE/HAPSE lowers time and costs for post-deployment modifications of fielded software.	Doubled productivity in post-deployment enhancement/change of operational software.

Estimates for Data Range were made by Technion International, and are based on a wide variety of published and unpublished reports. However, data in the right column should be used for no purpose other than to plan data collection efforts.

SELECTED BIBLIOGRAPHY

(Alfo81) Alford, M. W., and Davis, C. G. " Experience with the Software Development System". In Software Engineering Environments, H. Hunke, Editor, North-Holland, 1981.

Paper presents a methodology and supporting environment for developing very large, complex, real-time systems. The environment emphasizes importance of discovering errors early in the development process. The methodology consists of four major tasks: (1) Data Processing Systems Engineering (DPSE) - translate systems objectives into a consistent, complete set of subsystem functional and performance requirements (uses techniques based on verification graphs, petri nets, finite state machines, and graph models of decomposition for expressing requirements), (2) Software Requirements Engineering Methodology (SREM) - express functional and performance requirements as a graph model in Requirements Statement Language (RSL) and analyze with the Requirements Engineering Validation System (REVS), (3) process design engineering - translate requirements into a process design language, verify design, and evolve the design into code, (4) verification and validation - perform at all stages.

(Arth83) Arthur, Lowell Jay. Programmer Productivity: Myths, Methods and Murhphology. New York: John Wiley. 1983.

Book describes software productivity improvement methods used by a unit of American Telephone and Telegraph company. Reusable code is an important element in the program described.

(Barn82) Barnard, P., Hammond, N., MacLean, A., and Morton, J. "Learning and Remembering Interactive Commands." In Proceedings of Human Factors in Computer Systems, Washington, D. C. Chapter, ACM, Gaithersburg, MD, 1982.

Paper presents an experiment to determine how the choice of command names influences interactive performance of users by measuring access to an on-line help system. The authors propose that help assistance is a relatively passive cognitive strategy for learning and that it leads to less efficient operation retention if commands have general names.

(Bars84) Barstow, D. R., et al. Interactive Programming Environments. New York: McGraw-Mill Inc. 1984.

Book is a collection of papers by authorities in the field of interactive programming environments. It presents developments to about 1982 from the fields of programming methodology, artificial intelligence and software engineering. The book educates the reader to save time and increase productivity using interactive programming environments.. The book discusses UNIX, LISP and PASCAL with reference to interactive environments.

(Basi84) Basili, V. R., and Perricone, B.T. "Software Errors and Complexity: An Empirical Investigation." Communications of the ACM, Vol. 27, No. 1, Jan 1984, pp. 43-52.

(Baye81) Bayer, M., et. al. "Software Development in the CDL2 Laboratory." In Software Engineering Environments, H. Hunke, Editor, North-Holland, 1981.

Paper describes a programming environment that supports the CDL2 programming language. The components of the system include a command interpreter, a dedicated editor/with formatting and cross-reference, a program database (underlies all tools), a file system (underlies the program database), a local and global analyzer, an intermodule interface checker, a local and global optimizer, a segment builder, an abstract code-generator, and a concrete code generator. The paper includes an in-depth discussion of the system architecture.

(Boeh81) Boehm, Barry W. Software Engineering Economics. Englewood Cliffs: Prentice-Hall, 1981.

Book presents a highly detailed description of the COCOMO cost estimating system. It also gives a 63 project database, from which the COCOMO equations were derived.

(Boeh84) Boehm, Barry W. et. al. "A Software Development Environment for Improving Productivity." In Computer, Vol. 17, No. 6, June 1984.

Paper presents the background and status of TRW's Software Productivity System (SPS). The background includes discussion of a 1980 software productivity study and TRW corporate motivation for the study (e.g., increased demand of software, limited supply of software engineers, rising software expectations, reduced hardware costs). Based on an internal assessment, an external assessment, and a quantitative assessment, the productivity study recommends that TRW initiate a long range effort to develop a software development environment. In the short term, the study recommends the development of a prototype environment. The architecture and components of the prototype that was developed (called SPS-1) include: the work environment (improved office conditions), the hardware (a network of VAX's, LSI-11/23's, terminals, and communication equipment), a master project database (composed of a hierarchical file system, a source code control system, and a relational database), general utilities (menu, screen editor, forms package, date/time, report writer), office automation and project support (tool catalog, mail system, text editor/formatter, calendar, forms management, interoffice correspondence package), and software development tools (requirements traceability tool, SREM (Alfo81), program design language, Fortran-77 analyzer). Experience with the use of the prototype shows a definite improvement in productivity and also that immediate access to a good set of tools has the highest payoff.

(Booz81) Booz, Allen and Hamilton, Inc. "Final Report: Defense Acquisition Study." Washington, D.C., 1981.

This is the official document describing the USAF "Acquisition Improvement Project", headed by Col. Don Sawyer. Among other matters, the report describes the delays in acquisition of ADP-related equipment caused by multiple levels of required review and approval within the Air Force. The report differentiates delays caused by duplicative internal procedures (which routinely exceed a year). USAF has the potential ability to control these delays by modifying its review and approval procedures. Study isolated internal USAF variables from factors such as congressional directives over which USAF has little direct influence.

(Bran81) Branstad, Martha A., Adrion, W. Richards, and Cherniavsky, John C. "A view of Software Development Support Systems." In proceedings of NEC, Chicago, IL, October, 1981.

Paper presents views and examples of support environments and proposes four classes of support levels. The views include the toolbox approach, the VHLL (very high level language) approach, and the software development support system approach (an integrated system with an underlying database). Examples include PWB/Unix, Interlisp, Howden's environments (Howd82), and the Ada Programming Support Environment (APSE). The proposed four classes of support levels are: D1 (what most operating systems provide), D2 (the capabilities of D1 plus integration with an underlying database), D3 (D2 plus support for the entire life cycle), and D4 (D3 plus features that are current research topics). Extensions of each level are also proposed for critical applications.

(Bran81a) Branstad, Martha A., and Adrion, W. Richards, Editors. NBS Programming Environment Workshop Report. National Bureau of Standards, NBS SP 500-78, June 1981.

Report presents the results of the Programming Environment Workshop, Rancho Santa Fe, CA, April 1980. The goals of the workshop were to assess the current technology, indicate needed standards for tools, develop guidance for near term environment development, and determine future research directions. The workshop attendees were divided into four groups: (1) contemporary software development environments, (2) software environment research - the next five years (3) advanced development support systems, and (4) high level language programming environments. The results of each of the four groups is reported by their respective leaders, W. Howden (Howd82), L. Osterweil (Oste81), T. Standish, and M. Zelkowitz.

(Broo75) Brooks, Frederick P., Jr. The Mythical Man-Month: Essays on Software Engineering. Reading, Massachusetts: Addison-Wesley, 1975.

Book is an early collection of essays describing development of software as seen by the responsible manager. Brooks managed development and support of IBM's OS/360 system, and his examples are drawn from that experience. Many of his insights are portable to other software projects, such as those in defense systems.

(Buxt80) Buxton, J. "Requirements for Ada Programming Support Environment: STONEMAN." U.S. Department of Defense, Washington, D.C., February 1980.

Report lays out the requirements for Ada environments. The general requirements include support for the entire lifecycle, integration of tools, and exploitation of modern hardware. The more specific requirements include: the Ada Programming Support Environment (APSE)/Minimal APSE (MAPSE)/Kernal APSE (KAPSE) model, database support, and tool support at the different APSE/MAPSE/KAPSE levels. The APSE/MAPSE/KAPSE model is a four level model where level 0 is the host level, level 1 (KAPSE) is a standard interface to level 0 that supports database interactions, communications, and run-time, level 2 (MAPSE) is a minimal tool set (editor, translator, linker, debugger, configuration manager), and level 3 (APSE) is a set of tools for full support (life cycle, documentation, and management support).

(Buxt80a) Buxton, J. "An Informal Bibliography on Programming Support Environments." In ACM Sigplan Notices, Vol. 15, No. 12, December 1980.

Bibliography with brief notes and commentary on 40 papers that deal primarily with architectures of programming support envirronments. Included with the bibliography are short summaries of the following proposed and existing programming support environments: Cheatham's PDS Model, Cooprider's Thesis, CADES, C-MESA, Softech's MSEF, Stenning's Foundation System Model, and Tichy's Model.

(Camp84) Campbell, Roy H., and Kirslis, Peter A. "The SAGA Project: A System for Software Development." In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Gaithersburg, MD, April 1984. Authors discuss the SAGA Project and its current status. The project proposes to develop a lifecycle support environment for small to medium size projects. At the heart of the system is a proposed language-oriented editor generator that can synthesize a language-oriented editor for each life cycle language (i.e., requirements language, design language, implementation language, etc.). The current system is UNIX based and includes a prototype for a language-oriented editor (implementation language), prototype version control components, and a production documentation tool.

(Cast84) Castor, Virginia L., et. al. "Evaluation and Validation (E&V) Team Public Report", Vol. I., Interim Technical Report for Period 1 October 1983-3Ø September 1984. AFWAL TR 85-1Ø16. Wright-Patterson AFB, Ohio, 45433-6543.

Report is a collection of papers developed during the year and dealing with the U.S.A.F. research into use of software development and support environments.

(Cowe83) Cowell, Wayne R., and Osterweil, Leon J. "The Toolpack/IST Programming Environment." In Proceedings of SoftFair, (IEEE Order No. 83CH1919-Ø), July 1983.

The paper discusses a portable, Fortran-oriented programming environment. The architecture of the environment includes a command interpreter, tool fragments (commands may invoke several tool fragments), and a virtual file system (files created by tools can be recreated by tools). The tools in the environment include: data flow analyzer, program instrumenter and debugger, documentation generation aid, program formatter, syntax-controlled editor, macro processor, text formatter, program structurer, Ratfor processor, portability checker, program transformer, and various file-handling utilities.

(Cox83) Cox, Brad J. "The Message/Object Programming Model." In Proceedings of SoftFair, (IEEE Order No. 83CH1919-Ø), July 1983.

A tutorial on object-oriented programming, the paper discusses the operator/operand model and the message/object model. It then presents a case study developed on Smalltalk-8Ø.

(Deli84) Delisle, Norman M., Menicosy, David E., and Schwartz, Mayer D. "Viewing a Programming Environment as a Single Tool." In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Gaithersburg, MD, April 1984.

Paper presents an interactive programming environment called Magpie. The user of Magpie deals with two states, the status of the source code and the status of execution. The environment features overlapping windows, a mouse pointing device, pop-up menus, a browsing capability, language-directed editing, incremental compiling, and debugging capabilities.

(Fair8Ø) Fairley, Richard E. "Ada Debugging and Testing Support Environments." In Proceedings of the ACM-Sigplan Symposium on the Ada Programming Language, December 198Ø (see SIGPLAN Notices, Vol. 15, No. 11, November 198Ø).

A review of the requirements specified in (Buxt8Ø) and a discussion of the issues associated with them. Analysis considerations, source level support and debugging, KAPSE consideration, and data abstractions are covered.

(Feil81) Feiler, Peter H., and Medina-Mora, Raul. "An Incremental Programming Environment." In Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

Paper reviews the support required by programming ennvironments and the traditional method for providing this support (i.e., editing, translation, linking, loading, and debugging). It then presents the Incremental Programming Environment which features syntax-directed editing, common representation, incremental program translation, and language oriented debugging. This is followed by a discussion of design and implementation issues for such an environment.

(FIPS99) National Bureau of Standards. FIPS PUB 99. "Guideline: A Framework for the Evaluation and Comparison of Software Development Tools." March 1983.

Guideline presents a framework (a taxonomy) for identifying, discussing, classifying, evaluating, and comparing software development tools and environments. The taxonomy includes almost 1ØØ features that are presented in a hierarchy. At the top level, features are divided into input/output categories or function categories. The functions include transformation, static analysis, dynamic analysis, and management. The appendix includes a set of event sequences for the acquisition of tools.

(Fisc84) Fischer, C. N., et. al. "The Poe Language-Based Editor Project". In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments, Gaithersburg, MD, April 1984.

Paper presents an overview of POE, a Pascal programming environment, and presents some of the technical issues associated with the development of the environment.

(Fox82) Fox, Joseph M., Software and Its Development, Englewood Cliffs, Prentice-Hall, 1982.

Seldom does a successful manager write a book; most seem to be action people, who prefer to manage rather than to analyze and describe what is actually involved in managing software through its life cycle. In this book, the author draws on his experience with software development and support in large systems. His vantage point was that of manager of IBM's Federal Systems Division. His account differs from most literature, in that he reports the system and software life cycle as it appears from the top, not as he imagines it or wishes it were. His insights are extremely valuable for those who wish to understand the "big picture".

(Fran84) Frankel, S. "Introduction to Software Packages." NBS Special Publication 500-114. April 1984.

This report introduces use of applications software packages and directs potential users to sources of information. A review of the benefits of such a usage is made and use of software pellets versus in-house development is discussed.

(FSTC83) ________. "Software Tool Evaluation and Selection Guidelines." Report OIT/FSTC-83/016. August 1983.

(FSTC83a) U. S. General Services Administration, Office of Software Development and Information Technology, Federal Software Testing Center. "Programmer Productivity Aid Catalog." Report OSD/FSTC-83/017, Sept. 30, 1983.

Report lists over 100 productivity aids that were available in 1983. "The productivity aids that are included in this catalog address the system life cycle areas of design, development (code and application generation), testing, maintenance, documentation, and project management. A single aid may have application in only one phase of the life cycle or may be used in several phases. . . . These products have not been tested or validated by FSTC. . ." (pp i-ii). Many of the products listed have more than 1000 users. (Also see Houg82a).

(GAO81) U. S. General Accounting Office. Federal Agencies Maintenance of Computer Programs: Expensive and Undermanaged." AFMD-81-25, Feb. 26, 1981.

(GAO83) ________. Greater Emphasis on Testing Needed to Make Computer Software More Reliable and Less Costly. GAO/IMTEC-84-2, October 27, 1983.

This paper reports on a survey of federal agencies (in which USAF organizations are included). The survey shows that more than 60 percent of computer programs are modified because requirements have changed. About 80 percent of programs reported were modified in some way each year.

(Gutz81) Steve Gutz, Anthony I. Wasserman, and Michael J. Spier. "Personal Development Systems for the Professional Programmer." In Computer, Vol. 14, No. 4, April 1981.

This paper reviews the problems with existing development environments, proposes a programmer's personal machine, and examines the advantages and disadvantages of such a machine. The proposed programmer's personal machine consists of: (1) an intelligent terminal with 1 Meg local storage, CPU and address space equivalent to a 32-bit mini, graphics capability, (2) hard disk (4Ø Megs) and floppy disk, (3) networking capability (with other PPBS's), (4) audio input/output, (5) pointing device (mouse, tablet, or light pen), and (6) capability to add more memory and other devices (e.g. a quality printer). The potential advantages of such a machine include constant response time, a comprehensive set of tools, less dependence on a single machine, integration of software development and office automation. The potential disadvantages include tool expense, training expense, communications, device dependence.

(Guya84) Guyard, Jacques, and Jacquot, Jean-Pierre. "MAIDAY: An Environment for Guided Programming with a Definitional Language." In Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84CH2Ø11-5), March 1984.

Paper discusses an environment under development that is oriented around an object-oriented language and an algorithm design methodology. The environment enforces the methodology through a set of control functions. The user views a development session through a set of windows that present the development level, messages, the object being defined, objects remaining to be defined, the stored algorithm, and the current command.

(Hall8Ø) Hall, Dennis E., Scherrer, Deborah K., and Sventek, Joseph S. "A Virtual Operating System." In Communications of the ACM, Vol. 23, No. 9, September 198Ø.

Paper presents a UNIX-like environment that can be implemented on top of a vendor-supplied operating system to make in-effect a virtual operating system. The environment consists of four layers: (1) the vendor-supplied operating system (the innermost layer), (2) the virtual machine (consisting of primitives such as opening and closing files, reading and writing to files), (3) utilities (a set of tools written in portable Fortran including Kernighan and Plauger's Software Tools), and (4) an integrated command interface.

(Haus81) Hausen, Hans-Ludwig, and Mullerburg, Monika. "Conspectus of Software Engineering Environments." In Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

A paper which discusses the issues associated with the coverage of software engineering environments. The issues include support for full life cycle development, quality assurance, product control, management, specific applications, specific methodologies, and representation schemes. Also discussed are issues related to the integration of tools and motivations for developing environments. The appendix defines the criteria that must be met for a system to be considered an environment. These include: a software engineering orientation, the use of at least one recognized scientific concept, applicability to more than one life cycle phase, and some level of tool integration. The authors present short summaries of environments that they feel meet these criteria. They include AIDES, APSE, ARGUS, ASSET, CADES, CDL2-Lab, COSY, DREAM, Gandalf, Gypsy, HDM, ISES, PWB/UNIX, SDEM/SDSS, REVS, and SEF.

(Hech81) Hecht, H. Final Report: A survey of Software Tools Usage. NBS Special Publication 500-82. November 1981, p. 53.

This special Publication from NBS/ICST comprises a software tool usage study as a part of an effort to develop methodologies and standards for software quality control. The report gives a survey of software tool usage and an analysis of the findings.

(Hech82) Hecht, H., and Houghton, R., Jr. "The Current Status of Software Tool Usage." Proceedings of COMPSAC 82, November 82.

(Hech82a) Hecht, H. The Introduction of Software Tools. NBS Special Publication 500-91. September 1982, p. 35.

This special NBS/ICST publication discusses specific needs for software tools in programming for management information systems and for Scientific applications. Steps for the successful introduction of tools are discussed and measures are described to deal with organizational obstacles and difficulties posed by existing computer installations.

(Houg82a) Houghton, Raymond C., Jr. Software Development Tools. National Bureau of Standards (NBS) Special Publication 500-88. March 1982.

Data base of more than 400 software tools, classified according to the FIPS99 taxonomy. Appendices printe the database in different sorts, such as language written in, and hardware that tools are intended for. Data in this data base are current as of 1982. Data base is updated and maintained by RADC, but not classified according to the FIPS99 taxonomy.

(Houg82b) __________. "A Taxonomy of Tool Features for the Ada Programming Support Environment (APSE)". National Bureau of Standards, NBSIR 82-2625, December 1982.

A review of the APSE (Buxt80), the ALS (Wolf81), and the AIE (the Navy's Ada Integrated Environment) based on (FIPS99). The taxonomy includes a comparison of features in the areas of management, static analysis, dynamic analysis, transformation, and input/output. A set of underlying tool primitives is defined that support these features.

(Houg84a) __________. "Help Systems: A Conspectus." In Communications of the ACM, Vol. 27, No. 2, February 1984.

Paper discusses online assistance that is provided by various types of environments. It includes a discussion of the types of assistance and the issues associated with the development of online help systems.

(Houg84b) __________. "Comparing Software Development Methodologies for Ada: A Study Plan." National Bureau of Standards. NBSIR 84-2827.

The study outlines the Support Systems Task Area of the DoD STARS Program on the Software Developmental Methodolgoies and Ada. The report treats the methodology as a "black box" in an effort to simiplify the earlier model that was partitioned to design and corrective implementation phases.

(Howd82) Howden, William E. "Contemporary Software Development Environments". In Communications of the ACM. Vol. 25, No. 5, May 1982.

Paper proposes four levels of tool support that could be provided by software engineering environments. For each level, the type of project, the estimated cost, and the support provided is detailed. For example, environment I has an estimated cost of $35K and is for medium-sized projects, while environment IV has an estimated cost of $3M and is for large scale projects. Requirements, design, coding, verification, and management tools and techniques are presented for each environment level.

(Huff81) Huff, Karen E. "A Database Model for Effective Configuration Management in the Programming Environment." In Proceedings of the 5th International Conference on Software Engineering. IEEE Order No. 81CH1627-9. March 1981.

Paper addresses configuration mangement issues (i.e., configuration identification and configuration control) in a software engineering environment and presents a model for effectively handling them.

(Hunk81) Hunke, H., Editor. Software Engineering Environments. Amsterdam: North-Holland. 1981.

Book contains the proceedings of a symposium held at Lahnstein, Federal Republic of Germany, June 1980. Some of the papers include (Snow81), (Ridd81), (Alfo81), and (Mats81). Papers related to some in the book are (Tayl84), (Stu83), (Buxt80), and (Kern81). Other papers discuss issues and tools related to software engineering environments including functional aspects of environments, computer aided design, support for concurrent and distributed systems, human factors, description languages, productivity, formal verification, performance, system decomposition, and version control. The book concludes with a bibliography by Hausen, Mullerburg, and Riddle that contains more than 350 citations from 1968 to 1980.

(Ichb84) Ichbiah, J., "Ada: Past, Present, Future." In Communications of the ACM. Vol 27, Number 10, October 1984, pp. 991-997.

The article decribes the genesis, conception and reality of Ada and is outlined in the form of an interview with the Principle Designer of Ada.

(IEEE82) "An American National Standard-IEEE Standard Glossary of Software Engineering Terminology." ANSI/IEEE Std 729-1983. Sponsored by the Software Engineering Technical Committee of the IEEE Computer Society, and approved by the American National Standards Institute, 1982.

The terminology developed and presented in this report is a representation of a consensus on the subject within the Institute as well as those outside of IEEE with similar interests. The "Glossary" documents the increasing amounts of new terms and new meanings that are being adopted for existing terms. Its purpose is to "promote clarity and consistency in the vocabulary of software engineering" and serve as a useful reference for software engineers. This version has little terminology dealing with software development environments, but does present descriptions of some software tools.

(Kern81) Kernighan, Brian W., and Mashey, John R. "The UNIX Programming Environment." In Computer. Vol. 14, No. 4, April 1981.

A paper which extols the benefits of the UNIX programming environment. It reviews the underlying interface, i.e., the hierarchical file system and the seven primitive functions: open, create, read, write, seek, close, and unlink. It reviews the overlying interface (the user interface), i.e., available tools, input/output redirection, and various operators available to the user. It then discusses how a user can avoid programming by building a shell procedure of simpler tools available on the system. Finally, the attributes of the system are discussed, e.g., support for medium size projects, support primarily for the latter stages of software development, loose integration of tools and facilities, general support for all applications.

(Kern84) Kernighan, Brian W., and Pike, Rob. The UNIX Programming Environment. Englewood Cliffs: Prentice-Hall. 1984.

Book presents a more detailed account of the UNIX programming environment, suitable for reference.

(Klum85) Klumpp, A. R. "Space Station Flight Software: HAL/S or Ada." In Computer. March 1985, pp. 20-28.

Paper presents pros and cons to NASA of programming in Ada as opposed to HAL/S. Conclusion is that Ada is feasible for new software.

(Kuo83) Kuo, Jeremy, Ramanathan, Jay, Soni, Dilio, and Suni, Markku. "An Adaptable Software Environment to Support Methodologies." I n Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

Paper describes an environment that can be tuned to support different software development methodologies. The control mechanism is based on the gathering of project data through a forms-based interface. The forms are defined at the start of development.

(Lebl84) Leblang, David B., and Chase, Robert P., Jr. "Computer-Aided Software Engineering in a Distributed Workstation Environment." In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments. Gaithersburg, MD, April 1984.

Paper discusses an Apollo-based distributed software engineering environment. Because instances of the system can be running at various nodes in the environment, the system consists primarily of managers that keep track of what is going on. The managers include: A history manager (source code control), a configuration manager (version control), a task manager (tracks interrelationships among development products), monitor manager (watches user defined dependencies), and an advice manager (tracks general project information).

(Ledg82) Ledgrad, M. F., and Singer, A. "Scaling Down Ada." Communications of the ACM. Vol 25, Number 2, February 1982, pp. 121-125.

This article stresses that through Ada is an ambitious programming language its size and complexity plague its technical success. The paper gives means of streamlining the language and providing an authorized subset in an effort to scale it down.

(Love83) Love, Tom. "Experiences with Smalltalk-80 for Application Development." In Proceedings of SoftFair. IEEE Order No. 83CH1919-0. July 1983.

Paper extols the benefits of the single-user, single language environment called Smalltalk-80. An example is presented of a graphics program that was developed using the object-oriented development methodology that is supported by the system. The "mode-less" user interface and the performance benefits of the interface structure and mouse are also discussed.

(Mage84) Magel, Kenneth. "Principles for Software Environments." In ACM Software Engineering Notes. Vol. 9, No. 1. January 1984.

Paper lists and discusses a set of environment principles that include the following: generality (full life cycle support), adaptability (portability), user orientation (designed for a specific community), tailorability (adaptable to many types of interface devices), extensibility (new tools can be added), consistency (consistent use from part of the system to another), unification (user can anticipate how unfamiliar tools will operate), abstraction (hide as many details as possible), aggregation (bigger tools from smaller ones), incremental preparation, efficiency, predictability, subsetable, ability to group resources, and recoverability.

(Mats81) Matsumoto, Y., et. al. "SWB System: A Software Factory." In Software Engineering Environments. H. Hunke, Editor. North-Holland, 1981.

Paper discusses a large scale software factory that consists of three physical buldings, 2000 employees, a methodology, a software environment, and management and quality control. The software products are for critical applications (nuclear power stations) and there is an emphasis on using reusable code. The software environment consists of a number of tools and techniques that empphasize the latter part of the life cycle (language and library processors, editors, debuggers, etc.). The plans for the environment include the addition of tools for the front end (SADT, design languages, etc.).

(Metz81) Metzger, J. J.. and Dniestrowski, A. "PLATINE: A Software Engineering Environment". In Proceedings of SoftFair. IEEE Order No. 83CH1919-0. July 1983.

Paper describes an environment that consists of a methodology (the PLATINE methodology) and a set of tools (the PLATINE tools). The environment supports the entire life cycle, is adaptable to product size, supports different types of users, and supports host/target development. The methodology consists of defining a software structure hierarchy (software structuration) which produces typed abstract objects which are then associated with elements (source, listing, binary, map, nomenclature, or status). The methodology also includes the production of software (merging of the elements), project management, and evolution. The user interface consists of a command language and a set of full screen panels. The tools include LSTR (specification of real time embedded systems, derived from PSL/PSA), SDL (system design representation), Metacomp (YACC like), EPCS (a project management tool based on DEC's project control system), a formatter (DEC's runoff), a screen editor (DEC's EDT), documentor (editor from source code), mail (DEC's), crossrf (data dictionary cross reference), complex (a complexity measure), a configuration controller, and a comparator.

(Myer85) Myers, W. "An Assessment of the Competitiveness of the United States Software Industry." In Computer. March 1985, pp. 81-92.

(Oste81) Osterweil, Leon. "Software Environment Research: Directions for the Next Five Years." In Computer, Vol. 14, No. 4, April 1981.

Paper discusses research issues associated with software engineering environments, in particular, the breadth of scope and applicability, user friendliness, reusability of internal components, tight integration of tool capabilities, and use of a central database. A five-year research plan is presented which includes studies of existing support systems, tool fragment studies, data base studies, construction, and test beds for configuring environments.

(Oste82) __________. "Toolpack - An Experimental Software Development Environment Research Project". In Proceedings of the 6th International Conference on Software Engineering, (IEEE Order No. 82CH1795-4), September 1982.

An implementation of (Oste81). Paper presents an environment under development that concentrates on tight integration of tool capabilities (use of tool fragments) and an underlying central database (virtual file system). See also (Cowe83).

(Pari84) Parikh, Girish. Programmer Productivity: Achieving an Urgent Priority. Reston, Virginia: Reston Publishing Company. 1984.

(Phis79) Phister, Montgomery, Jr. Data Processing Technology and Economics. 2nd ed. Santa Monica, Calif.: Digital Press, 1979.

In this unique book, the author has collected nearly 7ØØ pages of technical and economic history relating to data processing hardwre and software. The author's careful economic interpretation of technical data (through 1978) is the most thorough and complete in the public domain. His analysis shows clear trends that probably remain true in the 198Øs.

(Pren81) Prentice, Dan. "An Analysis of Software Development Environments". In ACM Sigsoft Software Engineering Notes, Vol. 6, No. 5, October 1981.

A paper which emphasizes the problems. The issues discussed include lack of hardware support, high cost, user resistance to change, and poor user interfaces.

(Ridd81) Riddle, W. E. "An Assessment of Dream." In Software Engineering Environments. H. Hunke, Editor, North-Holland, 1981.

Paper reviews the DREAM system. DREAM is oriented to the development of concurrent systems using DREAM Design Notation (DDN), a language that can be used to model a total system including hardware, software, and wetware (people, etc.). The model reflects the externally observable characteristics of a system and is an adequate basis for preparing implementation plans. The DREAM system tools include a data base core that stores DON fragments, bookkeeping tools (entry and retrieval), and decision-making tools for paraphrasing (a re-structured presentation), extraction (simulation), and consistency checking. The paper concludes with lessons learned and problems for the future.

(Ridd83) __________. "The Evolutionary Approach to Building the Joseph Software Develcpment Environment." In Proceedings of SoftFair, (IEEE Order No. 83CH1919-Ø), July 1983.

Paper reviews an effort to build an environment that was cut short due to the closing of Cray Labs. The Joseph environment was 3Ø% completed. The paper includes a user scenario of the proposed environment which includes capabilities to view database information, extract database information, perform notation-directed editing, analyze changes, and deposit information into the database. The paper then discusses the work that was accomplished which includes a layered environment with UNIX at the core, a set of integrated tools in the next layer (the crypt), and a requirements definition tool and a design definition tool in the outer layer (pharaoh and oasis). Pharaoh and oasis include viewing, notation-directed editing, and version control capabilities of requirements and design specifications. They use a notation that consists of keywords and description fragments.

(Rube83) Rubenstein, Burt L., and Carpenter, Richard A. "The Index Development Environment Workbench." In Proceedings of SoftFair, (IEEE Order No. 83CH1919-Ø), July 1983.

Paper presents a methodology and an associated environment for building application systems (informations systtems for business applications). The methodology divides an application system into a dialogue manager, a database processer, an output processor, an action processer, an extended data dictionary, and a control monitor. The environment includes facilities for data dictionary specification, structured graphics, screen definition, output processing (for developing mock-ups), defining control between components, and generic data entry. An example of the use of the system is presented.

(Sava82) Savage, Ricky E., Habinek, James K., and Barnhart, Thomas W. "The Design, Simulation, and Evaluation of a Menu Driven User Interface." In Proceedings of Human Factors in Computer Systems, Washington, D.C. Chapter, ACM, Gaithersburg, MD, 1982.

Paper discusses experiments relating to the user interface of an environment. An analysis of human errors led to the design of a system that provided an extensive hierarchy of menus for the inexperienced user and a variety of shortcuts to system functions for the experienced user.

(Shne8Ø) Shneiderman, B. Software Psychology: Human Factors in Computer and Information Systems. Cambridge, Mass.: Winthrop. 198Ø.

Book discusses a broad range of issues related to human factors in computers. Of particular interest to the developers of software engineering environments are the chapters on interactive interface issues and designing interactive systems. These chapters cover the user interface (control), response time, text editing, menu selection, error handling), the goals for interactive system (simplicity, power, user satisfaction, reasonable cost), and the design process (from a human factors standpoint).

(Silv85) Silverman, Barry G. "Software Cost and Productivity Improvements: An Analogical View." Computer. May 1985, pp. 86-96.

(Skel82) Skelly, P. G. "The ACM Position on Standardization of the Ada Language." In Communications of the ACM. Vol 25, Number 2, February 1982, pp. 118-120.

(Snow81) Snowdon, R. A. "CADES and Software System Development." In Software Engineering Environments. H. Hunke, Editor. North-Holland, 1981.

A review of one of the early large scale software engineering environments. The paper presents a history of CADES dating back to the early 1970's. CADES was established to provide a mechanism by which information relating to the structural model of a system could be made available to system designers early in the development process. Underlying CADES is a hierarchical database called Pearl. The establishment of CADES led to the development of a structural modeling methodology: Isolate functions, data, and constraints, produce data tree, produce function (holon) tree, consider next level of detail, code in Systems Description Language (SDL), and compile. Although CADES was developed to support the earlier phases of development, the author claims that CADES solutions are always sought for development or production problems and there is an increasing trend towards support for implementation.

(Solo84) Soloway, Elliot. "A Cognitively-Based Methodology for Designing Languages/Environments/Methodologies." In Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on Practical Software Development Environments. Gaithersburg, MD, April 1984.

Paper discusses issues relating to use of an environment. In particular, the author claims that environments should be developed based on a methodology that derives design implications based on tested hypotheses of why software developers work the way they do.

(Sten81) Stenning, Vic, et. al. "The Ada Environment: A Perspective." In Computer, Vol. 14, No,. 6, June 1981.

A paper that reviews the objectives (i.e., life-cycle support, open-ended environment, support for Ada, configuration control, project team support, portability, and host characteristics) and the architecture (i.e., the KAPSE/MAPSE/APSE approach, the database, KAPSE functions, the user interface, intertool interfaces, and tools) of the Ada Programming Support Environment (APSE).

(Stuc83) Stucki, Leon G. "What about CAD/CAM for software? The ARGUS concept." In Proceedings of SoftFair. IEEE Order No. 83CH1919-Ø. July 1983.

Paper proposes that software can be developed using a CAD/CAM approach with the aid of a software engineering environment. An overview of such an environment (ARGUS) is presennted. ARGUS is a micro-based environment that was built on top of UNIX. Argus is menu driven with a single key stroke approach. Six toolboxes are available at the top level; they are the management toolbox (scheduling tools, action item tracking tool, electronic spread sheet, and phone list update and retrieval system), the designer's toolbox (kernel CAD/CAM capabilities with a graphics/forms based approach), the programmer's toolbox (language-based, project-specific code template capability provided by a customizable editor and language specific syntax generation macros), the verifier's toolbox (analysis tools), and general/utility tools (general editing and communication tools). A noted CAD/CAM feature of ARGUS is the automatic projection of data to documentation and code templates from the underlying database.

(Stue84) Stuebing, H. G. "A Software Engineering Environment (SEE) for Weapon System Software." In IEEE Transactions on Software Engineering, Vol. SE-1Ø, No. 4, July 1984.

Paper presents a large scale environment called FASP that is hosted on multiple, large scale commercial computers. FASP primarily supports the latter stages of software development, but the extension to the requirements and design phase is discussed. The author attributes a two-fold increase in lines per month to FASP and an increase in software quality due to the tools, standards, and testing associated with the environment. The environment includes an underlying database made up of libraries: Source library, object library, test library, interface library, production data library, and documentation library. The system is command driven where the commands consist of lower level system commands (command procedures). Testing is supported by the ATA (Automated Testing Analyzer) and there is support for multilanguages and multitarget computers.

(Taj84) Tajima, Denji, and Matsubara, Tomoo. "Inside the Japanese Software Industry." Computer. March 1984, pp. 34-43.

(Tayl84) Taylor, Richard N. and Standish. Thomas A. "Steps to an Advanced Ada Programming Environment." In Proceedings of the 7th International Conference on Software Engineering. IEEE Order No. 84CH2Ø11-5, March 1984.

Paper presents a research environment for exploring concepts and issues related to software engineering environments in general and the Ada programming language in particular. The environment called Arcturus currently includes interactive Ada (a Pascal superset), template assisted editing, performance measurement (histograms or color), mixed compilation and interpretation, and an Ada program design language. Some concepts and issures being explored include complexity (does it scale up?), AVOS (Ada Virtual Operating System, i.e. an Ada command language), user interface issures (an Ada shell), mixing interpretation and compilation, layered architecture (i.e., device level, user interface level, tool level, foundation level), and analysis, testing, and debugging of tasking programs.

(Teit81) Teitelman, Warren, and Masinter, Larry. "The Interlisp Programming Environment." In Computer, Vol. 14, No. 4, April 1981.

Paper presents a look at the Interlisp environment. Interlisp is an environment for users of Lisp (a non-procedural list processing language). The environment is very much language dependent and is intended for use by Lisp experts. Some representative facilities in Interlisp include: File package, masterscope (help analyze the scope of a change), DWIM (do-what-I-mean spelling corrector), iterative expressions, and the programmer's assistant.

(Teit81a) Teitelbaum, T., and Reps, T. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment." In Communications of the ACM, Vol. 24, No. 9, September 1981.

Paper discusses an interactive programming environment with facilities to create, edit, execute, and debug programs written in a subset of PL/I. Editing is syntax-directed with underlying tree structures, predefined templates, and phrases to fill the templates. Execution of programs can be gear-shifted forward or backward with various controls on speed.

(Teit84) Teitelman, W., "A Tour Through Cedar." In Proceedings of the 7th International Conference on Software Engineering. IEEE Order No. 84CH2Ø11-5. March 1984.

Paper presents the facilities of the programming environment called Cedar. The Cedar environment emphasizes the use of parallel operation, multiple windows on a screen, and user interaction with a mouse pointing device. The environment supports the use of an "industrial strength" pascal-like programming language. The tour makes stops at the display (bitmapped and object-oriented with the use of icons), viewer window package (supports multiple levels of windows which are tiled on the screen), whiteboards (work windows), tioga editor and document preparation system (supports tree structured documents, editing with the mouse, syntax-directed templates), user executive (programming interface), interpreter (for debugging), automatic storage management (garbage collector), rope (string) interface, bug tracker, electronic mail, support for parallelism, and icon editor (pixel oriented, graph editor).

(Tomo84) Tomoharu, Mohri, et. al. "PDAS: An Assistant for Detailed Design and Implementation of Programs." In Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84CH2Ø11-5), March 1984.

Paper presents an environment that uses a forms-oriented approach to standardize document format and to prevent inconsistencies between documents and programs. There are 1Ø types of forms for design which are based on a forms-oriented language. The system structure consists of a forms-oriented editing subsystem, a document generation subsystem, a program construction subsystem (generation is based on module algorithm descriptions), a design database, and a component database (interchangable program components). An interesting aspect of the environment is automatic Japanese to English translation from algorithm descriptions.

(Wass81) Wasserman, Anthony I. Tutorial: Software Development Environments. IEEE Order No. EHØ187-5. 1981.

Tutorial is a reference collection of 39 papers including most of the landmark papers on software engineering environments.

(Wass83) __________. "The Unified Support Environment: Tool Support for the USE Software Engineering Methodology". In Proceedings of SoftFair, (IEEE Order No. 83CH1919-Ø), July 1983.

Paper presents an overview of the USE Software Engineering methodology and the tools in the environment that support it. The methodology involves users in the early stages of development and addresses user interactions with information systems. The tools in the USE environment include: The TROLL relational database (underlies and is used by other tools), RAPID (rapid prototyping tool oriented to the development of information systems), PLAIN (a procedural language oriented to the development of information systems), FOCUS (screen-oriented editor and browser, and IDE (a software management and control tool).

(Werl83) Werling, P. R. Alternative Models of Organizational Reality: The Case of Public Law 89-306 (the Brooks Act). D.P.A. Dissertation submitted to the University of Southern California School of Public Administration. 1983.

This dissertation addressed two major issues. In the first, the economic aspect of computing, the author demonstrates that computing cost/performance improved by more than 30 percent annually over the years 1958-1980. During these years, a new "generation" of computing technology was born every five years.

In the second, the author questions why the Federal government fell farther and farther behind the state of the art after enactment of the Brooks Act in 1966. By 1980 Federal computers averaged five years older than those in the private sector, and were much less productive economically. The difficulties were traced to fundamental discrepancies among three models of organizational reality that describe behavior of separate groups of public servants: a) the General Accounting Office and regulatory agencies exhibit expectations and behavior that correspond to the "classical management" model taught in business schools; b); Authors of legislation and implementing procedures function as though guided by the "adversary proceedings" model (taught in law schools); and (c) Those in operating agencies act in accordance with the "organizational process" model, taught in the school of hard knocks.

Tests of 7 substantive hypotheses showed that the "classical management model" is useful for the first estimate of results, while the "organizational process" and "adversary proceedings" models provide valuable insights for anticipating disfunctions in implementation.

(Wert82) Wertz, Harald. "The Design of an Integrated, Interactive, and Incremental Programming Environment." In Proceedings of the 6th International Conference on Software Engineering. IEEE Order No. 82CH1795-4. September 1982.

A paper that presents the details of a proposed environment that integrates editing, executing, and annotating programs.

(Wino79) Winograd, T. "Beyond Programming Languages." Communications of the ACM. 22:7 (July 1979), pp. 391-401.

Paper analyses the shortcomings of programming languages as they exist currently, and gives possible directions for future research. It points out that just as higher level languages enabled the programmer to escape the intricacies of machine order code, the environmental approach can provide a better means to understand and manipulate complex systems.

(Wirt81) Wirth, N. "Lilith: A Personal Computer for the Software Engineer." In Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

Paper discusses the development, features, and architecture of the Lilith programming environment for Modula-2. The system provides a high bandwidth between the user and the system partly through the use of a mouse pointing device and the hardware structure. The display is suitable for text, diagrams, or graphics.

(Wolf81) Wolfe, Martin I., et. al. "The Ada Language System." In Computer, Vol. 14, No. 6, June 1981.

Paper discusses the Ada Language System which is currently under development at SofTech. The system will provide capabilities at the MAPSE level (Buxt8Ø). Issues relating to the development of an Ada compiler are also discussed.

(Zelk84) Zelkowitz, M. V., et al. "Software Engineering Practices in the U. S. and Japan." In Computer, June 1984, pp. 57-65.

Paper gives the results of an in-depth survey of 3Ø companies and reveals the actual goings-on in software production. Results show that through practices are 1Ø years behind research, the tools available can narrow this gap.

BIBLIOGRAPHY — CROSS REFERENCE A

Categorization of References

Overview of software engineering environments:

(Bars84)
(Bran81)
(Bran81a)
(Buxt8Øa)
(FIPS99)
(Fisc84)
(Fran84)
(Genr83)
(Haus81)
(Howd82)
(Hunk81)
(Oste81)
(Wass81)

Issues in building software engineering environments:

(Barn82)
(Gutz81)
(Houg84)
(Huff81)
(Mage84)
(Pren81)
(Sava82)
(Shne8Ø)
(Solo84)

General software engineering environments:

(Boeh84)
(Hall8Ø)
(Kern81)
(Kuo83)
(Lebl84)
(Mats81)
(Metz83)
(Ridd83)
(Stuc83)
(Stue84)
(Tomo84)

Systems development environments:

(Alfo81)
(Ridd81)
(Rube83)
(Snow81)
(Wass83)

Programming environments:

(Baye81)
(Buxt8Ø)
(Camp84)
(Cowe83)
(Cox83)
(Deli84)
(Fair8Ø)
(Feil81)
(Fisc84)
(Guya84)
(Houg82)
(Love83)
(Oste82)
(Sten81)
(Tayl84)
(Teit81)
(Teit81a)
(Teit84)
(Wert82)
(Wirt81)
(Wolf81)

BIBLIOGRAPHY -- CROSS REFERENCE B

General References on
Software Engineering Environments

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

2. Hunke, H., Editor, Software Engineering Environments, North Holland, 1981.

3. Proceedings of the 5th International Conference on Software Engineering, (IEEE Order No. 81CH1627-9), March 1981.

4. Proceedings of the 6th International Conference on Software Engineering, (IEEE Order No. 82CH1795-4), September 1982.

5. Proceedings of the 7th International Conference on Software Engineering, (IEEE Order No. 84CH2011-5), March 1984.

6. Proceedings of the ACM Sigsoft/Sigplan Software Engineering Sympsium on Practical Software Development Environments, Gaithersburg, MD., April 1984.

7. Proceedings of SoftFair, (IEEE Order No. 83CH1919-0), July 1983.

8. Wasserman, Anthony I., Guest Editor, Special Issue on Programming Environments, Computer, Vol. 14, No. 4, April 1981.

9. Wasserman, Anthony I., Tutorial: Software Development Environments, (IEEE Order No. EH0187-5), 1981.