TOWARD AN OPTIMUM PROGRAMMING LANGUAGE
FOR COMMUNICATIONS COMPUTERS

by
Benjamin Donald Blood, Jr.

A Thesis Submitted to the Faculty of the
DEPARTMENT OF SYSTEMS AND INDUSTRIAL ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE
WITH A MAJOR IN SYSTEMS ENGINEERING
In the Graduate College
THE UNIVERSITY OF ARIZONA

1973
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Benjamin O. Blood

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

R. L. BAKER
Associate Professor of Systems and Industrial Engineering

Date

Dec 14, 1973
This thesis resulted from the melding of two different aspects of my life in recent years. First, of course, was my studies at The University of Arizona in Systems Engineering. In my coursework, I was exposed to a formalized systems engineering methodology for the design of any system whether limited in scope or involving large, complex, man-machine environments. During a seminar entitled BIGOPS in the Spring of 1973, I participated in the application of this methodology to a real-world problem. Although the seminar was fruitful in its objectives, I felt I needed to understand the methodology better and, perhaps, carry out a systems engineering methodological study in a more comprehensive manner than was possible at the seminar.

At this point the second aspect fortuitously joined with my above needs. My education was being sponsored by the U. S. Army and I felt that I might partially repay them by studying one of their actual problem areas. I, therefore, journeyed to Fort Huachuca, Arizona, and after numerous discussions with various officers and civilians at U. S. Army Communications Command, settled on the problem area considered herein. In effect, they unofficially became my client and the contents of this thesis accordingly reflect some of their desires. Nevertheless, the thesis is my work as a student and does not imply or state any official U. S. Army position.
I would like to thank the officers and civilians at U. S. Army Communications Command at Fort Huachuca for their assistance as well as the faculty and staff of the Systems and Industrial Engineering Department of The University of Arizona. In particular, I would like to thank Doctor R. L. Baker, Associate Professor of Systems and Industrial Engineering, for his advice and criticism, Doctor A. Wayne Wymore, Professor and Head of the Department of Systems and Industrial Engineering, for his sometimes personal instruction of the systems engineering methodology he authored and Doctor G. D. Ripley, Assistant Professor of Computer Science for his instruction and his advice on programming languages and translators. Most important of all, I am indebted to my wife, Louise, for her patience as well as for her assistance in typing and proofing this manuscript.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF ILLUSTRATIONS</td>
<td>vii</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>viii</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Computer Systems and Their Costs</td>
<td>1</td>
</tr>
<tr>
<td>Communications Computer Systems</td>
<td>3</td>
</tr>
<tr>
<td>Systems Engineering Approach</td>
<td>9</td>
</tr>
<tr>
<td>2. LITERARY PROBLEM FORMULATION</td>
<td>12</td>
</tr>
<tr>
<td>Statement of Need</td>
<td>12</td>
</tr>
<tr>
<td>Statement of Need Satisfaction Criteria</td>
<td>14</td>
</tr>
<tr>
<td>Statement of Available Resources</td>
<td>18</td>
</tr>
<tr>
<td>Statement of Resource Utilization Criteria</td>
<td>19</td>
</tr>
<tr>
<td>Statement of Trade-Off Criteria</td>
<td>25</td>
</tr>
<tr>
<td>Statement of System Testing Requirements</td>
<td>28</td>
</tr>
<tr>
<td>3. SYSTEM THEORETIC PROBLEM FORMULATION</td>
<td>31</td>
</tr>
<tr>
<td>Project Name</td>
<td>31</td>
</tr>
<tr>
<td>Input/Output Specification</td>
<td>32</td>
</tr>
<tr>
<td>Technology</td>
<td>35</td>
</tr>
<tr>
<td>Input/Output Merit Ordering</td>
<td>40</td>
</tr>
<tr>
<td>Technology Merit Ordering</td>
<td>49</td>
</tr>
<tr>
<td>Feasibility Merit Ordering</td>
<td>61</td>
</tr>
<tr>
<td>Procedure for System Testing</td>
<td>67</td>
</tr>
<tr>
<td>4. PRELIMINARY SYSTEM DESIGN</td>
<td>73</td>
</tr>
<tr>
<td>Circumscription of Feasible Solutions</td>
<td>73</td>
</tr>
<tr>
<td>Existing Higher Level Languages</td>
<td>78</td>
</tr>
<tr>
<td>New Higher Level Language</td>
<td>80</td>
</tr>
</tbody>
</table>
TABLE OF CONTENTS—Continued

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5. CONCLUSIONS</td>
<td>87</td>
</tr>
<tr>
<td>Synopsis of Results</td>
<td>87</td>
</tr>
<tr>
<td>Areas of Future Study</td>
<td>90</td>
</tr>
<tr>
<td>APPENDIX A: FUNCTIONAL REQUIREMENTS FOR A PROGRAMMING LANGUAGE FOR COMMUNICATIONS COMPUTERS</td>
<td>94</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>101</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Hardware/Software Cost Trends</td>
<td>4</td>
</tr>
<tr>
<td>2. Potential Communications Computer System</td>
<td>5</td>
</tr>
<tr>
<td>Interconnection</td>
<td></td>
</tr>
<tr>
<td>3. Summary of Need Satisfaction Criteria</td>
<td>15</td>
</tr>
<tr>
<td>4. Summary of Resource Utilization Criteria</td>
<td>20</td>
</tr>
<tr>
<td>5. Summary of Trade-Off Criteria</td>
<td>26</td>
</tr>
<tr>
<td>6. Input/Output Diagram</td>
<td>33</td>
</tr>
<tr>
<td>7. Candidate Existing Higher Level Languages</td>
<td>79</td>
</tr>
</tbody>
</table>
ABSTRACT

A systems engineering methodology is used to study the problem of determining an optimum programming language for communications computers. Definition of this problem in detail begins with a literary formulation of the need, need satisfaction criteria, available resources, resource utilization criteria, trade-off criteria and system testing requirements. A system theoretic formulation of this problem follows with an input/output specification; technology listing; merit orderings of the set of systems that satisfy the input/output specification, that are buildable in the technology, and that are feasible in both; and a procedure for system testing. The best approaches to preliminary design of the target optimum programming language is then limited to modification of existing higher level languages or utilization of a macro language. Finally, the technical characteristics of the proposed programming language for communications computers is outlined.
CHAPTER 1

INTRODUCTION

In the limited and specialized field of software for communications computer systems, a phantom that continues to plague managers and planners of all computer systems is looming larger. This phantom is increasing cost. The cost of software for new procurements is compounded by increased personnel salaries, requirements for more versatility and usually less time to do the job. How can this cost rise be combatted? What new techniques will lower the expense of software development? How can they be applied to communications software development? As a basis for definitive answers to these questions, a cursory look at computer systems and their costs seems an appropriate beginning.

Computer Systems and Their Costs

An automatic data processing system (ADP) is defined (Sippl and Sippl 1972, p. 435) as an interacting assembly of procedures, processes, methods, personnel and equipment to perform a complex series of data processing operations. Broadening this definition a bit, the short phrase, "computer system", is a system housed in an appropriate facility composed of devices — hardware — which perform various operations based on instructions — software — which
have been written by people. Each of the components — hardware, software, personnel and a facility — are essential for a computer system. Each of these components has developed significantly during the twenty to thirty years of the modern computer era.

The development of two components, hardware and software, has been used as a measure of overall development from technology and cost standpoints of computer systems. In the case of hardware technology the measurement has traditionally been in terms of generations (Bartee 1972, pp. 2-6). The first generation was characterized by the utilization of vacuum tubes, the second by the utilization of transistors, the third by the utilization of integrated circuits, and the fourth by the utilization of medium-scale integration (MSI) and large-scale integration (LSI) technology. Some companies have even postulated a fifth generation of computers, primarily as a basis for advertising. As each generation succeeded the other, the unit cost of an equivalent computing capability, such as core memory went down significantly.

In the case of software, the divisions are not as clear cut and certainly not equivalent to the generations listed for hardware. Some people (Sammett 1967, pp. 2-8) measure software progress in terms of operating system development. Others in the development of programming languages — machine language to symbolic assembly language to higher level languages. Still others (Boehm 1973, p. 50) measure software progress in terms of the complexity
of the problems that could be programmed. In computer systems, even with advances in programming techniques, the cost of software has been steadily rising.

The direct cost of programming (Boehm 1973, p. 49) has been estimated at 5% in 1950, 50% in 1965 and is variously estimated today at 66% to 80% of the total cost of establishing a computer system (Fig. 1). If the software versus hardware cost appears lopsided now, consider what will happen in the future. As hardware becomes less expensive and software (primarily people) costs go up, software expenditures could increase to 90% of total computer system costs in the 1985 time frame. Reductions of this cost are receiving increasing attention from managers at all levels. For these same reasons, managers of communications computer systems must investigate means of reducing software costs.

Communications Computer Systems

To understand better the above statement and what a communications computer system does, the following paragraphs will highlight the hardware and software of selected civilian and military systems.

A communications computer system (Fig. 2) or a computer network is usually composed of two or more dependent or independent computer systems interconnected by a transmission means — commonly telephone lines — to transmit and receive information between users. The computer system could be hardwired computers (program is
Fig. 1. Hardware/Software Cost Trends
Fig. 2. Potential Communications Computer System Interconnection
pre-wired on removable boards) or programmable computers of all sizes. The users can be people or computers who interface manually to the system via local or remote input/output devices; ADP computers that interface electrically with the communications computer; or computers that contain both the communications and ADP functions in the same processor which interfaces directly to the network. Some networks are so vast that certain computers called message switches serve primarily as a means of properly routing message or data traffic to its destination computer system.

A typical non-military computer network is the ARPA (Advanced Research Project Agency) network (Farber 1973, p. 36). It is a nationwide system designed both to explore network technology and to interconnect and service ARPA-sponsored research centers. Its key aim is to allow the accessing of programs, services and data from any place in the network. It is composed of two electrically-connected parts. One consists of the computers which will provide the computational services. The other is composed of computers whose function is to service the communications needs of the network.

The communications portion of the ARPA network consists of modified Honeywell computers called nodes which are connected via leased telephone lines. The communications system operates in a message-oriented, store-and-forward manner. That is, the messages are sent from node to node until they reach their destination. The nodes also break up long messages into multiple smaller messages.
called packets. Later, at the terminal node, the communications computer reassembles the packets for transmission to the destination ADP computer. The communications computers also insure optimum routing of messages and efficient utilization of transmission facilities.

Another communications computer system (Farber 1973, p. 39), the TSS network, was developed by IBM and some of its customers as a network of homogenous computers. Each of the computers, IBM 360/67's, serve both as a communications computer and a job-oriented data processing installation. The computers utilize point-to-point circuits and the communications software operates as a user program via an access method. In this experimental network, because of equipment similarity, program and data interchange is available as well as dynamic file access and remote batch operations.

Other civilian networks (Farber 1973, pp. 36-39) are the CYBERNET, the Distributed Computer System, the MERIT network, the Triangle Universities Computational Center network and the OCTOPUS system.

An important military network is the Defense Communications System (DCS) Automatic Digital Network (AUTODIN). It is the largest system of its type in the world. It is a worldwide Department of Defense (DOD) computerized, general purpose system which provides for the transmission of narrative and data pattern, classified or unclassified traffic on a store-and-forward (message switching)
basis, and to a lesser degree on a direct point-to-point (circuit switching) basis. It consists of seventeen AUTODIN switching centers (ASC) in the United States and overseas. Each ASC is equipped with 100 to 300 terminations for interswitch trunks and access lines. Terminal equipments connected to the ASC's via leased or government-furnished circuits run the gamut from simple teletypewriters through hardwired computer terminals and programmable small computer terminals to direct interface with a large computer which has both the traditional ADP functions and communications in the same central processor.

Another DOD system run by the Air Force is the Automated Weather Network (AWN). It is a real-time computer system for collecting and editing data, delivery of data to weather centrals and forecast facilities, and distribution of generated products around the globe. It is composed of several worldwide Automatic Digital Weather Switches interconnected via trunks. Each ADWS has numerous terminals. In many aspects it is similar to AUTODIN but its unique weather mission and special character sets require it to be a separate system.

Other military systems include Worldwide Military Command and Control System (WWMCCS).

For each of the above systems or networks the hardware that is installed usually reflects the latest in proven computer equipments and related devices. Yet in the area of communications
software, the programmer is usually limited to coding his programs in the assembly language of the particular manufacturer. An adequate, accepted higher level programming language or macro language for communications computer programming does not exist.

This lack has the effect of driving the already steeply increasing software costs still higher. It causes users to acquire identical hardware to augment existing networks in order to reduce software costs. This in turn tends to cause inflexibility in the replacement of computer hardware by not allowing the use of state-of-the-art hardware developments. It also dulls the effect of competitive bidding techniques to lower overall new system costs.

If new or different equipment is acquired, software costs are still large. Scarce computer programming personnel must be cross-trained in new assembly languages. Existing software modules must be reprogrammed and new software must be developed for the new hardware. And finally, leadtime for system development and implementation is greatly extended.

**Systems Engineering Approach**

The above three paragraphs outline a problem area that is being explored by the civilian as well as the military world of communications software system analysts and managers. Some work has been done in providing limited communications extensions to existing higher level languages such as COBOL. Others (Hess and Martin 1970, pp. 151-157) have attacked similar software problem
areas by using a subset of PL/I. Still others are designing a communications macro language. This thesis will explore this same problem - developing an optimum programming language for communications computers - by utilizing a systems engineering methodology.

This methodology (Wymore 1973) begins by identifying a client. In this case, it is U. S. Army Communications Command (USACC) at Fort Huachuca, Arizona. The client serves a multiple role by providing information for problem definition, deciding on specific alternatives when design choices must be made, and finally approving the rationale for and relative merits of the optimum solution or final system design.

After client identification, the system design process enters the problem definition phase. This phase starts by determining answers to the following questions:

1. What is the system supposed to do, basically?
2. How do we judge how well the system will do it?
3. What do we have at our disposal to build the system?
4. How do we judge how well we use our resources?
5. How do we resolve conflicts between performance and efficient resource utilization?
6. How will we determine desired versus actual system performance?

Answers to these questions will serve as a basis for a detailed and complete literary formulation of the client's problem.
This literary formulation, once coordinated with the client, is then used to produce a precise and comprehensive system theoretic formulation of the client's problem. From there, preliminary system design is begun and eventually a final system design is completed, implemented and tested. The following chapters delineate such a system engineering approach to the extent of problem definition and a start toward a preliminary system design.
CHAPTER 2

LITERARY PROBLEM FORMULATION

The literary formulation of a problem is in essence a preliminary distillation in written form of the client's desires concerning the definition of a particular problem. It normally follows initial discussions and meetings with the client. It usually contains a minimum of mathematical and engineering terminology, and covers such areas as need, need satisfaction criteria, available resources, resource utilization criteria, trade-off criteria and system testing.

For this particular client and problem the information was not only obtained through discussion, but also via examination of numerous pertinent documents provided by the client and a survey of the state-of-the-art from current literature in all aspects of the problem. These sources of information are the basis for this chapter and the next.

Statement of Need

The beginning of the formulation of a systems problem is a complete statement of the need as seen by the client. This statement should leave no room for ambiguous interpretation and the extent of the meaning of key words and phrases should be defined.
In this case, the statement of need is: There is a compelling need to develop an improved method of programming suitable for general use in a communications oriented environment.

The need is compelling because of constantly climbing software costs and the lack of flexibility in hardware procurement that was previously mentioned. In addition, the new roles and requirements for communications computer systems are rapidly increasing in such areas as reduction of manual interfaces, utilization of optical character reader (OCR) and cathode ray tube (CRT) equipment and expansion of the use of remote input/output devices. Correspondingly, this increase in requirements means that efficient, inexpensive new software must be designed and implemented.

The words, "method of programming", encompass the whole range of possible languages used to prepare computer programs for execution. It includes, but is not limited to, machine codes, assembly languages and macro language with their associated assemblers, and higher level languages with their associated compilers or interpreters. The words, "general use", mean use in the communications application area as well as interaction with the operating system and other systems control functions. The users are usually programmers but could involve anyone who plans, programs or implements a communications computer system. The phrase, "in a communications oriented environment", includes only programmable equipment, not hardwired computer terminals. It means that the
system could be a small stand-alone, communications-only computer terminal with minimal input/output; a large multiple-processor computer terminal with numerous input/output and remote devices as well as high speed interface with separate data processing facilities; or a large computer system connected to a communications network that has communications applications programs processing concurrently with other software.

**Statement of Need Satisfaction Criteria**

In the systems methodology, the need satisfaction criteria (Fig. 3) are means by which candidate systems may be compared as to how well they meet the needs of the client. The criteria are not associated with the physical or actual composition of the system, but deal strictly with the performance of the system in meeting the basic need of the client. The following are the criteria:

1. Average amount of time required for translation of the program requirements. This criterion measures two areas. The first is the average amount of time necessary for all users to translate programming requirements into the candidate programming language. This includes any refresher training in the peculiarities of the particular programming language as well as the actual coding process. The second is the average amount of time necessary for all source programs in the candidate programming language to go through the machine translation process. In the case where the programming language is a machine language, the second part would always be zero.
Need Satisfaction Criteria

1. Translation Time
   a. Programmer
   b. Machine
2. Object Program Core Size
3. Object Program Execution Time
4. Output Programs
   a. In Useable Form
   b. In Incomplete Form

Fig. 3. Summary of Need Satisfaction Criteria
These are both objective measures where the first part is in terms of actual man-hours used and the second is in terms of seconds of computer time used. The preferred system is one which uses less time in both areas of interest.

2. Average amount of core storage required for the object program. This criterion is one measure of the relative efficiency of the object programs that are generated by the candidate programming language. Object programs that require less core storage reduce certain hardware costs. This is particularly true when the size of computer memory is to be determined for a hardware procurement. This is an objective measure in terms of words. The preferred system is one which uses less memory.

3. Average amount of time required for execution of the object program. This criterion is another measure of the relative efficiency of the object programs that are generated by the candidate programming language. Object programs that are executed in less time can handle communications traffic in a quicker manner. Thus, speed of service for a communications computer could be improved. This is an objective measure in terms of seconds of computer time used. The preferred system is one which uses less time.

4. Percentage of users who contact the system and get a useable output. This criterion describes two areas. The first measures to what extent the users who contact the system with appropriate program requirements receive a workable object program.
If an object program is not output, the second part measures the availability and quality of diagnostics and other debugging aids such as traces and dumps. The first part is an objective measure in terms of a percentage of the input program requirements. The second part is essentially a subjective rating in terms of a percentage of the input requirements. The preferred system is one in which both parts are higher.

These criteria will suffice to compare candidate systems when one system is better in every aspect than the other from the need standpoint. But, if one system takes less man-hours to translate programming requirements into its programming language and more computer time to translate the source programs in its programming language to object code than another system, how are the two systems compared? Somehow, a more elaborate comparison system must be developed. It could include a weighting of the criteria in some manner so that a single numerical value is calculated for each system. This value would easily lead to a ranking of all candidate systems.

An algorithm to obtain such a single numerical value has not been defined herein, although some of the trade-off criteria do resolve certain conflicts. Further study and negotiation with the client may resolve the remaining potential conflicts.
Statement of Available Resources

Intrinsic to the formulation of any problem is the listing of resources which are available for use in constructing a solution to the problem. This list includes all present resources and all possible future resources.

For USACC the resources presently available are the expertise of certain personnel and the information available in the software library and certain files. These resources are principally from the Telecommunications Automation Division (TAD) of the U. S. Army Communications-Electronics Engineering Installation Agency (USACCEIA), a subordinate command of USACC. The expertise includes specific knowledge of communications computer functional requirements, of existing assembly language programs for all operational Army communications computers, and from initial investigations into the state-of-the-art in higher level languages for communications computers. Presently there is no active investigation of a higher level language for communications computers or any direct funding of such efforts.

Potential resources, therefore, become important if any extensive research, development or implementation is to be done. Outside expertise can be found in other commands of the Army, other military services such as the Air Force or the Navy, other governmental agencies such as ARPA, numerous commercial companies and many universities and colleges such as the University of Arizona.
Proprietary information known only to particular organizations as well as public information published in trade journals, scientific books, and so on is also available. In particular much information is available on many existing higher level languages and their compilers such as FORTRAN, COBOL with its communications extension, ALGOL, PL/I, JOVIAL and many others. Finally, these potential resources of expertise and existing information can only be effectively tapped through appropriate funding, that is, direct donation of such resources or the use of funds to pay for any effort to acquire such resources. Funds could be made available from the Army, from other military services, from other governmental agencies and from commercial companies.

Statement of Resource Utilization Criteria

These resources if broken down into their components would make a very long list. Also the combinations of these components to form a possible solution to the problem is extremely long. Therefore, it is necessary that some means of judging between the possible combinations be established. This judgement can be made by defining certain criteria to rank the various candidate systems as to how well resources are utilized (Fig. 4). The following are the criteria:

1. Degree to which communications functional areas are encompassed in the programming language. This criterion specifies the amount and relative importance of the communications functions
Resource Utilization Criteria

1. Capability
2. Independence
3. Legibility
   a. External Documentation
   b. Self-Documentation
4. Translator Core Size
5. Cost
   a. Initial
   b. Recurring
6. Planning & Building Time
7. Extensibility
8. Translator Correctness

Fig. 4. Summary of Resource Utilization Criteria
listed in Appendix A that should be integral to the programming language and can be used to implement software in a communications computer. It will enable the client to determine which programming language best includes his functional requirements. This is an objective measure in terms of number of functional areas available. The preferred system is one which has more functional areas available.

2. Degree of independence from a particular hardware configuration. This single criterion encompasses two aspects of independence. The first concerns how independent the programming language itself is from a particular computer or class of computers. The second involves how independent its translators, if any, are from a particular computer or class of computers. In the first case, if the programming language makes reference to hardware that is unique to a given computer such as sense lights or the use of a fixed word length capability, then that programming language is not independent. For the most part, the standardized languages such as COBOL would be considered independent. In the second case, the utilization of a compiler in two different classes of computers is unusual unless each class has similar hardware characteristics. Thus, judgement of compiler independence is based on how many different compilers are available to implement a standard programming language over many classes of computers. This criterion is an objective numerical rating. The preferred system is one which has
a higher value of independence since the client would then become less dependent upon particular manufacturers.

3. Degree of legibility of the programming language. This criterion is dependent upon two aspects. The first is the documentation that is external to the programming language itself such as programming manuals and a description of the translators. The second is the self-documentation inherent in the language statements and their order. The first and second aspects will enable the system designer to determine how well a programming language assists in training a user to program in that language. For instance, the more cumbersome the language constructs are, the more difficult it is to learn. Yet, the converse is true only to a certain point. If a programming language is too brief and concise, it also is not easily understood. In addition, the second aspect alone will aid the programmer by conveying as much information as possible about the way a particular program does its job. These are both measured subjectively with the preferred system maintaining a higher level of legibility.

4. Amount of core storage required to handle the translators of a candidate programming language. This criterion measures the total amount of computer memory required by all translators needed in a particular translation process of a programming language. This indicates to the client one aspect of the relative efficiency of the translators. This is an objective measure of the number of words of
core storage used. The preferred system is one which uses less core storage.

5. Amount a candidate programming language costs. This criterion can be broken into two parts - initial and recurring. Initial costs include the cost of planning, designing, building and testing. The programming language and its associated translators could be bought outright in whole or in part and/or it could be developed and implemented in whole or in part. Specific costs could include personnel costs such as salary, benefits and training; facilities costs such as office space and equipment, utilities and computer test-bed time; and information acquisition such as existing programs and their documentation. The second part is the recurring cost of operating and maintaining the programming language and its associated translators. Again, specific costs on perhaps an annual basis could include personnel and facilities costs necessary to produce the communications software for a new or modified hardware configuration. This is an objective measure in terms of dollars. The preferred system is one which costs less on both an initial and a recurring basis.

6. Amount of time to plan and implement a candidate programming language. This criterion specifies the total amount of time that would be expended to plan, design and implement a candidate
programming language. This is an objective measure in terms of man-hours. The preferred system is one which takes less time to implement.

7. Degree of extensibility of the programming language. This criterion measures the degree to which a candidate programming language can be modified or added to within the framework of the language itself, without having the compiler modified. It will enable the client to determine how easily a programming language could be modified to include new or altered functional requirements of the future communications environment. This is measured subjectively with the preferred system exhibiting a higher degree of extensibility.

8. Degree of correctness of the translators. This criterion measures how correctly the translators of a programming language conform to the semantic description of their job. That is, whether or not they translate in a proper manner. A so-called "software failure" is really a lack of correctness in the original program description. This is an objective measure in terms of number of failures. The preferred system is one which has fewer failures of the translators.

Again there exist conflicts between the various criteria from a resource utilization standpoint. If one candidate system has low initial costs and high recurring costs as compared to
another candidate system, which system is better? Even though the trade-off criteria helps somewhat, these conflicts also remain to be resolved with the client.

**Statement of Trade-Off Criteria**

For most "real world" problems there will be candidate systems which meet the client's needs and can be constructed from the available resources. Those candidate systems are known as feasible solutions to the problem. Since more than one feasible solution usually exists, a method of comparing feasible solutions must be devised. This is done by formulating trade-off criteria (Fig. 5).

These criteria are a combination of two or more criteria listed previously either in the need satisfaction statement or the resource utilization statement. Usually the criteria will evaluate systems on the basis of how well they meet the needs of the client versus how well they perform technically. The criteria can also be based on two or more technical criteria as well as two or more need criteria. In all cases the one criteria when invoked to its best rating tends to cause the other criteria to approach its worst rating. The intent of the trade-off criteria then is to state a manner for both criteria to be optimized and/or to provide a method of rating feasible solutions.

Theoretically, all combinations of the need satisfaction criteria and resource utilization criteria could be used to formulate
## Trade-Off Criteria

1. Cost-Effectiveness

2. Object Program Efficiency

3. User Efficiency

4. Translator Efficiency

---

**Fig. 5. Summary of Trade-Off Criteria**
trade-off criteria. In practice, however, certain combinations are more valuable than others. Therefore, the following trade-off criteria are considered the most useful:

1. **The cost-effectiveness of the programming language.** This criterion measures the total cost per program requirement to include a factor as to how well the program requirement was completed. The preferred system costs the least per successfully completed program requirement.

2. **The average efficiency of the object program.** This criterion measures the programming language capability in terms of the execution time as well as the object program size. Often, as execution time decreases, the object program size will increase. Thus, this criterion can resolve certain conflicts in the need criteria as well as serve as a trade-off criteria. The preferred system is more efficient during execution time when this criterion has a higher value.

3. **The average efficiency of the user.** This criterion measures the average amount of time spent translating a program requirement into a programming language divided into its capability, independence, legibility and extendibility. Thus, the relative efficiency of the user in accordance with the characteristics of the particular programming language is measured. The preferred system is more efficient during translation by the user when this criterion has a higher value.
4. The average efficiency of the translators. This criterion measures the average amount of time spent translating the program requirement written in the candidate programming language to the particular machine language times the amount of core needed for all translators divided into the capability of the system. It also contains a second measurement of the correctness of the translators. The preferred system is more efficient during computer translation when both parts of this criterion have higher values.

As before, conflicts between two candidate systems may not be wholly resolved. Further refinement to an ultimate means of ranking two candidate systems may be required.

**Statement of System Testing Requirements**

After a particular system design has been selected and the actual system has been implemented, there should be a test made of the resulting implementation. This is done in problem definition to insure that any system design requirements for testing are included in overall design, and that both the client and designer fully understand how the system test will be conducted.

The test is to see how the actual system performs versus the desired operating characteristics that molded the final system design. It also determines how well the best programming language compares to the present utilization of assembly language by the client.

The testing plan should have a procedure designed to measure all aspects of the programming language. This procedure could be
based on all of the criteria that have been previously identified. In turn this procedure would be used on the design model to produce a set of standard operating characteristics. It could also be used on a model of the existing system or the actual existing system itself, to determine minimum operating characteristics. Then, the procedure would be used on the actual implemented system. The results would be compared to the standard and minimum operating characteristics to determine if the actual system is acceptable.

In the case of a programming language the following procedure could be used. A gamut of programs which encompass the range and frequency of the desired communications functional areas will be written by programmers with an average experience level. The program would then be run on a computer with the required machine characteristics and input/output capability. Parameters that would be measured are:

1. Amount of time it takes the user to understand a programming language and translate program requirements into it.

2. Amount of time it takes a computer to translate a program requirement written in the programming language to machine language.

3. Amount of core storage needed by object programs.

4. Amount of execution time required by object programs.

5. Number of program requirements that output as object programs.
6. Quality of the diagnostics and other debugging aids.
7. Number of functional areas present and their usage rate.
8. Amount of independence from particular hardware.
9. Degree of legibility.
10. Amount of core storage needed by all translators.
11. Amount of initial and recurring system costs.
12. Amount of time needed to implement the system.
13. Degree that the programming language can extend its capabilities.
14. Number of failures caused by incorrect translators.
CHAPTER 3

SYSTEM THEORETIC PROBLEM FORMULATION

The steps in the formal methodology for the definition of a system design project (Wymore 1973, pp. 6/10-6/13) cover seven areas - project name, input/output specification, technology, input/output merit ordering, technology merit ordering, feasibility merit ordering and system test plan. The next seven sections of this chapter will explore these seven areas with respect to a programming language for communications computers. It should also be noted that the last six sections are derived from the corresponding six sections of the literary problem formulation in the previous chapter.

Project Name

The first step is to give the project a name. Since the client, USACC, has done some preliminary investigation on the problem and has already formulated a name, that name will be adopted as the client's choice for a project name. Henceforth, the project will be known as the MITCOL (Machine Independent Tele-Communications Oriented Language) project.
Input/Output Specification

In the title of this section is the inherent idea that the inputs and outputs for a system to solve the client's problem must be specified (Fig. 6). These inputs and outputs must also be described as to the manner in which they are organized in time as they arrive at and depart from the system. In this form they are known as input or output functions. Finally, the relationships between each input function and a set of output functions must be described. This relationship is called a matching. The following is a formal delineation of the steps of an input/output specification (Wymore 1973, pp. 2/10-2/25) for the MITCOL project.

1. Let MITCOLSPEC be the name of the input/output specification meeting the need of the client.

2. The inputs to MITCOLSPEC are either none, or a subset of the set of all possible users of the programming language. Users can include individual programmers as well as groups of people such as one consisting of programmers, systems analysts, managers and administrative support personnel. Each user will be characterized by two categories of attributes. The first is the user's program requirements. This category includes a correct description of software functional requirements, hardware constraints and limitations such as core size and input/output configuration of the machine that will eventually use the program as operating software, and the program identification. The second category is the user's expertise. This includes the user's
INPUT ATTRIBUTES

1. Program Requirements
2. Expertise

OUTPUT ATTRIBUTES

1. Program Requirements
2. Expertise
3. Program Status

Fig. 3. Input/Output Diagram
level of training in all aspects of the particular programming language, the user's documentation capability and procedures, a rating of the user's intellectual capability whether individual or collective and a rating of his experience.

3. The input functions of MITCOLSPEC are discrete. The users will arrive at the system singly or in bunches. For all values of time of an input function other than integer values, the input function is a null input. Therefore, any users that might arrive between integer values will be considered to have arrived at the next integer value. In this case, the integer values will be hours.

4. The outputs from MITCOLSPEC are either none, or a subset of the set of all possible users of the programming language. Each user will be characterized by the same two categories of attributes as defined in the inputs. In addition, there is a third category, program status. This category will specify whether the program was successfully completed in accordance with requirements or not. It would include status of successful program completion, listing of diagnostics and messages, and anywhere from none, to partial, to complete output of a program for communications computers in the programming language and/or object code.

5. The output functions of MITCOLSPEC are discrete. The users will depart the system in the same manner as they arrived.

6. For every input function of MITCOLSPEC there is matched a set of output functions such that for each user the category of
input attributes called program requirements is the same on output. Also each user who appears in an input function must appear in an output function and vice versa.

Technology

Before a system to meet the client's need can be designed, there must be a listing of all the possible components that are available for system construction. The possible components include components that would have to be built as well as existing components. It could include the use of de facto but unstated sub-assemblies or parts of a component. Finally, it could include multiple use of the same component.

The formal definition of a technology (Wymore 1973, pp. 3/2-3/11) encompasses a name for the technology and definitions of the possible components.

1. Let MITCOLTECH be the name of the technology from which will be constructed a programming language for communications computers.

2. The primary area of technology available to build a programming language is the various methods of programming (Gries 1971, pp. 2-3). The first of these methods is machine language programming. Machine language is the actual set of symbols (usually binary 1's and 0's) which the hardware can interpret for execution. Any program or series of instructions written in machine language can be executed directly without any intermediate translation and is often called the object program.
The next method is assembly language programming. Assembly language is usually just symbolic representation of machine language statements. Now, prior to actual execution of the program, the source program written in assembly language must usually be translated into machine language. The program that performs this translation process is called an assembler. Thus, in this method, an additional component - the assembler - is needed to provide an executable program.

Another method is macro language programming (Kent 1969, pp. 183-196). A macro facility in an assembly language is a way of extending that language. The user can introduce new statements into the language by defining how such a statement is to be translated into statements of the original assembly language. These are known as macro instruction or macro calls. Thus, the macro language program could consist of assembly language statements and macro instructions. Again, an assembler program is still required, but now it must also contain a means of macro expansion or a way of replacing a macro instruction with the appropriate assembly language statements.

Still another method is the broad category of higher level programming languages. This catchall term can cover such classes of languages as procedure-oriented languages, problem-oriented languages, problem-solving languages and application-oriented languages. For the purpose of this thesis a higher level language is one which has
the following characteristics. Each source program statement is normally translated into more than one machine instruction and its notation is somewhat closer to the specific problem being solved than machine language or assembly language. In addition, other characteristics of a higher level language are that usually knowledge of machine code is unnecessary and there is a certain amount of machine independence. Again, prior to execution of the user's program, some translation must be done. This can be done by a compiler or an interpreter. A compiler translates the higher level language into either assembly language or machine language. Then the program either goes through an assembler and is executed by the machine or it is executed directly. An interpreter usually translates the higher level language into an internal form. It then interprets or executes the internal form directly.

In each of the above methods of programming except for machine language, there are two components. One is the programming language (assembly or macro and higher level languages); and the other is a translation program (assemblers and compilers or interpreters). The techniques of building these components are important sub-technologies. In particular, the technologies of higher level language design and compiler or interpreter construction are of consequence.

The design of a higher level language is dependent upon several areas of technology (Sammett 1967, pp. 65-126). These
areas include the form of the language such as character set, basic elements (identifiers, operators, delimiters, etc.) and type of input form; the structure of the program such as types and characteristics of subunits (declarations, loops, functions, subroutines, procedures, comments, etc.); the data types and their computation such as type of data variables, types of arithmetic and rules of evaluation of expressions; executable statement types such as assignment statements, control statements, and interaction with the operating system; non-executable statement types such as storage allocation and compiler directives; and the structure of the language and compiler interaction such as self-extension of the language, debugging aids and error checking.

Closely related to the design of a higher level programming language is the construction of its compiler or interpreter (Gries 1971). The technologies here concern the construction of the parts of a compiler or interpreter. The first part is usually the scanner or lexical analyzer which scans the characters of the source program and builds the actual symbols such as identifiers, integers, and reserved words. The second part is the syntax and semantic analyzer which disassembles the source program, builds an internal form of the program and puts information into symbol tables and other tables. It also completes a syntax (sentence structure) check and a semantic (sentence meaning) check of the program. The next part will prepare for code generation by allocating storage plus optimizing and
manipulating the internal form. At this point an interpreter will actually execute the internal form of the source program. A compiler, however, generates the actual machine or assembly language that becomes the object program.

These previous paragraphs outline the technologies available to design new higher level programming languages and their compilers and interpreters, but there are also technologies available to adapt or extend existing languages and compilers or put them on new hardware configurations.

Higher level languages in themselves such as FORTRAN and COBOL can be relatively machine independent, but usually their interpreters or compilers are dependent upon a particular machine. Thus, for each new hardware configuration a new compiler or interpreter must be written for each higher level language that is required by the user. For widely-used, higher level languages most manufacturers usually have readily available a compiler whose development cost is prorated over many users. Whereas, for new or less widely used higher level languages a new compiler must be designed and implemented for each manufacturer and/or hardware configuration.

To combat this problem several areas of technology are available. Microprogramming involves a hardware capability in certain machines to emulate or have the ability to imitate various other machines. Thus, if a compiler or even an object program existed for another machine, it could be run on a machine with microprogramming.
capability that was emulating that other machine. Another area is the various types of translator writing systems (Gries 1971, pp. 436-446), which are programs that aid in writing compilers (often called a compiler-compiler), interpreters and assemblers. Still another area consists of the various techniques of compiler bootstrapping (Gries 1971, p. 458) or the method designed to bring a compiler into a desired state of capability by means of its own action.

These technologies are not a complete list, yet it illustrates the components and methods available to construct a programming language for communications computers.

Input/Output Merit Ordering

An input/output merit ordering is a ranking or ordering of systems as to how well they fulfill the needs of the client as expressed in the input/output specification (Wymore 1973, pp. 4/1-4/164). To reach this ranking of systems, four distinct steps are usually taken. The first step is to define a performance index or the summary of the behavior of a particular system in an individual experimental situation. The components of the performance index correspond to the criteria in the "Statement of Need Satisfaction Criteria". This performance index must now be calculated in accordance with a probability distribution for each component over the set of all possible system experiments. This calculation is known as computing expected values with respect to a probability distribution. Thus, the definition of a finite probability
distribution is the second step, and the third step is the list of expected values of each component of the performance index which becomes the figure of merit. Finally, the merit ordering of all candidate systems' figure of merit in accordance with ordering criteria completes the input/output merit ordering.

The input/output performance index will be defined in accordance with the following format. A name will be given to the index. A system Z and the output function z will be defined. Each component of the performance index will be summarized from the previous chapter, given a name, given a set of possible values, given a rule by which it is calculated with an example and given a rule which specifies preferred values.

1. Let IOINDEX be a performance index for a candidate system Z with respect to the output function z.

2. Let Z be any discrete system that is a member of the set of systems that satisfies the input/output specification MITCOLSPEC.

3. Let z be any output function for Z which was defined in MITCOLSPEC.

4. The components of IOINDEX are:

4.1 Average amount of time required for translation of the program requirements. This criterion measures two areas. The first is the average amount of time necessary for a user to translate programming requirements into the candidate programming language. The second is the average amount of time necessary for a source program in
the candidate programming language to go through a translation process. In the case where the programming language is a machine language the second part would always be zero.

4.1.1 Let TRANSTIME be the name of the first component.

4.1.2 TRANSTIME is a pair \((PRTIME, PLTIME)\) where both \(PRTIME\) and \(PLTIME\) are real numbers between 0 and \(\infty\).

4.1.3 TRANSTIME is calculated as follows:

\[
PRTIME = \frac{\sum (R-S)}{n}
\]

Where \(R\) is the time a program requirement has finished being translated into a candidate programming language by the user. \(S\) is the time a program requirement entered into the system. The summation is over all program requirements that entered the system; \(n\) is the total number of program requirements that entered the system during a systems experiment.

\[
PLTIME = \frac{\sum (U-V)}{n}
\]

Where \(U\) is the time a program requirement in the candidate programming language enters the first translator. \(V\) is the time the same program requirement was output from the last translator. The summation is over all program requirements that entered the first translator; \(n\) is defined above.
Example: \( n = 4 \) The elapsed times for PRTIME (R-S) are 500 hours, 320 hours, 50 hours, 10 hours, and for PLTIME (U-V) are 22 seconds, 15 seconds, 17 seconds, 2 seconds.

\[
PRTIME = \frac{500 + 320 + 50 + 10}{4} = 220 \text{ hours}
\]

\[
PLTIME = \frac{22 + 15 + 17 + 2}{4} = 14 \text{ seconds}
\]

4.1.4 A smaller value of TRAHSTIME is better than a large one. This is true for both parts (PRTIME, PLTIME) of the pair.

4.2 Average amount of core storage required for the object program. This criterion measures the relative efficiency as to core storage needed by the object programs that are generated by the candidate programming language.

4.2.1 Let RUNCORE be the name of the second component.

4.2.2 RUNCORE is a real number between 0 and \( \infty \).

4.2.3 RUNCORE is calculated as follows:

\[
RUNCORE = \frac{\sum A}{m}
\]

Where \( A \) is the amount of core storage occupied by each object program that was produced by the candidate programming language during a system experiment. The summation is over all
program requirements that entered the system and left as object programs; \( m \) is the total number of program requirements that entered the system and left as object programs during a systems experiment.

Example: If \( m = 4 \) and core storage is 500 words, 750 words, 1750 words, 3000 words, then

\[
\text{RUNCORE} = \frac{500 + 750 + 1750 + 3000}{4} = 1500 \text{ words}
\]

4.2.4 A smaller value of RUNCORE is better than a large one.

4.3 Average amount of time required for execution of the object program. This criterion measures the relative efficiency as to the time needed for execution by the object programs that are generated by the programming language.

4.3.1 Let \( \text{RUNTIME} \) be the name of the third component.

4.3.2 \( \text{RUNTIME} \) is a real number between 0 and \( \infty \).

4.3.3 \( \text{RUNTIME} \) is calculated as follows:

\[
\text{RUNTIME} = \frac{\sum B}{m}
\]

Where \( B \) is the amount of execution time each object program took that was produced by the candidate programming language during a system experiment. The summation and \( m \) are the same as defined in RUNCORE.
Example: If \( m = 4 \) and execution times are 1.5 seconds, .5 seconds, 2.65 seconds, .15 seconds, then

\[
\text{RUNCORE} = \frac{1.5 + .5 + 2.65 + .15}{4} = 1.20 \text{ seconds}
\]

4.3.4 A smaller value of RUNTIME is better than a large one.

4.4 Percentage of users who contact the system and get a useable output. This criterion measures to what extent the users who contact the system with appropriate program requirements receive a workable object program and when an object program is not output, the availability of diagnostics and other debugging aids such as traces and dumps is measured.

4.4.1 Let OUTPROG be the name of the fourth component.

4.4.2 OUTPROG is a pair \((\text{PROGOOD}, \text{PROGBAD})\) where PROGOOD and PROGBAD are real numbers between 0 and 100.

4.4.3 OUTPROG is calculated as follows:

\[
\text{PROGOOD} = \frac{m}{n} \times 100
\]

Where \( m \) is the same as defined in RUNCORE and \( n \) is the same as defined in TRANSTIME.

\[
\text{PROGBAD} = \frac{\Sigma C}{n} \times 10
\]

Where \( C \) is a subjective rating of the quality of diagnostic printouts and other debugging aids such as dumps. The values for \( C \)
would be from 1 to 10 with 1 reflecting a program with no diagnostic
printouts. Succeeding numbers would each reflect better diagnostics
and debugging aids until 10 which would be the best and most complete
diagnostics possible. The summation is over all program requirements
that departed the system in a form other than an object program
during the system experiment; \( n \) is the same as defined in \( \text{TRANSTIME} \).

Example: \( n = 5 \), \( m = 2 \) and the C ratings are 4, 3 and 3.

\[
\text{PROGOOD} = \frac{2}{5} \times 100 = 40
\]

\[
\text{PROGBAD} = \frac{4+3+3}{5} \times 10 = 20
\]

and \( \text{OUTPROG} = (40, 20) \)

4.4.4 A larger value of \( \text{OUTPROG} \) is better than a small one.
This is true for both parts (\( \text{PROGOOD} \), \( \text{PROGBAD} \)) of the pair.

4.5 Specifically, the performance index, \( \text{IOINDEX} \), is
\( (\text{TRANSTIME}, \text{RUNCORE}, \text{RUNTIME}, \text{OUTPROG}) \).

\( \text{IOINDEX} \) is a representation of the input/output behavior
of a candidate system in only one experimental situation. This rep­
resentation must be extended to cover the set of all possible system
experiments. To accomplish such a comprehensive representation, all
possible experiments must be evaluated and a probability distribution
for them be constructed. This is used to calculate the expected values of each component of the performance index or its figure of merit called IOMERIT.

The construction of a valid probability distribution and subsequent calculation of expected values can require much time and effort to include use of computers and the aid of various statisticians, programmers and so on. Perhaps, an alternative approach using experience and knowledge could save time and effort and yet be almost as valid.

A scenario approach to acquiring expected values is not new or unique. A system experiment could be designed to encompass the range and frequency of all aspects of the input specification. The program requirements could reflect the original requirements of currently operational communications computers in various typical configurations. This utilization of program requirements is one approach used in benchmark tests (Drummond 1973, pp. 202-206). The user's expertise could be shown by several programmers and/or groups which reflect the spectrum of experience, intellectual capability, documentation practices and training of all. The timeframe would be from the time of input of all programs to the time of output of the last program requirement in some form. By this quasi-empirical method, the probability distribution will have a value of 1 for this particular experiment and 0 elsewhere. Therefore, the values measured for the performance indices would become the expected values.
A definition (Wymore 1973, p. 441) of a merit ordering over each figure of merit of candidate systems follows five steps. They are to give the merit ordering a name, define the figure of merit, define the ordering and define the rule of ordering two candidate systems. The MITCOL input/output merit ordering is as follows:

1. Let IOMERITORDER be a merit ordering defined in terms of the figure of merit IOMERIT and the ordering of merit IORDER.

2. The figure of merit IOMERIT is defined by one of a set of lists each of which are the expected values of (TRANSTIME, RUNCORE, RUNTIME, OUTPROG).

3. The ordering is defined as follows:
   3.1 Let IORDER be the ordering.
   3.2 The set over which IORDER is defined is the set specified in 2 above.
   3.3 Let (TRANSTIME₁, RUNCORE₁, RUNTIME₁, OUTPROG₁) and (TRANSTIME₂, RUNCORE₂, RUNTIME₂, OUTPROG₂) be two lists assigned by the figure of merit IOMERIT to two candidate systems. To this pair of lists IORDER assigns yes or that IOMERIT₂ is at least as good as IOMERIT₁ only if TRANSTIME₁ ≥ TRANSTIME₂, RUNCORE₁ ≥ RUNCORE₂, RUNTIME₁ ≥ RUNTIME₂, and OUTPROG₁ ≤ OUTPROG₂.

3.4 According to the rule given in step 3.3 above, IORDER assigns yes to any pair of the form (IOMERIT₁₁, IOMERIT₁) for any list in the set of lists.
3.5 If IORDER assigns yes to the pair of lists \((\text{IOMERIT}_1, \text{IOMERIT}_2)\) and to the pair of lists \((\text{IOMERIT}_2, \text{IOMERIT}_3)\) then it assigns yes to the pair of lists \((\text{IOMERIT}_1, \text{IOMERIT}_3)\).

4. If \(Z_1\) and \(Z_2\) are candidate systems and IORDER assigns yes to the pair of lists \((\text{IOMERIT}_1, \text{IOMERIT}_2)\) of these systems, then IOMERITORDER assigns yes to the pair of systems \((Z_1, Z_2)\).

In the above steps the ranking of various candidate systems is achieved based on which is totally better than the other. There is no mention of how a system is ordered if one component is better in one system, say \(\text{TRANSTIME}_1 < \text{TRANSTIME}_2\), and another component is better in the other system, say \(\text{RUNCORE}_1 > \text{RUNCORE}_2\). In these cases, a merit ordering of candidate systems is undetermined and usually will only be resolved by further refinement of the figure of merit, IOMERIT, or trade-offs in the feasibility merit ordering or both.

**Technology Merit Ordering**

A technology merit ordering (Wymore 1973, pp. 4/165-4/211) is a ranking or ordering of systems as to how well the resources are utilized that are available from the stated technology. This ranking is achieved in a manner similar to the input/output merit ordering. A performance index is defined whose components are based on the "Statement of Resource Utilization Criteria" of the previous chapter. If possible, a probability distribution for each component should be described. However, if the component is defined directly
from a general system description, a probability distribution is not necessary. Then, in turn, the figure of merit and a merit ordering of the various candidate systems should be defined.

The following is a formal listing of the technology performance index:

1. Let TECHINDEX be a performance index.
2. Let Z be any discrete system that is a member of the set of systems determined by the technology MITCOLTECH.
3. Let \( z \) be any output function for Z whose values are in the set defined in step 4.
4. Each element of the set from which the performance index, TECHINDEX, assigns a symbol to each experiment on Z is of the form of a vector with components as follows:

4.1 Degree to which communications functional areas are encompassed in the programming language. This criterion specifies the amount and relative importance of the communications functions listed in Appendix A that should be integral to the programming language. The relative importance of each functional area could be based on a strictly subjective rating by the client or could be calculated on their actual appearance in a series of typical program requirements.

4.1.1 Let CAPABILITY be the name of the first component.
4.1.2 CAPABILITY is an integer number between 0 and \( \infty \).
4.1.3 CAPABILITY is calculated as follows:

\[ \text{CAPABILITY} = \sum F(\sum W) \]

Where \( F \) is a value of 1 if the functional area is present in the programming language and 0 if not. \( W \) is a value of 1 if the functional area is present in a program requirement and 0 if not. The first summation is over all the functional areas that have been defined. The second summation is over all the program requirements that entered the system during the systems experiment.

Example: There are a total of five functional areas required in the programming language. In this case there are three programs to be translated. In this programming language all functional areas are present except for the fifth one. In the program requirements, program one has all functional areas present, program two has all except the fifth one present and program three has all except the second and third functional areas present.

\[ \text{CAPABILITY} = 1(1+1+1)+1(1+0+1)+1(1+0+1)+1(1+1+1)+0(1+1+0) = 10 \]

4.1.4 A larger value of CAPABILITY is better than a small one.

4.2 Degree of independence from a particular hardware configuration. This criterion has two aspects. The first concerns
how independent the programming language itself is from a particular computer or class of computers. The second is how independent its translator, if any, is from a particular class of computers.

4.2.1 Let INDEPENDENCE be the name of the second component.

4.2.2 INDEPENDENCE is an integer number between 0 and ∞.

4.2.3 INDEPENDENCE is calculated as follows:

\[ \text{INDEPENDENCE} = D(IL) \]

Where \( D \) is a value of 1 if the candidate programming language has nothing that makes it alone dependent on any class of computers and 0 if it is dependent. \( L \) is the number of distinct compilers available on a class of computers. The summation is over all classes of computers.

Example: The programming language is not dependent on a computer and it has two compiler implementations on one class of computers and one each on five other classes of computers.

\[ \text{INDEPENDENCE} = 1 \times (2 + 1 + 1 + 1 + 1) = 7 \]

4.2.4 A larger value of INDEPENDENCE is better than a small one.

4.3 Degree of legibility of the programming language. This criterion is dependent upon two aspects. The first is the documentation that is external to the programming language itself such as
programming manuals and description of a translator. The second is the self-documentation inherent in the language statements and their order.

4.3.1 Let LEGIBILITY be the name of the third component.

4.3.2 LEGIBILITY is a pair \((DOC, SELFDOC)\) where DOC and SELFDOC are integer numbers between 0 and 3.

4.3.3 LEGIBILITY is calculated as follows:

\[
DOC = \begin{cases} 
0 & \text{if no external documentation exists} \\
1 & \text{if external documentation is incomplete and/or cumbersome to understand} \\
2 & \text{if external documentation is complete and can be understood with average ease} \\
3 & \text{if external documentation is complete and is easily understood} 
\end{cases}
\]

\[
SELFDOC = \begin{cases} 
0 & \text{if language constructs and their order convey no information} \\
1 & \text{if language constructs and their order convey minimal information} \\
2 & \text{if language constructs and their order convey an acceptable amount of information} \\
3 & \text{if language constructs and their order convey maximum information} 
\end{cases}
\]
These assessments would have to be made on the basis of subjective evaluation by computer science and, perhaps, linguistics experts.

Example: A programming language is evaluated and found to be average in all aspects of documentation and self-documentation.

\[ \text{LEGIBILITY} = (2,2) \]

4.3.4 A larger value of LEGIBILITY is better than a small one. This is true for both parts (DOC, SELFDOC) of the pair.

4.4 Amount of core storage required to handle the translators of a candidate programming language. This criterion measures the relative efficiency as to core storage required by all translators needed in a particular implementation of a programming language.

4.4.1 Let TRANSCORE be the name of the fourth component.

4.4.2 TRANSCORE is a real number between 0 and \( \infty \).

4.4.3 TRANSCORE is calculated as follows:

\[ \text{TRANSCORE} = \sum E \]

Where \( E \) is the amount of core needed for a translator. The summation is over all the translators necessary in a particular implementation of a programming language. TRANSCORE is zero if no translator is required.
Example: A particular programming language requires 500 words for an assembler and 2500 words for a compiler.

\[ \text{TRANSCORE} = 500 + 2500 = 3000 \]

4.4 A smaller value of TRANSCORE is better than a large one.

4.5 Amount a candidate programming language costs. This criterion compares the various candidate programming languages as to how much each will cost. It has two parts. The first is the total costs of planning, designing, building and testing a programming language and its associated translators. The second is the recurring cost of operating and maintaining the programming language and its associated translators.

4.5.1 Let COST be the name of the fifth component.

4.5.2 COST is a pair \((\text{ONECOST, RECURCOST})\) where \(\text{ONECOST}\) and \(\text{RECURCOST}\) are both real numbers between 0 and \(\infty\).

4.5.3 COST is calculated as follows:

\[ \text{ONECOST} = \sum G \]

Where \(G\) is a specific component of the cost to plan, design, build and test a programming language. The summation is over all these specific cost components.
RECURCOST = ΣH

Where H is a specific component of a periodic cost to operate and maintain a programming language once it is implemented. The summation is over all these specific cost components.

Example: A particular programming language will cost $10,000 to plan, $100,000 to design, $100,000 to implement, $5,000 to test, $20,000 a year to maintain and $100,000 a year to operate.

ONECOST = $10,000 + $100,000 + $100,000 + $5,000 = $215,000

RECURCOST = $20,000 + $100,000 = $120,000/year

4.5.4 A smaller value of COST is better than a large one. This is true for both parts (ONECOST, RECURCOST) of the pair.

4.6 Amount of time to plan and implement a candidate programming language. This criterion specifies the amount of time necessary to plan, design and implement a candidate programming language.

4.6.1 Let BUILDTIME be the name of the sixth component.

4.6.2 BUILDTIME is a real number between 0 and ∞.

4.6.3 BUILDTIME is calculated as follows:

BUILDTIME = ΣM
Where M is a specific component of the total time necessary to plan, design, build and test a programming language. The summation is over all these specific components of time.

Example: A particular programming language will take 4,000 hours to plan, 4,000 hours to design, 15,000 hours to build and 1,000 hours to test.

\[
\text{BUILDTIME} = 4,000 + 4,000 + 15,000 + 1,000 = 24,000 \text{ hours}
\]

4.6.4 A smaller value of BUILDTIME is better than a large one.

4.7 Degree of extensibility of the programming language. This criterion measures the degree to which a candidate programming language can be extended within the framework of the language itself, without having the compiler modified.

4.7.1 Let EXTEND be the name of the seventh component.

4.7.2 EXTEND is an integer number between 0 and 3.

4.7.3 EXTEND is calculated as follows:

\[
\text{EXTEND} = \begin{cases} 
0 & \text{ - no capability for extension} \\
1 & \text{ - a minimal capability for extension} \\
2 & \text{ - an acceptable capability for extension} \\
3 & \text{ - an excellent capability for extension} 
\end{cases}
\]
These subjective assessments would be made by computer software experts based on such language characteristics as ability to use macros or ability to add a language construct to a programming language with only an addition to the translator.

Example: A programming language is evaluated and found to have a macro capability for self-extension.

\[
\text{EXTEND} = 2
\]

4.7.4 A larger value of \text{EXTEND} is better than a small one.

4.8 Degree of correctness of the translators. This criterion measures how correctly the translators of a programming language conform to the semantic description of their jobs.

4.8.1 Let \text{TRANSFAIL} be the name of the eighth component.

4.8.2 \text{TRANSFAIL} is an integer number between 0 and \(\infty\).

4.8.3 \text{TRANSFAIL} is calculated as follows:

\[
\text{TRANSFAIL} = \Sigma P
\]

Where \(P\) is the number of failures of the translator itself, not the program being translated. The summation is over all the translators necessary in a particular implementation of a programming language. \text{TRANSFAIL} is zero if no translator is required.
Example: A programming language has a compiler and an assembler. Two compiler failures are recorded and no assembler failures.

\[ \text{TRANSFAIL} = 2 + 0 = 2 \]

4.8.4 A smaller value of TRANSFAIL is better than a large one.

4.9 Specifically, the performance index, TECHINDEX, is

\[ (\text{CAPABILITY}, \text{INDEPENDENCE}, \text{LEGIBILITY}, \text{TRANSSCORE}, \text{COST}, \text{BUILDTIME}, \text{EXTEND}, \text{TRANSFAIL}) \].

TECHINDEX like IOINDEX is a representation of the technology of a candidate programming language in only one system experiment. Unlike IOINDEX, however, certain components need not be evaluated over the set of all possible system experiments. Once a value for such components as INDEPENDENCE, LEGIBILITY, TRANSSCORE, the ONECOST portion of COST, BUILDTIME, and EXTEND is chosen, these values would be the same for any system experiment. Nevertheless, to calculate the expected value of CAPABILITY, the RECURCOST portion of COST, and TRANSFAIL, either a probability distribution must be formulated or an optimum system experiment (scenario) could be designed. Thus, these values would then become the component values of the technology figure of merit TECHMERIT.

As was done in the input/output merit ordering, the MITCOL technology merit ordering is as follows:
1. Let \textsc{tecmath\text{-}ordering} be a merit ordering defined in terms of the figure of merit \textsc{tecmath} and the ordering of merit \textsc{techorder}.

2. The figure of merit \textsc{tecmath} is defined by one of a set of lists each of which is an expected value of \((\text{capability}, \text{independence}, \text{legibility}, \text{transscore}, \text{cost}, \text{buildtime}, \text{extend}, \text{transfail})\).

3. The ordering is defined as follows:

3.1 Let \textsc{techorder} be the ordering.

3.2 The set over which \textsc{techorder} is defined is the set specified in 2 above.

3.3 Let \((\text{capability}_1, \text{independence}_1, \text{legibility}_1, \text{transcore}_1, \text{cost}_1, \text{buildtime}_1, \text{extend}_1, \text{transfail}_1)\) and \((\text{capability}_2, \text{independence}_2, \text{legibility}_2, \text{transcore}_2, \text{cost}_2, \text{buildtime}_2, \text{extend}_2, \text{transfail}_2)\) be two lists assigned by the figure of merit \textsc{tecmath} to two candidate programming languages. To this pair of lists \textsc{techorder} assigns yes or that \textsc{tecmath}_2 is at least as good as \textsc{tecmath}_1 only if \(\text{capability}_1 \leq \text{capability}_2, \text{independence}_1 \leq \text{independence}_2, \text{legibility}_1 \leq \text{legibility}_2, \text{transcore}_1 \geq \text{transcore}_2, \text{cost}_1 \geq \text{cost}_2, \text{buildtime}_1 \geq \text{buildtime}_2, \text{extend}_1 \leq \text{extend}_2, \text{transfail}_1 \geq \text{transfail}_2\).

3.4 According to the rule given in 3.3 above, \textsc{techorder} assigns yes to any pair of the form \((\text{tecmath}_1, \text{tecmath}_1)\) for any list in the set of lists.
3.5 If TECHORDER assigns yes to the pair of lists
(TECHMERIT_1, TECHMERIT_2) and to the pair of lists (TECHMERIT_2,
TECHMERIT_3), then it assigns yes to the pair of lists (TECHMERIT_1,
TECHMERIT_3).

4. If Z_1 and Z_2 are candidate programming languages and
TECHORDER assigns yes to the pair of lists (TECHMERIT_1, TECHMERIT_2)
of these systems, then it assigns yes to the pair of systems
(Z_1, Z_2).

Again, the methods of ranking systems based on TECHORDER is
undetermined when some components of one system are better than
those same components in the other system and the rest of the
components of the first system are worse than the rest of the com­
ponents of the second system. Here again, further refinement of
the figure of merit, TECHMERIT, or trade-offs in the feasibility
merit ordering should resolve most of these conflicts.

**Feasibility Merit Ordering**

A feasibility merit ordering (Wymore 1973, pp. 4/213-4/290)
is a ranking or ordering of systems which satisfy the input/output
specification and are implementable in the technology. To reach
this final ranking of candidate programming languages the following
seven steps define a feasibility or trade-off merit ordering. They
are to give it a name, define the input/output specification,
define the technology, define the input/output and technology merit
orderings, define the merit ordering of feasible systems, and prove that the merit ordering of feasible systems is consistent with the input/output and technology merit orderings.

1. Let MITCOLMERITORDER be a merit ordering of all feasible programming languages.

2. The input/output specification, MITCOLSPEC, is defined beginning on page 32.

3. The technology, MITCOLTECH, is defined beginning on page 35.

4. The input/output merit ordering IOMERITORDER, is defined beginning on page 40.

5. The technology merit ordering, TECHMERITORDER, is defined beginning on page 49.

6. The ordering is defined as follows:

6.1 Let MITCOLORDER be the ordering.

6.2 The ordering MITCOLORDER is defined over the set of systems that belong to both MITCOLSPEC and MITCOLTECH.

6.3 The cost-effectiveness of the programming language.

6.3.1 Let COST/REQUIREMENT be the name of the first component.

6.3.2 COST/REQUIREMENT is a real number between 0 and ∞.

6.3.3 COST/REQUIREMENT is calculated as follows:
\[
\text{COST/REQUIREMENT} = \frac{\text{ONECOST} + \text{RECURCOST}}{\text{PROGOOD} + \text{PROGBAD}}
\]

Where ONECOST and RECURCOST are defined under COST in the technology merit ordering, and PROGOOD and PROGBAD are defined under OUTPROG in the input/output merit ordering.

Example: Given values calculated in previous examples.

\[
\text{COST/REQUIREMENT} = \frac{215,000 + 120,000}{40 + 20} = 3,916.66
\]

6.3.4 A smaller value of COST/REQUIREMENT is better than a large one.

6.4 The average efficiency of the object program.

6.4.1 Let RUNEFFICIENCY be the name of the second component.

6.4.2 RUNEFFICIENCY is a real number between 0 and \(\infty\).

6.4.3 RUNEFFICIENCY is calculated as follows:

\[
\text{RUNEFFICIENCY} = \frac{\text{CAPABILITY}}{\text{RUNTIME} \times \text{RUNCORE}}
\]

Where RUNTIME and RUNCORE are defined in the input/output merit ordering, and CAPABILITY is defined in the technology merit ordering.

Example: Given values calculated in previous examples.
RUNEFFICIENCY = \frac{10}{1.2 \times 1500} = .00555

6.4.4 A larger value of RUNEFFICIENCY is better than a small one.

6.5 The average efficiency of the user.

6.5.1 Let USEREFFICIENCY be the name of the third component.

6.5.2 USEREFFICIENCY is a real number between 0 and \infty.

6.5.3 USEREFFICIENCY is calculated as follows:

\[
\text{USEREFFICIENCY} = \frac{\text{CAPABILITY} + \text{INDEPENDENCE} + \text{DOC} + \text{SELFDOC} + \text{EXTEND}}{\text{PRTIME}}
\]

Where PRTIME is defined in TRANSTIME of the input/output merit ordering. DOC and SELFDOC are defined in LEGIBILITY which with CAPABILITY, INDEPENDENCE and EXTEND is defined in the technology merit ordering. Note that these latter components could be weighted to reflect the client's opinion of their relative importance.

Example: Given values calculated in previous examples.

\[
\text{USEREFFICIENCY} = \frac{10+7+2+2+2}{220} = .104
\]

6.5.4 A larger value of USEREFFICIENCY is better than a small one.
6.6 The average efficiency of the translators.

6.6.1 Let TRANSEFFICIENCY be the name of the fourth component.

6.6.2 TRANSEFFICIENCY is a real number between 0 and ∞.

6.6.3 TRANSEFFICIENCY is calculated as follows:

\[
\text{TRANSEFFICIENCY} = \left( \frac{\text{CAPABILITY}}{\text{PLTIME} \times \text{TRANSCORE}}, \frac{1}{\text{TRANSFAIL}} \right)
\]

Where PLTIME is defined in TRANSTIME of the input/output merit ordering. TRANSCORE, CAPABILITY and TRANSFAIL are defined in the technology merit ordering.

Example: Given values calculated in previous examples.

\[
\text{TRANSEFFICIENCY} = \left( \frac{10}{14 \times 3000}, \frac{1}{2} \right) = (.0023, .5)
\]

6.6.4 A larger value of TRANSEFFICIENCY for both parts of the couple is better than a small one.

6.7 Specifically, the feasibility figure of merit, MITCOLMERIT, is \((\text{COST/REQUIREMENT}, \text{RUNEFFICIENCY}, \text{USEREFFICIENCY}, \text{TRANSEFFICIENCY}, \text{BUILDTIME})\).

6.8 Let \((\text{COST/REQUIREMENT}_1, \text{RUNEFFICIENCY}_1, \text{USEREFFICIENCY}_1, \text{TRANSEFFICIENCY}_1, \text{BUILDTIME}_1)\) and \((\text{COST/EFFICIENCY}_2, \text{RUNEFFICIENCY}_2, \text{USEREFFICIENCY}_2, \text{TRANSEFFICIENCY}_2, \text{BUILDTIME}_2)\) be two lists assigned by the figure of merit, MITCOLMERIT, to two candidate programming languages. To this pair of lists MITCOLOORDER assigns yes or that
MITCOLMERIT_2 is at least as good as MITCOLMERIT_1 only if

\[ \text{COST/REQUIREMENT}_1 \geq \text{COST/REQUIREMENT}_2, \text{RUNEFFECTIVENESS}_1 \leq \text{RUNEFFECTIVENESS}_2, \text{USEREFFECTIVENESS}_1 \leq \text{USEREFFECTIVENESS}_2, \text{TRANSEFFECTIVENESS}_1 \leq \text{TRANSEFFECTIVENESS}_2, \text{and BUILDTIME}_1 \geq \text{BUILDTIME}_2. \]

6.9 According to the rule given in 6.8 above, MITCOLORDER assigns yes to any pair of the form (MITCOLMERIT_1, MITCOLMERIT_1) for any list in the set of lists.

6.10 If MITCOLORDER assigns yes to the pair of lists (MITCOLMERIT_1, MITCOLMERIT_2) and to the pair of lists (MITCOLMERIT_2, MITCOLMERIT_3), then it assigns yes to the pair of lists (MITCOLMERIT_1, MITCOLMERIT_3).

7. If Z_1 and Z_2 are candidate programming languages and MITCOLORDER assigns yes to the pair of lists of these systems, then it assigns yes to the pair of systems (Z_1, Z_2).

This is consistent with IORDER and TECHORDER because when appropriate values assign yes to the pair (Z_1, Z_2) for each of these orderings, the same values assign yes to the same pair for the feasibility ordering MITCOLORDER.

This feasibility merit ordering should resolve most conflicts between the relative "goodness" of any two candidate programming languages. Nevertheless, since there is not yet just one value to use to compare two systems, there is still a possibility for an undetermined ranking for certain candidate systems. The ranking of these indeterminate systems can only be accomplished by further refinement of the feasibility merit ordering MITCOLMERITORDER.
Procedure For System Testing

The design of a detailed procedure for testing (Wymore 1973, pp. 5/1-5/68) the optimum programming language that has been planned, designed and built in accordance with previous sections of this chapter is the last subject area in problem definition. This design process covers three areas - a definition of the test decision framework, a means of determining whether the actually constructed system $Z_{real}$ is faithful to the model that resulted from the planning and designing process, and a means of deciding whether or not $Z_{real}$ is acceptable.

Overall it will be known as the system test plan and will encompass the following areas: Give the system test plan a name; define the input/output specification, technology, and the feasibility merit ordering; define a system test decision framework consistent with feasibility merit ordering; and define a system test. This methodology for the candidate programming language is as follows:

1. Let MITCOLTESTPLAN be a system test plan.
2. The input/output specification, MITCOLSPEC, is defined beginning on page 32.
3. The technology, MITCOLTECH, is defined beginning on page 35.
4. The feasibility merit ordering, MITCOLMERITORORDER, is defined beginning on page 61.
At this point, the methodology to define a test decision framework is to give it a name, define the input/output specification, technology and the feasibility merit ordering and define the acceptable values to be measured consistent with the feasibility merit ordering.

5. The system test decision framework is defined as follows:

5.1 Let MITCOLTESTAREAS be the test decision framework.

5.2 The input/output specification, technology and feasibility merit ordering are the same as those defined in 2, 3 and 4 above.

5.3 The values acquired when the components of the feasibility merit ordering MITCOLMERITORDER are measured by a system experiment on the design model Z' will serve as the test decision framework.

At this point the methodology to define the system test is to give it a name, define the input/output specification, technology, and test decision framework, define the tests on $Z_{real}$ to determine what values it has consistent with the feasibility merit ordering, determine if the system theoretic model $Z'$ is an adequate model of $Z_{real}$, and define a procedure for deciding whether $Z_{real}$ is acceptable.

6. The system test is defined as follows:
6.1 Let MITCOLTEST be the system test for those candidate programming languages in the feasibility set of systems with respect to the test decision framework MITCOLTESTAREAS.

6.2 The input/output specification, technology and system test decision framework are the same as those defined in 2, 3 and 5 above.

6.3 The system experiment as described on page 47 can be run on $\mathbb{Z}_{\text{real}}$. The components and their calculation are specified in the feasibility merit ordering MITCOLMERITORDER. The specific values and the method of measuring them are as follows:

6.3.1 COST/REQUIREMENT consists of measuring values for ONECOST, RECURCOST, PROGOOD, and PROGBAD.

6.3.1.1 The value for ONECOST is in dollars. It will be measured by recording all monies expended to design, build and test the best programming language. In this case some test costs would have to be estimated.

6.3.1.2 The value for RECURCOST is in dollars. It will be measured by recording all monies expended to operate and maintain the best programming language during the period of the system test.

6.3.1.3 The value for PROGOOD is percent of program requirements that are output as an acceptable object program. It will be measured by recording total number of input program requirements and total number of acceptable object programs.
6.3.1.4 The value for PROGBAD is percent of program requirements that are output in an incomplete form. It will be measured by recording total number of input program requirements and recording a subjective evaluation by the test team on the "goodness" of the diagnostics and other debugging aids for each incomplete output.

6.3.2 RUNEFFICIENCY consists of measuring values for RUNTIME, RUNCORE, and CAPABILITY.

6.3.2.1 The value for RUNTIME is in seconds. It will be measured by placing each object program that is output from the programming language on its computer configuration and running it through a specified cycle. The object program would contain, in itself, instructions which would count and output on a printer or other device the amount of time it took for program execution.

6.3.2.2 The value for RUNCORE is in words of computer memory. It will be measured in the same manner as RUNTIME except that separate instructions would count the amount of memory used and output this information on a printer or other device.

6.3.2.3 The value for CAPABILITY is the number and importance of functional areas. It will be measured by evaluating and recording how many times each functional area appears in the program requirements of the system test and whether the functional area is in the best programming language or not.
6.3.3 USEKEFFICENCY consists of measuring values for PRTIME, CAPABILITY, DOC, SELFDOC and EXTEND.

6.3.3.1 The value for PRTIME is in hours. It will be measured by counting and recording the actual man-hours expended to translate each programming requirement into the programming language.

6.3.3.2 CAPABILITY is specified in 6.3.2.3 above.

6.3.3.3 The value for DOC is a subjective number. It will be measured by a subjective evaluation of the test team of the documentation that is external to the program itself.

6.3.3.4 The value for SELFDOC is a subjective number. It will be measured by a subjective evaluation of the test team of the self documentation that is inherent to the language statements and their order.

6.3.3.5 The value for EXTEND is a subjective number. It will be measured by a subjective evaluation of the test team of the extensibility of the programming language.

6.3.4 TRANSEFFICIENCY consists of measuring values for PLTIME, TRANSCORE, CAPABILITY, and TRANSFAIL.

6.3.4.1 The value for PLTIME is in seconds. It will be measured by having each translator contain, in itself, instructions which would count and output the amount of time it took for the programming requirement to go through the machine translation process.
6.3.4.2 The value for TRANSSCORE is in words of computer memory. It will be measured in the same manner as PLTIME except that separate instructions would count the amount of memory and output that information.

6.3.4.3 CAPABILITY is specified in 6.3.2.3 above.

6.3.4.4 The value for TRANSFAIL is in failures of each translator. It will be measured by analyzing all software failures and recording those that result from an incorrect translator.

6.3.5 The value for BUILDTIME is in hours. It will be measured by counting and recording the actual man-hours expended to plan, design, build and test the best programming language. In this case, the total test time would have to be estimated.

6.3.6 After measurement and calculation of values for \( Z_{\text{real}} \), they are compared to \( Z' \) as follows: If they are all better or the same as the values for \( Z' \), then \( Z' \) is an adequate model of \( Z_{\text{real}} \). If some are better or the same and some are worse, a better trade-off method must be designed to determine model adequacy. If they are all worse than the values for \( Z' \), then \( Z' \) is not an adequate model of \( Z_{\text{real}} \). In that case, this whole problem definition stage should be reanalyzed.

7. Let \( Z, Z' \) and \( Z_{\text{real}} \) be as previously defined. If \( Z \) is the best system in the set of feasible systems and \( Z' \) is an adequate model of \( Z_{\text{real}} \), then "\( Z_{\text{real}} \) is acceptable". Otherwise, the outcome of this step is, "\( Z_{\text{real}} \) is not acceptable".
CHAPTER 4

PRELIMINARY SYSTEM DESIGN

The previous chapter has attempted to state the problem of developing a programming language for communications computers in considerable detail. Now a solution to this problem should be constructed. Such a solution must satisfy the input/output specification, MITCOLSPEC, and the components that form the solution must also be in the technology, MITCOLTECH. When both of these prerequisites are satisfied, that solution will be one of many feasible solutions.

In fact, the number of feasible solutions could be endless. The construction and merit ordering of each of a possible endless number of solutions would, in turn, be endless. Obviously, the client has only a specified amount of resources he is willing to expend on the detailed design of a solution. Thus, the system designer must limit his design efforts to those possible solutions that are most attractive.

Circumscription of Feasible Solutions

The various methods of programming were outlined in the Technology section of Chapter 3. The elimination of any of these
general categories by a somewhat cursory analysis is one way of circumscribing or limiting the feasible solutions. The following analysis states positive or negative aspects of various methods of programming, relates those statements to the various components of the figures of merit and then specifies whether that method should be discarded or not.

The first and earliest method of programming is machine language programming. Writing programs directly into machine language is extremely tedious and slow, or the \( PRTIME \) value of the \( TRANSTIME \) component is extremely large. Few programmers are trained to use a particular machine language let alone those of several manufacturers, or \( INDEPENDENCE \) is zero. Documentation is difficult or \( LEGIBILITY \) is small, and automatic error-detection is almost non-existent, or the \( PROGBAD \) value of \( OUTPROG \) is small. The only really positive aspect is that no translator is required, or the \( PLTIME \) value of \( TRANSTIME \) is zero. Most programming organizations do not program directly into machine language. This method was discarded by them as an economically infeasible approach, or \( COST/REQUIREMENT \) is high. As such, all solutions which encompass the direct use of machine language programming have been discarded.

The second method of programming is assembly language programming. This is the current method used by the client. Again, utilization of this method is tedious and slow, or the \( PRTIME \) value of \( TRANSTIME \) is large. This category of languages is
not machine independent, or INDEPENDENCE is zero. Thus, programmers
must be trained and become proficient in more than one language, or
procurement for new systems must be limited to one manufacturer.
As assembler program in addition to the communications program is
required. These are usually available from the manufacturer as part
of initial computer system procurement. Thorough documentation is
still difficult to produce, or LEGIBILITY is small, and automatic
error correction is minimal, or the PROGOOD value of OUTPROG is
small. Training of programmers must be extensive to overcome the
tendency for error and the resultant problems of effectively and
quickly debugging programs, or the USEREFFICIENCY value is small.

On the other hand, an advantage of an assembly language is
its ability to achieve a better balance between storage utilization
and execution time. It also allows access to specific and, perhaps,
unique machine characteristics. Finally, optimum code for time
critical portions of programs can be written. In effect, the
RUNEFFICIENCY of such a method is high.

Since the client is already using an assembly language
approach for programming communications applications, and feels a
better method is available, this method is acceptable only as a
minimum solution. As such, solutions which include the use of
assembly language programming alone have been discarded. It should
be noted that a solution that allows portions of a communications
application to be programmed directly in assembly language is still acceptable.

The third method of programming is macro programming. The possible solutions using this method could vary from an assembly language with a few macro instructions in it to a programming language consisting entirely of various levels of macro instructions. A tendency toward the former solution would essentially have the advantages and disadvantages of assembly language programming while a tendency toward the latter solution seems to be a good approach, or COST/REQUIREMENT tends to be low. In both cases, a macro expansion capability would have to be added to the assembler. Thus, a programming language composed primarily of macro instructions is a candidate solution which should be developed in more detail.

The final method of programming is higher level language programming. This method is made up of two essential parts - the actual programming language and its translators. Usually, a higher level programming language in its standardized form is machine independent, or INDEPENDENCE is not zero. Inexperienced programmers learn it easily, write programs with fewer errors and can debug programs faster. Improved documentation reduces program maintenance and program conversion to successor machines is simpler, or USER EFFICIENCY is high.

The primary translators of higher level languages can be in two forms - compiler and interpreter. Both have their advantages
and disadvantages. The client, however, operates many small communications computer terminals in its network whose operating system, application and any other software is in either machine or assembly language. The primary reason for this approach, of course, is that the programs were written in assembly language. Nevertheless, in the future if software were written in a higher level language, there would not be a compiler or interpreter in each small terminal. The software would probably be pre-compiled at a central facility after elimination of all known errors. The resulting machine or assembly language program would be used on the terminal. Software maintenance and modifications would be performed at the same central facility. Only the larger terminals and switching centers in the network would retain a translation capability. With this approach to software preparation, maintenance and modification, an interpreter is not an acceptable method of translation. Thus, another method that should be developed as a candidate solution is a higher level programming language whose primary translator is a compiler.

For each method that seems a legitimate candidate solution, there exist three ways to find an actual programming language and its translators. First, an analysis can be made of all existing programming languages and their translators. Secondly, if none of the existing languages will fulfill the requirement as they are, an analysis should be made to see if they could be used with an acceptable minimum of modification. Finally, if a major modification is
required to make a programming language acceptable, then it would probably be better to design a new programming language that meets the optimum merit orderings as closely as possible.

Existing Higher Level Languages

An analysis of all existing higher level languages and their compilers is again an extensive undertaking. Before such an analysis is made in detail, existing higher level languages should be culled to obtain those languages that best meet certain general requirements. These general requirements are: Is it considered a multi-purpose language, or is CAPABILITY a high value? Has there been some standardization effort, or is INDEPENDENCE larger than zero? Is it used with existing compilers on a wide range of computer systems, or is INDEPENDENCE a high value? And, does it seem to include the rest of the characteristics and requirements stated in Chapter 3?

With these general requirements in mind, the programming languages in the first column of Fig. 7 were chosen as the best candidates for further detailed study. As stated above, these higher level languages will probably not meet all the detailed requirements and/or may have superfluous capability. Here again the use of existing extensions or subsets may prove acceptable. Those extensions or subsets which also should be studied in further detail are listed in columns 2 and 3 of Fig. 7.
<table>
<thead>
<tr>
<th>Higher Level Languages</th>
<th>Extensions</th>
<th>Subsets</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL/I</td>
<td></td>
<td>XPL</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TACPOL</td>
</tr>
<tr>
<td>JOVIAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COBOL</td>
<td>COBOL with Communications Extension</td>
<td></td>
</tr>
<tr>
<td>FORTRAN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALGOL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SNOBOL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7. Candidate Existing Higher Level Languages
New Higher Level Language

If substantial changes must be made to an existing higher level language, extension or subset, or a new language must be designed from "scratch", then a description of its characteristics, features, structure and form must be specified in greater detail than has heretofore been stated in Chapter 3 and Appendix A. The following phrases attempt to outline the detailed description of a communications computer programming language. This description assumes some knowledge of computer and software terminology and only defines the meaning of phrases that may be ambiguous.


1.1 ASCII character set
1.2 ASCII, ITA#2, Hollerith input/output data sets

2. Basic Elements

2.1 Identifiers
2.1.1 Used as data names and labels
2.1.2 Preferably not used as reserved words
2.1.3 Preferably maximum length of 20 characters
2.1.4 Composed of letters, digits and other selected characters such as the underline ‘_’

2.2 Operators

2.2.1 Arithmetic - addition, subtraction, multiplication, division
2.2.2 Relational — equal to, greater than, less than, greater than or equal to, less than or equal to, not equal to
2.2.3 Boolean — and, or, not
2.3 Delimiters and Punctuation
2.3.1 Use as necessary to insure language clarity for user
2.3.2 Blanks have minimal significance except for readability and separation of key words
2.4 Constants
2.4.1 Literals
2.4.1.1 Composed of any string of characters except character chosen as delimiter
2.4.1.2 Delimiter is single quote mark at beginning and end of string (double quote mark can be used internally)
2.4.2 Numbers

3. Program Structure
3.1 General characteristics
3.1.1 Language will contain executable and non-executable statements
3.1.2 Beginning and end of statements must be delimited
3.1.3 Embedding of statements within another is authorized, e.g., IF A=B THEN A=A+2 ELSE A=A+5 contains two assignment statements embedded in a conditional statement
3.2 Declarations
3.2.1 Contains data description for identifiers
3.2.2 Contains attributes, e.g., initial values
3.2.3 Specifies storage allocation with capability of either static or dynamic storage
3.2.4 Outlines segmentation requirements
3.2.5 Located preferably at beginning of program and/or block
3.2.6 Specifies scope of variables
3.3 Loops
3.3.1 Contains range or set of statements which are executed
3.3.2 Contains the value(s) of parameter(s) in the range
3.3.3 Contains a terminating condition
3.4 Procedures and Functions
3.4.1 Parameters
3.4.1.1 Can have none
3.4.1.2 Can be called by value or reference
3.4.1.3 Can be an expression
3.4.2 Can be invoked from main program
3.4.3 Prefer that it can be recursive
3.4.4 Contains blocks and declarations
3.5 Blocks
3.5.1 Contains designator for beginning and end
3.5.2 Consists of statements
3.5.3 Can be referenced from control transfer statements

3.6 Comments

3.6.1 Can be used flexibly

3.6.2 Will be delimited at beginning and end

4. Data types

4.1 Arithmetic

4.1.1 Integers or Fixed Point numbers

4.1.2 Floating Point numbers

4.1.3 Precedence is division and multiplication first with addition and subtraction second

4.2 Logical

4.3 String

4.4 Pointer - conveys information about the location of another data item

4.5 Hierarchical - concatenation of other data types

4.6 Combinations

4.6.1 Tables

4.6.2 Arrays

4.7 Global scope unless specified as local in a function or procedure

5. Executable Statements

5.1 Assignment Statements

5.1.1 Assigns results of arithmetic computation to a variable
5.1.2 Data type of single variable on left of assignment sign will govern any arithmetic conversion

5.2 Control Transfer Statements

5.2.1 Unconditional jump

5.2.2 Switch or transfer of control based on existing or prior conditions, e.g., computed GOTO

5.3 Conditional Statements

5.3.1 Contains IF Boolean statement format

5.3.2 Contains any type of subordinate statement or block

5.3.3 Option to use ELSE or multiple ELSE type statements

5.4 Loop Control Statements

5.4.1 Permits nesting

5.4.2 Integer iteration of loop

5.4.3 Boolean condition on loop termination, e.g.,

DO WHILE (X<0)

5.5 Error Condition Statements

5.5.1 For data or programmer errors

5.5.2 For hardware errors, particularly input/output

5.6 String Handling Statements

5.6.1 Concatenation

5.6.2 Deconcatenation

5.6.3 Insertion of strings between others

5.6.4 Deletion of portions of strings

5.7 Input/Output Statements
5.7.1 Read a record
5.7.2 Write a record
5.7.3 Open, close, update and position a file
5.7.4 Continuous string input (physically)
5.8 Miscellaneous Statements
5.8.1 Stop Statement
5.8.2 Null Statement
6. Non-executable Statements
6.1 Declaration Statement
6.2 File Description Statement
6.3 Procedure invocation and return statements
6.4 Format Description Statements
7. Library Functions or Procedures
7.1 Count string length
7.2 Search for a specified string
7.3 Define the location in storage of an argument
7.4 Specify a null value location
7.5 Get current time
7.6 Get current date
7.7 Perform code conversion

This outline is a step closer in terms of detail to defining a new programming language for communications computers. Certainly, more detailed and complete descriptions can be designed. For the purposes of this thesis, this is the extent of preliminary design of the actual programming language itself.
For a new programming language, the work in designing the language itself is small compared to the effort of designing and implementing enough compilers for different classes of computers to make the programming language truly machine independent. While standardized languages such as FORTRAN have compilers available and supported by most hardware manufacturers, universities and/or software companies, a new programming language must get its compiler implementations from the user either through in-house effort or through a contract with some software company. Unless the cost of such work can be lowered by utilizing translator writing systems or bootstrapping methods, this factor alone is a major reason not to implement a new programming language. Thus, this area of compiler implementation must receive preliminary design emphasis before a decision to utilize a new higher level programming language is made.
CHAPTER 5

CONCLUSIONS

What are the results of this thesis in terms of practical value? A somewhat detailed definition of the problem in a systems methodology and some preliminary design of a solution to that problem have been examined. Yet, there is no detailed final solution and no estimate of resources necessary for designing and implementing that final solution. A review of the results will show that a constructive beginning to a solution of the client's problem has been accomplished.

Synopsis of Results

The statement of a system design problem is the first important task of any project. If it is sketchily or improperly formulated, in all likelihood the solution to the problem will have the same characteristics. Or, the problem definition phase of the project will have to be revised considerably in some intermediate stage of the project. Thus, the content of problem definition can reflect the potential results of the complete project.

In the case of designing a programming language for communications computers, a detailed problem definition was
accomplished in Chapter 3 in terms of rigorous systems engineering methodology. After naming the project, an input/output specification was defined. Then, a listing was made of major areas of technology available for system development. In turn, a means of ranking systems that meet the input/output specifications, a means of ranking systems that can be built from the technology and a means of ranking feasible systems from both a technology and input/output standpoint was specified. Finally, a procedure for system testing was written. This precise definition of the communications programming language problem served as a solid basis for preliminary design.

In Chapter 4 concerning preliminary design, the various generalized approaches for solutions to the problem were discussed. It concluded that certain design approaches should be investigated in detail and others should be discarded from further immediate consideration. The best design approach is to investigate existing higher level languages as listed in Fig. 7 and their compilers to see if any of them meet all the technical characteristics and compare favorably with respect to the merit orderings for a communications programming language. Probably, none will.

If it is true that no existing higher level programming language meets all the stated technical characteristics or compares favorably with respect to the merit orderings, then the previous investigation is a natural and important precursor of the next design approach. This second best approach is to investigate existing
subsets or extensions as listed in Fig. 7 of higher level languages and their compilers. Here, the programming languages will probably be closer to meeting the stated technical characteristics or comparing favorably with respect to the merit orderings, but will not do so in every case. Also, implementation will be more difficult because compilers will not exist for all classes of computers that could be used in the communications network.

The third best design approach again leads from the previous two. It is to design modifications and/or extensions to the programming language previously investigated which was most amenable to modification or extension. Again, implementation will be more difficult because no compilers will exist. Probably concurrent modification or extension of existing compilers could be designed at the same time as the programming language itself is changed.

At the same level of "goodness" is to design a macro assembly language composed entirely of various levels of macro instructions. Again translator implementation will be more difficult because in most cases only the assemblers themselves exist for each class of computers. A macro expansion for each macro will have to be written in assembly language for each class of computers. This set of macro expansions will then have to be integrated into each appropriate assembler.
The last of the recommended approaches is to design a new higher level communications programming language that meets the technical characteristics and compares favorably with respect to the merit orderings in every aspect. To begin such a design effort, a more detailed description of the programming language was outlined. Such a description is important not only for this design approach but also as a basis for comparing existing languages with overall requirements. Finally, the important reason that this is the last of the recommended approaches is that a new compiler must be designed and implemented for each class of computers.

Areas of Future Study

Since the methodology used here is dynamic, continued development of the design of a communications programming language will require revision of previous design efforts as well as the definition of the problem itself. This revision process has been characteristic of what has been done so far. As such, future iterations of this process should consider the following areas:

1. A more detailed description of input/output characteristics.

2. Investigation of other areas of existing technology that have been overlooked.

3. Define additional performance indices to cover aspects of the programming language that are not presently measured.
4. Redefine performance indices where possible that rely on a subjective evaluation of certain behavioral aspects. Sometimes the state-of-the-art is such that subjective opinions by experts is all that is available.

5. Define a valid probability distribution by either classical statistical methods or by utilizing acceptable preference function methods. Thus, a better computation could be made of the input/output, technology and trade-off figures of merit.

6. Complete the definition of the feasibility merit ordering by formulating additional trade-off criteria and by devising a formula or means of resolving the ranking of all candidate systems, not just the majority.

7. Refine the procedure for systems testing.

8. Improve the client's knowledge of the methodology used, and the relationship between the client and the researcher so that the results more truly reflect the client's desires.

Some areas of technology have been mentioned briefly as possible candidates for inclusion in the final system design. The brevity of their reference in most cases reflects the state of their development, not their potential. The following is a listing of those areas which should be researched carefully to see if they would apply towards a solution of the communications programming language problem, or need to be developed as a proven technique in the design of programming languages in general.
1. Development of microprogramming techniques and microprogrammable computers. One utilization of these techniques and capabilities would be to have a microprogrammable computer at the central software facility to serve as a test bed via emulation for development, test and maintenance of all communications software for any class of computers.

2. Development of a viable translator writing system. The utilization of this technique would allow improved design in a shorter time of the many compilers that might be required. Such a technique would accept a definition of a source language in terms of some target language and would produce a program capable of translating from the source language to the target language.

3. Development of the technique of bootstrapping. Such a technique envisions implementing a compiler for a source language by writing it in the source language. One way to do this is to write a compiler for a small subset of the source language in assembly language and then rewrite it in the source language itself. Now the bootstrapping takes place in a series of steps. At each step we extend the previous compiler to a compiler for a bigger subset of the language by implementing other features of the desired language. Also, old parts of the current compiler can be rewritten in the new subset of the source language to take advantage of these new features. Eventually, the desired complete source language is contained in the last version of the compiler. A somewhat similar approach might also
be used to bootstrap an existing compiler on one computer for a specific programming language to another computer.

Ultimately, the most important area for future study is to finish the present project. The tasks remaining can be summarized as finishing the design plus implementing and testing a programming language for communications computers that satisfies the client.
APPENDIX A

FUNCTIONAL REQUIREMENTS FOR A PROGRAMMING LANGUAGE FOR COMMUNICATIONS COMPUTERS

Functional requirements of a communications programming language are the procedures and capabilities in that language that a programmer requires to write a program for a computer that processes message and data traffic in a communications computer system. Ideally, such functional requirements should be applicable to any user with a communications computer programming demand. That was the intent when the following list of functional requirements was compiled. Nevertheless, there is some bias toward the client, the U.S. Army Communications Command, and the computer communications network it participates in, DCS AUTODIN.

The sequence or grouping of the functional requirements is not intended to indicate the sequence or the modularity of an actual communications program. Rather, the organization used to indicate the functional requirements is an attempt to state them in the most complete and logical manner possible.

1. Executive Control - The system supervisor must have the capability to maintain complete control through a series of software routines to provide for internal sequencing and passing of control to
and from vendor and user supplied programs. Specific subfunctions that might be included are:

1.1 Restart and Recovery - The system must be able to recover from hardware and some software errors. There should be some type of self-monitoring control to determine these types of errors. This function therefore could be either system or operator initiated. The system should periodically store any data and register contents needed for restarting the system or recovering messages due to system failure. This function must retain data or rebuild queues at the time restart is performed.

1.2 Dumps - The system must have the ability to output a full or selective printout of main or auxiliary storage. It should be initiated either by the system or the operator during on-line or off-line processing.

1.3 Retransmission - The system should have the ability to retransmit a message either from external or internal storage upon demand from another system in the communications network or the operator.

1.4 Program Scheduling - There must be a facility for controlling and sequencing the execution of subprograms.

1.5 Priority Interrupt Scheme - There must be a flexible system of priorities so that the supervisor can interrupt a low priority processing function and start a higher priority one.
No loss of data must occur in order that the low priority function can be fully reinstated when all higher priority tasks are completed.

1.6 Overlay Control - The supervisor should be able to bring in library modules for execution in an overlay area.

1.7 Program Relocation - There must be some facility for relocating programs within main storage.

1.8 Dynamic Core Allocation - The supervisor must be able to assign variable amounts of main memory to executing modules and, through program relocation, conserve the maximum amount of core storage for other use.

1.9 Interaction with Communications Devices - The supervisor must have facility to determine if any of the communications devices or non-communications devices are operational or require service.

1.10 Input/Output Control - The supervisor should have control over assignment and scheduling of input/output channels and functions, whether the devices are used in communications or in background batch processing tasks.

1.11 Switchover/Fallback - The system should include a means for switchover for redundant systems and/or components as well as a software procedure to implement a graceful degradation of hardware components as failures occur.

2. Input/Output Processing - These requirements perform the processing steps necessary to bring data into the computer where it
is examined and positioned for output processing. The following are considered input/output processing functions:

2.1 Acknowledgement - The system must be able to correspond with other systems in the network to acknowledge correct or incorrect data transmission or to reject such transmission when the system is saturated.

2.2 Input/Output Interface - The system must be able to control hardware buffers to compensate for varying data rates and provide for data set interfaces for various modulating and demodulating communications equipment.

3. Message Processing - This requires manipulation of the data itself by content and meaning. The following are considered message processing functions:

3.1 Priority Processing - The system should process multi-level precedence messages on a first in/first out basis for each level of precedence. Higher precedence messages will be processed before those with lower precedence. This function must suspend processing of a message when one of a higher precedence enters the system, and return to the interrupted message when processing of the higher precedence message has been completed.

3.2 Message Routing - The system must recognize certain predetermined routing designators and output messages to the network according to the control information contained in the message.
3.3 Message Distribution - The system must be able to automatically distribute incoming messages to remote terminals or output devices based on control information or on message content or on information stored in the system.

3.4 Recognition of Control Information - The system must be able to recognize predetermined control characters or information such as end-of-message characters, framing characters, etc., and act accordingly upon recognition of this information.

3.5 Validity Checking - The system must be able to analyze a specified portion of the data pertaining to record content, format, length, classification, routing, etc., and verify the validity of this data. The software must also be able to detect invalid data patterns by using such methods as vertical redundancy checks, longitudinal redundancy checks, or cyclic redundancy checks.

3.6 Cancellation of Transmission - The system must have the ability to terminate processing for a given message prior to the transmission of the end-of-message control characters.

3.7 Data Manipulation and Code Conversion - The system must be able to manipulate data at the bit level regardless of the structure of the machine. Conversion routines for communications procedure formats such as JANAP 128 to ACP 127 and for data codes such as ASCII to EBCDIC must be provided.

3.8 Message Bookkeeping - The system must be capable of logging messages and keeping pertinent records pertaining to each
message. This function also keeps track of various statistics and measurements within the system.

4. History/Storage - This requirement involves the storage of messages and bookkeeping data for the purpose of maintaining a systems history and/or journal of events. These functions are:

4.1 Decayed Data Removal - The system must be able to remove or update messages or data which are outdated or no longer needed for backup.

4.2 Storage - The system must be able to temporarily store messages before or after processing for backup or reprocessing as required when the system is saturated. It should permanently store statistical and other relevant information processed by the system.

5. System/Operator Communications - This requirement provides the necessary man/machine interface communications to inform the computer or operator of any conditions necessary for the operation and maintenance of the system. These communications may be system or operator generated. The following are considered system/operator functions:

5.1 Message Intercept - The system should have the facility to route messages either by system or operator command, to a specified device before, during or after processing for alteration, checking and analysis. This includes the ability to output,
on a selected device, selected portions of a received or transmitted message.

5.2 Device Selection - The operator must have the option to select or substitute input/output devices other than those configured as standard on-line equipment for the system, provided hardware interface differences can be managed through software.

5.3 On-Line Modification - The system should allow for on-line software patching and debugging, and permit operational changes or substitute program modules during system operations. This will require the translator, if used, to generate man-readable symbolic machine language corresponding to the higher level instructions.

5.4 Systems Control Rule Changes - The system must permit operation parameter changes such as machine configuration, processing sequence, input/output format, control characters, and message routing either by recompilation or by operator command.

5.5 Fault Detection - The system should allow fault detection and diagnostics which can require either certain automatic machine operations or corrections and/or operator intervention and correction.
LIST OF REFERENCES


