UNCLASSIFIED

AD NUMBER
AD919267
NEW LIMITATION CHANGE
TO Approved for public release, distribution unlimited
FROM Distribution authorized to U.S. Gov't. agencies only; Test and Evaluation; OCT 1973. Other requests shall be referred to Space and Missiles Systems Organization, Los Angeles, CA.
AUTHORITY
SAMSO ltr, 13 Jul 1979

THIS PAGE IS UNCLASSIFIED

THIS REPORT HAS BEEN DELIMITED
AND CLEARED FOR PUBLIC RELEASE
UNDER DOD DIRECTIVE 5200.20 AND
NO RESTRICTIONS ARE IMPOSED UPON
ITS USE AND DISCLOSURE.

DISTRIBUTION STATEMENT A

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED.

AD919267

UNITED STATES AIR FORCE

INFORMATION PROCESSING/DATA AUTOMATION IMPLICATIONS OF AIR FORCE COMMAND AND CONTROL REQUIREMENTS IN THE 1980s (CCIP-85) (U)

VOLUME IV
TECHNOLOGY TRENDS: SOFTWARE

OCTOBER 1973

D D C
RECEIVED
MAY 28 1974
D

SAMSO TR 72-122

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author)
Space and Missile Systems Organization
P.O. Box 92960, Worldway Postal Center
Los Angeles, CA 90009

2a. REPORT SECURITY CLASSIFICATION
Unclassified

2b. GROUP

3. REPORT TITLE
Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s (CCIP-85) (U), Volume IV, Technology Trends Software

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)

5. AUTHOR(S) (First name, middle initial, last name)

6. REPORT DATE
October 1973

7a. TOTAL NO. OF PAGES
70

7b. NO. OF REFS
44

8a. CONTRACT OR GRANT NO.

b. PROJECT NO.

c.

d.

9a. ORIGINATOR'S REPORT NUMBER(S)
SAMSO TR 72-122

9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

10. DISTRIBUTION STATEMENT "B" Distribution of this report is limited to U.S. Government Agencies only; Test and Evaluation, Oct 73. Other requests for this document must be referred to Hq SAMSO/XRS, PO Box 92960, Worldway Postal Center, Los Angeles, CA 90009.

11. SUPPLEMENTARY NOTES

12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

Command and control software will become more important in 1985 than it is today. It is crucial to determine whether future software technology, as projected from current trends, will be able to provide the techniques necessary to build the appropriate software of the future.

This volume provides a brief introduction to software technology and defines the kinds of software that will be required to build and operate 1985 C&C systems. Having established requirements the report then focuses on relevant software technology to forecast what it may be able to achieve by 1985. Both application and executive software are considered, with special emphasis on response time, adaptability to unforeseen situations, suitability, and ease of transfer from one machine to another. Of particular importance in C&C systems are methods for the design, production, and validation of software, and the management techniques necessary to administer large software-development projects. Current tools and practices are assessed.

Finally, the estimates of 1985 software technology capabilities are compared with projected 1985 requirements for C&C software. The concluding section of the report outlines studies, projects, and R&D investments that the Air Force might undertake to narrow the expected gap between requirements and technology and to alleviate future problems in implementing and operating command and control software.

DD FORM 1 NOV 65 1473

UNCLASSIFIED
Security Classification

UNCLASSIFIED

INFORMATION PROCESSING/DATA AUTOMATION IMPLICATIONS OF AIR FORCE COMMAND AND CONTROL REQUIREMENTS IN THE 1980s (CCIP-85)

NOTICE TO RECIPIENTS

The views presented in this report are those of the Study Group and do not necessarily reflect the policy or position of the Air Force or the participating Commands on any issue. The work of the Study Group has been reviewed for technical quality and adequacy by an Advisory Review Group of qualified operational and technical specialists.

D D C
RECEIVED
MAY 28 1974
D

UNCLASSIFIED

STUDY REPORT: LIST OF VOLUMES

Volume I	Highlights
Volume II	Command and Control Requirements: Overview Annex A: Strategic Requirements Annex B: Air Defense Requirements Annex C: Tactical Requirements
Volume III	Command and Control Requirements: Intelligence
Volume IV	Technology Trends: Software
Volume V	Technology Trends: Hardware
Volume VI	Technology Trends: Sensors
Volume VII	Technology Trends: Integrated Design
Volume VIII	Interservice Coordination Trends
Volume IX	Analysis
Volume X	Current Research and Development
Volume XI	Integrated Research and Development Roadmaps

PRECEDING PAGE BLANK-NOT FILMED

UNCLASSIFIED

TABLE OF CONTENTS

Page

I. INTRODUCTION 1

A. The Role of Software in C&C Systems 1
B. Objectives of this Report 3

II. SOFTWARE TECHNOLOGY 5

A. A Working Definition 5
B. Types of Software 6
C. Software Production 6
D. Command and Control Software 7

III. SOFTWARE REQUIREMENTS FOR COMMAND AND CONTROL SYSTEMS 9

A. Increasing Demands on Command and Control Systems 9
B. Functional Requirements 10
C. Qualitative Requirements 12
D. Economic Requirements 14

IV. TRENDS IN SOFTWARE TECHNOLOGY 17

A. History 17
B. Current State of the Art 22
C. Projection of Current Trends 26
D. Trends Affecting Command and Control Software . 32
E. Unique Software Requirements of Command and Control Systems 39
F. Recent Research 40

V. SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS . . 49

A. Requirements versus Capabilities 49
B. Measures to Narrow the Gap 49
C. Conclusions 64

UNCLASSIFIED

PRECEDING PAGE BLANK-NOT FILMED

LIST OF ILLUSTRATIONS

Figure		Page
IV-1	USAF Software Expenditures	2
IV-2	Hardware/Software Cost Trends	3
IV-3	Aggregation of Software Components	5
IV-4	The System Development Process (1970)	15
IV-5	Growth in Software Requirements	27
IV-6	Computing Speed (1955 - 1970)	29
IV-7	Computing Speed (1960 - 1985)	30
IV-8	The Cost of Computing Power (1960 - 1972)	31
IV-9	Programmer Productivity (1955 - 1985)	33
IV-10	Estimated Relation of Effort to Size of Software Projects	35
IV-11	Air Force Information-Processing Technology Staff: Recommended Manpower Plan	59
IV-12	The Software-First Machine: Possible Configuration	63
IV-13	The System Development Process, 1970 and 1985 (potential)	66

LIST OF TABLES

Table		
IV-I	Effort on Four Software Systems	7
IV-II	1985 Requirements versus Capability: Function	50
IV-III	1985 Requirements versus Capability: Productivity & Timeliness	51
IV-IV	1985 Requirements versus Capability: Reliability	52

Table		Page
IV-V	1985 Requirements versus Capability: Acceptability	53
IV-VI	1985 Requirements versus Capability: Adaptability	54
IV-VII	1985 Requirements versus Capability: Security	55
IV-VIII	Suggested Information-Processing Staff Functions	60
IV-IX	Summary of R&D Recommendations	65

GLOSSARY

ADP	Automated Data Processing
AED	Algol Extended for Design
AI	Artificial Intelligence
C&C	Command and Control
CPT	Chief Programmer Team
CPU	Central Processing Unit
FGSS	Flexible Guidance Software System
HOL	Higher-Order Language
MTBF	Mean Time Between Failure
R&D	Research and Development
RFP	Request for Proposal
THE	Technische Hogeschool Eindhoven

I. INTRODUCTION

Command and control systems in the 1980s will undoubtedly operate in a different environment, employ different hardware devices, and be required to respond in ways not possible with today's systems. The CCIP-85 study examined the functions to be performed by future C&C systems and assessed what the new functions will require of the computer hardware and software that must perform them. A preliminary comparison of those requirements with projections of future information processing capabilities suggests that future software design procedures, analysis techniques, and production methods might not adequately meet C&C requirements. The current study discusses that problem and outlines promising R&D strategies to decrease the gap between future C&C requirements and future software technology.

A. THE ROLE OF SOFTWARE IN C&C SYSTEMS

By nearly every measure, software is a vitally important element of today's ADP systems for command and control. The nature of software technology and its intellectual and production components are described in Section II. Briefly stated, "software" refers to the computer programs (and their associated descriptive documentation) that give purpose and direction to computer hardware, tailoring it to serve the information needs of a user and to support his decision-making processes. Software is the critical element in ADP systems because:

- It is most expensive. As shown in Figures IV-1 and IV-2, at least 70 percent of the current Air Force investment in ADP systems is spent on software. That figure should rise to over 90 percent by 1985.

- It has absolute control over ADP system response. Although hardware may influence system speed and display media, software determines the manner in which C&C requirements are satisfied. It directs the hardware to aggregate, organize, and transform data into usable information.* Contrary to the popular view, C&C hard-

*Information is distinct from data. "A datum is a fact in isolation. Information is an aggregate of facts so organized or a datum so utilized as to be knowledge or intelligence. Information is meaningful data, whereas data, as such, have no intrinsic meaning or significance."[1]

ware in fact supports software, which in turn supports mission functions.

- It provides essential adaptability. Regardless of possible inflexibilities in the design of a particular program, software (unlike hardware) is inherently easy to modify. That quality is critical in C&C systems, where functional and response requirements may change rapidly as the Air Force evolves toward the dynamic force-management operations of the 1980s. It is now infeasible to track continually shifting functions by frequent hardware alteration.

- It is on the "critical path" in system development. The availability of a new or updated C&C capability has -- historically -- depended directly on the completion of software. Hence, achieving a given capability within a given time requires the development of workable software well within that time.

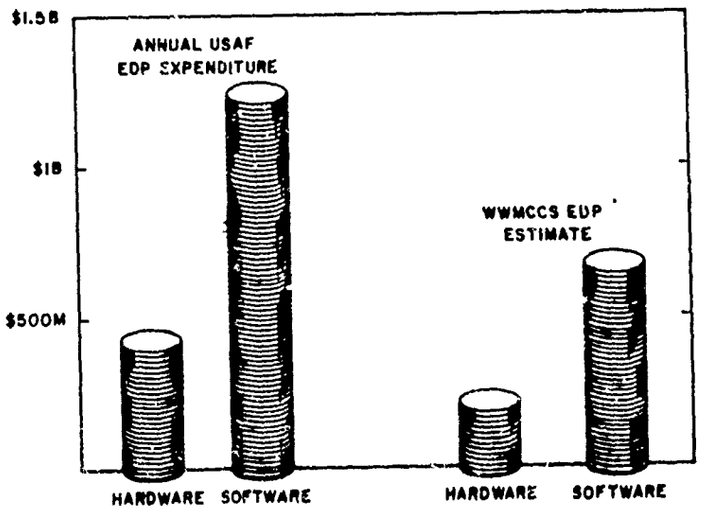

Figure IV-1. USAF Software Expenditures

The importance of C&C software is underscored by the severity of the penalties attending late or ineffective software. Often, the result is simply the temporary loss of capability. More dangerous is the accidental execution of an unintended command. On one recent C&C project, failure to complete programming delayed the completion of the entire system by six months. The delay not only added another $2 million to direct development costs, but it also prevented use of the system by the operating command for those six months. Since the system was designed for a seven-year operational life, at a total cost of $1.4 billion, the loss of six months represents an opportunity cost of $100 million.[2]

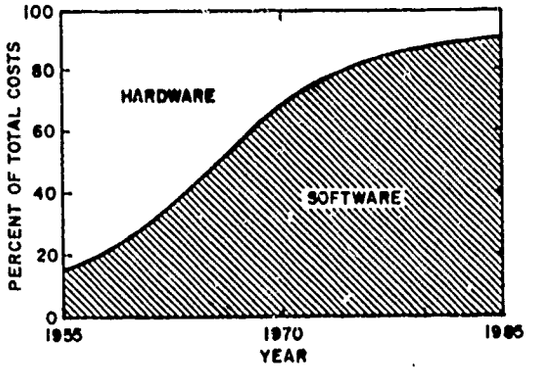

Figure IV-2. Hardware/Software Cost Trends

As an illustration of the dangers attending functional failure of software, consider this scenario:

> During a fairly tense period, the Soviet Union launches a new satellite. The United States decides to use a previously untried combination of sensors on an inspection satellite to take a look -- an option available under the computer program that controls sensor sequencing. But, under a certain unlikely combination of conditions, which had not been checked out, this software option also activates a high-intensity electron beam for warhead detection. This happens during the mission, and the electron beam kills a crew of six in the satellite.[3]

The above situation could quickly precipitate a dangerous strategic confrontation. Software errors, some perhaps as serious as the above hypothetical case, are continually being found in C&C systems today; in the SACCS software, for example, the rate is approximately one error per day.[2]

B. OBJECTIVES OF THIS REPORT

Command and control software will become more important in 1985 than it is today, as greater command and control precision and flexibility are demanded, more capacious hardware becomes available, and military budgets and manpower shrink. It is crucial to determine whether future software technology, as projected from current trends, will be able to provide the techniques necessary to build the appropriate software of the future. After a brief introduction to software technology (Section II), the kinds of software that will be required to build and operate 1985 C&C systems will be defined (Section III). The approach used here is a relatively conservative one, extrapolating future requirements from historical developments and technological trends. The findings of the three agencies contracted by the CCIP-85 study to deter-

mine specific information-processing requirements for strategic offense, strategic defense, and tactical warfare in 1985 are used to determine the qualitative, quantitative, and functional requirements of future C&C software.

Having established requirements, the report then focuses on relevant software technology to forecast what it may be able to achieve by 1985. Again, evolutionary development from the present, following historical trends, is assumed (Section IV). Both executive and application software are considered (see page 8), with special emphasis on response time, adaptability to unforeseen situations, suitability, and ease of transfer from one machine to another. Of particular importance in C&C systems are methods for the design, production, and validation of software, and the management techniques necessary to administer large software-development projects. Current tools and practices in those areas are assessed, to project the probable modes of software development in the 1980s.

Finally, in Section V, the estimates of 1985 software technology capabilities are compared with projected 1985 requirements for C&C software. The concluding section of the report outlines studies, projects, and R&D investments that the Air Force might undertake to substantially narrow the expected gap between requirements and technology and alleviate future problems in implementing and operating command and control software.

II. SOFTWARE TECHNOLOGY

Many aspects of computer software are not widely understood, and it is often difficult to discuss software without resorting to a specialized vocabulary. This section explains what software is, how it is produced, and, specifically, what distinguishes C&C software from other types.*

A. A WORKING DEFINITION

Computer software may be defined as the collection of elementary commands or instructions describing the machine operations necessary to perform a defined task. The commands or instructions may be linked to form successively more complex units, as shown in Figure IV-3. Instructions may be grouped, first, into subroutines or modules of a particular program, then into a computer program. One or several programs may comprise a software subsystem. Additionally, one or several programs or subsystems may be joined to form a software system.

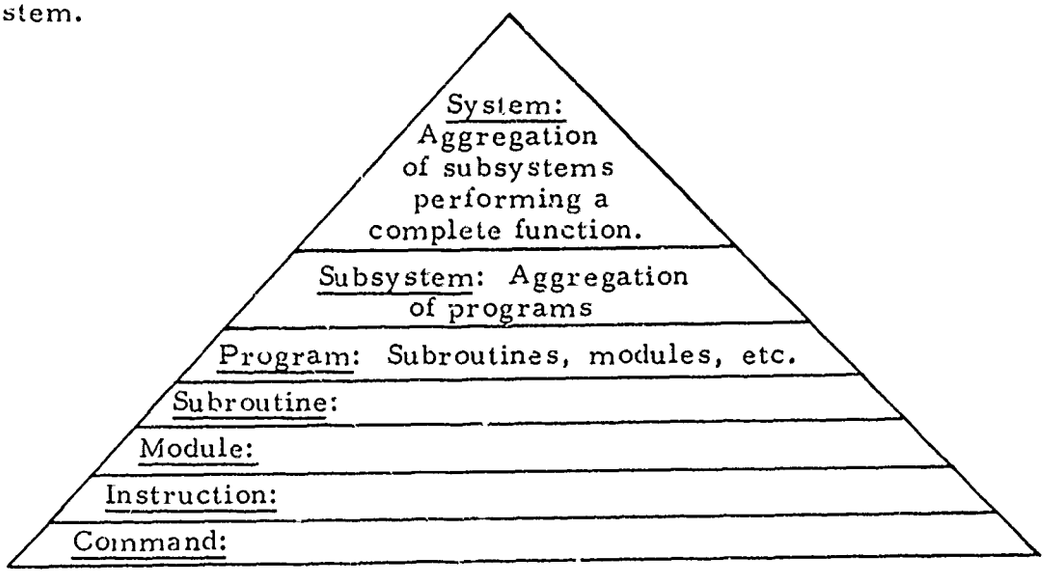

Figure IV-3. Aggregation of Software Components

*To help the nonspecialist understand the more technical discussions in later sections, he is referred to the "Prose Glossary," published as Annex A in the Highlights (Volume I) of this report.

Although not physically part of a computer program, documentation (the set of verbal and pictorial descriptions of software functions and underlying concepts) is considered part of software. Without it, software lacks the transferability and flexibility that distinguish it from hardware.

B. TYPES OF SOFTWARE

The term software may refer to any or all of the components shown in Figure IV-3 -- any series of instructions to a computer. In use, software is differentiated into three general types:

- Executive software performs resource-management functions such as input-output and scheduling of different (but concurrent) program executions;
- Utility software performs often-used general-purpose functions such as tape printing and translating man-written instructions into machine-readable commands; and
- Applications software performs special-purpose tasks requested by the computer user. The software in most C&C systems, specifically written to perform the tasks required by the commander, is of the applications type.

C. SOFTWARE PRODUCTION

Software production may be divided into five elements:

- System design -- A statement of the user's requirements is translated into a functional design specifying the tasks to be performed by each program in the total software system;
- Program design -- Each program in the system is designed in detail, with definition of the algorithms to be used and steps to be performed;
- Program coding -- The specific machine-executable instructions necessary to perform the task are prepared;
- Program checkout -- The coded program is run to verify, to a specified level of certainty, that it performs properly; and
- System and program documentation -- The prepared software is described in writing for the benefit of the users, future system designers, and those desiring to understand, alter, or further verify the program.

It is apparent that software production relies heavily on intellectual rather than physical effort. Most of the tasks involve translation of requirements into the processes and steps necessary to perform a function. The actual writing down of the program code is currently the smallest part of the total effort. The checkout phase involves the tedious process of finding errors in the program and is currently the most demanding and time-consuming. Table IV-I shows the percent of effort expended upon each phase in producing four large software systems. (Generally, the documentation phase adds about ten percent to the effort.)

TABLE IV-I

EFFORT ON FOUR SOFTWARE SYSTEMS[4]

System	Percent of Effort		
	Analysis and Design (1, 2)	Coding (3)	Checkout (4)
SAGE	39	14	47
NTDS	30	20	50
GEMINI	36	17	47
SATURN V	32	24	44

The figures in the table, which show that the majority of software production is conceptual development (design) and the laborious verification of program logic (checkout) provide some insight into the difficulties of gaining large increases in software productivity. These difficulties are discussed in more detail in Section IV.

D. COMMAND AND CONTROL SOFTWARE

Critical to this document is a question that has aroused some controversy: what's different about command and control software? If one views the programs themselves, there seems to be little difference. Command and control software performs the same general tasks (file management, data reduction, report generation, and the like) as other types of software. Moreover, C&C software is procured and produced in the same manner as software for most other applications.

The vital difference between C&C software and all other types lies not in the nature of the software, but in the nature of command and control, which drastically alters the cost-benefit environment in which the software must operate. Failures in C&C software -- in real life -- can be directly translated into lost defense objectives. Similarly, in hardware, there is little functional difference between a Spartan missile and an Agena launch vehicle for a weather satellite: each has propulsion,

guidance, and a payload. If the weather satellite is not successfully injected into orbit, the consequences are not serious: it can be tried again. But if a Spartan aborts its mission, then national defense may be seriously and irreparably compromised. Therefore, mission software must be better than conventional software, especially in reliability and flexibility, because the penalties of software error may be truly staggering.

III. SOFTWARE REQUIREMENTS FOR COMMAND AND CONTROL SYSTEMS

As stated in the Introduction, software has become the biggest item in ADP system cost and the critical factor in system performance and timely completion. This has not happened recently but perhaps has only recently been recognized. This section discusses past and future requirements for command and control ADP systems, with particular emphasis on the quality and quantity of the required software.

A. INCREASING DEMANDS ON COMMAND AND CONTROL SYSTEMS

The evolution of automated C&C systems has been governed by a single force: increased requirements (e.g., for processing speed, reliability, and functional capability) due to the increasing speed, sophistication, and power of the weapon systems available to and arrayed against the force commander. When it took several hours for manned bombers to penetrate, C&C systems could have response times of several hours. As missiles became the primary weapon systems, C&C response times had to be significantly shorter. In turn, as response time requirements shortened, other requirements became more critical. For example, C&C systems had to be more reliable: the 15 minutes of "downtime" allowable in the face of an air-breathing threat would be disastrous confronting ballistic missiles. Similarly, the consequences of a C&C system failure became more critical in another sense: the less time there is to detect and rectify an error, the more likely it is that a system error will be taken as a real threat. When events are played out in minutes instead of in hours, there is less time, for instance, to verify by other means (i.e., through a non-C&C system) whether a perceived threat is real.

More flexibility is now required of C&C systems. The existence of an extensive range of weapon systems, response options, and engagement scenarios requires C&C systems of corresponding sophistication and versatility. Also, because more data are available (through sensors, satellites, and OTH-B radar), C&C systems have to have a greater capacity for digesting data and reporting them in a form useful to decision-makers.

These new requirements have affected the shape and size of C&C systems considerably, primarily in the area of ADP support (both hardware and software). For the most part, hardware has been able to respond adequately; over the past decade, hardware speed and reliability have increased dramatically. Hardware flexibility has not eased soft-

ware requirements, however; nor has it reduced the complexity of the tasks software must perform.

Software has shown no dramatic progress. Therefore, it has become the least reliable and least secure component of the C&C ADP systtem, due in part to the increasing complexity of the functions it must perform and in part to the methods by which it is produced. The remainder of this section discusses the requirements that C&C software must meet in the future, as a prelude to surveying the potential of the software industry for fulfilling these requirements. Three classes of requirements are considered: functional, describing the sorts of tasks that C&C software must perform in 1985; qualitative, discussing acceptable standards in reliability and flexibility; and economic, providing a framework for discussing the impact of requirements upon software development cost.

B. FUNCTIONAL REQUIREMENTS

Certain ADP requirements for strategic, tactical, and air defense areas have emerged from the CCIP-85 study. This section summarizes the implicit and explicit requirements provided by those studies. The following list enumerates six general functions that C&C software must be able to perform in the 1980s. Most of them represent evolutionary continuations of tasks now performed by the more advanced C&C systems.

1. Data Management and Display

Fulfillment of this function requires maintenance of automated files of data on the status of changing conditions (such as our forces, enemy forces, weather, airspace assignments, etc.) that are necessary for decision-making. This function includes facilities for displaying selected information, means of selecting relevant information from the total, and all necessary underlying update and retrieval procedures.

2. Computational Assistance

Missiles, bombers, and fighters require guidance, targeting, tracking, and routing parameters to be calculated rapidly for maximum flexibility.

3. Optimization

Optimal use of available force relies on various mathematical algorithms and models to select weapons, set force levels, and assign weapons and targets in the most effective way possible under given constraints.

4. Real-Time Data Reduction/Data Entry

High-speed sensor-generated data and manually entered data must be "massaged" to remove noise, isolate important features, provide summary information for retention in the data base, and for command decisions.

5. Real-Time Command Decision Aids

Command and control decisions must often be made so rapidly that ways of arranging and evaluating situation information must be predetermined to take all relevant factors into account in a decision. This function includes maintenance of programs to assist in attack/response assessment, damage assessment, and response selection.

6. Real-Time Simulation and Exercise

All automated military systems must be tested and exercised in simulated combat situations for shakedown, training, and tuning; they must include programs for performance monitoring, reset, and recovery.

Such software will always be a specialized product tailored to fit the information needs and processes of the military user. It will never be available off the shelf. However, none of these functions is beyond today's software technology. Given sufficient development investment, programs can be produced that would perform these six kinds of jobs to some extent. The technology is lacking, however, in the production of "intelligent" C&C systems. Intelligent, in this context, refers to programs that perform three classes of operations: pattern recognition (visual or aural), induction (generalization), and complex decision-making -- tasks normally done only by the human mind.

A primary and rapidly growing requirement of current and future C&C systems is for rapid and intelligent reduction of multisource sensor data. Stated requirements include:

- Pure pattern recognition, as involved in identifying a truck or radome under a wide range of shadow patterns, and assessing bomb damage;[4]
- More powerful methods for the automated extraction of useful information from sensor data;[4]
- An image-processing capability: automatic target detection on the photographic image type of information; and
- Fusion of image and textual intelligence information.

Other functional requirements call for more intelligent (decision-making and resource allocation) capabilities for the reoptimization of SIOP options, reprogramming and retargeting of ICBMs and bombers, and guidance on when to withdraw from a force engagement.[5]

C. QUALITATIVE REQUIREMENTS

It is in quality that requirements for C&C software differ most from other applications. An ADP system controlling inventory levels, for example, can tolerate numerous errors in counting expended stock; a command and control system cannot tolerate any errors in counting incoming intercontinental ballistic missiles. A multi-access university time-sharing system can tolerate unauthorized access; a C&C system cannot. An obsolete finance and accounting system can be leisurely replaced through reprogramming for a new configuration; an obsolete C&C system degrades national security. Essentially, C&C software must be the most reliable, secure, transferable, and adaptable to changing needs. Let us consider the five important dimensions of software quality and the requirements in these areas for future command and control systems.

1. Reliability

Reliability, in this context, means the capability of the software to operate without failure, without a software-caused inaccuracy in data output. Software errors may be caused by a variety of design or coding inconsistencies; or they may be introduced in the requirements analysis phase through imprecise or contradictory specification of functional requirements.

It is difficult to specify a quantitative requirement for reliability (or for any qualitative aspect). The criteria used here are those that would remove software from the critical path of system development or would make software other than the weakest system component. By 1985, hardware systems are likely to achieve a mean-time between failures (MTBF) of 15,000 to 40,000 hours.[6] Software components of C&C systems should be required to reach a similar mean-time between failures. Since the MTBF currently associated with very complex software is eight to 24 hours, considerable effort must be made to advance the state of the art.

2. Acceptability

In this context, acceptability may be defined by the degree to which the implemented ADP system meets true user needs. The difficult process of analyzing user requirements usually has one of four outcomes, three of them bad:

- The user eventually gets a system that meets his true needs;
- The user eventually gets a system that does not meet his stated needs;
- The user eventually gets a system that meets his stated needs, but the stated needs are found to be inconsistent with his true needs; or
- The user gets a system that is satisfactory, but only after several interruptions during the development process and numerous expensive and time-consuming design and programming retrofits.

The final case is the most common; this and practical experience indicate that, in many cases, decision-makers are not able to clearly articulate their complex information needs. These needs are often best discovered by showing the user the information system output in an operational context and then letting him judge the utility of various items, reports, and displays.

Finding better methods of determining users' requirements would remove a big obstacle to the rapid and inexpensive simulation of systems. Feasible approaches to more precise and effective ways of defining user requirements will be discussed later. For this discussion, it is sufficient to state that finding a solution to the problem should receive very high priority.

3. Flexibility

One difficulty associated with software is the high cost of making even minor functional changes. Retrofit of software can cost four or five times more than initially producing particular routines.[3]

Command and control software is particularly liable to retrofit because it must reflect the constantly changing force structures and firepower of both friendly and hostile forces. Modifications will be needed at an even greater rate in the future, particularly as constant or declining defense spending brings about increasingly sophisticated weapon technology to meet changing needs. The requirement here is for software that is

- Easily and rapidly alterable;
- Alterable at a "reasonable" cost; and
- Rapidly and easily certified, once it has been altered.

To achieve it, either the very nature of software systems (currently complex and highly interdependent) will have to be changed, or the cost and time it takes to modify software will have to be reduced drastically.

4. Security

Preventing unauthorized access to, and subsequent compromise of, software is a complex problem involving questions of hardware, personnel, and facilities, as well as software. Software becomes the critical security factor only after physical access to the ADP system has been gained. In most recorded cases, even the most "secure" software can be compromised once someone has access to the system.

Security requirements cannot be explicitly stated here, because information regarding various levels of classification must be protected. Suffice it to say that each level of information has associated with it a different value, both to the protector and to the person who wishes to gain access to it. To be secure, software must have mechanisms able to insure that the cost of violation (to the violator) exceeds the value of the information gained. There are no absolutes in security -- only that relative cost-benefit relationship.

D. ECONOMIC REQUIREMENTS

All of the requirements described above (with the exception of those for intelligent systems) are feasible with sufficient economic investment. With severely constrained budgets, however, it is not currently possible even to approach those goals.

We conclude this section by looking at some attainable needs with respect to the cost of developing software systems and their timely completion.

1. System Development Cost

The current economics of computer programming are discussed at length in later sections of this report (see Section IV). In brief, it may be stated that the entire system-development process -- design, programming, and checkout -- remains essentially an artisan, labor-intensive process. Moreover, the "labor" is highly skilled, educated, and paid. These factors together keep system development and production an expensive and time-consuming process.

Program development time varies with the individual, type of task, working environment, language being used, and so forth. It is generally estimated in the range of one to 10 machine-executable instructions per day, with the order-of-magnitude range accounted for by variation in the parameters mentioned above.[7] More productivity is needed:

to have a 2-million-instruction system operational by 1987, 115 programmers would have to begin coding today. Total programmer cost (at $40,000 per year per man) would amount to $64,400,000. And as hardware speed increases and requirements become more complex, more instructions per system will be necessary.

2. Timeliness

There are two aspects to timeliness: first, that of the software itself (i.e., whether it is produced on schedule) and, second, that of the entire system (i.e., whether parts of software design and production slow the total development process regardless of the speed of software production). It may be possible to make systems more timely by software-related means that allow an earlier start to software production but do not affect its timely completion.

Software affects C&C ADP system timeliness both directly (as mentioned above) and indirectly. Software development is a serial process: the phases of requirements analysis, system design, program design, coding, checkout, system test, etc. follow each other. (See Figure IV-4, which also shows the various hardware-related decisions that attend the development and acquisition process.)

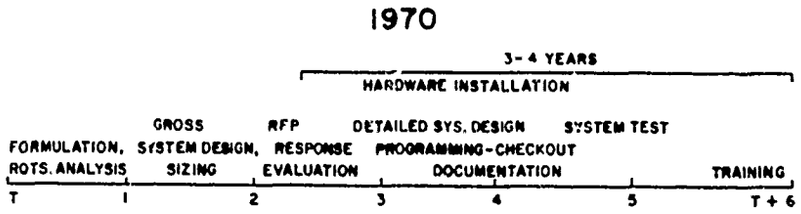

Figure IV-4. The System Development Process (1970)

What is needed is to reduce the time involved in this process, preferably by overlapping these steps. In the Conclusions (Section V), this problem is addressed. Specific R&D projects are suggested to solve it.

Looking at Figure IV-4, one implication of system timeliness is furnished by the issuance of the Request for Proposal (RFP) for hardware about a third of the way along the process. It should be pointed out that the six-year time span shown in Figure IV-4 is illustrative and typical but does not apply to the development of all command and control systems. The WWMCCS projects seven years for development, and SAGE required eleven years to complete. The phasing of the steps is the important aspect here. Essentially, the specification of hardware commits the system to a relatively narrow range of configurations. This means that altered requirements -- either new or newly perceived -- will require major, expensive design retrofits. In addition, early

specification of hardware makes the system vulnerable to early obsolescence with advances in hardware technology. Therefore, it is necessary to:

- Phase the system-development process so as to reduce total time-to-complete, and
- Find a way to delay hardware selection and acquisition until the latest possible time.

IV. TRENDS IN SOFTWARE TECHNOLOGY

With few exceptions, growth in software capabilities has been driven by user requirements on one side and machine characteristics on the other. For example, if the user needed a tracking program, he got a tracking program; if the user had drum storage, the program would use drum storage. Hence, forecasting the C&C functions that will require automation in 1985 and projecting the kind of hardware that will be available then largely determines the qualitative nature of future C&C software.

There are other characteristics of software, however, that are as critical to the user as the overall job it must do. Because there are so many ways to write a computer program to perform a given task, such parameters as how much the program costs, how quickly it was constructed, how efficiently it uses hardware resources, how reliable it is, and how suitable it is to the user have become essential considerations in determining software quality. Trends in these and other related characteristics must also be evaluated to derive an accurate picture of future software technology.

The approach here will be to outline the history of software development, describe the current state of the art, and then project the directions in which software technology is likely to grow. Limits and mitigating factors will be pointed out where necessary. Special sections will be devoted to exotic information processing, unique C&C requirements, and recent work in software program production and validation methodologies that may significantly alter current trends. The effects of software technology trends on managing development efforts and on the organizational framework in which development takes place will also be pointed out.

A. HISTORY

1. The First Decade

In the first generation of computing, software was extremely machine-constrained. Much effort was expended in fitting programs into the small machine memories and in devising algorithms efficient enough to produce the desired results in a reasonable time. Programs were full of clever methods to capitalize on machine quirks and wring out as much processing speed as possible. Although standard, prewritten library programs were considered,[8] they were not used except perhaps to do floating-point arithmetic.

Programs were also required to cope with the fairly unreliable equipment: the MTBF was often less than 20 minutes.[9] This forced programs to be very adept at using short bursts of computing power (e.g., by providing frequent storage of the contents of main memory on a magnetic drum for use in restart). Since computers were initially thought of as being very fast arithmetic machines, and the earliest programs did great quantities of scientific and engineering calculations, machines were designed with very limited input-output gear. Typically, this consisted of on-line, low-speed, punched-card equipment. Hence, few programs were written to process large quantities of data. The interaction of machine characteristics and the calculational workload tended to make programs small, limited in input-output, and highly machine-dependent.

At about the same time, the possibility of processing large volumes of business data and real-time radar data on electronic computers was beginning to be explored. Concurrently, hardware technology was allowing faster, more reliable machines to be built with larger memories and considerably improved input-output equipment. The demand for computer processing increased, and the character of software changed.

The most important differences were in the size and complexity of programs attempted, the emergence of higher-order languages (HOLs), and the appearance of supervisor systems.

Users had no trouble in conceiving programs that could utilize the increased speed and memory size. Logically complex programs, with more options and more program paths, require more space and are naturally difficult to segment and reduce below a certain size (related to the level of complexity). Various optimization and simulation programs appeared, not to mention more detailed engineering calculations. Real-time programs in particular (e.g., the SAGE prototypes) were great consumers of computing cycles and main memory.[10]

The demand for more programs to be written prompted the development of the first higher-level computer languages geared to the user. The increase in machine capability now allowed the writing of programs large enough to translate such languages into instructions acceptable to the machine. Though programs produced by these translators, called compilers, took more storage and took longer to run than hand-coded programs, the increased machine capability made storage and running time less costly than programming talent. Subroutine libraries were also being built up through the formal or informal interchange of programs among users. Common tasks (e.g., output of data to printers) often did not have to be reprogrammed from scratch for each new job.

The inefficiency of operating a computer by hand (program setup often took longer than program execution), the standardization of input-

output functions, and the advent of higher-order languages led to the development of supervisory "operating systems." These were collections of programs that loaded programs into memory from tapes, directed output to tapes, assembled user-written programs together with the necessary library routines, and automatically maintained tape files of intermediate data produced in multistep procedures.

Later, when the independently operating "data channel" device was introduced to permit input-output to be overlapped with computation, the operating system incorporated the very complicated input-output control and buffering programs necessary to take full advantage of the new hardware. Supervisory programs and compilers were among the largest programs; however, the costs (in terms of speed and space) of using higher-order languages were considered too great for these applications. Hence, they were coded almost entirely in machine language.

It is interesting to note that the military requirements for the SAGE system placed it beyond the leading edge of software technology in this period (the late 1950s). The number of programs required to perform identification, monitoring, tracking, weapon assignments, intercept direction, etc., plus the various support programs used for testing, recording operations, simulation and other purposes, made SAGE an exceptionally large system of coordinated programs (over one million instructions). The speed at which these functions had to be performed to keep up with radar data on incoming bombers in real time demanded automatic scheduling and supervision of the hardware, although this control was implicitly spread throughout the operational programs rather than being centralized in an "operating system." Emphasis on speed also required the development of extensive measurement and simulation software not previously seen in the field. SAGE became one of the first system to include immediate, interactive man-machine communication via display, light guns, and switches.

2. The Second Decade

As increasing generality was pursued in the early 1960s, software technology grew more elaborate. More powerful machines allowed, and often required, more complex support programs. Applications for programs and higher-order languages to construct programs proliferated. It became clear that user requirements for software of all kinds were growing at an exponential rate.

Several factors contributed to the use of more elaborate programs. Foremost was the degree of generality required by supervisor system programs. While it was sufficient for a typical application program to solve a specific problem with a specific calculation on a specific machine, system programs had to be designed to run on a range of machine configurations to support a wide variety of problem specifications.

Indeed, one of the primary justifications for system programs was their ability to increase the productivity of nearly all programmers, no matter what their task. The need for generality was also found in language translators, where all logical combinations of statements must produce proper machine code. And as more generality was required, programs grew to accommodate all the necessary tests and options. System programs were further elaborated to protect the system from user actions that could erase or otherwise abuse the system, to interpret various command languages, and to account for who should pay for the use of the system.[11]

Improvements in hardware also widened the scope of system programs. In particular, the independent data channel, the availability of substantial core memory, and the advent of interrupts to signal events (combined with the disparity in speed between the relatively slow input-output and relatively fast computing) encouraged the development of multiprogramming systems. In return for increases in work done per unit time and in hardware utilization, a multiprogramming operating system acquired the additional burdens of 1) keeping track of the progress of several programs simultaneously; 2) running system tasks to transfer information from low-speed to high-speed memory devices; and 3) maintaining queues and tables to schedule hardware and software system resources, as well as performing many other coordination functions. Further enlargements were necessitated by time-sharing of several active programs, multiprocessing, decoupling programs from slow readers and printers through "spooling," and the maintenance of a variety of system and user files.

In the most capable systems of this period (e.g., CTSS),[12] the supervisor alone could take up as much core memory as some similar machines had in toto. The supervisor generally exceeded the amount of core memory that could be devoted to it, and some programs had to be kept on the recently introduced disk-storage devices (from which they could be quickly loaded as needed).

Programs that actually solved problems (as distinct from system programs) developed in several ways. The most exotic "artificial intelligence" programs were also becoming more elaborate. Systems that answered questions, recognized patterns, or attempted to solve problems in more general ways were of great interest,[13] compared with the relatively simple game-playing programs of the 1950s. A much larger effort, however, was going into developing techniques to make it easier to use the computer for various applications. "Packages" of programs were developed for various applications to which one could give numbers and get back answers (e.g., the FORTRAN scientific subroutine package, SCERT/COMIT, etc.) Languages were specialized for solving problems in certain areas. For example, JOVIAL (for command and control), COBOL (for business), and SIMSCRIPT (for simulation), among others, were developed, and many specific appli-

cation programs were written. In 1963, IBM maintained a library of nearly 6500 of these kinds of programs, and SHARE had a collection of over 1800.[14]

The quickest solution to a fairly wide (although explicitly restricted) set of problems was perhaps furnished by the closed-end, algebraic, time-sharing languages such as JOSS and BASIC.[15] These systems provided (and still provide) a facile yet flexible means for engineering and scientific computing by strictly shielding the user from the machine he was using. With this development, the rest of software technology reached the level SAGE had attained some years earlier.

The third generation of computing hardware was introduced in the mid-1960s, and what had been research projects became commercial products. Special machinery was included to do things particularly difficult for software, such as memory protection and dynamic address translation to achieve "virtual storage." Thus, multiprogramming, time-sharing, and numerous languages became widesperad. The forefront of software technology moved on to attempt programs that were increasingly more complex.

Typical large efforts during this period were OS/360, TSS/360, the SABRE II airline-reservation system, and the Apollo real-time system. Some characteristics of these systems are as follows:

- Control and utilization of large collections of hardware. The Apollo center uses five IBM 360/75 computers, functionally bound together, plus hundreds of displays, consoles, and other input-output devices.

- Very large programming workforce, with multilevel management. The OS/360 took at least 5000 man-hours of effort and involved many levels of management, directing some 900 people.

- Greater integration and generality of functions. SABRE II was designed to handle not only real-time seat reservations but aircraft status and all American Airlines' business data-processing as well.

- Greater reliability requirements. In contrast to most previous systems, which could fail, be corrected, and be restarted without permanent damage, the cost of failure for the Apollo system was unacceptable.

- Hardware saturation. The gains in hardware speed and memory capacity were entirely consumed by the requirements for these projects; moreover, control programs

continued to be written primarily in assembly language to achieve maximum efficiency.

Besides the large-scale efforts described above, a great variety of development continued along traditional lines. As graphics hardware became less costly, software packages were written to give the user convenient modes of using it. New languages were developed for more and more applications, not the least important of which were for constructing other languages. The existence of direct-access storage devices stimulated many efforts to produce systems to give users easy access to data recorded on those devices. The availability of time-shared systems prompted all types of text editors and conversational programming systems.

B. CURRENT STATE OF THE ART

In discussing today's software, one must distinguish between typical computer usage and advanced computer systems. We will call the first the "mainstream" and the second the "leading edge" of software technology.

1. The Mainstream

One does not now have to be a highly trained programmer to use computing facilities. Engineers and military men can cover more problem-solving ground per day than programmers could in previous years. Computer utility has broadened through:

- The use of on-line, interactive, and sometimes graphic techniques to specify calculations and other manipulations of quantitative data (e.g., graphic ROCKET[16]);
- The use of data-management systems that allow retrieval of information from data bases by using fairly formal query procedures (e.g., MARK IV[17]); and
- The use of parameterized application packages and collections of routines that perform specific operations on the users' data (e.g., BMD[18]).

"Programs" written by such users, if any are written at all, are likely to be small and do not significantly change the power of the system being used. However these facilities can sometimes be tailored to a specific user's preferences, vocabulary, and working habits with a little extra programming.

The professional programmer is also more productive. He is still required to build the tools described above and to write programs when

no previously programmed system satisfies a user's requirements. His software tool kit includes:

- Multipurpose higher-order languages (e.g., PL/I) and special-purpose languages (e.g., SIMSCRIPT II) for different kinds of jobs;
- Libraries of subroutines for particular applications (e.g., IGS[19]);
- Tool-building tools, such as meta-compilers (systems that write compilers to order), cross-referencers (which construct "indexes" to program elements), and simulators; and
- On-line conversational text editors, file managers, and debuggers that help automate previously manual tasks.

Today's typical programs are one- or two-man products that operate in a non-interactive fashion and no longer require the effort, or receive the refinements, that similar programs once did. Relative freedom from machine and language constraints has brought out new limits on computer utilization, however. Some of these are a high frequency of errors, the likelihood of inflexibility, and the lack of program transferability.

Finding and eliminating errors is a significant part of all programming. Although it is usually fairly easy to code what one wants to have done, it is difficult to be perfectly correct or logically complete in writing a program. In a recent study,[20] debugging consumed far and away the largest amount of time in program development, usually 45 to 65 percent. Moreover, the ease in writing a program provided by higher-order languages has not generally been extended to debugging. Programmers must still be quite knowledgeable about the intricacies of the operating system and machine operation in order to isolate and correct errors.

Providing flexible software is another problem. The need for a particular software capability tends to change over time as the surrounding environment changes or design deficiencies are recognized by users. Though computer programs are inherently modifiable, programmers often find it difficult to alter tightly intertwined program logic quickly and to know precisely where to enter the modification in the original design. Also, the user may simply be averse to changing a piece of software that finally works after a long development effort.[21]

Despite the prevalence of higher-order language programs, software capabilities that exist at one installation cannot generally be transferred

to another installation because of compiler differences from machine to machine, operating system differences, lack of a particular language for a particular machine, or lack of sufficient documentation. Thus, an appreciable amount of development effort goes into redoing software, rather than into advancing software capability. Moreover, much programming is done in languages that are not well matched to the application because those languages are more universally compilable (e.g., FORTRAN). Again, software technology is not advanced through these efforts.

The above problems are serious and pervasive; on the whole, however, the average computer user is much better served than he was five, ten, or fifteen years ago.[7]

2. The Leading Edge

As in the past, the software projects that determine the upper limits on current software technology are, by definition, those that are the largest and most elaborate, that make the most severe demands on the power of the hardware, and that strain the ingenuity of the designers. In this class are most of the larger operating systems (e.g., TSO/360 and MULTICS) and the real-time control systems (e.g., Apollo mission control and the newer airline reservation systems.* Important characteristics of these large operational systems and their problems are discussed below.

a. Enormous Effort -- Time taken to develop these projects is measured in years, cost in millions of dollars, labor in hundreds of man-years, and documentation in tons. A labor-intensive artisan approach to software design, production, and testing is used, incorporating several layers of management. President Nixon's former science advisor, Dr. E.E. David, Jr., has commented:

> Among the many possible strategies for producing large software systems, only one has been widely used. It might be labeled "the human wave" approach, for typically hundreds of people become involved over a several year period. This technique is less than satisfactory.[7]

b. Overexpectation -- The imagination of systems designers, salesmen, and others seeking to push technology is limitless, but their time and cost quotations to the customer cannot be. As it is nearly impossible to write definitive specifications for large-scale systems before the design is begun, more powerful system functions may be

*Another kind of limit is set by the state of research in more esoteric uses of computers (e.g., artificial intelligence), which will be discussed later.

promised than can be delivered. Moreover, with such long associated lead time, perceived user requirements can shift, forcing redesign, retrofit, or cancellation of the project. Continues Dr. David:

> [The "human wave" approach] is expensive, slow, inefficient, and the product is often larger in size and slower in execution than need be. The experience with this technique has led some people to opine that any software system that cannot be completed by some four or five people within a year can never be completed; that is, reach a satisfactory steady state.[7]

c. Efficiency Requirements -- The desire to exploit the hardware to its maximum potential generally requires that at least a part of large-scale systems be written in machine-dependent assembly language. This need for operational performance also often requires discarding some software tools, such as higher-order languages or standard input-output packages, because the overhead resulting from their generality is too costly. Some projects misjudged the effects of this overhead (e.g., UNIVAC's reservation system for United Airlines) and were abandoned. As it was put by Hopkins, "If you have the prospect of 3000 inputs a minute and your system has always been just one step ahead of the sheriff, you are not going to tell people that a microsecond isn't important; too many times they have had to rewrite code to save a microsecond or two."[22]

d. Responsibility -- The cost of failure, even temporary failure, is fairly high. First, these systems are more tightly coupled to the environment. Second, they handle many tasks concurrently. For example, this scenario:

> Consider a time-sharing computer which operates perfectly for, say, 50 hours, then malfunctions for two seconds. Suppose that, during the two seconds, all communication between the on-line users (there may be several dozen) and the computer are lost; program pointers to data are confused; dictionaries of files are destroyed; and key portions of the operating software have been disabled. Each user is going to be very, very unhappy.[23]

If these users happen to be key defense commanders or air-traffic controllers, some people depending on the system may be more than unhappy.

e. Reliability -- Equipment is often duplicated to provide fail-safe operation in the event of hardware failure, but there is no simple means of validating software. It is impossible to say with certainty that there are no errors in a large software system, so the system

may fail in unpredictable situations. In critical applications, particularly in command and control, validation must nevertheless be done. Brute-force methods have required as much as $750,000 for an 8000-instruction program.[24] Typically, the cost of validation for a missile-guidance program is $4 per machine instruction. Since this cost is so high, errors are usually shaken out of larger software systems through actual use. For example, it is said that around 1000 errors are found in each new release of OS/360.[22]

f. Integration -- Current systems attempt to perform multiple functions with the same hardware and a common data base. One express purpose of MULTICS, for example, was to "provide multiple access to a growing and potentially vast structure of shared data and shared program procedures."[25] Thus, the construction, compilation, testing, editing, debugging, and storage of computer programs would all be available through compatible system software "at one's fingertips." Similarly, in business data processing, the potential for using data for many purposes (e.g., inventory data for marketing, forecasts, and product evaluation, as well as for inventory control) has been realized in some instances. That is also the objective of SAI (SO's Flexible Guidance Software System (FGSS) being developed for rocket-vehicle guidance and control.

C. PROJECTION OF CURRENT TRENDS

Forecasting software capabilities is somewhat different from predicting future hardware. While many appropriate measures exist for gauging hardware production and performance -- cost per bit, add time, transfer rate, byte capacity, etc. -- similarly comprehensive and meaningful metrics for software are lacking. Hence, it is hard to define what is forecast. Though copious quantitative data have been collected on hardware production and performance, the importance of such data for software is just beginning to be recognized. Thus, there is no firm data base from which extrapolations may be made. Most importantly, while the laws of physics are well known and can be used to predict and control progress in hardware development, the "laws of software" are not well known; there are few invariant relationships to bound projections of future software characteristics.

Nevertheless, it is most likely that some patterns can be abstracted from the history of software development so that the kinds of software technology that might be available in the 1980s can be suggested. As mentioned above, certain forces greatly influence the course of software technology and produce certain trends in software capabilities. The intention here is to use these to derive a first-order approximation of future software characteristics.

1. The Driving Conditions

Three factors seem uppermost in influencing the development of software technology. The computing requirements and economic resources of the user are most important in determining the functional characteristics (i.e., what tasks the program should do). These interact with the characteristics of the available hardware to determine more precisely how programs will operate (e.g., what degree of detail is feasible and what response time can be expected). Further affecting the technology are evolutionary forces -- those that result from the available stock of implementation tools, ideas, and experience with previous attempts to do a similar job. Let us examine these in more detail.

a. User Requirements -- Growth in software requirements has continued at an exponential rate. Few capabilities have been discarded and many, many have been added. Increases have occurred in the number of applications, operating systems, and languages, and in their size and complexity. As one index of this growth, Figure IV-5 presents the amount of code (machine instructions) provided as standard programming support (Type 1 programs in IBM terminology) for a variety of computers. This curve shows that the amount of software required doubles about every 1.4 years. Projecting this curve out to 1985 would indicate a need for about six billion instructions of programming support for a particular computer system.

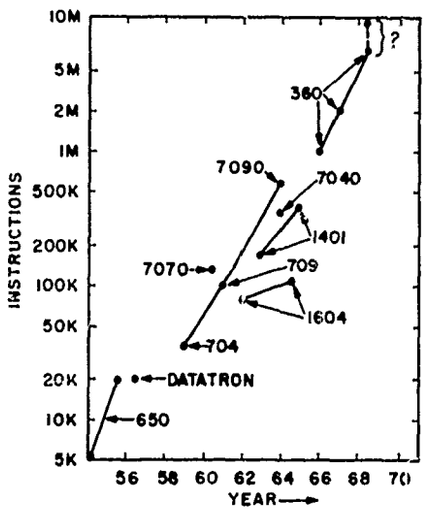

Figure IV-5. Growth in Software Requirements

Of course, this growth may slow down before 1985. The industrial market is already beginning to be saturated, the growth of the programming profession is slowing down, and in general the computer industry is beginning to mature. Because software is no longer supplied "free" with IBM equipment and an independent software industry has appeared, the above measure of software requirements, as defined, probably cannot be projected accurately.

However, the growing market forces surrounding software and the continuing search for generality may perpetuate the current growth rate. In the 1960s, there was only one operating system and set of languages, supplied by the machine's manufacturer; now, competition may induce various firms to produce several different systems and language implementations with different characteristics for the same machine. This is already visible in certain software components, such as data-access methods and compilers.

The search for generalized software had haunted software technology as the search for the philosopher's stone haunted alchemy. People believed that many of their troubles would disappear, if only there existed a completely generalized operating system, or a universal programming language, or a general management-information system. Much programming effort has gone into achieving these ideals -- OS/360, PL/I, GIS (Generalized Information System[21]) -- but it has turned out to be extraordinarily difficult to obtain the expected benefits. And generality is a primary factor operating to escalate user requirements.

There is no reason to believe that the imagination and ambition of software users, designers, and implementors will diminish, thereby limiting the variety and scope of program products. For example, the historical expansion in operating system functions suggested by the lists below will probably continue, perhaps to handle demands for increased integration, reliability, security, and performance. Thus, if "supporting software: is redefined to mean all that which is commercially available for a given class of machines, the projection may be borne out.

1955 Support Software[11]

Program loader
Floating point subroutines

1970 Support Software[26]

Job management:
- Scheduling
- Resource allocation
- Program loading
- Interrupt event monitoring
- Program termination processing
- Input-output scheduling
- Buffering control
- Device manipulation
- Remote terminal support
- System startup
- Recovery processing

Management utilities:
- Peripheral device support
- System simulation routines
- System measurement
- Display routines

Data management:
- File access control
- I-O blocking & label control
- Data file generation, maintenance, retrieval

Sorting and merging programs

Diagnostic error processing:
- Hardware error correction
- Program error notification
- Interface error control

Processing support:
- Timing service
- Testing/debugging service
- Logging and accounting
- System status monitoring

Operating system management:
- System generation
- System reconfiguration

Program maintenance:
- Libraries and directories
- Load module generation

Compiler interfaces:
- Executive routine support
- Library support
- System utilities

b. Hardware Characteristics -- The CCIP-85 study of hardware trends demonstrated that computing power has also been growing at an exponential rate. As an index of this, Figure IV-6 shows the increase in computing speed over the last 15 years. According to this curve, the speed of commercially available processors doubles about every two years. Note that this is not quite as rapid as the growth in our index of software requirements. The CCIP-85 study has projected this curve out to 1985, as shown in Figure IV-7. The decrease in the rate of growth shows the influence of approaching physical limits on processor circuitry speed.

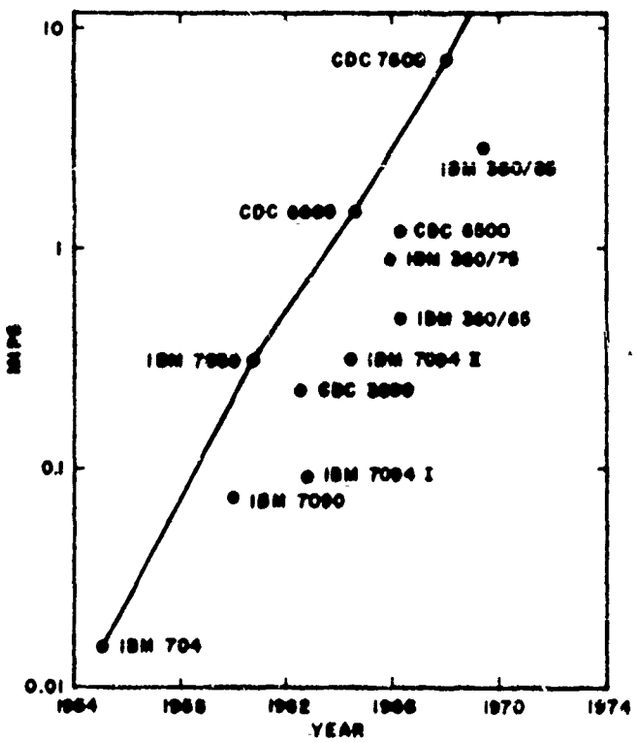

Figure IV-6. Computing Speed (1955-1970)

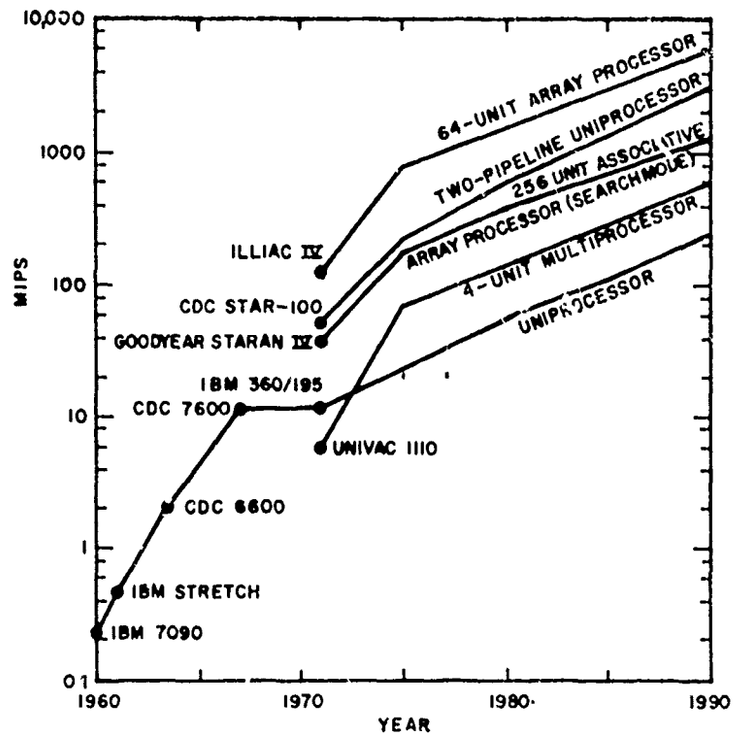

Figure IV-7. Computing Speed (1960 - 1985)

More important to software technology than the increase in power is the decrease in cost per instruction executed, as shown in Figure IV-8. Decreasing cost and increasing power have allowed more elaborate software aggregates to become feasible, and more to become cost-effective. The availability of larger core memories and random-access storage has also encouraged this trend. It appears that the cost of computing power will continue to shrink.

Although the cost of power has decreased, computer hardware is still expensive, and there is a continuing interest in using it more efficiently. Where processing bottlenecks have been identified, various forms of multiplexing and parallelism have been used to overcome them (e.g., overlapping data retrieval from various portions of memory or simultaneous central processing and input-output by means of the data channel). This desire for system efficiency also greatly influenced software design in the past: more efficient utilization means more real work done per dollar spent on hardware. Hence, the concept of program paging and overlaying was developed to conserve main memory; input-output buffering was introduced to ease the disparity between input-output rate and processing rate; and multiprogramming evolved to utilize the central processing unit (CPU) more efficiently. The value of system efficiency is still prized, even though cost per instruction executed has decreased by about three orders of magnitude since 1955. It appears

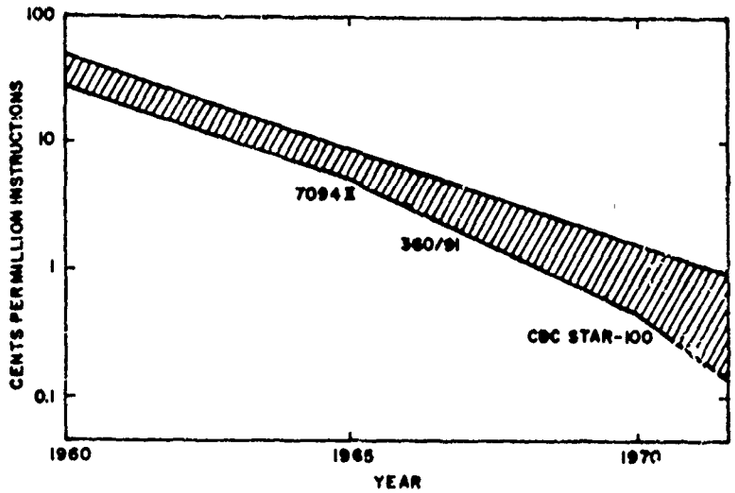

Figure IV-8. The Cost of Computing Power (1960 - 1972)

unlikely that efficiency will be less sought after in the future, particularly in C&C systems accommodating higher data rates, more sophisticated options, and reduced defense budgets.

2. Evolutionary Forces

Ordinarily, new software is heavily influenced by previous software and by the stock of previously developed tools available to build and operate it. Previous software has such a great effect because progress has typically been made in software by the cut-and-try method. Next-generation designs attempt to introduce features in reaction to problems arising from use of the old software. The lack of appropriate and well understood metrics has frustrated attempts to evaluate designs by other means. Hence, software nearly always goes through various versions -- FORTRAN I-IV, JOVIAL 1-6, OS/360 releases 1-20 (so far) -- and classes of software often develop in reaction to previous versions of that class, as MULTICS is a reaction to CTSS. Such reactions have occurred approximately every four or five years (e.g., the time between FORTRAN I and FORTRAN IV), which roughtly corresponds to the time between generations of hardware.

The software development environment also has great effect. For example, the existence of FORTRAN governed the way many other programs were written (static storage allocation, no recursion, no string handling, etc.), and the existence of time-sharing operating systems spawned text editors and conversational compilers. This effect is strongest when low development cost is considered important and weakest when user requirements make more stringent demands on the available hardware.

This problem-driven evolution of software thus means that predicting software capability more than one evolutionary step ahead is extremely hazardous. Yesterday's problems are solved today, but tomorrow's problems cannot be reliably anticipated.

D. TRENDS AFFECTING COMMAND AND CONTROL SOFTWARE

A number of specific trends have resulted from the general conditions just discussed. Assuming that the driving conditions continue, the trends can be projected into the future and their effects on command and control software in the 1980s can be examined.

1. Demands on Hardware Capacity

The largest and most powerful software systems have always severely strained the largest and most powerful hardware available; future large systems will make similarly severe demands on future hardware. The insatiable growth in user requirements and projected hardware capabilities will continue to encourage complex designs employing very large programs to manipulate larger and larger collections of equipment and data. If the requirements for military C&C systems continue to be the most extensive ones at any given time (as they always have been), C&C software projects will face more difficult design problems and more constraints on efficiency and response time than other software efforts. Neither more capacious future hardware nor evolutionary forces seem likely to reverse this trend.

2. Use of Higher-Order Languages

Stringent performance requirements for large systems have necessitated the use of assembly language to achieve maximally efficient equipment utilization. Traditionally, higher-order languages have not allowed a very close coupling between the user and the real machine instructions (perhaps because of transferability concerns). The inability to exploit low-level machine features and the overhead typically imposed by generalized input-output, setup, or storage-management routines have usually ruled out HOLs as system-building tools. In recent years, however, there have been attempts to combine the advantages of both kinds of programming by using a layered design strategy or by using specially developed system-programming languages. Layering involves using an HOL for most of the code, with performance-critical sections written in assembly language. The wider availability of techniques for measuring program performance has aided this approach. System-programming languages provide access to machine code through machine-dependent statement types and explicit provisions for intermingling assembly code with higher-level statements. The MULTICS implementation, for example, has used both techniques.

In 1969, Corbato estimated that the performance of compiled object coding was two times worse than that of hand coding.[28] Until these capabilities improve, C&C software may not be able to afford much use of HOLs or other generalized programming tools.

3. Increasing Programmer Productivity

Programmer productivity will not continue to increase through the use of software tools if only assembly language is used. Higher-order languages, operating system facilities, and subroutine libraries have allowed speedier production of programs to perform a given task. Assembly-language programmers as a group have not been able to achieve this. To illustrate, Figure IV-9 shows the estimated average number of checked-out machine instructions produced per man-month in 1955 and 1970, with projections for 1985. The numbers are related to the tools prevalent at the time. (The tools shown for 1985 will be discussed later.) The width of this estimate reflects the variance found in an SDC study of 169 USAF programs.[29] Also shown are estimates provided by Aron,[22] summarizing historical data from IBM projects involving system programs (1410, 7040, 7030, OS/360) and applications systems (Mercury, SABRE, FAA) done primarily in assembly language. A fairly large variance was found in these data as well.

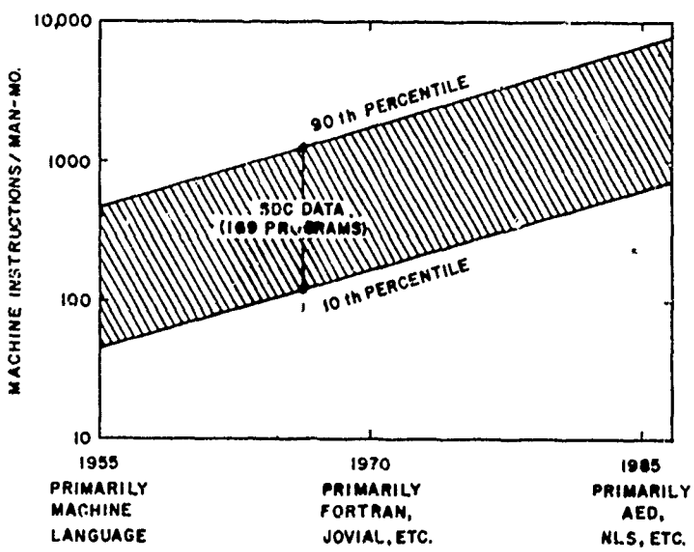

Figure IV-9. Programmer Productivity (1955 - 1985)

Such productivity calculations are derived by dividing the total number of instructions in a completed system by the total number of man-months taken to design, code, and test the system. Productivity can

therefore be increased by reducing the time spent in any of these three phases of software development, as long as the time saved does not reappear in another phase. Software tools have affected all three phases. Design and coding have become easier since the tools provide standard solutions to many subproblems and a means of communication with the machine better oriented to the problem being solved. Testing and checkout are somewhat easier because the tools have provided some chunks of pre-debugged code for inclusion in the final system. Many tools have been developed for enterprises with many users (e.g., business data processing). To the extent that C&C software overlaps with these activities, such tools would allow future C&C systems to be developed more quickly or with less effort (even though they might degrade efficiency).

4. Shifting Evaluation Criteria

Software evaluation criteria will continue to shift as computing power becomes cheaper, programming skill more expensive, and user requirements more critical. Software attempts to satisfy many needs. Run time and storage space may be traded off against user convenience, flexibility against efficiency, generality against ease of learning, and so on. Changing hardware characteristics alter the choice of software characteristics of greatest importance to the user at a given time. For example, in compiler evaluations reported by Haverty in 1962, compilation speed and object code running time received all the attention.[30] Rubey's compiler evaluation in 1968 showed that the speed with which a program could be written and debugged, program production costs, and the frequency of errors made with a particular language nearly submerged the data on execution time.[20] Changing requirements also alter evaluation criteria. As systems have been vested with more responsibility, for example, software reliability has become more important. But reliability usually demands freezing the development, which halts attempts to obtain greater efficiency. Projecting this trend into the 1980s, one can expect hardware-dependent software characteristics to become less important than user-dependent criteria.

5. Development Effort Disproportional to System Size

The effort required to complete a software project seems to increase nonlinearly with the size of the project. Although there are few quantitative data to support it, practitioners (e.g., Schwartz in Reference 22) feel that this curve resembles that in Figure IV-10. There are many reasons for such nonlinearity, but probably chief among them are simply the consequences of scale in the complexity of the task and the number of man-man and man-machine interfaces required. Although increasing numbers of projects have used traditional tiers of managers to coordinate the many people involved and to handle communica-

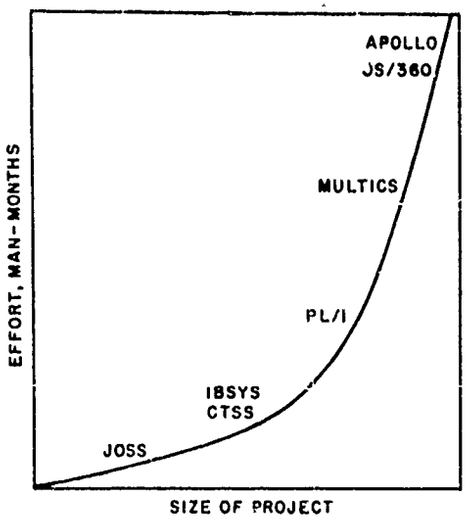

Figure IV-10. Estimated Relation of Effort to Size of Software Projects

tion among different groups working on different pieces of the project, there is some controversy over how well such scheduling, reporting, and control methods work. Another cause of disproportionate effort may be that, as software evolves, the harder problems tend to exceed our increasing ability to solve them at a geometric rate: what appears to be twice as much capability actually requires four times the effort to provide. It is also thought that software tools that make small problems easier may exacerbate larger ones. Checkout and reliability problems, which also increase with software size and complexity, are another factor increasing the required effort. To automate more functions, C&C software systems of the 1980s must be larger than today's, and the nonlinear increase in required effort may be the key barrier to their timely implementation.

6. Growth in Special-Purpose Software

The search for generality will continue in software research, but special-purpose software will become more cost-effective for many applications. Up to now, evolutionary forces have combined with the desire for more integrated computer "utilities" to push toward general or multipurpose languages, operating systems, and application programs. However, the increased performance of special-purpose software, its greater suitability to a particular problem, and its relative availability in time-sharing system libraries will expand its use in the future. These packages will proliferate even further because of market forces, the ease with which special-purpose software is built, and the increased use of higher-order languages as computing power becomes cheaper. Compatibility and maintainability considerations, however, will increasingly inhibit radical software development. It will be easier and cheaper to learn how to use a new software product than to build one. Unless specification methods become considerably more sophisticated, it appears unlikely that an organization will ever be able to use on-line application software without adjustment, tuning, and tailoring. High overhead in these items may be tolerated since they will be less costly than other alternatives. Although this trend will benefit C&C support software, it will be a challenge to use commercial packages in the highly demanding C&C operational environment.

7. Need for Certification

Debugging, testing, and validation practices have remained virtually unchanged since the beginning of computer programming, but interest in changing them is increasing. Except where the cost of failure has been very high (e.g., spaceborne software), not much effort has gone into streamlining those tasks, even though they are known to consume about 50 percent of the production effort. Few languages have been designed to include any but the most rudimentary debugging aids; few special tools (e.g., input generators, output analyzers, checkers for source code consistency) are available; and personnel try to avoid thinking about testing because it is so tedious. Most of the research on this topic has attempted to recapture the on-line debugging capability of the first generation of computing. The trend toward software systems with greater responsibility in the physical world may force greater emphasis on certification, however. Considerable work will be necessary to achieve even modest gains because of the intellectual difficulty of the problems and the scarcity of basic data defining their boundaries. Current techniques for verifying the validity of software are based on exhaustive testing, but these methods can never guarantee a perfectly correct product. It will probably continue to be far beyond the state of the art to guarantee that software systems of the kind envisaged for 1985 command and control will be completely correct.

8. Changing Hardware

Changing hardware characteristics have had far-reaching effects on software technology, and this will continue. The desire to use new or special-purpose hardware efficiently probably does not vary with the speed or capacity of the equipment. The CCIP-85 hardware study[27] forecast that some different types of equipment will be available in the 1980s. Some of these, and their software implications, are listed below.

- Pipelining, multiprocessing, array processors, and other forms of parallelism will require different kinds of languages for the user to communicate the parallelism of his problem to the machine. With only a little guidance, compilers may themselves be able to discover and exploit parallelism in a given program. Attempts to couple this high processing speed to fairly non-parallel real-time applications through clever scheduling are also expected. Debugging and optimization problems will take on new dimensions, even with careful design.

- Associative processing or other special processing devices will require tight scheduling since they probably will be an expensive resource. The ability to trade off software for special hardware should be welcomed

although users' desires for greater flexibility may inhibit the widespread adoption of these devices.

- Self-repairing computers and miniprocessor networks will enormously intensify coordination and testing problems but they may be worth it for relaibility. The extent to which software can recover and reconfigure itself during and after hardware failure is a current research question.

- User-microprogramming and hardware-implemented operating systems seem able to greatly boost effective CPU speed for certain well defined applications. However, the introduction of a whole new level of possible errors, transferability and maintenance problems, and compatibility considerations may prove too burdensome for the average user. Widespread use of microprogramming would stimulate all sorts of micro-prefixed software: micro-assemblers, micro-executives, micropaging software, etc. Whether the flexibility inherent in these techniques can be put to good use is another question for research.

9. Artificial Intelligence

One branch of software technology research has remained fairly independent of current functional requirements and hardware capabilities. Investigators of the kind of information processing that has come to be known as artificial intelligence (AI) have pursued much larger problems: the simulation or duplication of human perception, planning, problem-solving, and general intelligence and understanding.

Success in this area could make it possible not only to augment the human operator but also to replace him in some cases. In earlier years, some even argued that

> If artificial intelligence can imitate the intelligence of a man, then with more effort it can outwit and surposs that intelligence; the ability to consider large numbers of different factors cogently, rapidly, accurately, and adaptively implies that management and control systems of the future will be run virtually by computer.[31]

It is precisely this kind of problem, however, with which computer programs have been least successful. According to Feigenbaum, the second decade of AI research has shown that it is extraordinarily difficult to achieve the program generality and representational flexibility required to economically "consider larger numbers of different

factors" in solving a problem.[32] Because heuristic search* is still the basic strategy in AI projects, working in a very rich problem space has been too costly. Moreover, generalized programs can always be overpowered by more specialized ones that can take advantage of special representations. For example, character-recognition software operates well for certain stereotyped printing; but it cannot read correctly as wide a range of character styles as humans can.

There has been success, on the other hand, with very task-specific programs. Very good checker players and fairly good chess players have been built. An elaborate program to infer the composition of organic molecules from mass-spectrograph data has demonstrated an ability equal to that of a skilled chemist in some domains. Natural-language processing has advanced to the point where it can "understand" and solve most story problems typed in from a sixth-grade arithmetic text. Recognition of geometric solids and block constructions from TV images has been accomplished.

Current research is attempting to push beyond the current limits on generality. Computer-controlled mechanical robots are providing a test bed for integrating various task-specific programs into a coordinated whole. It turns out that using the richness of the real world is easier than simulating it for programs designed to operate in real-world situations. Work has been particularly vigorous on visual (TV) scene analysis with application to solving such problems as traversing an obstacle-filled room.

Research on deductive reasoning by means of the logic manipulations used to prove theorems is also very active. This appears to be a powerful representation capable of use as an action-planning mechanism in question-answering systems and in robots. (An application in program validation is discussed later.) Machine learning and self-organization for problem-solving has not been successfully grasped. Investigations into how humans perform these tasks, how they make decisions, and how they incorporate values into these decisions is also not as active as it once was.

Evolutionary forces strongly affect AI research since the stock of ideas remains painfully hard to enlarge. Success with systems of limited generality suggests that certain 1985 Air Force command and control needs could be met if they were sufficiently limited, sufficiently well defined, and sufficiently funded. For example, it may be possible to update a data base by scanning a text employing a few hundred words of vocabulary, simple sentence structure, and a restricted frame of

*A technique based on experimenting with many trial solutions in an organized way.

reference. It seems very unlikely that, even with very great computing power, a general language capability or general image-interpreting capability will be available. Greater understanding of natural intelligence and its relation to artificial intelligence may also be necessary to further extend the applicability of AI research to the command and control arena.

A larger question is whether it would be more effective to replace or merely augment the human in the command post by a computer, particularly as regards the manual backup for preserving system viability. In any case, the C&C system and its software must be organized so that key decisions remain in the hands of human commanders.

E. UNIQUE SOFTWARE REQUIREMENTS OF COMMAND AND CONTROL SYSTEMS

General trends in how software technology might affect C&C software have been discussed up to this point. In the next few paragraphs, the issue is examined from the other side: how will C&C software differ from other kinds of software and, hence, make unique demands on technology?

- The uncertainties in the military environment require extremely adaptable programs. A change in the strategic situation may require reorientation of ADP support in a matter of hours, making reprogramming and retesting infeasible. Generality has been the principal means of achieving adaptability, but unforeseen situations and performance (response time) limitations make this an only marginally effective solution. Certain software techniques based on easily reconfigurable tables or other data structures (or a series of "plug-compatible" software modules) might provide better means. As there are less consequential uncertainties in other environments, commercial organizations will not have the impetus to make extensive commitments to these methods.

- Decision times in C&C systems are vey short compared with other information systems. It is thus critical 1) to shorten decision time; 2) to devise decision aids, menus of choices, interactive strategic simulations, and different types of adjustable displays; and 3) to tighten the interface with human decision-making. Commercial systems typically do not face such time constraints; hence, commercial software will not move to incorporate these functions or to investigate human decision-making under conditions of military stress.

- Commercial systems will probably always be able to tolerate a level of reliability that is an order of magnitude below that which a C&C system could tolerate. Incorrect programs are not as dangerous in commercial environments. Hence, expensive certification efforts will not be as important to non-defense industry as they will be to the military.

- Security and system integrity are also much more vital to a military system than to a commercial one. A competitor hardly presents the same caliber of threat to operations as an enemy. The hostile environment imposed on C&C systems will require different kinds of software for verification of access, detection of sabotage, and so on. Note that security is not the same as data privacy, which is receiving some attention in research.

- The difficulty of envisioning all the consequences of logically complex design decisions has necessitated a long test period for commercial systems under actual operational conditions. Command and control systems cannot be tested under the actual conditions for which they are designed until it is too late. Therefore, simulation and other exercise methods, which are not likely to be developed outside the military, will be necessary.

F. RECENT RESEARCH

Most current research extends software in an evolutionary way, without much changing its direction (e.g., Balzer[33]). There are a few laboratory developments that may alter this assessment. Only a few experimenters have had experience with them, however, and evaluation data are lacking. In spite of the rapidity with which software technology changes, it is not clear how fast research results can be made available for Air Force production. For example, the first attempts at "automatic programming" (higher-order language translation) occurred around 1952, but the first commercial compilers did not appear until around 1957 and were not widespread until 1959 or 1960.

1. Software Production Methods

Two problems associated with recent large-scale development efforts -- producing the software on time and assuring its high quality -- have led to investigation of three new approaches to program design and production. These are the building-block approach, the on-line approach, and the structured programming approach.

a. Building-Block Approach -- Essentially, this is an extension of the development of higher-order languages, but a large enough extension that "higher-order language" does not accurately describe it. Basic to this method are a number of software components, or building blocks, which can be assembled in many different ways by means of a more elemental language. The existence of libraries of these components would relieve a software project of starting from scratch with each new system when similar systems have been built and similar software already constructed. Also essential to this technique are:

- Careful attention to interfaces and module definition;
- A cohesive, coordinated selection of software components;
- A very flexible programming language to "glue" the components together;
- System-generation tools for tailoring the components and building special components; and
- A software-production discipline to guide the system builder in his use of the tools.

If operational system efficiency is also required, the performance of the software produced is measured, and the efficiency of critical modules (i.e., those most often used) is increased by stripping out excess generalities.

The foremost example of this approach is the AED (Algol Extended for Design) system.[34] The components are generally suited to developing new kinds of languages rather than, say, processing real-time sensor data or constructing new operating systems. The AED system has been used for a variety of applications including a system for formal algebraic manipulation, language translators, information retrieval systems, econometric modeling, ship design, and others. The prewritten and pretested building blocks do seem to speed production and reduce the need for debugging and testing to achieve correctness. A job estimated to take six man-months was accomplished with AED in about two man-weeks.

Work on AED began in 1959 and an operational system has been around for a number of years; but it does not play a major role in the computing community either in theoretical contribution or in practical utility.[15] This seems to be caused by a lack of necessary documentation; difficulty in using the manuals that exist; lack of training courses for potential users; errors still in the AED systems themselves; and

the crude error-detection-and-notification schemes employed when processing programs are written by the user. It is not known to what extent AED or an AED-like system could cope with the more severe C&C software requirements, particularly the efficiency constraints. At least a partial test is underway, as AED is being used in the compiler and operating system for the B-1 avionics software.

b. On-Line Approach -- In large projects, keeping track of all the specifications, flow charts, design changes, development progress, test cases, and other interfaces is a mammoth job. Moreover, integrating all the assemblers, compilers, cross-reference listings, data files, debuggers, text editors, simulators, and other currently used tools into a commonly accessed, terminal-oriented data-management system cuts down tremendously on the relatively nonproductive bookkeeping and card-shuffling common in large projects. Various systems are being used (e.g., CLEAR-CASTER at IBM) or are under development (e.g., NLS at Stanford Research Institute[35]) to perform these functions. Reducing various delays in the current process of producing programs cannot help but speed development. Although the reduction of clerical errors by this approach may improve system reliability, using the same programming techniques on-line is not likely to dramatically improve the quality of C&C software.

c. Structured Programming Approach -- The most far-reaching, and therefore least predictable, approach has been laid out by Dijkstra. He proposes to change the design and production methodology and the way programmers write programs.[36] This would involve, for example, eliminating unconditional branching in programs and following both a top-down design scheme and a top-down implementation procedure. The block programming structure that results would reduce the probability of undiscovered error by clarifying the logical reasons for its execution at the beginning of a block. This would also reduce program production time by forcing out logical inconsistencies in the design phase, thus shortening the testing phase. It is claimed that individual programming efficiency can be increased by about five times.[22] Unfortunately, data to substantiate these claims are not available and are probably unattainable except through controlled experiments, which have not been done.

IBM has tried mixing the foregoing approaches with a new management technique called the "Chief Programmer Team." This and other aspects of structured programming will be discussed further in Section V.

2. Software Validation Techniques

Errors and oversights in the design and implementation of computer programs have always been very common. Currently, a reasonably correct program that does what it is supposed to do is developed by an iterative program-and-test process known as "debugging." Confidence

that a program always performs as it should, and does not perform as it should not, is currently supplied by a period of exhaustive testing. The term "validation" has come to mean the process by which one gains confidence that a program is completely correct. Currently, validation usually consists of another round of testing performed by a group independent of the one that developed the programs. A completely "reliable" program is one that operates without error, or "failure," every time it is executed.*

Although programmers have long known that programs are logical entities capable of precise mathematical definition, this characteristic has only recently begun to be heavily exploited by researchers interested in proving program accuracy.** Standard testing is unsatisfactory for economic and theoretical reasons: the cost of detecting and correcting all errors in large systems is usually very large; testing cannot ever demonstrate the absence of errors, only presence.[36] Two new approaches, both relying on the underlying logicality of programs but using different degrees of formality, have been summarized by Liskov and Towster.[39] These are briefly described below.

a. The Analytic Approach -- The bulk of work has gone into the more formal analytic approach. The given program is treated as a theorem to be proved of the form:

> Program X with input Y will terminate, and its output will be Z, where
>
> X is a listing of the program,
>
> Y is the range of input variable values, and
>
> Z is the resulting output.

It is usually assumed that the semantics of the language used to define the program are well known and unambiguous and that the input and output can be well defined. Usually this main theorem is broken up into a number of subtheorems, called assertions, concerning what should be true at various points in the program before the output is issued. Although the problem of determining in general whether a program terminates is insoluble, many practical programs can be shown to do so. Research has been undertaken to define appropriate notation, construct the rules of inference, prove necessary supporting theorems, and extend the concepts to more types of programs. Initially, only recursive programs with no branches were analyzed

---

*For a discussion of current testing and validation techniques, see Hetzel.[37]

**Formal recognition of this logicality may be found in McCarthy.[38]

for accuracy; but later work has encompassed programs with loops and branches[40] and asynchronous parallel programs.[41] This approach is concerned primarily with proving the correctness of programs after they have been written.

Besides the establishment of theoretical and methodological foundations, some work (principally by London) has applied proof techniques to non-trivial real-world programs that were written without any consideration of proving correctness. The list of achievements, compiled by Liskov and Towster,[39] is highlighted by proofs for:

- A program that performs error-bounded arithmetic (433 Algol statements, 46 pages of proof);
- An input routine that accepts bridge hands (167 Algol statements, 20 pages of proof); and
- A program that makes the opening bid in a bridge game (370 Algol statements, 55 pages of proof).

Although these programs had been debugged by traditional methods, the process of proving them correct uncovered several previously undetected errors.

A few methods for automating part of the proof procedure have been devised and work in this area is continuing. In one such method,[42] a special compiler has been written to accept both 1) a statement of a program in an Algol-like language and 2) assertions (supplied by the programmer) about what the program is supposed to do, in terms of changes to the values of its variables. The compiler then verifies that the assertions are true and, hence, that the program is valid.

Another method allows the machine to do part of the work while a man does the part for which a human is more qualified. An interactive system has been written[43] that manipulates the assertions and keeps track of them, while the programmer supplies the proofs. When all assertions have been proved, the complete proof is assembled and displayed by the system.

The "verifying" compiler has been used to prove the correctness of, for example:

- A program that determines whether a number is or is not prime (6 Algol statements);
- A program that sorts an array of integers into ascending order (12 Algol statements); and

- A program that multiplies two integers by successive addition of ones (23 statements).

The interactive system has aided in providing the correctness of an Algol program that reverses the elements in an array (11 lines). Some thought has been given to extending the interactive system to include more aids for proving intermediate theorems, but further study will be required.

b. The Constructive Approach -- A more pragmatic and heuristic approach to program reliability has been advanced by Dijkstra.[36, 39] The central premises of his approach are:

- The ability to prove program correctness should be kept clearly in mind while the software system is being designed and implemented.

- The structure of a program, both data and algorithm, greatly affects one's ability to prove it correct; hence, programs should be structured so as to make this task easy.

- Certain design strategies (e.g., using a hierarchy of levels of abstraction to partition the programs to be written); certain methods of directing the flow of control (e.g., using IF-THEN-ELSE and WHILE statements, but not GO TO statements); and certain programming disciplines (e.g., making the physical appearance of the source code correspond to the logical behavior of the program) should be used in structuring a program.

- A program structured into a hierarchy of levels by the above means requires only a small number of relevant test cases per level to verify the correctness of that level. Hence, the program can be validated by a complete case analysis of all levels. Testing must proceed from the lowest level upward.

Several small routines have been validated by this means, mainly in Europe. Dikjstra's major achievement so far has been the validation of a small operating system, the THE-Multiprogramming system, that he and his colleagues built.[44] Other benefits of imposing a structure on program development appear to be faster implementation (as mentioned above), greater facility in understanding and managing program complexity, and greater ability to change and adapt programs to changing tasks.

3. Applicability to Air Force Software

Although it is possible to validate some programs by various procedures different from those currently employed, questions remain about whether such techniques are applicable to Air Force software projects. A few are listed below.

- The more formal analytic proof procedures may cost as much and take as much time as testing the program, and doing the proof seems to be more difficult and complex than either testing or writing the program initially. Proofs of programs have always been much longer than the programs themselves. However, formal proofs are the only way known to achieve mathematical certainty that a program is correct.

- Automated program proving is not yet viable for practical programs, because of both the software necessary and the depth of experience required to use it effectively. Programs to help prove the correctness of other programs must be able to use a very broad and general set of axioms so that axioms that are relevant to a given assertion may be chosen. This generality, as we have mentioned, is very difficult to incorporate. Moreover, stating assertions is now an art that must be learned by human program validators; little is known about automating it. Automation of these procedures may be of only slight benefit in very large systems.

- The constructive approach may not be appropriate where high program efficiency is required.

- It is not known whether the benefits of structured programming could be obtained by large development groups.

- The details and boundaries are sufficiently unclear to produce ludicrous results if the "rules" of structured programming are applied slavishly. For example, a large program completely lacking subroutines and perhaps having an excess of repeated code is entirely possible. This may be a problem in transferring the methodology.

- Programs cannot be validated without being specified correctly. As Liskov and Towster point out,[39] standard techniques for specifying what the program has to do -- usually a combination of natural language and mathematical notation -- do not allow sufficiently complete or

precise design descriptions. It is therefore impossible to check formally on whether a program meets informally stated requirements. Although a more formal problem-statement language could help, it might simply shift the validation burden to another level. Moreover, some means would be required to determine errors in the problem statement.

Further research will be needed to determine whether the above difficulties are surmountable. Both approaches seem worth pursuing -- the analytic for the strength of its guarantee, and the constructive for its apparently easier application to practical problems.

V. SUMMARY, CONCLUSIONS, AND RECOMMENDATIONS

In this section, C&C software requirements are summarized and compared with functional performance, quality, and production capabilities projected for the industry as a whole in Section IV. Some R&D and procedural measures that could narrow the gap between capabilities and requirements are also discussed.

A. REQUIREMENTS VERSUS CAPABILITIES

Tables IV-II through IV-VII summarize projected 1985 C&C software requirements and current and expected state-of-the-art capabilities in six aspects: function, productivity and timeliness, reliability, acceptability, adaptibility, and security. In each table, the first column on the left shows the C&C software requirement in 1985; the center column shows the current state of the art; and the extreme right column indicates the expected state in 1985 and, hence, the disparity between requirements and available technology and methodology. This judgment assumes continued research in current directions and continuation of current procurement and management methods. It also assumes continuation of the slow rate at which research results are put to operational use in software production. In general, the tables indicate that the "software problem" in 1985 will equal or surpass that of today.

B. MEASURES TO NARROW THE GAP

Detailed below are a series of technical approaches and procedural alterations that could lead toward better and more expeditiously produced software. Before such steps can succeed, however, a new way of thinking about computer software must be adopted.

1. A New Perspective on Software Production

Folklore holds that the cost of software is increasing. In reality, software cost is not increasing: at worst, it is constant per programmed task, and at best, it is gradually decreasing. What has risen is the ratio of software to hardware costs, a reversal from 1:9 in 1960 to 5:1 or larger in 1971. That is because:

- Hardware costs have decreased markedly. The cost of a given processing rate (instructions per second) has decreased 8 to 40 times.

PRECEDING PAGE BLANK-NOT FILMED

UNCLASSIFIED

TABLE IV-II

1985 REQUIREMENTS VERSUS CAPABILITY: FUNCTION

1985 Requirements	Current Advanced Capabilities (State of the Art)	Expected 1985 Capabilities
"Management Information System": near-real-time update, retrieval, and display	Real-time single-purpose management information systems and networks available (e.g., SABRE).	Multipurpose capabilities available, but some tailoring needed.
Computational assistance and decision aids -- guidance, targeting, tracking, and routing.	Full capability available (e.g., Apollo).	Available.
Optimization -- weapons and force level selection and assignment.	Limited simple optimization algorithms.	Questionable -- intermediate levels available; more sophisticated levels unavailable or not sufficiently reliable.
Real-time data reduction and storage.	Available, primarily hardware-constrained.	Available.
Real-time simulation and exercise.	Low level of realism and validation.	Available, but limited capabilities in scenario generation and protocol analysis.
"Intelligent" algorithms:	Generally unavailable:	Limited availability:
- Pure pattern recognition.	- No significant operational capability.	- Nascent capability in structured situations
- Photo-image recognition.	- No significant operational capability.	- Intermediate capability.
- Automatic SIOP re-optimization.	- Insufficient level of sophistication and confidence.	- Intermediate capability.

UNCLASSIFIED

UNCLASSIFIED

TABLE IV-III

1985 REQUIREMENTS VERSUS CAPABILITY: PRODUCTIVITY & TIMELINESS

1985 Requirements	Current Capabilities	Expected 1985 Capabilities
Major software system (1 - 10 million instructions) implementation in 1 - 3 years.	Major system implementation 4 - 8 years; current programmer productivity 1 - 10 instructions/day.	Historically, increases in requirements have outdistanced advances in production technology; expected to continue.
Hardware selection and acquisition undertaken late in system development process.	Hardware chosen, initially; software tailored to hardware and application; hardware nearly obsolete prior to implementation completion.	Some amelioration due to microprogramming and other hardware-tailoring techniques.

UNCLASSIFIED

TABLE IV-IV

1985 REQUIREMENTS VERSUS CAPABILITY: RELIABILITY

1985 Requirements	Current Capabilities	Expected 1985 Capabilities
Rapid methods for insuring validity to specified levels in particular areas.	Limited crude methods of specifying criteria, objectives. Brute-force testing approach is primary technique. Checkout often consumes 50 percent of total production effort; few automated aids. Typical cost of thorough validation (for missile guidance program) is $4 - $6 per instruction.	Assessing software reliability through testing still expensive, uncertain. Little change expected: assistance through increases in HOL usage will be offset by increases in complexity. Full validation not crucial to commercial software technology.
MTBF equal to hardware (10 - 40,000 hours) for continuous on-line systems.	MTBF* in large systems (SAGE, Apollo) 8 - 48 hours.	Serious mismatch: typical software far behind hardware in reliability.

*Here, "failure" refers to any software problem, many of which can be overcome by human judgment in today's systems.

UNCLASSIFIED

TABLE IV-V

1985 REQUIREMENTS VERSUS CAPABILITY: ACCEPTABILITY

1985 Requirements	Current Capabilities	Expected 1985 Capabilities
Verified, repeatable methods for determining: - Decision-maker information needs - Effect upon total C&C system performance of change in ADP system speed, reliability, etc. - Impact upon ADP system of changing environment: force structure, threat, etc. - Alterations required in the above case.	Presently an imprecise, fallible art: few automated aids in prototype stage. - Little emphasis in current research or literature - Simulation infrequently used	Rule-of-thumb, intuitive techniques still predominant.
Precise software specification methods.	Though voluminous, current specifications still too vague and incomplete to serve as baseline for acceptability, reliability and performance evaluation.	Special-purpose techniques available (e.g., for specifying compilers); no widespread general-purpose methods.
Effective methods for determining system performance and feasibility under hypothetical stress conditions.	Simulation & exercise - insufficiently realistic: - Difficult to validate - Results often not used toward system adaptation - Often degrades operational performance during exercise	Considerable disparity between actual and possible, although technology will be available.

UNCLASSIFIED

TABLE IV-VI

1985 REQUIREMENTS VERSUS CAPABILITY: ADAPTABILITY

1985 Requirements	Current Capabilities	Expected 1985 Capabilities
Rapid retrofit of major and minor design alterations.	Poor - retrofit effort may exceed initial fitting by order of magnitude.	Some improvements via modularity, but still considerable mismatch.
Rapid and efficient transfer of software between different hardware configurations.	Poor - transfer of SACCS to WWMCCS hardware will require a 200-programmer effort for three years.	Uncertain - microprogramming, increased standardization, and increased use of HOLs should decrease problems.
Minimization of time required for transfer of software responsibility between individuals.	Minimal - very sensitive to style, methods, and techniques used by individual programmers; HOLs somewhat lessen the problem.	See above.

UNCLASSIFIED

TABLE IV-VII

1985 REQUIREMENTS VERSUS CAPABILITY: SECURITY

1985 Requirements	Current Capabilities	Expected 1985 Capabilities
Installations and communications links secure from compromise, tampering, or sabotage.	All software currently fallible: system certified by NSA as "secure" broken with six man-weeks of effort.	Improved: promising research efforts currently underway.
Compartmentalized information secure at various levels within a multi-access system.	Classified computing not allowed in multi-access environment; only certified protection involves physically securing the system.	Improved: combination of hardware, software, and encryption techniques currently under development.

UNCLASSIFIED

- New computing requirements affect software much more directly than hardware. An increase in hardware speed or capacity creates new demand, which must always be filled by more, and more complex, software.

Thus, an increase of hardware speed by 8 to 40 times has meant that 8 to 40 times as much software is generated to use a faster processor. Indeed, the cost of software has risen in direct proportion to the decreasing cost of hardware (per instructions/second executed). The fact that the ratio has not increased to 40:1 indicates that the productivity of programmers has actually risen, although unspectacularly.

Second, it must be realized that software, unlike hardware, is not a "breakthrough" technology. Progress in software comes from the gradual accretion of knowledge, expertise, and methods. Improvements in software production cannot be hastened except by attacking problems on several fronts with several weapons -- which raises the price of research and development. As compensation, however, raising the productivity of programmers by one instruction per day would save the Air Force $25 to $50 million per year.

2. Specific Recommendations

a. Procedures for Keeping Better Records of all USAF Software Development Projects -- Despite repeated recommendations, few data are being collected -- in government or industry -- on the history, progress, results, and personnel participating in software development projects. Efforts to improve the software production process -- to assess qualitative and quantitative aspects of the process and product -- are severely handicapped by a lack of data concerning past efforts. Managers of software development projects are similarly hindered because the lack of data makes it difficult to, say, estimate required effort. Data are needed to assess the utility of various programming languages, institutional approaches, and methods of choosing personnel.

Before a comprehensive software data base can be established, developers must deal with a number of questions:

- What data should be collected?
- How do different organizations subdivide the software development effort for consistent reporting?
- What are effective, unobtrusive procedures for collecting such data?

- Precisely what sorts of analyses would be performed? (This may be a primary question, as it defines the formats and data items.); and

- What are the costs (both institutional and direct) of different levels and techniques of data collection?

Answers to the above questions will provide a framework for the rapid and efficient collection of data. In the near term, these data can be used to gain insights into the nature of software production, the utility of various production methodologies, and programmer productivity in different project organizations. In the long term, the data collected should serve as a base for both research and development of new techniques and management tools for estimation, planning, and control of Air Force software development projects.

b. Allocate the Budget for Information-Processing R&D so as to Reflect Software's Pre-eminent Share in ADP System Cost and Performance -- Although information plays a crucial role in decision-making, the current state of knowledge about software and its production methods is hazy. The USAF investment in software procurement is estimated at over $1 billion per year, yet the funds allocated for research on software and its production methods is small. In FY 1972, only about $10 million was invested in all information-processing R&D (6.1, 6.2, 6.3, 6.4, and 6.5).* Further, only about 30 percent of that $10 million was devoted to software problems.

These R&D budget figures in toto are extremely low relative to the magnitude of software procurement and the overall importance of information. To illustrate by way of comparison, total USAF R&D expenditures (6.2 and 6.3) on structures and materials are about $40 million, but the success or failure of future Air Force operations will likely depend at least as much on information structures as on physical structures.

c. Find Better Ways to Analyze Requirements and Specify Software Design -- One possibility involves testing and exercising C&C systems by carrying out artificial (although realistic) procedures similar to those that might be encountered operationally. Such operational simulations are intended to provide the C&C system with inputs that duplicate as closely as possible the information that would be re-

*This figure excludes direct development, such as that carried out through SPOs, which undoubtedly has some additional research value.

ceived in a real threat. The system's functioning under these conditions can suggest its effectiveness, efficiency, and reliability.

Besides its significant potential for improving future combat-readiness, operational simulation is an aid to requirements analysis. Current methods are haphazard and generally ineffective. Operational simulation can be used to show the commander the results -- in a lifelike environment -- of the information needs he has specified. He may then alter his judgment of needs or approve the service provided by the system as simulated. The use of operational simulation can thus serve as an iterative aid to eliciting information requirements, with each simulation-alteration phase successively refining the precise needs of the commander.

Simulation improves combat-readiness by revealing "bugs" and weak points in systems, and frequent testing by this method helps to maintain a high level of operational preparedness, both in hardware and software components and in the personnel charged with operating the system. Nor should the value of such exercises in building morale be overlooked.

There are additional benefits from building a repeatable simulation capability into a C&C system early in its life cycle. The most significant is that it can serve as a prototype of the system and thus allow early validation of the requirements analysis. Determination of simulation parameters can often point to design deficiencies and weak spots in the system's overall conception. Another benefit is a potential cost saving, as retrofitting of the simulation capability can often be more expensive and is certainly more costly in time and disruption of C&C system operations.

Operational simulation is not a panacea, however. It can do little to alter the shape of a system that is poorly designed or executed. The value of operational simulation in other respects remains open to question. It is very difficult to provide an atmosphere that is sufficiently realistic to introduce the stress that probably would accompany actual situations. Therefore, the level of confidence in a successful simulation is relatively low. Finally, any simulation degrades operational capability to some degree during the simulation period.

d. Establish a USAF Information-Processing Technology Staff -- Such vital functions as data gathering, technology dissemination, and configuration management standards cut across all of the classical R&D categories; a single organization is necessary to provide coherent and efficient performance of such information-processing staff functions. An organization should be formed to provide user services, to build and maintain libraries and tool inventories, information gathering and analysis, standards, interservice coordination, evaluation of experi-

mental developments, and long-range planning. It should not, at this stage, compete with existing organizations in information-system research, development, operations, and management. Figure IV-11 shows a time-phased plan by which these staff functions could be introduced and the manpower to allot to each.

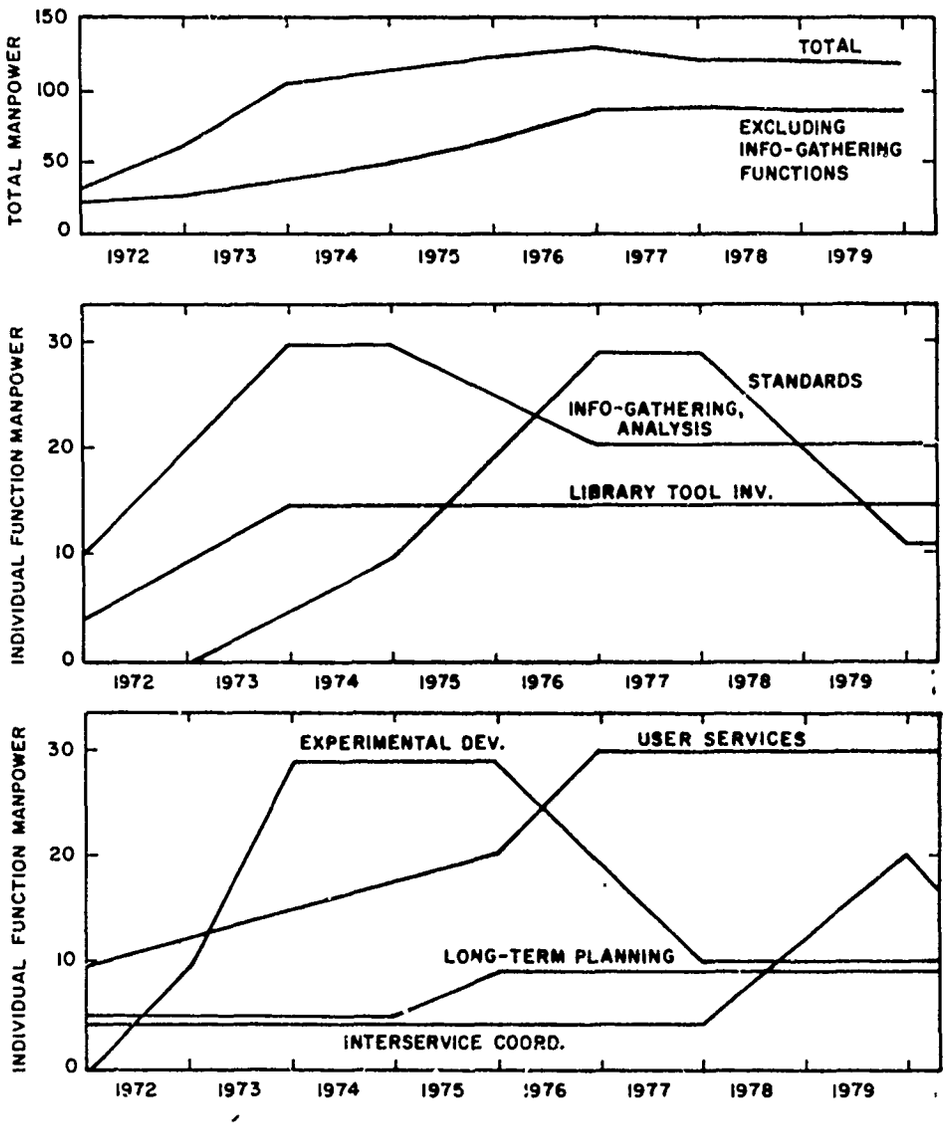

Figure IV-11. Air Force Information-Processing Technology Staff: Recommended Manpower Plan

Table IV-VIII specifies the types of tasks performed in each function. Initially, this organization should undertake projects with potential early payoffs, to demonstrate its feasibility. Although this organiza-

TABLE IV-VIII

SUGGESTED INFORMATION-PROCESSING STAFF FUNCTIONS*

Function	Tasks	USAF Manpower on This Function** (percent)	Existing Manpower Sources
Standards	Set Standards for Industry Products Documentation OS, DMS, compilers Configuration Management	20	ESD, Air Staff, RADC, User Commands
Interservice Coordination	Joint Standards Software Library	20	ESD, Air Staff, User Commands
Information-Gathering & Analysis	Plan What to Gather Monitor Data Collection Analyze, Replan Develop IGA Guidelines	20	ESD, SAMSO, Air Staff
Experimental Developments	Select Tools, Guidelines for Test Experimental Designs Monitor Experiments Analyze, Replan	5	RADC, HRL, ESD, SAMSO
Library, Tool Inventory	Develop, Maintain Software Library Plan, Direct Tools Comparisons	25	SAMSO, ESD, DSDC
User Services	On-Site Experts Consultant Log Workshop Coordination Specialized Courses	50	various
Long-Term Planning	Develop, Monitor USAF Computer Resources Plan	40	Air Staff, ESD, RADC

*Excluding R&D, management and individual applications

**To define by example, the 20 percent in the top row means that for every staff member working on standards, four others are working on standards elsewhere in the Air Force.

tion should support the full range of Air Force activities, several C&C problems are worthy of special attention: software/system certification, data security, information system design/analysis methodology, and computer system performance analysis.

Three technical concepts are described below, which appear likely to reduce future Air Force software difficulties. Although these are by no means the only technical areas for fruitful research, they are of paramount promise. Other topics for investigation will emerge through these and other studies, particularly the data collection and analysis effort outlined above.

e. Investigate the Applicability of Structured Programming to Air Force C&C Software -- Though the term "structured programming" has been used for other specific endeavors, it best describes a variety of techniques including higher-order languages such as AED; programming techniques as exemplified in Dijkstra's Technische Hogeschool Eindhoven (THE) operating system; and innovative structuring of software production such as the IBM Chief Protrammer Team (CPT) experiment. Although each of these activities is somewhat different, they all represent an attempt to bring a "top-down" approach to software production and to minimize logical errors and inconsistencies through structural simplification of the development process. In the case of the THE system, the approach includes requiring system coding to be free of discontinuous program control ("GO-TO-FREE"). In the CPT approach, a single individual is chosen to do the majority of actual design and programming; he tailors a support staff around his function and talents.

None of these systems or concepts has yet been rigorously tested. Indications are, however, that the structured approach can significantly shorten the software development process. In one case, the use of AED reduced the man-effort of a small system from an estimated six man-months to two man-weeks. An experiment using the CPT concept (on a system for the New York Times) halved expected project costs and reduced development time to 25 percent of the initial estimate.

Because structured-programming processes produce relatively error-free and well documented software, they might be particularly well suited to C&C information processing. A cursory analysis shows that, if certain "folklore" assumptions are true, increasing the degree of system code structure or project structure can simultaneously reduce both programming costs and the probability of a mjaor undetected error in either coding or system integration. In addition, some methods of structured programming provide for easy transfer to program maintenance responsibility (through simplified coding and self-documenting systems), thus reducing costs over the lifetime of a system. On the other hand, too much of the wrong kind of structure produces an inflexible and untransferable system.

Current USAF software-procurement procedures do not permit specifying any form of structured programming as the required approach. If and when they do, a great deal of care will be needed because, even more than in most applications, structured programming requires very senior personnel, highly trained and skilled in both systems analysis and programming. Very few such people exist in the Air Force, either in commissioned or in civilian status. All Air Force career paths for officers and civilian counterparts lead to management positions. There is currently no extensive, technical "software" career path. Should such a path be established, however, retaining these skilled specialists would become a problem.

Experiments have shown that these approaches can remarkably reduce the time and effort required for software development. However, only a small number of experiments have been done; little has been reported about the ultimate quality or usefulness of the programs produced; and the results of experiments cast as experiments must be viewed with caution (because of the Hawthorne effect*).

Nevertheless, the promising results justify further, broader experimentation. An initial recommended step is the design of a series of controlled experiments to determine:

- The difference in development time and effort between "structured" and usual approaches;
- The differences in program quality (reliability, core utilization, speed, transferability) between structured and usual approaches; and
- The types of analysts and programmers most effective with each approach.

To insure the long-term validity and relevance of such experimentation, related steps must be taken:

- Investigation of other methods of bringing structure to the programming process, ranging from establishment of extensive program-quality standards to more sophisticated techniques of "software engineering."

*Interference of the observers with the subject being observed. Named after industrial psychology experiments conducted at GE's Hawthorne Works, wherein productivity rose no matter how working conditions were changed because workers worked harder just because they were the subjects of attention.

- Development of career paths for both commissioned and civilian personnel, allowing career advancement in technical disciplines.

- Development of training programs within USAF to motivate and instruct commissioned and noncommissioned officers in highly technical areas.

- Review and possible revision of retention incentives and software-procurement regulations.

f. Investigate the Feasibility, Costs, and Benefits of a "Software-First" Machine -- The "software-first" machine is a highly generalized computer capable of simulating the behavior of a wide range of hardware configurations. It would allow the systems designer first to configure the software, then use the software machine to determine an effective hardware configuration. A software machine would also allow testing of software for implementation on hardware not yet constructed. A possible configuration of such a machine is shown in Figure IV-12.

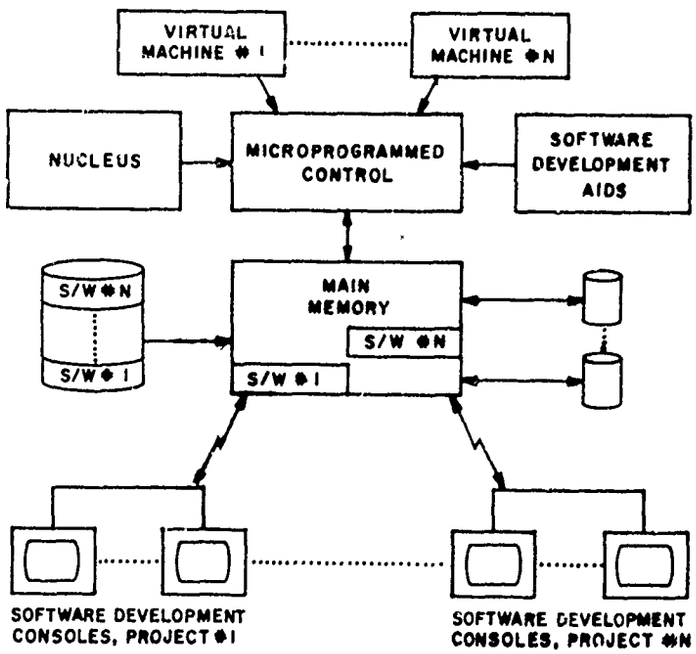

Figure IV-12. The Software-First Machine: Possible Configuration

Use of the software-first machine could shorten the time from conception to implementation of an integrated hardware-software system. In the usual procurement process, the hardware is chosen first. If special-

purpose hardware is required, the software cannot be debugged until the hardware has been delivered. With the software machine, debugging could be done independently. Since the checkout phase can consume 40 to 60 percent of the total software development effort, that means a huge saving in elapsed time. This saving translates directly into increased system operating life. A danger is that the software machine might have a "centrifugal effect" on the hardware development: allowing designers to tailor hardware to software might result in the proliferation of similar though critically different computers, each used for a special purpose. For WWMCCS-like systems, where standardization is vital, this aspect is critical; for radar processors, guidance computers, or communications processors, it is less important.

We do not know, of course, whether the software-first machine can be built and, if so, whether its cost would be "reasonable." The appearance of hardware radically different from conventional digital computers (such as the CDC STAR and ILLIAC IV configurations) has greatly expanded the hardware options predictable for 1985. What might be expected to appear, and seems technologically and economically feasible, is a range of software-first machines, each capable of efficiently simulating the performance of a class of computers. Otherwise, the software machine's configuration would have to be so general that its efficiency for any given task would be seriously degraded. However, the range may not need to be too broad to encompass most Air Force C&C tasks.

Developments related to the basic software-first machine are also likely to be useful in solving other Air Force problems. As a current example, the USAF Satellite Control Facility has acquired some microprogrammed computers that will first be made to emulate existing second-generation computers there. Later, the Facility's software and system functions will be upgraded using a microprogrammed base. In this way, they can avoid system down time that would result from a simultaneous hardware and software transition. Thus, the great potential benefit of the software-first machine and its potential spinoffs demands that it be seriously investigated.

The topics suggested for future research are summarized in Table IV-IX, with a description of the potential benefits and drawbacks of each, and specific recommendations for near-term action.

C. CONCLUSIONS

The technical innovations suggested here should be investigated. If successfully implemented, they could improve the development of C&C systems in three ways. First, the work of developing the software for a particular C&C project could be significantly reduced by using structured approaches to software design and production.

UNCLASSIFIED

TABLE IV-IX

SUMMARY OF R&D RECOMMENDATIONS

Concept	Potential Benefits	Drawbacks	Near-Term Steps
Structured Programming	Reduced design/programming effort. More error-free software. Better documentation.	May require rare, uniquely skilled individuals. Lack of USAF "software career" paths. Lack of USAF retention incentives. Inability to specify development method in software procurement.	Controlled experimentation. Data collection, instrumentation of software project. Develop software career paths. Review/revise incentives. Establish training/OJT programs.
Operational Simulation	Improved exercising capability. Aid to requirements analysis. Step toward simulation of future systems.	Expensive. Low confidence. Degrades capability during simulation period.	Limited implementation. Further exploration/experimentation.
Software-First Machine	Allows integrated hardware/software design. Allows program test prior to hardware fabrication or delivery.	Expensive. Feasibility/range of options questionable. Centrifugal effect on hardware.	Study of feasibility. Explore costs/benefits of other options (e.g., networks and software simulation).
Information-Processing Technology Staff Organization	Concentration of talent might lead to development of better methodology.	Concentration of talent might degrade command expertise.	Establish pilot center under this organization to provide training; develop methodology.

UNCLASSIFIED

Second, the work of developing an entire C&C system could be shortened considerably, primarily because the software-first machine would allow overlapping of system design and software production and testing, and because the use of operational simulation would shorten the training period required for commanders and system operators.

Finally, and perhaps most important, the elapsed time between hardware specification (and acquisition) and system implementation would be materially reduced. In operational terms, the total useful life of a system would be extended because the hardware would become obsolete much later than it now does after software implementation. The potential system-development process in 1985 -- under the effects of the innovations described here -- is shown in Figure IV-13, contrasted with the current process. The six-year time span is representative of a typical current system and was chosen to illustrate the time that can be saved.

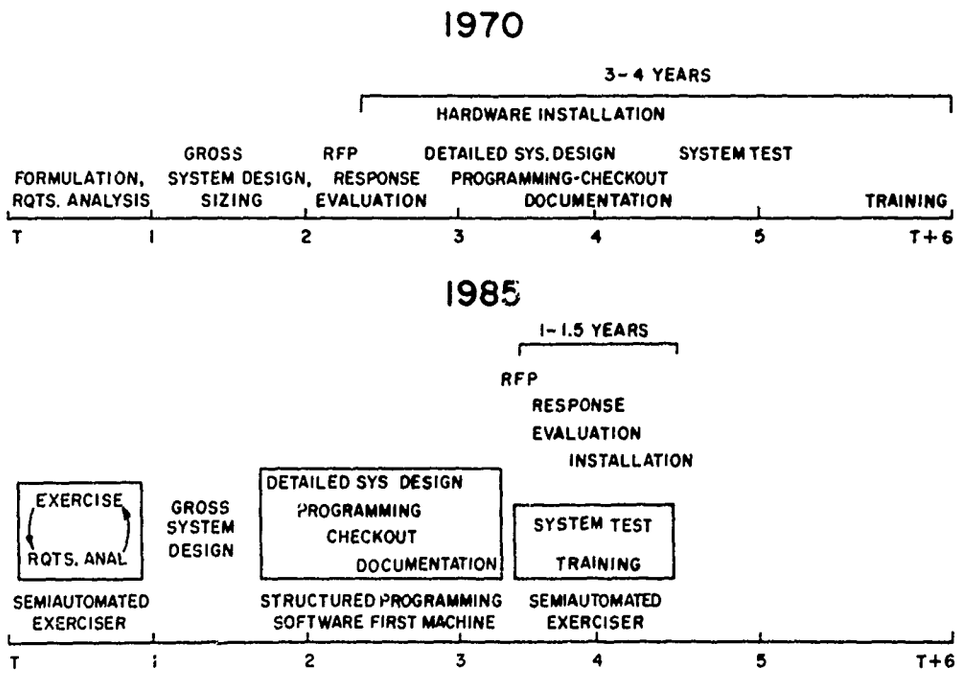

Figure IV-13. The System Development Process, 1970 and 1985 (potential)

REFERENCES

1. P. E. Rosove, Developing Computer Based Information Systems, John Wiley & Sons, New York, 1968.

2. Personal communication with B. W. Boehm.

3. B. W. Boehm, "Some Information Processing Implications of Air Force Space Missions: 1970 - 1980," RM-6213, The RAND Corporation, Santa Monica, California, 1970.

4. "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s, Volume III, Command and Control Requirements: Intelligence," SAMSO/XRS, AFSC, January 1973, SECRET.

5. "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s, Volume II, Command and Control Requirements: Overview, Annex A: Strategic Requirements," SAMSO/XRS, AFSC, June 1972, SECRET.

6. "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s, Volume V, Hardware Forecast," SAMSO/XRS, AFSC, January 1973, UNCLASSIFIED.

7. Software Engineering, ed. by Peter Naur and Brian Randell, Scientific Affairs Division, NATO, Brussels, Belgium, January 1969.

8. M. V. Wilkes, D. J. Wheeler, and S. Gill, The Preparation of Programs for an Electronic Digital Computer with Special Reference to the EDSAC and the Use of a Library of Subroutines, Addison-Wesley Press, Inc., Cambridge, Massachusetts, 1951.

9. Saul Rosen, "Electronic Computers: A Historical Survey," Computing Surveys, 1, 1, March 1969, pp. 7 - 36.

10. H. Sackman, Computers, System Science , and Evolving Society.

11. R. F. Rosen, "Supervisory and Monitor Systems," Computing Surveys, 1, 1, March 1969, pp. 37 - 54.

12. F. J. Corbato, M. M. Daggett, and R. C. Daley, "An Experimental Time-Sharing System," Proc. SJCC, 21, Spartan Books, Baltimore, Maryland, 1962, pp. 335 - 344.

13. Computers and Thought, ed. by E. A. Feigenbaum and J. Feldman, McGraw-Hill Inc., New York, 1963.

14. "A Survey of the Computer Field," Computers and Automation, 12, 1, Industrial Securities Committee, Investment Bankers Association of America, 1963, pp. 15 - 25.

15. Jean E. Sammett, Programming Languages: History and Fundamentals, Prentice-Hall, Inc., Englewood Cliffs, N. J., 1969.

16. B. W. Boehm and J. E. Rieber, Graphical Aids to Aerospace Vehicle Mission Analysis, P-3660, The RAND Corporation, Santa Monica, California, October 1967.

17. "Mark IV File Management System Reference Manual," 2nd ed., Document No. SP-70-810-1B, Informatics, Inc., Canoga Park, California, 1970.

18. BMD -- Biomedical Computer Programs, ed. by W. J. Dixon, University of California Press, Berkeley, California, 1970.

19. G. D. Brown and C. H. Bush, "The Integrated Graphics System for the IBM 2250," RM-5531-ARPA, The RAND Corporation, Santa Monica, California, October 1968.

20. R. J. Rubey, "Comparative Evaluation of PL/I," ESD-TR-68-150, Electronic Systems Division, AFSC, April 1968.

21. A. Kreger and J. Nathanson, "The Tribulations and Triumphs of GIS," Datamation, 17, 20, October 15, 1971, pp. 20 - 25.

22. Software Engineering Techniques, ed. by J. N. Buxton and B. Randell, Scientific Affairs Division, NATO, Brussels, Belgium, April 1970.

23. G. W. Armerding, Computer Software: The Evaluation Within the Revolution, P-3894, The RAND Corporation, Santa Monica, California, July 1968.

24. B. W. Sine, " An Autonomous Quick Reaction Software System, STS," TOR-0059 (6758-03)-3, Aerospace Corporation, El Segundo, California, October 26, 1970.

25. J.F. Corbato and V.A. Vyssotsky, "Introduction and Overview of the Multics System," Proc. SJCC, 27, Spartan Books, Baltimore, Maryland, 1965, pp. 185 - 202.

26. Clinton S. McIntosh, et al, "Analysis of Major Computer Operating Systems," ESD-TR-70-377, Electronic Systems Division, AFSC, August 1970.

27. R. Turn, "Air Force Command and Control Information Processing in the 1980s: Trends in Hardware Technology," R-1011-PR, The RAND Corporation, Santa Monica, California, October 1972.

28. F.J. Corbato, "PL/I as a Tool for System Programming," Datamation, May 1969, pp. 68 - 76.

29. "Air Force ADP Experience Handbook (Pilot Version)," ESD-TR-66-673, Electronic Systems Division, AFSC, December 1966.

30. J.P. Haverty, "The Role of Programming Languages in Command and Control: An Interim Report," RM-3293-PR, The RAND Corporation, Santa Monica, California, September 1962.

31. Data Processing and Display, Project FORECAST Final Report, available from the FORECAST Special Project Office, HQ AFSC (SCGF), Andrews AFB, Maryland, January 1964.

32. E.A. Feigenbaum, "AI: Themes in the Second Decade," CS Memo No. 67, Stanford University, Computer Science Department, Stanford, California, August 1968.

33. R.M. Balzer, "On the Future of Computer Program Specification and Organization," R-622-ARPA, The RAND Corporation, Santa Monica, California, August 1971.

34. D.T. Ross, "Fourth-Generation Software: A Building-Block Science Replaces Hand-Crafted Art," Computer Decisions, April 1970, pp. 32 - 38.

35. D.C. Engelbart, Advanced Intellect-Augmentation Techniques, SRI Project 7079, Stanford Research Institute, Menlo Park, California, July 1970.

36. E.W. Dijkstra, Notes on Structured Programming, IH-Report 70-Wsk-03, Department of Mathematics, Technological University, Eindhoven, The Netherlands, April 1970.

37. Program Test Methods, ed. by William C. Hetzel, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

38. J. McCarthy, "A Basis for a Mathematical Theory of Computation," Computer Programming and Formal Systems, ed. by P. Braffort and D. Hirschberg, North Holland Publishing Co., Amsterdam, 1963, pp. 33 - 70.

39. B.J. Liskov and E. Towster, "The Proof of Correctness Approach to Reliable Systems," ESD-TR-71-222, Electronic Systems Division, AFSC, July 1971.

40. R.W. Floyd, "Assigning Meaning to Programs," Proceedings of Symposia in Applied Mathematics, American Mathematical Society, 19, pp. 19 - 32.

41. E.A. Ashcroft and Z. Manna, "Formalization of Properties of Parallel Programs," Memo AIM-110, Stanford Artificial Intelligence Project, Stanford, California, 1970.

42. J.C. King, "A Program Verifier," (thesis), Carnegie-Mellon Computer Science Department, September 1969.

43. D.I. Good, "Towards a Man-Machine System for Proving Program Correctness," (thesis), University of Wisconsin, University of Texas Computation Center, June 1960.

44. E.W. Dijkstra, "The Structure of the 'THE' - Multiprogramming System," CACM, 11, 5, May 1968, pp. 341 - 346.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Space and Missile Systems Organization P.O. Box 92960, Worldway Postal Center Los Angeles, CA 90009	2a. REPORT SECURITY CLASSIFICATION Unclassified 2b. GROUP

3. REPORT TITLE
Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s (CCIP-85) (U), Volume IV, Technology Trends Software

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

5. AUTHOR(S) *(First name, middle initial, last name)*

6. REPORT DATE October 1973	7a. TOTAL NO. OF PAGES 70	7b. NO. OF REFS 44
8a. CONTRACT OR GRANT NO. b. PROJECT NO. c. d.	9a. ORIGINATOR'S REPORT NUMBER(S) SAMSO TR 72-122 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	

10. DISTRIBUTION STATEMENT

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY

13. ABSTRACT

Command and control software will become more important in 1985 than it is today. It is crucial to determine whether future software technology, as projected from current trends, will be able to provide the techniques necessary to build the appropriate software of the future.

This volume provides a brief introduction to software technology and defines the kinds of software that will be required to build and operate 1985 C&C systems. Having established requirements, the report then focuses on relevant software technology to forecast what it may be able to achieve by 1985. Both application and executive software are considered, with special emphasis on response time, adaptability to unforeseen situations, suitability, and ease of transfer from one machine to another. Of particular importance in C&C systems are methods for the design, production, and validation of software, and the management techniques necessary to administer large software-development projects. Current tools and practices are assessed.

Finally, the estimates of 1985 software technology capabilities are compared with projected 1985 requirements for C&C software. The concluding section of the report outlines studies, projects, and R&D investments that the Air Force might undertake to narrow the expected gap between requirements and technology and to alleviate future problems in implementing and operating command and control software.

DD FORM 1 NOV 65 1473

UNCLASSIFIED
Security Classification

UNCLASSIFIED
Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT

UNCLASSIFIED
Security Classification