REHAK, DANIEL ROBERT

COMPUTER AIDED ENGINEERING PROBLEMS AND PROSPECTS

University of Illinois at Urbana-Champaign PH.D. 1981

COMPUTER AIDED ENGINEERING
PROBLEMS AND PROSPECTS

BY

DANIEL ROBERT REHAK

B.S., Carnegie-Mellon University, 1973
M.S., Carnegie-Mellon University, 1976

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Civil Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1981

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

May 1981

WE HEREBY RECOMMEND THAT THE THESIS BY

DANIEL ROBERT REHAK

ENTITLED COMPUTER AIDED ENGINEERING PROBLEMS AND PROSPECTS

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

Director of Thesis Research

Head of Department

Committee on Final Examination†

Chairman

† Required for doctor's degree but not for master's

0-517

ABSTRACT

We are now entering the third decade of engineering computer applications. In the past twenty years, the computer has become a requisite tool in civil engineering. One is hard-pressed to imagine the analysis and design of structures such as high-rise buildings (exemplified by the Sears Tower, John Hancock Building, World Trade Center, etc.) without such a powerful computational tool. Unfortunately, the computer is still used as a basic tool, and has not been fully integrated into the design process. There has been a significant increase in the scope, range, and power of the computer applications, but there has been little progress in the development of an integrated computer aided engineering environment.

The computer has the potential to take a much larger role in the engineering design, analysis, construction, and project management processes. The use of computer systems to maintain the large volume of data present for a project, to verify the compliance with standards, and to provide project management, in addition to its traditional design and analysis role is desirable. Integration of the computer throughout the design process can produce better engineered systems by allowing the computer to assure consistency, completeness, and compliance, in a rigorous manner, throughout the design cycle; the current lack of these aspects is a major problem.

Attempts to advance computer utilization in engineering are being blocked by the current state of engineering software technology. Much of the software being used was developed in the mid-sixties. There have been some changes in the underlying software concepts utilized, but a large portion of current software is rooted in the computer technology of the sixties. In order to move forward and provide future advanced engineering systems, significant changes in engineering software systems are required.

Two problems, (a) the design of an integrated multi-disciplinary engineering design software system, and (b) interfaces to finite element systems, are presented to show: (1) why the current state of engineering software technology is not capable of supporting the development of advanced engineering computer systems, and (2) what types of capabilities are needed in these systems. Particular issues discussed in detail include: standards processing, data management and handling, program interfaces, and logic and process control.

To develop the next generation of engineering computer systems, advanced computer technologies must be integrated into engineering software. Topics such as relational database management and knowledge based artificial intelligence are discussed, and it is shown how aspects of these technologies can be applied to the problems currently limiting engineering software. These technologies provide the basis for a proposed software environment which may be used to develop advanced computer aided engineering software systems.

PREFACE

The presentation contained herein is the result of two pilot studies by the author:

(a) The design and implementation of a general purpose multi-disciplinary computer aided design system.

(b) The design and implementation of user interfaces to finite element software.

Both of these problems have a straightforward description, and the form and style of the desired solutions is known. Unfortunately, a straightforward implementation of the solutions, based on current engineering software practice, is not possible. The difficulties in developing complete solutions to these problems lies in the current state of software for engineering problems. These complex engineering software systems require complex software solutions, software beyond the scope of that currently used in engineering application programs. As a result, this thesis has evolved as a discussion of the software issues which need be addressed in order to advance the art of computer applications in engineering.

This thesis deals with the software engineering of a proposed new generation of engineering software systems. The resulting discussion, and the topics on which it is based, are interdisciplinary in nature. The presentation deals with both the engineering nature of the problems associated with developing such software systems, and with a variety of computer science topics and techniques which are used in the proposed solution.

Organization: Chapter 1 provides background information on computer applications in civil engineering. It presents a review of the evolution of computer utilization in civil engineering applications, a definition of computer aided engineering, and a presentation of the objectives and scope of this research.

In chapter 2, the two problem domains: (a) the design of an integrated multi-disciplinary engineering design software system, and (b) interfaces to finite element systems are presented. A description of the problem domain, the capabilities needed for a solution, and the status of solutions to the problems are presented.

Specific problem areas, which are independent of any application domain, and which limit the development of advanced engineering computer systems, are presented in chapter 3. This chapter deals only with the various problem areas. The problem areas are each treated individually and independently of any possible solutions.

Chapter 4 contains the description of a variety of tools and techniques which might be used to develop solutions to the problems discussed. Each of the solution techniques may be applicable to one or more of the problem areas of chapter 3. These techniques will be related to the problem areas, but each of the techniques will be treated individually.

Chapter 5 proceeds to describe a proposed prototype computer aided engineering software environment which could be used to develop advanced computer applications for engineering. All of the various solution techniques are combined in an integrated system to address all of the problem areas. The reader may skip directly from chapter 1 to chapter 5 if he desires only information on the proposed solution.

A summary and discussion of the proposed software system, along with its application to the specific problem domains presented in chapter 2, are contained in chapter 6.

Many of the tools and techniques used in the proposed engineering software environment are taken from state-of-the-art research in computer science and software engineering and are foreign to civil engineering and many of the readers of this thesis. For the readers convenience, a short introduction to some of these topics is presented in the appendices. Additionally, there is a glossary to present definitions for many of the terms used throughout the text.

Acknowledgements: The author wishes to express his appreciation to his advisor, Professor L. A. Lopez, for his assistance throughout this study.

The Civil Engineering Systems Laboratory of the University of Illinois (CESL) and the Department of Civil Engineering provided unlimited access to the Burroughs B6700 and PDP 11/04-GT41 computing facilities. The availability of these resources, and the interaction with the user community and faculty provided invaluable insight and experience throughout this study.

Financial support provided by University of Illinois Fellowships and Research Assistantships is gratefully acknowledged.

TABLE OF CONTENTS

Page

Page

APPENDIX Page

LIST OF FIGURES

1. INTRODUCTION

The creation of engineered systems is an ill-defined and complex task. It is a cooperative effort between a sponsor or client, a design team, a constructor or manufacturer, and possibly the users of the product. Each member of this group has his own (potentially conflicting) concepts and goals for the project, and these affect the final product. Decision making, communications, and information management play major roles in the design process. As projects grow more complex, informal methods for communications, information management, and decision making tend to break down, resulting in a decrease in the quality of the final product (measured in time to produce, costs, or by some physical quality attribute). As a remedy, engineering oriented design systems based on modern computational technologies can potentially provide a formal communications, information management, decision support environment to assist in the engineering process, in addition to providing the more traditional analytical and computational tools.

The work contained herein is a discussion of the various problems which limit the development of such computer systems, and a discussion of techniques for addressing these problems. The presentation is oriented towards, and based in, the civil engineering design domain. Civil engineering represents, possibly, a worst case situation: the various groups involved in the design of a project are usually associated with different organizations; the projects are long-term; individual projects are unique; numerous subdisciplines are involved; there are a variety of governing constraints, regulations, specifications, and standards; absolute measures for judgement and comparison do not exist; the design process is ill-defined and ill-structured; and of course, everything is subject to time varying change. This situation is not limited to civil engineering, but rather, it is the norm in civil engineering. As a result, the presentation which follows should not be viewed as limited to civil engineering, but as a discussion of a general problem common to all engineering disciplines, and which is exemplified in civil engineering.

1.1 Computer Applications in Civil Engineering

Computer utilization within civil engineering is entering its third decade. Since the introduction of COGO [MillC61] and STRESS [FenvS64], usage has increased to the point where costs of computer utilization in structural mechanics alone are measured in billions of dollars per year [SchaH78]. It is difficult to imagine the design and construction of modern structures without computer assistance.

COGO and STRESS were among the first general purpose applications, and they were responsible for setting the tone and style of future developments. From the user's point of view, some modern applications appear similar to these original programs. Additionally, STRESS and COGO are still actively used. Their popularity is due to their effectiveness and ease of use. They have often been replaced by similar programs with extended capabilities, but engineers are reluctant to change their tools without need. If they are to be accepted by the profession, new engineering computer systems must provide more than a better way to do the same thing.

From these beginnings, computer utilization has expanded into numerous problem domains, including: hydrology, transportation planning, project control and scheduling, estimating, automated drafting and detailing, finite element analysis, geotechnical analysis, and component design and selection. This horizontal expansion extends into all areas where the mathematical procedures can be easily converted into automated computational processes. Additionally, within each domain, there has been a vertical expansion, with newer systems having extended the capabilities present in their predecessors. However, in spite of the horizontal expansion, structural mechanics remains the preeminent application area.

Structural analysis has seen a large vertical expansion of capabilities. With the completion of STRESS, the developers felt the need for a system with improved analytical capabilities. STRUDL [LochR67] was thus born, but its development was hampered by the then current state of software (the techniques used in STRESS required extremely complex hand coding). This resulted in the development of ICES [RoosD66] to provide support for general civil engineering applications (although there is nothing particular to the ICES concepts which limit it to civil engineering).

The finite element methodology emerged at the same time. It requires an advanced computational capability, and it is readily adapted to current computer technology. The methodology and its software realization each

contributed to the success and further development of the other. Software development for finite element applications has continued at a rapid rate, with all of the new programs attempting to overcome difficulties in, and to provide capabilities lacking in, previous systems.

The large, general purpose finite element analysis systems all rely on some type of underlying software support to assist in the program development task and to provide run-time support. NASTRAN[1] relies on DMAP and GINO [MacNR71, McCoC72, NASA72a, NASA72b], SESAM on NORSAM [Bell K73, Egel074, Mo078], ASKA on DRS and now DVS [SchrE74, SchrE77, SchrE78, SchrE79], and FINITE on POLO [LopeL72a, LopeL72b, LopeL77a, DoddR80]. In the latter case, the development of the application (FINITE) was hindered due to insufficient capabilities in the then available support software, and this necessitated the development of a general support-supervisory system (POLO) prior to completion of the application.

The support-supervisory systems (ICES, POLO, DVS, etc.) were developed to ease the burden of programming large application systems. The typical analysis program is written in FORTRAN, which does not provide any facilities for data structuring, database support, or memory management. A generalized, large-scale application requires complex data organizations to store and utilize problem data. Additionally, many practical problems exceed the available physical memory resource of current production computers. Explicit programming of the details of handling all the data structures, and development of techniques to fit needed data into the limited memory resource, is a complex process. It results in programs in which the analytical component is totally obscured by the details of resource and data management. The support-supervisory systems attempt to eliminate this burden. They provide data structuring facilities and run-time support for resource management, databases, and input language translation. Although applicable to any type of engineering analysis, the major systems supported are all structural mechanics or finite element analysis systems.

Most organizations consider the computer to be only a computational tool. In a recent U.S. General Accounting Office survey [GA080], the major reason given for computer utilization was "to carry out tasks which would not be practical using manual techniques," and the major task area was structural mechanics. Essentially, engineers use computers to solve complex, time consuming problems which can not be done by other means.

---

[1] NASTRAN is a registered trademark.

The limited application of computers outside structural mechanics is of concern. The success of computers in the structural mechanics and finite element analysis field is largely due to the timely development of the two cooperating technologies. The computer can do more, but application areas such as reducing the number of design errors and checking compliance with standards currently account for only 2% and 1% of computer utilization, respectively [GAO80]. The computer's information handling ability is well suited to the design process. The use of a computer based design system can eliminate much of the routine processing and data handling performed by the engineer. This will permit the engineer to become more productive. He will be able to evaluate more alternatives, and do a better job of design and checking without increasing design costs. A more effective use of the machine will allow the engineer to spend more time on the creative aspects of his task.

This extension of usage of the computer has not occurred as rapidly as one might like. Much of the previous software development effort has been oriented towards specific applications with well-defined computational procedures. Engineering software systems with decision support, information management, and multi-user communications capabilities are desirable, but these are more complex tasks than those which have been computerized in the past, and for their development they require the use of more complex software techniques that those currently used. The relationship between software technology and needs and requirements is shown in figure 1.1 [JensR79]. It is the premise of this work that the development of engineering software, particularly for nonanalytic processes, is difficult and hindered by the current state-of-the-art in engineering software development, and without advances in software technology, the range and scope of engineering computer applications can not be readily expanded.

1.2 Computer Aided Design / Computer Aided Engineering

There has been an increase in the number of attempts to extend the use of computers into all design domains, augmenting the computer's traditional analysis role. Active areas of development of software for design and engineering applications include mechanical engineering parts manufacturing and electrical engineering circuit and chip layout. Such efforts are denoted by a number of names and acronyms such as CAD (Computer Aided Design) and CAM (Computer Aided Manufacturing). These and various other names and areas of work are described in more detail in the glossary (section 2).

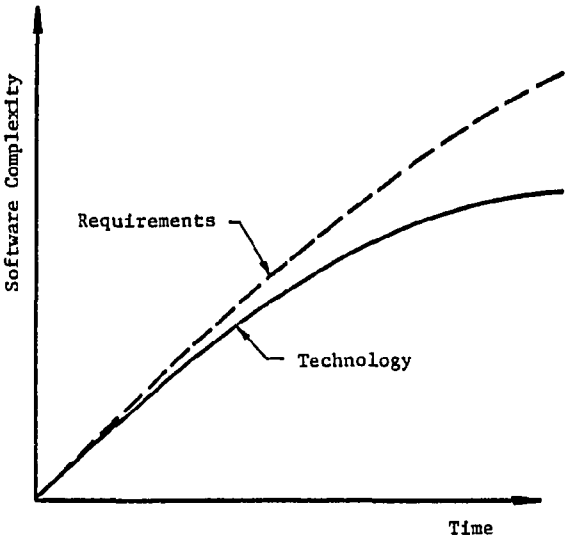


Figure 1.1. Software Technology

There is a problem in that there are no clear definitions of what constitutes a design and engineering computer application in a particular problem domain. Anyone may classify a process, technique, or program into any of the fields of computer applications to design and engineering. The problem is particularly acute when dealing with CAD. There have been numerous pieces of software which have been denoted as CAD systems. However, it is most often the case that this software only provides some graphics display capability or analysis function which is used in some particular phase of the design process. Such software usually has a very narrow scope. Although such capabilities do fit into the definition of having the computer provide some assistance in performing design tasks, one is left with the feeling that something is missing. Design is usually considered to be an ill-structured creative selection process, the process of selecting components and configuring the form of an engineered system. Analysis and presentation are important components of the overall engineering design process, but they are usually considered to be subservient activities to the total process.

As a result of the above situation, the phrase "computer aided engineering" has been used herein to describe the applications of computers in the more traditional design and engineering role. The following is used as a working definition throughout this presentation.

Computer Aided Engineering: The application of an integrated, man-machine, computational environment to the life-cycle process of creating multi-disciplinary engineered systems.

In the definition, the following phrases are important:

integrated: Design consists of a number of separate processes, each with their own data and computational needs. These processes and their data should be automatically linked, without the need for manual coupling of the various computational aids used in the design process.

man-machine: Complex computations can not be performed blindly by the machine. The engineer still must retain control, using the computer to perform in a manner which will be most helpful to the engineer. Engineering computer utilization must be a synergism of man and machine.

environment: Comprehensive design programs can not be regarded as simple tools to be picked up from the shelf, used, and returned. These systems require the computer be a constant

companion to the engineer. The various computational aids should be incorporated into all phases of the design process, and the procedures for design and engineering should rely on an integrated support environment provided by a computer based design system.

life-cycle: Design and engineering begins with the conception of a project, and it continues throughout all steps, until the project is constructed. For many projects the work continues beyond construction, supporting changes, maintenance, and updates.

multi-disciplinary: The design of a large project is not the work of an individual, or of a group of experts from a single discipline, but rather, it involves the cooperation of engineers, specialists, and technicians from a variety of areas.

Computer aided engineering systems should deal with all of these aspects of engineering and design. Any system claiming to do computer aided engineering must deal with aspects such as project control, data management, process integration, and user communications all applied to large-scale, multi-disciplinary, long-term, engineered projects. Any computer application which fails to deal with all of these aspects can not be classified as one doing true computer aided engineering. Unfortunately, most application systems available today fail to meet these criteria.

1.3 Objectives and Scope

This work deals with the design and implementation (the software engineering [JensR79]) of large, general purpose engineering software systems. Such systems are designed specifically to support engineering and design applications with the following attributes:

generalized: Specific applications are designed to solve one and only one problem (possibly with some minor parametric variations). Generalized or general purpose applications are designed to solve all members of a large class of problems (e.g., one program for all types of finite element structural analysis as opposed to individual programs for flat plates, 2-D plane stress and strain, cylindrical shells, etc.). Generalized systems provide an extensive set of capabilities

which can be applied in a majority of situations (but which are possibly not the most efficient or most appropriate for any one case). They are preferred because they present a uniform problem solving approach for an entire problem domain, rather than different approaches for many similar tasks.

large-scale: Large-scale systems are not constrained to a particular maximum size of problem; they are designed to be applied throughout the range of potential problem sizes, from the smallest to the largest practical. Thus, the size of the problem being solved need not influence the solution approach.

ill-structured: Design and engineering are ill-defined and ill-structured tasks. Specific single component design may have a well-defined and well-structured problem solving methodology, but the "creative" design and engineering process which deals with an entire engineered system is ill-structured [SimoH73]. "Each small phase of the activity appears to be quite well structured, but the overall process meets none of the criteria we set down for WSPs [well-structured problems]." The interaction between, and complexity caused by, the individual subprocesses creates a process which is ill-structured in the whole.

Therefore, the scope of this work is that of the large-scale, generalized, computer aided engineering system for ill-structured problems. Such a system is different from that which is used for any specific, well-structured problem solving activity. This difference is due to both the level of sophistication required to implement the features of a generalized engineering software system, and the actual size of such a system (complexity grows in an exponential fashion with increasing size and sophistication). This difference necessitates an approach to software design and implementation which is different from the approach used for the smaller scope problems. Throughout this work, all of the discussion presupposes an orientation towards developing a generalized systems approach for large ill-structured and ill-defined problem domains.

Computer aided engineering systems (as defined in section 1.2) do not yet exist. The brute force approach of building a computer aided engineering system based on current software technology will not produce a system with the desired sophistication. However, the application of current advanced software

techniques does show promise. Just as with the development of the prior generation of support-supervisory systems, the current software base must be expanded to meet the needs of the new applications. Several state-of-the-art techniques from computer science such as relational database management and knowledge based artificial intelligence must be brought into usage in engineering applications.

The objectives of this work are to show: that there are several major problem areas which must be solved before a computer aided engineering system can be built, what capabilities are needed in such a system, techniques which are available to solve these problems, and the structure of a proposed prototype for the next generation of computer software for engineering applications.

2. TWO PROBLEM DOMAINS IN ENGINEERING SOFTWARE SYSTEMS

In order to better understand the difficulties in developing large-scale engineering computer systems, two problems are presented and discussed:

(a) The design and implementation of a general purpose, multi-disciplinary computer aided design system.

(b) The design and implementation of user interfaces to finite element software.

These problems are treated individually. The discussion includes the motives for solving the problem, a description and components of one possible solution, and a summary of the status of software available to solve the problem. The discussion is quite general, and does not address the details of any particular solution. Rather, the purpose is to provide a flavor for the types of problems which exist and which must be addressed in developing software for engineering systems.

2.1 Problem A — A Computer Aided Design System

Large design projects are multi-disciplinary in nature. They deal with large volumes of information, which must "flow" between members of the design team. Additionally, they are guided and constrained by various design standards. Information flow and standards present many problems in design. There is the need to communicate up-to-date information between the members of the design team, and to process the complex standards which govern the design. Design is an information processing task, and the computer is an effective information processor. A computer based design system could help with standards processing and with the multi-disciplinary nature of design tasks, potentially producing better designs at lower costs.

One philosophy for a computer based design system would be a generalized support software system which is independent of design tasks and standards, and which could be used as the base for developing specific, task oriented, design systems. The computer aided design system problem is, therefore: to design and implement a software system for use in a multi-disciplinary, long-term, project oriented, design environment (similar to that described in section 1.2).

2.1.1 Problem and Motivation

Engineering design is a complex process. It is iterative, subject to many constraints, and multi-disciplinary in nature. One possible view of a project oriented "design loop" is shown in figure 2.1. The project moves through four phases: from a synthesis or conceptual phase, to preliminary design, to detailed design, to construction.

Each of the first three phases of the design process consists of three steps: (1) selection (design), (2) analysis, and (3) evaluation. This process is performed by all disciplines, for all systems, subsystems, and components which comprise the design. The state of the design proceeds from the set of available information, with the processes providing new information for the next phase. The evaluation of a component may, at any time, result in a failure of the solution to meet criteria. This results in an iteration within the phase, iteration to an earlier phase, or possibly a complete failure of the process.

Throughout the design process, the design procedures are driven, and the results are controlled, by a variety of standards, specifications, codes, and constraints.[1] The various provisions of the standards sometimes form the basis for the engineer's design procedures. Often a design procedure will be an implicit application of some provision of a standard. The standards used in design may be either formal (and often legal) requirements, or they may be informal requirements, expressed as project specifications or a client's wishes. The attempt to conform to all of the governing standards influences the structure and content of the design process.

There are a number of difficulties with the incorporation of standards into programs. Currently, they are "hard-coded" (through explicit procedural language statements) into the body of design programs. Standards are subject to constant revision, and thus, they are constantly invalidating software. This makes software which incorporates standards very expensive to maintain. Standards are produced as compromises of committees, and as a result of the compromises, the standard may not have a unique, accepted interpretation. In addition, standards are subject to misinterpretation by programmers while converting the textual form into a computer processable form (programmers are usually inexperienced when compared to standards writers). Thus, it is not

---

[1] Standards, codes, and specifications are all considered synonyms in this discussion. Standards appears to be the preferred terminology and will be used throughout the text.

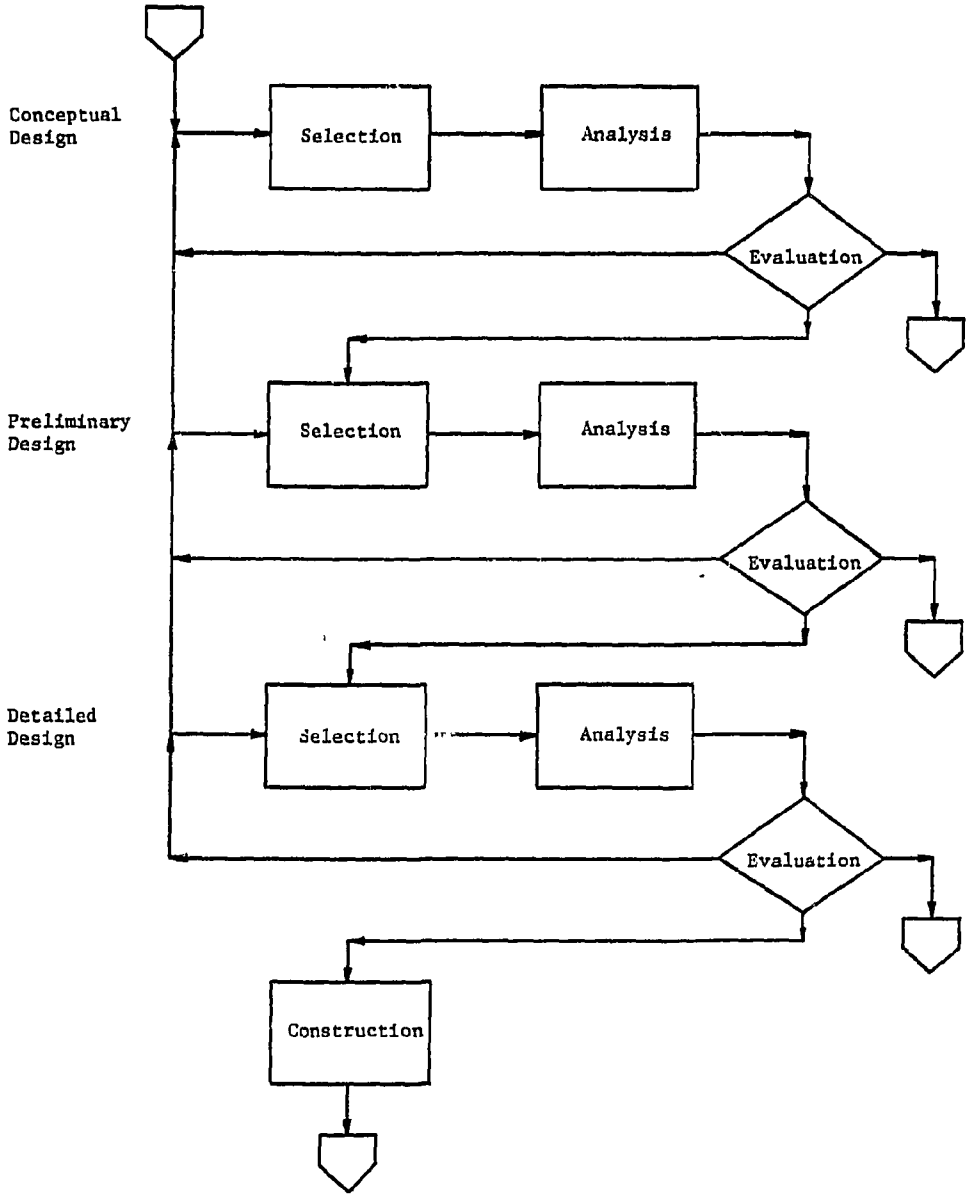


Figure 2.1. Design Loop

unreasonable to assume that provisions will be applied incorrectly, and there will be potentially serious errors of interpretation.

All design procedures require data. The results of one computation are often used as input for another. Information flows through the design process and is communicated between members of the design team. Although not commonly thought of as such, this information is the design. Problems result when engineers do not have the needed data, or if the data they do have is not correct or up-to-date. Data availability is insufficient to successfully complete a design. There is the need to verify that data and design results do not violate any constraints, and that they are not in conflict with other results and the remainder of the design. If such conflicts do exist, it can only be hoped that they will be detected before the design is completed. Unfortunately, there are no formal mechanisms for detecting such conflicts and errors. The longer they go undetected, the more difficult and costly are the resulting change orders.

Ideally, a data item will appear only once in the design data space (the data space being the set of all logical data items used in the design). A singular representation eliminates problems of data consistency and integrity. However, a singular representation of data is not always the most appropriate. There are multiple levels of representation and abstraction which are needed at the various steps in the design process.

Integrated design systems are often built as ad hoc systems; the various existing programs are pieced together to form the total system. Every program has its own set of data structures, data representations, and data needs. It is necessary to "map" the data between the various processes (integrating the processes by providing translations of data forms and content). The data mapping problem is complicated by data items which are inconsistent, or missing. Since each process may communicate with many others, there is a combinatorial expansion in the number of interconnections which must be made as the system grows. The currently available alternative of providing a centralized database (using a common data structure representation with all processes mapping data to and from the database) is not much better. The number of required mappings is smaller since it is proportional to the number of processes. However, all of the various problems of representation and missing data still exist. Both the distributed and centralized forms break down when it is necessary to change any component. The systems are tightly coupled; explicit data linkages exist (based on location, representation, and content), and these must be modified to make most changes.

Current integrated design systems do not have any information flow capability. The fundamental problem is that there are no methods to determine where data comes from, or what data will be affected by changes to other data. There is no way to determine if the correct data is being used, or if the data is consistent with known constraints. If a data inconsistency is detected it is nearly impossible to determine the effects of such an inconsistency. Similarly, if the data representations must be modified, there is no way to detect the impact of such changes.

Although computer based design systems can assist in producing better and more cost effective designs, and can eliminate some of the hand translation of data passed between individual programs, the current systems do not have the capabilities to deal with the combined procedural, data handling, standards processing, and integration problems outlined above. Systems which do not effectively address such problems are not adaptable or responsive to the needs of the engineering users. It appears that current systems do not successfully address these various problems. Thus, there is the potential for significantly better computer utilization in the design process.

2.1.2 System Description

As a result of the above situation, a project was initiated to design and implement an integrated system for computer aided design. The system was to be used for large-scale, long-term, multi-disciplinary projects, and was to address many of the problems detailed above. It was to be configured as a two level system.

The lower level was to be application independent, providing general system support software, but performing no actual design. This level would provide the data management, information flow, user and hardware interface, and standards processing capabilities to support design applications. The information flow capability included the processes necessary to determine what data was affected by changing another piece of data (data tracking). The standards processing component permitted the use of standards without directly coding them into the application programs. Standards processing would be based on decision table technology currently available [FenvS66, GoelS71, FenvS73]. By configuring the standards as decision tables, they could be treated as data to the program, and standards revisions could be accomplished by changing the decision tables. Thus, revisions would have minimal impact on the remainder of the design system. Additionally, the system would have the

capacity to integrate tasks through the database and data management facilities (through a data flow based system). This use of a common core of software to support the applications would be a major extension of the support-supervisory systems such as ICES [RoosD66], POLO [LopeL72a, LopeL72b], and GENESYS [AlwoR72].

The actual design systems would be implemented on top of the support level. The various design processing tasks, their databases, and the needed standards would be assembled into an executable program unit to assist in design. Various design domains, such as bridges, dams, power plants, buildings, etc., would each have their own separate design program, based and built on the common support software.

2.1.3 System Components

A system which could provide the capabilities described above would consist of a number of integrated subsystems. Each of these subsystems would be responsible for handling one aspect of the total problem. The following is a short description of the major components of a possible system and the tasks they would perform:

Database Management System: The database manager would be responsible for handling all data needs of the system and the applications. It would maintain all databases and provide all data access mechanisms.

Information Flow System: This is a component of the database management system which would perform all the information seeking, and data tracking to insure data correctness and consistency.

Database Definition System: Database definition permits the various components of the databases to be described, allowing the databases to be structured, created, and documented independently of any accessing process.

Standards Processor System: The standards processor would perform all work needed to check a component against any applicable provision of any standard. Whenever an action provided by a standard was needed, the design module would suspend activity and invoke this subsystem to perform the appropriate action.

Standards Definition System: This system would allow the standards to be defined and converted into their internal representation so they could be utilized by the standards processor and the application programs.

Report Generator: The report generator would be a programming tool to support the development of tabular and report output.

Graphics System: The graphics system would provide programming support for the development of graphical interfaces in application programs.

Input Language System: This is a software tool which would provide the translation facilities for user input languages for the application programs.

Multi-user Communications: The design tasks are performed in parallel by many engineers. It is necessary for them to communicate with each other, regarding the status of the project, and to resolve conflicts and errors detected by the system. The multi-user communications system would provide the necessary software to support these functions.

These components provide the basis for the support level. The support system itself includes the software framework into which all of these components are integrated.

In addition to the support system, there are a number of components which are application systems when considered from the support system point of view, but which are actually common to all design applications, and would be included as part of the support system. These components include:

Information Storage and Retrieval System: This system would be used by the engineer to access the various databases for data inquiry, and to update and create entries within the databases. It would provide a direct end-user interface to the database management system.

Project Definition System: The project definition system would be used to instantiate and control the projects of an application system. The concept of a project, and its alternative solutions, is independent of application domain. The concept is common to engineering, and structuring applications to work in a project oriented environment would parallel engineering practice.

Design Controller: The design controller would provide the executive system with which the engineers access the application tasks, and would handle all needed sequencing and control to supervise design.

Application Utilities: There are a number of engineering tasks which are quite general in nature. The alternative to each application providing the code for such tasks would be to provide a library of utilities which could be used by any application. Utilities might include structural analysis, optimization algorithms, and network algorithms.

The application tasks are developed using all of these support components. This common support software must be augmented by a body of application dependent processes, data structures, and standards. An application development system would then be used to describe the various components, their interrelations, and their structure, and to integrate these components into the complete design system. Each design system developed in such a manner could then be used.

2.1.4 Current Status

Some software is available which is used to perform design work; however, it is usually quite limited in scope. Existing programs usually operate by selecting a design from a set of possible choices within some range of parameters (by an iterative trial and error process). Many other design programs are simple graphics display programs. Some code checking programs exist, but the standards are "built-in" and the programs are invalidated by changes to standards.

A number of the basic tools used to implement such a support system as described above exist, either in engineering software systems, or as computer science techniques. These tools include: database systems, input language systems, graphics systems, and a variety of software and techniques for standards [HarrJ75a, HarrJ75b, WrigR75]. In addition, there are numerous basic engineering analysis and computational modules.

There have been some attempts at developing integrated support systems. ICES, POLO, and GENESYS are examples, but all of these have been used primarily to support large analysis applications. They provide only database management, language translations, and other run-time support features. None of the existing support systems provide any standards processing or

information flow capability. Other systems such as GLIDE [EastC76, EastC77, EastC80] include graphics capabilities, but do not address the standards processing and information flow problems. IPAD (Integrated Programs for Aerospace-Vehicle Design) [GarrC74, MillR74, BurnB78, IPAD80] is one of the most recent systems. It is the first to address the long-term nature and multi-user aspects of the problems, but it does not provide standards support or information flow capabilities.

In summary, there are many systems and system components in existence. However, none address all of the problems, and none have a technological basis which appears to be capable of addressing today's needs.

2.2 Problem B — User Interfaces for Finite Element Systems

Experience has shown that appropriate user interfaces to finite element software can have a significant increase on the productivity of the users. These interfaces are usually neglected when the software is developed. As the software for analysis becomes more complex, and is applied to larger problem domains, better interfaces will be needed, and the interfaces will become more complex.

2.2.1 Problem and Motivation

Finite element analysis is one of the major application area of computers in engineering. Users select a system based on the capabilities of that system to solve the problem, or class of problems, with which they deal. Since they desire the computer to be a tool, the selection is based, to a large degree, on the applicability of the tool. Other aspects, such as usability, maintainability, adaptability, etc., are not generally considered as prime factors in selecting a system. If considered, these attributes are used to select from different systems with comparable analytical capabilities.

Once a system is put into production for solving a given problem, the user is affected in three areas: (1) the preparation of the input, (2) the interpretation of the results, and (3) the direct execution costs. Using some particular system, and for a given solution and modeling procedure, the user has little control over the execution costs. However, the interfaces to the system can have a pronounced effect on the productivity of the engineer and the total solution costs.

It has been estimated that modeling, data preparation, and result interpretation account for 80%-90% of the total problem solving costs [HernE74] (such values are dependent of the type of system being used and the nature of the particular problem being solved). This is contrasted to the fact that 80% of the software development costs are associated with the mathematical and computational aspects of the program [HernE74], with only the remaining 20% being devoted to user features. Improved data generation and graphics capabilities are estimated to save from 40%-80% of the total costs [WilsJ76]. This situation is contrasted to other software (commercial and business systems), where interfaces and error handling are estimated to comprise over half the code [DeMiR79].

This sad state of affairs is quite understandable. Early finite element software was developed primarily in universities, or by other research organizations. The software was usually a tool used to test and implement the methodology, not to solve production problems. As such, immediate utilization was of the greatest importance. Results for the research project superseded software features. This attitude resulted in the development of a great deal of "throw away" software. This was, and unfortunately still is, particularly true in the universities. Such software was often created by a graduate student for a single project, and was then discarded because of no further need, lack of needed capabilities to solve other problems, or lack of documentation [LopeL77b, LopeL77c, LopeL79b]. Unfortunately, in some cases, some of this software survived and is now used by others, but the impact of the computational aspects remains.

As a direct result of the lack of usability of software, and of the high costs associated with the user aspects of the software, pre- and postprocessors evolved. These are after-the-fact programs, designed to enhance some part of the user interfaces. Most pre- and postprocessors are usable with only one finite element program, and provide only a limited number of features.

There have been some attempts at generalized, multi-host pre- and postprocessors, but due to the diversity of all potential hosts, it is extremely difficult to implement such a system. Each individual finite element program has its own style; ranging from simple programs which accept bulk, fixed format input for a single structure; to programs which accept problem oriented language (POL) input for substructured models. The underlying philosophical basis for the system's modeling and interface

components are so different that a pre- and postprocessor must select either (1) to implement a model and input/interface level which contains only the simplest components available in all host finite element systems, thereby eliminating the possibility of using any advanced features of any particular host, or (2) it must implement a level as high as or higher than that of any host and provide extremely complex translators to the lower level hosts. In the second case, some translations may not be possible because of the lack of particular capabilities in the low level hosts.

The types of features (model and results display, mesh generation, renumbering, etc.) found in the pre- and postprocessors are quite similar due to the structure of the majority of host systems. The underlying basis of the newer finite element systems are such that the techniques currently applied in pre- and postprocessors are insufficient to provide complete and adequate interfaces for users.

As an example, consider FINITE [LopeL77a, LopeL79a, DoddR80], which exhibits a number of user interface problems. FINITE has a problem oriented language input system, for describing hierarchically substructured models. There is a formal subsystem ("Library") for describing the input parameters, and output quantities of the individual types of elements and material models. Individual processes are written to compute element and material model quantities, such as stiffness, stresses, loads, etc., and these are then linked to the base system. The base system provides all input and output functions and allows any element to be used with any other elements, automatically. FINITE suffers serious drawbacks due to the lack of a comprehensive graphics capability. A simple mesh plotter was developed to fulfill the need for checking components of complex problems, but it is limited in that it deals only with individual substructures, and it can not handle the complex multi-level substructured model display problem.

Extensions of FINITE, via brute force programming and without the use of sophisticated support software, to display models and results is possible. Such an effort would be extremely complex and time consuming, but it would solve the problem. It appears that the basic software technology used to support FINITE has serious drawbacks and is not adequate to develop the needed interface capabilities. This is in spite of the fact that the support software technology underlying FINITE (provided by POLO) is one of the most advanced of those in use today.

The finite element interface problem is, therefore, to determine what basic interfaces need be present in the base system to provide the needed user capabilities, and what impact these features will have on the structure of finite element software.

2.2.2 Interface Description

The interface problem is now considered assuming FINITE as the host finite element system; however, the underlying concepts may be applied to any generalized finite element system. This problem is, simply: to determine and design all of the user interfaces needed to provide a convenient, effective finite element system. When complete, the new system will have all of the capabilities of the current system relative to structural modeling and analysis, plus the capability to perform graphical display of all components of the structural model and all results derived from the model.

2.2.3 Interface Components

A complete description of the design of the interfaces to FINITE is beyond the scope of this work. Rather, the following is a description of a proposed model of the system and the basic components of the solution

A proposed version of FINITE, with a complete set of interfaces, is shown in figure 2.2. The system consists of three basic groups of components. The first group is the data space used to store all problem and system data. The various processes used to model, compute, and present results, comprise the second group. The third group consists of the various processes used to interface to the external environment. The latter are dependent on the style of the interface — not the data content (problem oriented language or fixed format input and tabular or graphical output are each different styles and any can be used with either a substructured or a nonsubstructured model).

The system's operation can be viewed as a three level process. The user is at the highest level. He creates a mathematical model of the problem and obtains results in terms of this description. The problem description and the results are stored in terms of the mathematical model, the second level. The third level is the computation model which is used as the basis for performing the actual analysis. The mathematical model must be consistent with the computational model. There may be more than one mathematical model; however, the system will most likely support only a single model (this one model

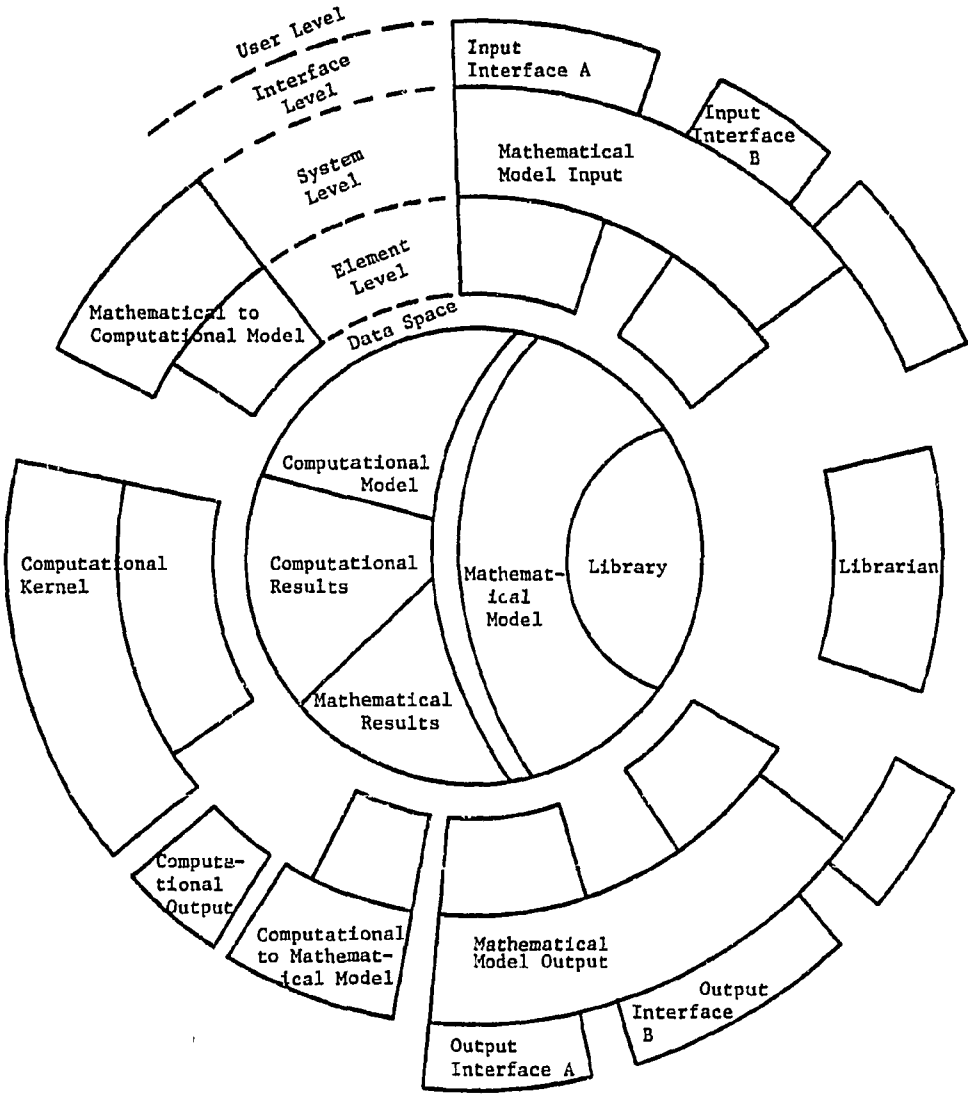


Figure 2.2. Finite Element System Configuration

assumption is used throughout the remainder of this discussion). The interfaces must also be consistent with the mathematical model, although several interfaces, each of a different type or style, are possible. In FINITE, the computational model is based on a blocked hypermatrix model. The mathematical model is one of simple multi-level substructures. The user level is a direct implementation of the mathematical model, with tabular output of model components and problem oriented language input for model descriptions and control requests.

The various components of the system include:

Data Space: The data space consists of the following different types of data groups:

Mathematical Model: This is a model of the problem being solved, and it is based directly on the user's view of the problem. It typically consists of descriptions of element types, topology, geometry, parameters, applied loadings, etc.

Mathematical Results: These are the results of the analysis, expressed in terms of the mathematical model. Typical quantities include element stress and strain resultants.

Computational Model: This is a representation of the problem expressed in the form used in the computational process. It is typically represented by the stiffness matrices and applied load vectors.

Computational Results: These are the primary results from the analysis, typically structural displacements.

Run Time Library: These are element descriptions, such as nodal degrees of freedom, which are used through the computational process.

Library: This is the complete description of the elements which is used throughout all processing steps.

Processes: The system consists of the following processes. Normal processing proceeds sequentially through the first six processes given below.

Mathematical Model Input: This process is used to support the individual input processors (model input interfaces). The process is independent of the style of the input, and is responsible for creating the mathematical model component of the data space.

Mathematical to Computational Model: This process is used to convert the mathematical model to the computational model.

Computational: This is the basic computational component of finite element analysis. It computes all quantities in the computational model and all computational results.

Computational Output: This process provides access to the computational model for output purposes.

Computational to Mathematical Results: This process converts the computational results into the set of mathematical results.

Mathematical Model Output: This process is used to support the individual output processors (model output interfaces). Each output process has its own style, and it is used to present results to the user in terms of his mathematical model.

Librarian: The librarian is used to build and maintain the library component of the data space.

Interface: The interface component provides the links between the user and the mathematical model input/output processes.

Model Input Interface: Each of these implements a single style of modeling. In FINITE, the user describes the mathematical model directly. If the system were used for a particular class of problems, such as regular framed structures or tubular pipe intersections, an input system could be developed which converts a higher level problem description into the corresponding mathematical model.

Model Output Interface: Each of these implements an output style, and corresponds to a given input interface.

Element and Material Modules: These modules are not part of the basic system. They exist for each type of element, and appear within every process to perform element dependent computations. There may be any number (fixed by the particular process) of types of these modules. Typical modules would include element stiffness, stress-strain, equivalent nodal loads, and residual loads routines. Similar functions exist for materials.

Interface Modules: It will not be possible to provide, at the system level, all the functions needed by all of the various interface systems. These modules permit the basic input and output processes to be augmented with specific routines to support the

interface developer's needs, such as special purpose data generators for particular model interfaces.

The above provides a suitable model for the various components of the system. Such a system would permit a variety of interfaces to be implemented.

FINITE users have expressed the desire for a number of additional interface and system features. From these, a "User's Wish List" has been compiled. This list, presented in appendix A, discusses not only the types of requests, but also what aspects of the current system are affected by providing such changes, and the order of magnitude of the proposed tasks.

In order to implement these requested and needed interfaces to FINITE, a number of basic support components and capabilities must be added to the support-supervisor (POLO). The following is a short description of the components needed to support the system model and the various user interface features:

Input Language System: POLO provides a token oriented language translation facility. Translation of higher level language constructs, and a system which is input device independent (supporting both textual and graphical devices) is required.

Graphics System: Portable, device independent graphics support software is needed for development of input, model display, and result display functions.

Report Generator: This tool would provide the software support for developing all forms of tabular output.

Information Storage and Retrieval System: This system would provide the end-users with the means to interrogate and modify any component of the system data space without explicit programming.

Log, System Status, and Error Handler: All handling of problem status, error handling and recovery, and the logging of system messages is currently performed in an ad hoc manner. A common set of support components for these features would provide the needed capabilities.

Engineering Data Management: The POLO data manager treats all data equally and recognizes only certain hierarchical structures. An extended data manager is needed to handle more of the types of data used in finite element analysis.

2.2.4 Current Status

There are a variety of algorithmic procedures available for implementing interfaces and performing many of the above tasks. A large number of algorithms have been published on data generators (both two dimensional, three dimensional, and special purpose), renumbering algorithms, stress averaging procedures, mesh display, etc. The problem lies in selecting which procedures are the best, and which have the most general applicability for a general purpose system.

Basic software tools for providing both graphics and problem oriented language translation also exist. Tools for tasks such as engineering report generators do not exist. However, the wisdom of using some of the tools is questionable. For example, consider the use of a machine and device independent graphics package, which would conform to the proposed (core system) standard [GSPC79, MichJ78]. There are benefits due to portability of such standard software, but there also are problems because such a system is not well suited to the application. The core system provides a number of low level graphics operations. Applications such as finite element mesh display require more abstract, higher level operations. The additional software needed to provide such operations is quite similar in form and capabilities to that of the core system. Thus, it may be appropriate to develop a special purpose package, and eliminate the redundant capabilities.

A variety of implementations of interfaces to finite element systems exist. Many of these are pre- and postprocessors. In most cases, the implementations are designed to address a number of deficiencies in the host system, and have no particular design philosophy or basis. They just exist as software to improve usability. This results in questions concerning the generality and applicability of the ideas and concepts to other systems.

There appear to be sufficient tools and techniques to provide the various features for, and extensions to, a system such as described above. However, it requires the development of a large body of software. Potentially, new and better software tools could significantly reduce this effort and simultaneously provide a better solution to the finite element interface problem.

3. SOME SPECIFIC PROBLEM AREAS

Attempts to implement solution systems for the two problem domains described in chapter 2 have not been successful. The lack of success is due to the complex nature of the problems combined with the current state of engineering software technology. There are a number of specific, fundamental problems which must be resolved in order to develop acceptable solution systems. Five of the most significant will be discussed below. Each of the specific problem areas will be treated individually, and a discussion of potential solution techniques and systems is deferred to chapters 4 and 5. The emphasis of the discussion is that of presenting a technical description of the types of problems which arise in developing engineering software. The implicit assumption throughout this discussion is that these technical problems must be successfully addressed in future software systems.

The five problems discussed are: (1) standards processing, (2) data handling, (3) control, (4) interfaces, and (5) the computer technology base. The last four problem areas are applicable to any problem domain (including those of chapter 2). Explicit standards processing does not appear in all types of problems (in some areas of engineering the explicit use of standards is not required). However, standards are a major problem area in civil engineering systems, and are required for a system which performs civil engineering design and checking.

3.1 Standards Processing

Standards have a great influence on the engineering design process. They have two basic uses: (1) compliance checking of a given design, and (2) providing a procedure for component selection. In the latter case, the various provisions often form the basis for design heuristics which are sometimes implicitly used by the engineer. Standards are usually thought of as formal bodies of provisions, such as AISC (American Institute of Steel Construction) [AISC80], ACI (American Concrete Institute) [ACI71], UBC (Uniform Building Code) [UBC76], or ASTM (American Society for Testing Materials). In many cases they are also legal requirements which must be met. Formal standards do not specify all criteria for a project; there are many informal criteria and client needs and wishes which must be combined to form

the complete set of project standards which are used to control the design. There are a number of issues to be dealt with in the computerized utilization of standards. Those of importance to this work are: (1) linkage, (2) access, (3) changes, (4) interpretation, and (5) feedback.

To date, the standards which are used are deterministic in nature. They describe explicit rules, procedures, and checks which must be made, or to which a design must conform. Reliability based standards are now being proposed. These standards are used to determine an overall measure of reliability of a system, as opposed to determining the safety of an individual component. With the exception that compliance checking in a reliability based standard is done on an overall system level, rather than at the level of each individual component, the problems of the computer utilization of such standards appear to be identical to those which arise in the use of traditional deterministic standards.

3.1.1 Linkage

Standards require data for their use. They must be "linked" to the various data structures and data present in the design space. However, standards are data context independent; they represent provisions which are applicable to any component in a given class of problems. They are used in a wide variety of contexts, and in each class of problem the data can exist in different representations (see section 3.2.3). For a given type of engineered system, there are many components, each with potentially different descriptions, which must all conform to the same standard. There must be a mechanism for linking the specific data structures and data representations used in the computer to the context independent description of the applicable provisions which constitute the standards.

As an example, consider the following provision for allowable stresses in tension members, taken from the current (1978) AISC Specification[1] [AISC80].

[1] In this section, Specification is used to denote the AISC Specification [AISC70, AISC80].

1.5.1.1 TENSION

Except for pin-connected members, F_t shall not exceed $0.60F_y$ on the gross area nor $0.50F_u$ on the effective net area.

For pin-connected members: $F_t = 0.45F_y$ on the net area.

For tension on threaded parts: See Table 1.5.2.1

The provision explicitly references (by symbolic name as used in the nomenclature of the Specification): F_u (ultimate stress), F_t (allowable stress in tension), and F_y (yield stress). The provision requires the net area (A_n) and the gross area (A), although these data items are referenced by generic name. The applied load (P) may be implicitly required for the computation of F_t. Also, each part of the provision is dependent on the conditions of "pin-connected" and "threaded parts." Proper utilization of this provision may require that all or part of these data items be known.

The nomenclature to the AISC Specification lists about 150 items, but over 400 different data items have been found to exist [FenvS69]. Any application program using this standard must be able to find and access all of these generic items for the design or checking of actual components.

3.1.2 Access

The access problem is one which is of concern not only in computerized standards processing, but also in the manual usage of standards. Basically, the problem is that of knowing what provisions of the governing standards are applicable to any step in producing the design. How does an engineer know that a provision exists or should be applied in a given case? Such information is not explicitly given in the standards.

For the provision shown above, the engineer must know he is dealing with a member that is governed by tension. How does he determine that the bending stress in a truss member is negligible, or that axial stress in a beam does not require that it be treated as a column, and that the corresponding provisions of the Specification are not utilized? The alternative is to exhaustively check all provisions in the Specification.

3.1.3 Changes

Many of the problems associated with standards would not exist if standards did not change. If they were invariant, the various linkages and access paths could be hard-coded into programs without serious consequences. Standards are currently used in programs in this manner. When a new edition of a standard is produced, all existing software based on the old version is invalidated. It is desirable to rapidly incorporate changes to standards into existing software with minimal impact on the software.

The provision shown above has changed from the previous (1969) version of the AISC Specification [AISC70].

1.5.1.1 TENSION

On the net section, except at pin holes:

$$F_t = 0.60F_y$$

but not more than 0.5 times the minimum tensile strength of the steel.

On the net section at pin holes in eyebars, pin-connected plates or builtup members:

$$F_u = 0.45F_y$$

For tension on threaded parts see Table 1.5.2.1.

The change is not a simple modification of the factors-of-safety, or of the equations used. It is a philosophical change, safety against yield is now based on the gross area, as opposed to the net area of the member. The change appears as a modification of the explicitly required data (gross area is implicitly required for the computation of net area, but this computation may be performed externally to the utilization of the provision). Changes to a program would require the addition of a new datum, gross area of a tension member, which was not present in the previous Specification. Such changes may be extremely difficult to implement when standards are hard-coded into application programs.

3.1.4 Interpretation

Standards must be translated from the written textual form to some computer processable form. The standard's developers are experts in their field. Typical software implementors are junior engineers or scientific

programmers (because of their knowledge of computers). These people do not have the expertise and experience to develop programs for standards processing without the risk of misinterpretation of the standard. The standards developers, standards experts, and experienced engineers are often too busy addressing complex engineering problems to devote their time to assist in software development (the problem of lack of commitment of experienced personnel to assist in software development, due to these people's apparent misconception that they are not important in the development of software, exists in most areas of program development [JensR79]).

The source of the interpretation problem lies not only with the software implementors, but also with the standard's writers. The form and style the writers use in developing the original standard is the cause of some problems [HarrJ80].

Interpretation errors may result in the incorrect encoding of a provision, access to incorrect data, or utilization of the wrong provisions. Equally as serious as the original interpretation problem is the lack of methods to verify the resulting software.

3.1.5 Feedback

Compliance with a deterministic standard is typified by a binary result: provision satisfied, provision violated. The actual usage is not so simple. When a criterion is violated, it is necessary to know why the criterion failed, and what changes are needed to satisfy the criterion. For example, if a tension member is unsatisfactory there are two possible causes, based on the yield stress and the ultimate stress criteria. For either failure mode, three different alternatives exist: reduce the load, increase the area, or increase the steel strength. At any step, the list of potential alternatives can become long and complex. Guidance is needed to determine which of the alternatives are plausible and likely to be successful. Similarly, if the check is successful, it may be the case that another, more economical, design would also be satisfactory.

The test of compliance of the design of a component with a provision is not an absolute test. For the provision shown, with steel (A36) with yield stress (F_y) of 36 KSI, the basic allowable stress limit is 21.6, but the Specification itself rounds the value to 22. An engineer judges all results within the range of acceptable engineering accuracy. The computer will perform only absolute numerical checks. This engineering accuracy is also

context dependent. When the engineer knows that a criterion does not govern or that the consequences of failure are reduced, he is likely to increase the range of what he judges to be acceptable. It is nearly impossible to build this type of judgement into the types of programs which are in use today.

The same type of problems exist for reliability based standards. For such standards, the individual compliance decisions are deferred until the entire system is checked, but the same types of judgements are needed. The problem of feedback to determine why an unsatisfactory probability of failure exists, or to determine a more economical design, are the same for this type of standard. Since more computations and more components may be involved in such a system check, the process of determining what alternatives to pursue may be more complex.

3.2 Data Handling

The engineering process deals with the creation and the manipulation of the data and information which describes and models the system being engineered. The basic cause for many of the data handling problems is the fact that the data is a resource which is independent of processes, but is only of value to particular processes. Problems result from, and are related to, this "corporate" nature of the data; the data logically "belongs" to the design, not to an individual design process or designer which uses the data. The data has traditionally been maintained by individuals and individual processes. The data is treated in the manner which is most appropriate for the individual involved, and the impacts of such data handling on others are not considered.

There are a number of specific problems to be dealt with in data handling. These include: (1) how to propagate the data through the design process, (2) how to be sure the current, correct data is being used, (3) how to represent the data, (4) how to integrate data and processes, and (5) how to access data. Each of these problem areas are discussed separately, although they are all interrelated.

3.2.1 Information Flow

Data is produced and modified by various processes. It is used (consumed) by other processes. This data is the design, and it moves and flows from one process to other processes which require the data. To

integrate processes, it is necessary to integrate the data and provide the mechanisms for the data to move from process to process. The other data handling problems all result from the attempts to produce an integrated system in which the information flows between processes. Capabilities must exist to support and provide this information flow.

3.2.2 Consistency and Integrity

The data used in design should be correct and up-to-date. A major problem in engineering is the performance of work based on the wrong or incomplete data. This produces errors and unsatisfactory designs. In some cases these errors are detected before the design work is completed; in other cases after-the-fact changes must be made. The problems of consistency and integrity deal with: (1) who currently has the data, (2) who changed or created the data, (3) is the data correct, (4) how to keep the data current, and (5) what the change of a data item implies to other data. If the "owner" of the data (the individual or process which is currently responsible for maintaining the correct value of the data) is known, then it is possible to access the data when it is needed by some other user or process. Knowing who changed the data permits placement of responsibility (and blame), and permits one to query the responsible individual to determine the rational used to obtain the current data value.

The attributes correct and current are difficult to characterize. The effects of incorrect or out-of-date data are known, but the problem is to determine if the data is correct or incorrect. Since data is dependent on, and derived from other data, a change implies that the derived data is potentially wrong. An information flow capability is needed to determine what data is affected by a change of some other data, and judgement is needed to determine the effects of such a change (recomputing some data due to a change in some other data item, every time a value changes, may not be required, and may be very costly). Unfortunately, in current engineering software systems, there are no mechanisms to attack these problems.

3.2.3 Data Representation

Data is usually associated with one major process, and this process determines how the data is organized and stored — its representation. There is both a logical and a physical representation of the data. Consider, for example, nodal loads from finite element analysis. Logically, these consist

of a force vector (direction and magnitude) and a location (node). Physically, this may be represented by a set of magnitudes of every degree of freedom component at every node stored as a vector of length equal to "Number of Degrees of Freedom per Node" times "Number of Nodes." An alternative representation is a data structure consisting of a node, direction, and magnitude for every specified load component.

In conventional programming practice, once a representation is selected, the access mechanisms for that representation are coded directly into processing modules. Other processing modules (either new, or replacement modules) will need the data at some future time. Their needs often will be different from those of the existing processes, and the selected storage representations (either logical or physical) are often not appropriate.

Consider the topology of a finite element mesh as an example. Various elements are associated with various nodes in the mesh. Each element has a list of the nodes upon which it is incident — the element incidences. The information is often logically represented as a list of nodes associated with each element. This is quite natural as the element matrices are often produced on an element-by-element basis, and then assigned to the global matrices through the incidence mappings. For a process such as stress averaging, it is necessary to know what elements are incident on each node — the nodal incidences. These two sets of data are the inverses of each other. Either can be determined dynamically from the other as needed, or an alternative data representation can be created, and the data duplicated in both locations. Selecting a single representation may require the repetitive use of complex code to transform the data when needed. The alternative of multiple representations is subject to problems of data consistency.

As more processes are involved, the selection of the appropriate representation becomes more important to insure efficiency, to insure that all potential accesses are possible, and to insure that the stored data is what is really needed by all of the accessing processes. An incorrect representation will restrict what can be done with the data. Mechanisms are needed to select proper representations and to isolate the physical representations from the logical representations.

Proper techniques for selecting data representations and constructing data access procedures permit applications to deal with data in an effective manner without requiring extensive, complex programming due to changes of representation. Current data management techniques applied to engineering software systems do not provide such capabilities.

3.2.4 Process Integration

To integrate individual processes into a complete system the processes' data must be integrated. The data can not always be in the proper representation. Each process or subsystem will have its own desired or required form. The data must be transformed or "mapped" between the various forms as it flows between processes. Many attempts at integrating programs address this problem. In manual usage, data is output from one program and transcribed, by hand, to the form required for the next program.

The most common integration alternative is a tightly coupled "N x N" system. In this scheme, each process has its own data storage and data representations. The processes each communicate with all other processes, with a data transformation occurring on any data communication path, as shown in figure 3.1.a. Such an integration is very complex. The number of mappings grows combinatorially with the number of components. A change to one process requires the modification of "N" mappings.

An alternative is to provide a single common representation. The data may either be distributed and stored with the individual processes, or it may be stored in a central database, as shown in figure 3.1.b and figure 3.1.c. One transformation is needed between each process representation and the common representation.

There are a number of problems with both alternatives. One of the most basic, for which there is no simple or convenient solution, is missing data. A process may need some data item which does not exist in a database, but which is logically associated with, and should be produced by, some other process. The process which should be responsible for creating the data may compute the data item and use it within its own processing. The data is computed by this process "on-the-fly" as a temporary quantity, but it is not "exported" to any other processes, and it is not available for other processes.

Another problem is the transformation of the data between representations. The mapping may not be possible; additional nonexistent data may be required to complete the transformation. Alternatively, the mapping may be very complex, not a simple one-to-one transformation. Some physical representations, such as the nodal loads example above, are straightforward. The mappings involved in the topology representation are much more complex. If multiple representations are present, how can it be insured that the data is consistent in all representations? If the data is changed in one location it must be changed in all locations.

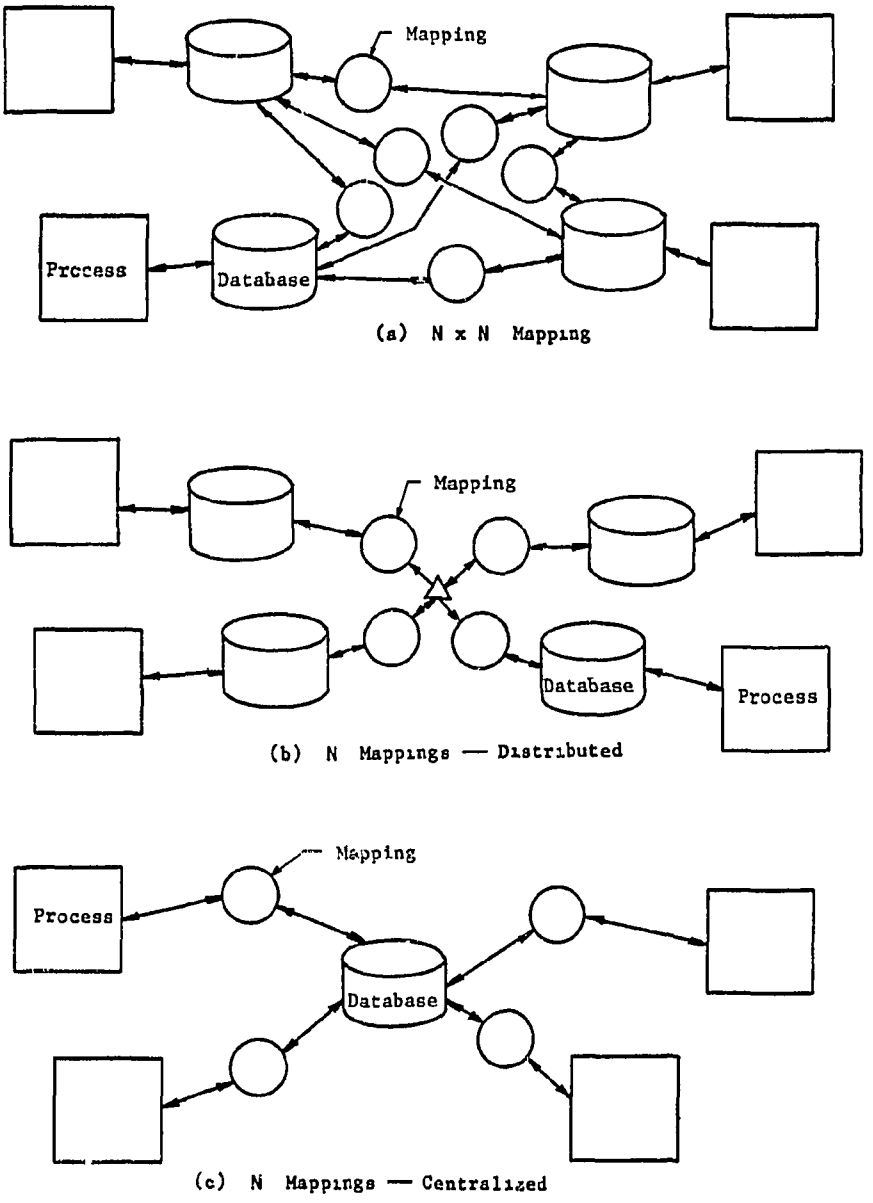


Figure 3.1. Process Integration Configurations

If multiple mappings exist, there are questions of completeness and symmetry; can any data item be transformed from any one form to any other form (is it possible to transform representation A to A', but not A' to A)?

In the centralized form, there is a problem of selecting an effective common representation for the data. The chosen data representation must support all required data accesses. There is also the problem of efficient access. A single form usually implies a single access path. In a multi-user environment, this is a potential bottleneck.

As with selecting a data representation for a single process, selecting an integration scheme can have a significant influence over the remainder of the system (the system design, performance, development, and maintenance). Effective techniques are needed to address these issues.

3.2.5 Context and Access

Data represents a particular problem. Individual processing steps are general engineering procedures, are independent of the particular problem, and often are independent of the class of problem. Some data, such as descriptions of standard components, is independent of any problem and may be used by many processes. In applications, data is usually stored in a manner which links the data to the particular problem being solved. Thus, if one has a procedure which designs a beam in a single building, and a project involves two or more buildings, an ambiguity exists (the context of which building in the project is not considered in the data access mechanisms). There must be some mechanisms for accessing the data and augmenting the context independent process descriptions to obtain the correct data. The current approach of building data access paths into programs defines the context in which the data is used. A coupling between problem dependent data and generic processes exists, and this coupling can not be changed without changing the program.

In current programming languages, this context coupling is accomplished through explicit linkages, either through calls to data management routines in which context dependent information is used to access data, or through the use of subscripts (representing context) in addressing data structures. Because of the explicit linkages, the processes can not be used in any other context, and they loose their value as generic procedures. A mechanism is needed to permit the dynamic linking of the generic processes to specific problem dependent data.

3.3 Control

Control deals with the issues of presenting the engineering design process in the form of algorithms. Two basic problems are present: (1) determining the actual algorithms, and (2) presenting the algorithms to the computer.

3.3.1 Design Algorithms

What are the algorithms for design? Design is a complex process, and unlike analysis, no specific procedures for performing design exist. There is a general procedure of selection, analysis, and evaluation, but beyond this level, the design process is ill-defined and unstructured. Each engineering firm has its own general procedures for attacking a project. Each individual engineer has his own personal process for design. Design algorithms are not taught, or explicitly available. They are acquired through experience and through observations of how others design. They often consist of assumptions, guesses, intuition, and implicit applications of analysis procedures and standards.

Consider the design of a plate girder as an example. The engineer may automatically assume an initial web thickness of 3/8", simply because he previously designed a girder for a similar span and loading condition, and that was the final result. To compute the remainder of the section, he may assume an allowable steel stress of 24 KSI. In doing so, he is: (1) assuming ASTM A36 steel, (2) assuming that bending governs the design, (3) assuming a compact section, (4) utilizing a provision of the AISC Specification which states allowable stress in bending is 0.66 of the yield stress of the material, and (5) utilizing the ASTM standard for A36 steel which provides a yield stress value of 36 KSI. He is utilizing his prior experience along with assumptions and implicit applications of standards.

Alternatively, consider the following quote from a structural design text [Gaylord72].

> Shear may determine the design of beams which support heavy concentrated loads near reaction points and of very short (small values of L/d) beams uniformly loaded.

This presents a provisional rule. There is a degree of uncertainty in what constitutes heavy, concentrated, near, and short. In design, the engineer is continually dealing with such descriptions and processes, and is successful in utilizing them to produce complete, detailed designs.

In analysis, the details of the computations are simple, well-defined, and generally lead to a direct set of processing flow paths. In design, the paths are more complex, more interconnected, and it is often difficult to determine how to select a given processing path from a number of alternatives. Additionally, there is the problem of starting the design process. Usually some initial guesses are required. The procedure commences at some point, it proceeds iteratively, and then is terminated when the design is judged satisfactory by the engineer.

In an integrated design process, there are problems resulting from data flow and timing. Some procedures require that certain data items be available before they can proceed. As the number of procedures increases, the degree of interconnection grows and can not be readily determined. Two conditions, termed contention and race, are possible. In contention, process A requires some information produced by process B, while process B is dependent on some other data produced by process A. The processes deadlock in contention for the original values of the data. Once the data is determined a race condition results. Process A can proceed but will affect the results of process B which will affect A, ... In manual processing, these types of conditions do not occur. The engineer will obtain needed data to perform the computations, and will delay determining the effects on other data until the process has terminated. Additionally, he will recognize iterations, and will make judgemental decisions regarding convergence, or if iterations are appropriate.

Engineering design is a loosely structured problem dealing in uncertainty and requiring significant experience and judgement. Such judgement and experience are difficult to codify. Conversion of design processes into computer code is difficult, if not impossible, using current techniques.

3.3.2 Presenting the Algorithms

Once an algorithm has been defined, there remain the problems associated with presenting the control processes to the computer. Simple procedures consist of only equations to be evaluated. Support code must be added to provide input, output, and resource control. As processes grow more complex, simple equations are not sufficient. Simple mathematical expressions such as

$$[K]_e = \int_v [B]^T [D] [B] \, dv$$

can not be expressed directly and require at least several lines of code to perform the integration, in addition to all the code needed to form the base matrices (In FINITE, the size of a complete element stiffness module doing

such a numerical integration ranges from 1000 to 3000 lines). As the complexity and size of the problems grow, the memory resource limits of the computer are reached. This necessitates more code to move data to and from secondary storage. Soon the computational process "gets lost" in all the support code associated with the details of managing the data and providing the primitives for the computations.

One solution to this dilemma is to provide "packages" of code to perform many of the common functions (i.e., disk I/O, memory allocation, matrix manipulation, etc.). This results in code consisting of numerous calls to subroutines. This approach allows the programmer to become more productive by eliminating some detailed coding, but the original problem simply reappears at a slightly higher level of abstraction. The complexity of the programs using such schemes soon exceed the skills of the programmers. All their effort is spent in trying to manage resources efficiently, and not to solve the real problem.

Support-supervisory systems like ICES and POLO were developed to overcome this problem. They provide a higher level language in which all control is programmed. The support software provides data and memory management functions so that the programmer need not be concerned with such details. However, even with these systems for support, the program (in the higher level language) often becomes lost in the supporting code, particularly when manipulating complex data structures. The programmer loses sight of the real problem.

The data and processing primitives are still at a low level when compared to the complexity of the processes. As a result, the algorithms are difficult to implement, with a great deal of the development effort applied to issues other than the desired procedure. Such code is difficult to maintain, to check, to modify, and to enhance. Alternatives to the explicit programming of all the details of the algorithms are required.

3.4 Interfaces

As stated in section 2.2.1, interfaces are often neglected in engineering software, even though they have a significant impact on productivity. There are two basic problems to be resolved in providing the interfaces: (1) what style of interfaces to provide, and (2) developing the software to support the interfaces.

3.4.1 Form and Style

There are a variety of forms and styles for interfaces. The objective is to be able to communicate the maximum amount of information in a manner best suited to the user. The interfaces should be flexible. The user should be able to direct the input rather than having to respond in a predefined sequence; the user should control the process, not be controlled by the program. Similarly, for output, the user should be able to select the style (tabular, graphical), content, and order of all data presented. Looking at thousands of lines of output to select a few numbers is inefficient, tedious, error prone, and nonproductive.

For input, the most common form currently used in engineering software is fixed-form bulk input. This is the easiest for the programmer to provide (it can be supported directly through programming language features), but is the most restrictive for the user. The other extreme is natural language voice input. Such input systems currently require dedicated computer facilities and are state-of-the-art research. However, they do provide the fewest restrictions; consequently they also provide the greatest possibility for ambiguity. In between these two extremes are a variety of forms. Problem oriented languages (POL's), menu driven graphical systems, and question and answer systems are the most popular forms. At first glance, question and answer systems appear to be useful and convenient for the infrequent user. Basically, the user is prompted and lead through the input process. For frequent users of a system, the prompting systems are boring. In addition to the boredom, there are other serious drawbacks. The worst of these appears to be that there are no user controlled mechanisms for backtracking and correction of erroneous input. Menu systems are graphics oriented; the user points to one of a set of possible alternatives. They are more flexible than prompting systems, but the number of potential responses is limited to the viewing area of the screen. POL's are the most flexible of the three. Users communicate in a subset of natural language, with restricted syntax. The vocabulary is tuned to the user and the problem domain, and the user has more freedom to direct the process than with the other schemes.

The form of the interface is dependent upon the data requirements. For input, the data may be either (1) processing and control requests, or (2) problem and model descriptions. The latter are generally more voluminous and require more complex input forms to reduce input requirements.

Similar problems exist for output. Here the two major forms of output are tabular and graphical. Tabular output is typically provided through programming language features. As such, the user generally has little control over content or order. Often there is no way to suppress unwanted portions of the output. Graphical output is much more desirable and flexible. It requires additional processing, but yields results which are more readily interpreted by the user. However, for graphical output, it is difficult to determine the manner in which to present the data. Thus, there is the need to provide capabilities to change the presentation of results for the user. In addition to these bulk output forms, there is the need to present status information, messages, errors, etc.

Increased user productivity is possible through appropriate interfaces. However, such interfaces are complex and require careful design to insure they provide the necessary capabilities and are truly useful.

3.4.2 Techniques

Only the simplest interfaces can be implemented by facilities provided in current programming languages. Development of interfaces is extremely complex and requires the programmer to consider and resolve many issues such as device dependency, terminal access procedures, and response criteria. To eliminate these problems, a variety of software tools have been developed or proposed in order to isolate many of these issues. The application interfaces are then built using these support tools.

Support software is available for graphics [GSPC79] and input language translation [RehaD79]. However, in both instances the tools provide only a basic level of support. Development of the sophisticated interfaces needed by the applications requires a significant effort. The primitives provided by the various tools do eliminate much detailed programming, but the level of support provided is such that a significant amount of complex, and often repetitive, code is required.

For example, graphics systems provide only basic drawing primitives for line segments, characters, and viewing transformations. Development of a complete graphics facility for FINITE would require support software to permit the programmer to handle more abstract concepts such as surface function plots (contour plots) or arbitrary cutting planes. To produce a contour stress, load, or deflection plot of an arbitrary cutting plane or surface of a

structure requires a complex program, with most of the detail being associated with producing proper line segments for display. An ability to handle the more abstract quantities at the support level is required.

Current technology provides only a first level of software support. There is a need to develop additional support which will allow system developers to deal with the engineering nature of the applications directly, without having to first translate all actions into device and basic operation oriented algorithms. Additionally, there is a need to develop tools in other areas such as tabular output and error handling; currently these functions are being provided in a totally ad hoc manner for most engineering applications. A complete set of tools and techniques for interfaces would enable programmers to provide more effective interfaces with significantly less development effort.

3.5 Computer Technology Base

The computer field continues to undergo a period of rapid technological development. There is a continuing revolution in hardware, languages, and systems. The effects of the ongoing computer revolution in these areas is presented below. The basic problems in all these areas arise from the ongoing changes.

It is necessary to configure engineering software systems so that they are adaptable, or, due to the rapid changes, they may be outdated before they are operational. The problem with all three aspects is to know what to select, and how to be prepared for future changes. Complete computer software systems are extremely costly to develop. The costs of constantly redeveloping software due to technological changes in the supporting systems is not acceptable.

One would like to ignore as many of the issues of selecting hardware, languages, and systems as possible. In fact, most of these issues should not be of concern to the end-users or to the application program developers. However, to achieve this isolation, these issues must be dealt with in the system software. Unfortunately, with current software technology, there are no formal mechanisms to deal with these problems and to minimize the effects of change.

3.5.1 Hardware

Semiconductor technology has caused the most dramatic changes in computer technology. The largest hardware systems of less than twenty years ago cost over a million dollars and occupied a room. Today the same processing capability is available on a single chip, costing less than ten dollars. A complete microprocessor system requires only a few chips, sits on a table top, and costs only a few thousand dollars. On the other end of the scale, there has been the introduction of the super computer, machines with extreme speed, currently approaching a billion operations per second. Additionally, there have been continuing advances in peripherals. The end is not in sight; prices continue to fall and the capacity of a single chip continues to increase.

The future of hardware to support engineering software is unforeseeable. Engineering software must be efficient and portable, even though the nature of hardware is contradictory (incompatibilities and inconsistencies exist between various hardware manufacturers and these limit portability). The lack of concern for portability and efficiency issues has plagued prior systems. Effective, adaptable, long-lived software must be conceived and designed to deal with the indeterminable nature of the host hardware.

3.5.2 Languages

New computer languages continue to be introduced. Large numbers of languages are continually developed for research, experimentation, and teaching purposes. The majority of these do not become widely accepted, due to resistance to change from users, and due to their lack of portability, distribution, and support. The new languages provide a variety of ideas and techniques, and one wonders how long the current mainstays, FORTRAN and COBOL, will continue to flourish. Possibly the most significant change in this area is just emerging with the introduction of Ada [DoD80] by the U.S. Department of Defense [DoD].

The choice of a language can have a significant effect on the ease of development and reliability of software. Engineering programming currently implies FORTRAN. Should programming in FORTRAN continue with the acceptance of its deficiencies (particularly with respect of data structuring facilities) in exchange for language acceptance and portability, or should there be a switch to a new language and risk a premature end of life of the application

programs due to the death of, or lack of support of, or unavailability of the base language? The benefits of any alternative language must be weighed against the potential costs.

3.5.3 Systems

Original engineering programs were designed for operation in the batch environment, the only alternative. Then came the addition of time-sharing. A number of other choices now exist, including: transaction processing, networking, and distributed systems (these are described in more detail in the glossary, section 5). On-line, interactive computer utilization for engineering is essential. The choice between centralized, networked, and distributed systems must be made. Distributing data and processing leads to problems of interconnections and access. Centralized systems have a potential for bottlenecks. The problem with selecting a system form is similar to those described above; what is the proper technology to select to provide the best support for engineering applications.

4. TECHNIQUES FOR ENGINEERING SOFTWARE SYSTEMS

There are a number of techniques which appear to have promise in developing advanced engineering software systems. These techniques potentially provide a means to address and solve many of the problems presented in chapter 3. Five techniques which appear to be most valuable are: (1) relational database management systems, (2) use of context, (3) knowledge based artificial intelligence systems, (4) virtual computer models, and (5) alternative programming languages. Each of these will be discussed separately: with respect to the technique, the problems which are addressed by the technique, and the potential advantages and disadvantages of using the technique. Due to the very complex nature of some of these areas, more detailed background information is provided in the appendices.

4.1 Relational Database Management Systems

Database management systems (DBMS) are used to fulfill a variety of the data handling needs of software systems, and allow the system's developer to concern himself with the engineering problem to be solved without dealing with all the details of data storage, data representation, and data manipulation. Database management systems provide a software package which is the interface between the applications program and the physical storage system. This software allows the programmer to deal with data on an abstract logical level, rather than at a physical level. Relational database management systems are the most recent development in this field. Additional information on database management is presented in appendix B.

4.1.1 Background

In the early sixties, database management systems evolved from report generators and disk management systems [FryJ76]. The database management systems were introduced to provide mechanisms to reduce program complexity and development effort. Attempts to integrate programs had lead to difficulties due to data representation and storage. Database systems were introduced to eliminate these difficulties. Three distinct types of database management systems have evolved: (1) Hierarchical [TricD76], (2) Network (CODASYL) [TaylR76], and (3) Relational [ChamD76, MichA76, KimW79]. A variety of

implementations now exist, but most are oriented towards business applications. Within engineering, the applications often had no database support. If some type of support was present, it was usually ad hoc disk management routines, or a special purpose engineering hierarchical database manager.

The use of a database management system helps, but it does not eliminate all of the problems encountered, and it does introduce some new problems which must be resolved. The relational database model is the latest and most advanced technology available. It appears to provide a number of features which reduce data handling problems.

4.1.2 Problems Addressed

Database management systems are used to reduce the magnitude of the data handling problems associated with process integration and data representation. All of the information in a database is created, accessed, and maintained by a single system. As such, a data representation which is best suited to all applications can be selected and used. No application "owns" the data, but all access it from the database. The database manager allows each application to have its own "view" of the data, permitting the programmer to work with a subset of the data without knowing all the representational details of all of the data. This feature, along with a set of common data operators and support functions, such as concurrent access control, permits the applications to be data independent; details of data management is the sole responsibility of the database management system.

The hierarchical and network systems both have two distinct levels for describing the data. The lowest level is the "data mapping level," where the data structures of the databases are described in terms of their physical organization and structure. The higher level is the "data definition level," which defines the various components of the data structures within the database and the relations between the components. The user (programmer) deals only with this higher, "logical level." The data is stored in logical records, and there are certain logical interrelations between data items and records. These relationships are represented in the logical organization and representation of the data in the database. To access or change the data, the user must explicitly deal with all aspects of the data representation: (1) content of records and data structures, (2) linkage (hierarchies, networks, pointers, etc.) between data structures, and (3) order of records

within data structures. As a result, the applications become tied to the "representational detail" of the database, and can not deal with the data in an effective manner. The applications are cluttered with code to access and manipulate the database, not the data in the database. The database organization can not be changed because the information about access is built into the accessing programs. No matter what the programmer tries to do, he is seldom dealing with the actual data. Rather, he is always explicitly operating on the logical organization of fields, records, and data structures, and is specifying all of the details of all representational manipulations. Such database systems are "representationally addressed." Although the database eliminates much of the detailed programming and provides a common data representation as a basis for integrating applications, the data handling problems (as described in section 3.2) are not all resolved. Relational database management systems attempt to solve the remaining problems by providing another level of data independence.

4.1.3 Advantages

The objective and advantage of the relational approach is the attempt to eliminate this last level of data representation dependence described above. In the relational model, the user deals only with the data, not its representation. The database is "content addressed." There is a logical content of data groups. The user can request any information in any manner he desires (by specifying the content of the requested data), and is presented with the data in the form of a relation. The access interface to the data, and the form of the data, is independent of the data organization and representation. The physical database structure is unknown to the user. The selection of storage and access mechanisms can be determined by the database management system. The definition of different views of the data can be constructed in a hierarchical layered fashion (relations defined in terms of other relations), allowing the user to treat the data in the manner he desires, independent of the form selected for the database.

The following is an example of the data management statements used with the POLO hierarchical database manager to access the element incidences from the mathematical model data structures in a finite element system. The example is based on the data structures used in FINITE (the statements correspond to the actual data structures used in FINITE, and these data structures were chosen for efficiency in a particular type of access, and the

example depicts the bias in data structure representation). A conventional procedural language embedded data manager would require several more lines of code than that required by the higher level data management commands supported by POLO. The first two commands deal only with the data representation; the actual data is not accessed until the third command.

```
GET_VECTOR    ( INCIDENCE_VECTOR,
                MATHEMATICAL_MODEL ( ELEMENTS, STRUCTURE_NAME,
                      INCIDENCE_POINTERS, ELEMENT_NUMBER ) ),
GET_POSITION  ( INCIDENCE_POSITION,
                MATHEMATICAL_MODEL ( ELEMENTS, STRUCTURE_NAME,
                      INCIDENCE_POINTERS, ELEMENT_NUMBER ) ),
GET_INCIDENCE ( MATHEMATICAL_MODEL ( ELEMENTS, STRUCTURE_NAME,
                      INCIDENCES, INCIDENCE_VECTOR,
                      INCIDENCE_POSITION ) )
```

For a relational database management system, the request which corresponds to the example is shown below. This request is independent of data organization and structure.

```
SELECT     INCIDENCES
  FROM     MATHEMATICAL_MODEL
    WHERE      STRUCTURE = STRUCTURE_NAME
     AND       ELEMENT   = ELEMENT_NUMBER
```

There are four distinct advantages of the relational approach: (1) simplicity, (2) data independence, (3) symmetry, and (4) theoretical foundation [ChamD76].

Simplicity: The user has only the single relational tuple data structure to deal with. All accesses are independent of storage organization, and the user deals only with data tuples, not access mechanisms or access paths.

Data Independence: The details of the storage structure are unknown to the user. Thus, the storage structure can be changed without affecting any applications. Anyone can access any data, simply by knowing that the data is present in the database. Applications are independent of the details of the data organization.

Symmetry: If the data is stored in some record oriented manner, then there must be a traversal of the records and the links to access the data. For certain requests, which do not map directly onto the data structure, complex programming is needed to obtain the data (e.g., going from element incidences to nodal incidences as described in section 3.2.3). This complexity limits the accessibility of the data and may imply serious performance problems. In the relational approach, since access mechanisms and data organization are hidden, any request can be formulated directly, and all requests are handled equally. The database is "symmetric" with respect to data access.

Theoretical Foundation: The relational model is based on the mathematical theories of relations and predicate calculus.

The first three advantages address the data representation dependence problems of prior database management systems. The last provides a formal basis for the concepts utilized in the relational model.

4.1.4 Disadvantages

Relational database models are a new and rather untested technology, with a number of questions concerning the viability of such systems. There are no very large databases which have been developed using such systems, so questions of effectiveness in large applications have been raised. Most relational systems have been developed to do research in the design and use of such systems. Major implementations are just being released [IBM81a, IBM81b]. Thus, there is no large body of experience of use in the production environment as there has been with the other database models.

There are also some questions concerning the operational speed and efficiency of relational systems. The majority of the work of the database administrator in using nonrelational types of database systems has the objective of determining the data representations and access paths which will be most appropriate for all users. The optimality criteria which are used in the selection of the data organization are: (1) speed of access, (2) minimum storage transfers, and (3) minimum storage space. Since the database management system has control over selecting the physical representations and access paths in the relational model, the system may not select the appropriate representation or access mechanisms, and the result may be

unacceptable performance. It is hoped that the magnitude of the optimal data representation and access path selection problem will be such that, for large systems, the machine can produce a solution which has overall better performance than one developed by a database administrator. For any individual access, a "hand tuned" system may be better, but for a very large system, the number of accesses will become so large that hand coding and hand tuning can not be considered, and on the average, the database management system will do a better job. (this is similar to the argument for use of higher level programming languages as opposed to assembler languages).

Perhaps the most serious question involving the use of the relational model for engineering applications is the question of available data primitives. Relational systems have been developed for information retrieval and business applications, and the data primitives are usually only names, integers, booleans, and character strings. In fact, some relational systems do not support real numbers. Primitive data types such as reals, integers, characters, and booleans, and other engineering data types and data primitives such as vectors, matrices, tensors, etc., are needed in engineering applications (such higher level data types are not currently supported by standard database management systems). The lack of such data types will be restrictive, making it difficult to develop programs which require such data types.

4.2 Context and Scope

Context and scope are not techniques, but rather, they are concepts. They are based on the methodology used to solve engineering problems, and are dealt with in an ad hoc manner in most applications. The formalization of the concepts appears to be of value in solving some of the data handling problems.

4.2.1 Problems Addressed

Analytic engineering processes and standards are usually context independent (in some fields such as nuclear power plants an individual standard may be developed for a single project). As stated in section 3.1.1 and 3.2.5, processes and standards can be applied to any problem or project by using the appropriate data. Their application requires the addition of context. In applications, this context information is presented in the form of data subscripts. In programming languages, the various data items needed

by the processes are stored and grouped in data structures which are addressed by subscripts ("subscripted") to indicate what part of the data is needed. In access to databases, a similar method of subscripting the data structures is used to obtain the correct information for the processes. This subscripting is explicitly built into current application programs. Thus, the application must have context information scattered throughout the processes. Any change of context requires recoding the processes. The concept of context is to separate the context information from all processes, just to use the generic processes. Context information would then be declared externally to the processes. The data management system can be extended to include a formal context system, and the database manager would augment data references with context information to obtain the correct data.

The concept of scope is also based on current procedures, but scope is more abstract. The data which is used in any processing step is dependent upon the type of process. In design, the same type of information is needed in both the detailed and preliminary design phases. Approximate values, derived from heuristics, are acceptable in the preliminary phase, but exact values are required in final computations. In analysis, many different types of computational processes are available which produce results under different assumptions. In some cases the results from one type of analysis may be acceptable in other situations (e.g., the use of results from a nonlinear analysis of a structure in place of results from a linear analysis). For many processes, different representations of the same data items are acceptable at different times. Consider a beam in a building. For structural analysis of the frame it is considered to be a line connecting two points, and the overall length is the only dimension of major concern. Once the beam is detailed, all of its dimensions, and those of the connections, become significant.

The concept of scope is to permit the application developer to state, external to the process, the scope and range of data that are acceptable to the process. Then the data management system can resolve all the data requests and provide the appropriate data, to the level required by the processes.

4.2.2 Advantages

The concepts of context and scope have advantages in all data handling situations. The complex context information present in all data references within a process will be reduced or eliminated. Processes will become generic

and can be used in any suitable context. Context could be declared globally, and hierarchically. As projects become more complex, higher levels of context can be added, and none of the applications will be affected.

Standards are an example of processes which are context independent. The application of the concept of context will permit standards to be used directly in design systems without dealing with the issue of explicit database linkages. Completely generic standards processing could be developed.

Scope has similar benefits of simplicity and process and data representational independence. The details of determining the acceptable types of data will be eliminated from the details of the process. When combined with a data flow architecture, scope can be used to control the automatic computation of data. This will cause the more detailed and exact computations to be deferred until explicitly required, but it will permit these more detailed results to be used in place of other results if they are available.

The following is an example of a conventional relational database access used to obtain stresses in a finite element system. The request will determine the principal element stresses for all elements of type "CST," in a structure called "BEAM," analyzed as a linear system, and subjected to loading condition "UNIFORM." The structure is part of one design alternative ("DESIGN_A").

```
SELECT      PRINCIPAL_STRESSES
  FROM      MATHEMATICAL_RESULTS
    WHERE     STRUCTURE   = BEAM
     AND      LOADING     = UNIFORM
     AND      ANALYSIS    = LINEAR
     AND      ALTERNATIVE = DESIGN_A
     AND      ELEMENTS    =
       SELECT     ELEMENTS
           FROM     MATHEMATICAL_MODEL
              WHERE     STRUCTURE   = BEAM
              WHERE     ALTERNATIVE = DESIGN_A
               AND      TYPE        = CST
```

Using context and scope, the request might be recoded as shown below. The three context and one scope statements (which are declared independently of the actual data access) are:

```
SCOPE          ANALYSIS     = LINEAR
CONTEXT        STRUCTURE    = BEAM
 AND           LOADING      = UNIFORM
 AND           ALTERNATIVE  = DESIGN_A
```

The data request then becomes:

```
SELECT     PRINCIPAL_STRESSES
   FROM     MATHEMATICAL_RESULTS
      WHERE     ELEMENTS =
         SELECT     ELEMENTS
            FROM     MATHEMATICAL_MODEL
               WHERE     TYPE = CST
```

4.2.3 Disadvantages

Context and scope are simply concepts at this time. They are based on techniques currently used in engineering, but these concepts have never been implemented and used in engineering software. Appropriate formalisms for using the concepts must be developed, and they must be implemented and tested to determine their practicality.

4.3 Knowledge Based Systems

Knowledge based systems are one of several types of artificial intelligence systems used to solve ill-structured problems. Engineering design is a typical ill-structured problem where many procedures are based on rules-of-thumb, experience, and intuition. Knowledge based systems provide a technique for describing such problem solving activity to the computer. A more complete description of artificial intelligence and knowledge based systems is presented in appendix C.

4.3.1 Background

Two basic types of artificial intelligence systems currently exist: weak solvers, and strong solvers [ErmaL80]. The original work in the area was in the development of weak solvers. Production systems are typical weak solvers. They have no built-in information about the problem being solved, and are composed of a number of simple premise-action rules. The production system can accept any set of rules, and will attempt to solve the problem by transforming a problem description from one state to another state through the use of the rules (theorem proving being a typical example). Such systems attempt generality, but are slow and unresponsive.

Due to the problems with the weak solvers, the strong solvers, which contain specific domain dependent knowledge were developed. In the strong solvers, the problem solving rules are more complex, and the problem solving strategies, which are built into the system, are tuned to the application being performed. DENDRAL [BuchB69], MYCIN [ShorE76], and Hearsay-II [ErmaL80] are all examples of knowledge based systems; each of these systems being built on the experience gained from the prior systems.

The knowledge in a knowledge based system consists of a body of rules, provided by experts from the application domain. Each rule consists of a premise and an action to be taken when the premise is found to be true. The rules are based on, and operate on, the current problem state, as represented by various data items. A controller monitors the data space, and determines when rules are to be invoked. From the current set of applicable rules, the controller will select those to be applied (based on problem knowledge), and invoke the processing of the corresponding actions. The process of rule utilization continues until some particular goal state is reached, or until the system determines that the goal is unreachable.

The knowledge rules are data for the controller. Thus, the problem solving data is not part of the system. The manner in which the problem is solved is determined dynamically by the controller. No explicit processing steps exist, and the problem solving strategy can be readily changed and tuned to different problems simply by changing the rules. Advanced capabilities permit the systems to learn and tune themselves through experience.

4.3.2 Problems Addressed

Much of the engineering process is ill-structured. Knowledge based systems can be applied in many areas, and they appear to provide a valuable technique for dealing with such ill-structuring. Specific areas where knowledge based systems appear most promising are: (1) standards processing and access, and (2) representing design procedures.

Standards: Computer processing of standards has been performed by using decision table based systems. Decision tables are a formalism for representing a variety of conditions and actions in a compact tabular form which can be readily processed. Decision tables are identical in nature to the rules and knowledge sources of knowledge based systems; only a different representational form is used. Thus, a standards processing system can be considered to be a form of a knowledge based system.

One of the most difficult aspects of standards use is that of accessing the correct provisions. Previous decision tables based systems have used "Switching Tables" as one method to control access [FenvS69, GoelS71]. Again, these decision tables fit into the premise-action structure of knowledge based systems. Current standards processing systems use only the standards themselves, with no additional data or rules which originate externally to the standard. Actual engineering practice augments the standard with additional information to gain access to, and to use, the various provisions of the standard. Engineers do not explicitly use all of a standard, exhaustively checking all provisions, as is the case in some computer based systems. Additional rules in an expert system, based on engineering practice, could allow the system to perform in a manner similar to the engineer. These additional rules would describe which provisions are applicable in any particular state. When necessary, such rules could be suppressed, and exhaustive, rigorous compliance checking could be performed. Techniques which are similar in nature to those currently used can provide an intelligent approach to use of standards in engineering computer systems.

Design Procedures: Design procedures do not exist as explicit algorithms, but rather they are a body of knowledge which is maintained by various engineers, each having different parts of the knowledge. Engineering activity relies on the cooperation of these individuals to pool their knowledge and experience to determine the procedure to design an engineered system. The information which constitutes the design process is processable by humans, but its structure and content are not explicitly known. Parts of it are represented by language in texts. Other parts are based on experience and are transmitted between individuals. All of the knowledge and design procedures are based on determining that the design, or the design process, is in a given state, and in this state certain conditions are true which cause the engineer to conclude that some action may be appropriate. This recognition of state and application of action is exactly what a knowledge based system does. Once the various rules have been formulated by the practicing engineer, a knowledge based system may be used to process these rules. The resulting system will solve the ill-structured design tasks in a manner similar to an engineer.

4.3.3 Advantages

The basic advantage of the knowledge based systems is that they provide a mechanism to address ill-structured problem solving tasks. The structure of such systems provides a number of other benefits as described below.

The knowledge based systems are flexible, and are not tightly coupled to the problem solving applications. The knowledge in such systems is expressed as data to the problem solver. This knowledge exists as a body of information, and it is not built into the system. Rules need not be explicitly linked to each other, and data accesses need not be explicitly coded. The various rules can readily be changed to tune the system to the problem solving task, and new knowledge and processes can be added without impacting existing components. The knowledge based systems can even be made to learn from experience, and to augment rules automatically.

The system can determine how to solve the problem, and the developer need not be concerned with all the details of potential interactions and conflicts between processing steps. A knowledge based system will determine what to do,

and will report on difficulties encountered during problem solving. Such systems can explain what they are doing, and why they are performing certain actions. Thus, the engineer can examine the workings of the application and determine when it is failing, or when modifications to the system's problem solving behavior are needed.

This flexibility is very important. Appropriate rules for design are unknown, and experience will be needed to develop systems which are usable and perform at the level of expert engineers. A system which requires extensive reworking when changes are required would not be responsive.

The MYCIN model [ShorE76] has been used for engineering applications. In one direct application [MeloR78], the medical consultant was changed to a finite element modeling consultant, simply by changing the knowledge rules. This "consultant" is designed to assist an engineer in determining the most appropriate modeling scheme for a nonlinear finite element problem. Unfortunately, the presentation does not show the power of the system. The other example is an extension of the model into component design, with the ability for the system to learn through experience [LatoJ77]. Thus, the technique does show promise in solving the ill-structured engineering design problem.

The following is an example of how a knowledge based system can be applied to standards processing (syntax and style based on MYCIN). The decision table representation of the tension stress provision described in section 3.1.1 is:

DECISION TABLE 1.5.1.1

THREADED PART	T	—	—
PIN-CONNECTED	—	T	F
USE TABLE 1.5.2.1	*		
($f_t = P/A_n$) ≤ ($F_t = 0.45F_y$)		*	
($f_t = P/A$) ≤ ($F_t = 0.60F_y$)			*
($f_t = P/A_n$) ≤ ($F_t = 0.50F_u$)			*

The knowledge based form of the decision table requires a single parameter to be defined (additional numeric data items will be required for the actual usage of the rule). The value of the parameter will be used to select the proper rule, and is defined as:

```
TENSION_MEMBER: <TENSION_MEMBER is the type of tension member>
       EXPECT: (ONE OF TYPES: (THREADED_PART
                PIN-CONNECTED  SIMPLE_TENSION) )
    LOOKAHEAD: (RULE_1.5.1.1.A  RULE_1.5.1.1.B  RULE_1.5.1.1.C)
       PROMPT: (Enter type of *:)
        TRANS: (THE TYPE OF *:)
```

The decision table is represented as three rules. In this example, there is a one-to-one correspondence of the rules and the columns of the decision table.

RULE_1.5.1.1.A

IF: 1) THE TYPE OF TENSION_MEMBER IS THREADED_PART
THEN: THEN USE TABLE 1.5.2.1

RULE_1.5.1.1.B

IF: 1) THE TYPE OF TENSION_MEMBER IS PIN-CONNECTED
THEN: ($f_t = P/A_n$) $\leq$ ($F_t = 0.45F_y$)

RULE_1.5.1.1.C

IF: 1) THE TYPE OF TENSION_MEMBER IS SIMPLE_TENSION
THEN: ($f_t = P/A$) $\leq$ ($F_t = 0.60F_y$)
AND: ($f_t = P/A_n$) $\leq$ ($F_t = 0.50F_u$)

4.3.4 Disadvantages

There are two serious questions associated with the application of knowledge based systems: (1) speed, and (2) development of knowledge sources.

Speed: Computers are fast when performing arithmetic computations because the primitive operators (addition, multiplication, etc.) are built into the hardware. It is questionable if a computer which was programmed to perform arithmetic in the manner of a human would be as fast as a human; the primitives are wrong. The cognitive processes present in design may

require excessive time when performed by a classic computer designed for arithmetic operations. Thus, with respect to design, the computer based system must be faster than the engineer or provide a number of benefits in order to be successful. If it is not faster, no advantages are gained. Without significant benefits, simply providing all of the base components, and letting the engineer provide all of the expertise and control to guide the problem solving behavior would be appropriate.

Knowledge Sources: A knowledge based system requires knowledge and rules. Someone must develop these rules, and then test them to determine if the system performs in an acceptable manner in a variety of situations. This task will require constant monitoring of the system and upgrading of its capabilities. Such tasks can be performed only by human experts, those with the judgement and experience to determine if the computer is performing as expected, and those who know what to do when it is not performing as desired. There has traditionally been a reluctance on the part of senior experts to handle such details, and they are usually relegated to junior personnel. For a knowledge based system to be acceptable, expert knowledge must come from, and be maintained by experts.

4.4 Virtual Machines

The concept of a virtual machine is to provide, via software, a computational environment in which the users of the virtual machine appear to be using a dedicated piece of hardware [CanoM80, GrovL80]. The configuration and capabilities needed in a computer can be designed and implemented using software on an existing system. The capabilities present in the virtual machine may not exist in any real system. All application programs are written and execute on the virtual computer which provides the resources and features not present in the host configuration.

4.4.1 Background

Virtual machines were created to provide computing environments which were not available on existing hardware. One of the first uses was in providing upward compatibility across new hardware systems. Introduction of new hardware invalidated many programs which were written in assembler language for the older machines. The costs of rewriting these programs, and the time involved, presented difficulties in maintaining the ongoing operations of facilities. The alternative to rewriting programs was to create an emulator for the old hardware running on the new hardware. The emulator (a virtual machine) would then execute the old programs directly, using the new hardware. Thus, only one program needed to be written, and conversions could proceed without affecting day-to-day operations.

The virtual machine concept has been extended in recent years. IBM has introduced a complete virtual computer system which is used to configure a proposed hardware system as a program running on some existing hardware configuration [CanoM80]. With the inclusion of all the details of timing and I/O transfers, a proposed system can be exercised and tested for performance evaluation without the expense of configuring a real system.

An identical approach is used to provide a variety of single or multi-user computer configurations operating on a single real machine. In this manner, each user has what appears to be a complete computer system for his use. He is operating on a multi-user system, but is never concerned with the other users. In fact, it is possible for him to execute the virtual computer system software, and provide a number of virtual computers, each running on his own virtual system. The base virtual computer system is used to provide the necessary multi-user support, and the applications can execute on the individual virtual machines without knowledge of the underlying support.

Engineering support-supervisory systems such as POLO [DoddR80] an DVM [SchrE79] can be considered to be virtual computers. They provide a computing environment which does not exist as a physical system, but a computer system which would be desirable for performing engineering applications. Such systems consist of a controller and a set of operators. These are analogous to the central processor on a real system. However, the basic virtual machine operators are better suited to the engineering applications. The engineering oriented operators permit programs to be written at a higher level than if they were written for a real machine. Virtual machines also include a memory subsystem, and disk or secondary storage systems. The software for such a

virtual machine consists of a monitor or operating system and a set of languages and their compilers. In addition to all these basic system components, the virtual machine model provided by POLO includes a number of features not commonly found in real systems. These include components logically equivalent to: (1) a writable control store which allows the applications system implementor to add new instructions to the basic repertoire, and (2) a virtual back-end database machine along with a data definition language and compiler which are used to provide database support for applications.

4.4.2 Problems Addressed

A major problem in designing any piece of software is configuring the set of basic components and the overall system organization and structure so that the software is flexible and performs the desired tasks well. The virtual machine provides a software structure model to address this problem; it provides the basis for the software configuration. A basic machine model can be used to provide the structure and the complete set of components with the features and capabilities needed to develop application software. Applications are designed, developed, and programmed for the virtual machine. The existence of the virtual machine to provide support may yield better structured software than ad hoc approaches.

4.4.3 Advantages

The basic advantage of the virtual machine approach to software development is that it provides a sound, structured basis for the development of software systems. Classic computer architectures have been used for over thirty years, and although there are questions about their effectiveness [BackJ78b], the basic von Neumann architecture is still used. By developing virtual machines which are well suited to engineering applications based on such a software model, all of the experience, research, and development which has gone into computer systems development can be utilized in developing the basic system software.

The use of virtual machine models results in clean programs. The applications deal only with high level concepts provided by the virtual machine. Applications are developed using a level of abstraction which is

closer to the real problem. Thus, they do not need to deal with a variety of details which clutter programs written in nonvirtual environments, and which make development and maintenance more complex.

By separating various functions into separate machine models, a significant portion of the complexity is eliminated, and each virtual machine can be tailored to a specific task. Formal models for machine interfaces and communications can be applied to these virtual machines to link the subsystems. Many of the complex issues involved with resource management, and other details such as providing multi-user support, can be relegated to the system level, and are not apparent to the application developer. This approach has proven to be of great value in developing applications such as FINITE.

4.4.4 Disadvantages

Such systems can become very complex. The software used to implement a virtual machine is not simple, and its development may present difficulties. Additionally, there is the potential problem of speed when using such an approach. The use of a complete virtual machine operating at the same level as the host machine is several times slower than the host hardware, due to system overhead (interpretatively simulating any operator such as multiplication or addition is much slower than letting the hardware do it directly). To be effective, the operators in the virtual machine must be powerful enough so that the system overhead becomes negligible.

4.5 Languages

Is FORTRAN the first, last and only scientific programming language? The question has been posed recently. FORTRAN is the de facto standard for development of engineering software. Other programming languages may provide alternative features and capabilities, but they are generally ignored by engineering users.

4.5.1 Background

There are hundreds of programming languages. Of these, FORTRAN and COBOL are the industry standards for scientific and business programming. Their popularity is due to their widespread availability and standardization. This is due to government selection of these two languages as requirements for

government computer systems. Both languages are quite old, dating back to the late fifties. Through recent years, COBOL has been updated and extensive database facilities have been added (CODASYL). FORTRAN remained unchanged for over ten years, but now is undergoing a number of changes, and future language additions and modifications may change the overall flavor of the language.

As a result of the lack of facilities in FORTRAN, COBOL, and other languages (their designs were not based on any particular set of principles, but they were developed to fit specific needs and hardware configurations [BackJ78a, SammJ78]), a number of alternatives have been developed. A few of the more common are ALGOL, ALGOL 68, PL/I, APL, LISP, Pascal, and Ada. These are described in the glossary (section 3). These languages all have a large user community, and are available on a variety of computer systems. In addition, there are numerous other languages, each developed to meet a particular set of perceived needs for some particular problem domain. Many of these languages have a number of interesting features. However, most are not well supported, are not portable, and have only a limited user community — the development team.

4.5.2 Problems Addressed

Software development is extremely complex and costly. Many of the problems of presenting the algorithms to the machine are due to the nature of the programming languages, due to their lack of abstraction. Alternative languages provide features to simplify program development and yield better programs.

4.5.3 Advantages

Each of the various languages has its own advantages. In general, each of the languages has some particular set of features which yield better programs, with less development effort, by eliminating some details of program development. All of the newer languages have improved control and data structuring features. Other features which some of these languages provide and which might be beneficial include: (1) operator overloading, (2) language extensibility, (3) language environments, and (4) data flow architectures. These various features are described in the glossary (section 4). Each of the features eliminates some of the details of program development and coding. They permit the programmers to be more productive and to deal with more

abstract concepts, concepts which are closer to the problem being "computerized," rather than dealing with the detailed presentation of the machine implementation of the process. Resulting programs are more flexible, more adaptable, and more reliable.

4.5.4 Disadvantages

The major disadvantages of any new computer language are the questions regarding its acceptance and portability. The selection of a programming language must deal with the realities of the user community. If languages are not accepted, or if the programs can not be moved, programs may die, or the extent of their use may be severely limited. Prior large engineering software systems have been long-lived, and portable, well supported programming languages are necessary to develop, maintain, and support such software.

Some languages lack particular facilities which can seriously impede their use for particular applications. For example, in current languages, FORTRAN lacks data structuring facilities, and Pascal lacks I/O and separate compilation facilities. The effort spent to overcome these deficiencies may outweigh any potential benefits.

In other cases, the fact that the languages are new, and have not had extensive use in large systems development, may mean that there are questions regarding their applicability to the production environment. Large-scale software development is quite different than other types of programming (program complexity grows exponentially with program size), and such software is often operating at the limits of the language. In new languages without extensive large-scale use, a potential for problems exists, and this tends to discourage the use of the languages.

5. A COMPUTER AIDED ENGINEERING SOFTWARE ENVIRONMENT

In chapter 4, a variety of techniques which can assist in developing engineering software, and which will overcome many of the current problems were discussed. These techniques are not directly applicable for use in developing engineering software systems. Many of the techniques are not implemented as production software tools. Others require extensions and further research. Even if all of the techniques were available as production tools, the problem of developing advanced engineering software systems would not be solved. Each of the techniques discussed address only a portion of the total problem. A complete solution will require the integration of all of the various tools into a single framework for engineering computer applications.

The Computer Aided Engineering Software Environment (CAESE [kas'ē]), as proposed herein, is designed to be a prototype research and production engineering computer system based on the techniques presented in chapter 4. CAESE is neither a single system nor a collection of programs, but rather it is a collection of system components which are shared by developers, researchers, and users, and which are applied to all of the steps needed to apply computers to the design and engineering process.

From the operational viewpoint, CAESE is basically a two level, three component system. It is patterned after the current generation of support-supervisory systems. However, it contains a number of new features and concepts which are significant and which make CAESE different from its predecessors.

The top level (the first component) is the application level — the application environment. It consists of all of the domain specific tools, programs, and procedures, as well as the data utilized in the computer aided engineering process for any specific application area (each separate problem domain has its own individual application environment). This level performs the actual design and engineering computations.

The bottom level (the second component) is the system level — the system environment. It consists of all of the components, data, and support software which are independent of any application domain. This level performs no engineering or design. However, it is used by all of the applications and the remainder of the system (including the support environment) as a run-time support system. The application environments are built on the system environment.

The third major component of CAESE (also at the bottom level) is the development support software — the support environment. This component provides the various tools which are used to develop and maintain both the system and application levels. The support environment is not used to perform or to provide run-time support for any computations. It is used only to create, configure, and maintain the remainder of the total system (both the system and the application environments).

The following is an introduction to these components, the problems they address, and the technologies they use. It is not meant to be a complete presentation of CAESE. Rather it is an introduction and a "strawman" design of the system, its features, and its capabilities. Each of the environments will be discussed separately in the following sections, although the environments are interrelated. A presentation of the overall structure and the relationships of the environments and the components which comprise CAESE follows the description of the individual environments.

5.1 The System Environment

The system environment is a collection of components which are used to provide a variety of system and run-time support features needed to solve the problems described in chapter 3 and to form the basis of a computer aided engineering system. Each of the components addresses one or more of the specific problem areas, and each is built from one or more of the various solution techniques. The major software components of the system environment include: (1) an engineering oriented database management system, (2) a knowledge based system kernel, (3) a standards processing system, (4) a complete set of interface systems, (5) a project management system, (6) a design supervisor, and (7) an overall organizational framework for all components. Each of these are discussed below.

5.1.1 Engineering Relational Database Management System

The engineering database management system is the common database manager used by all applications and system components for all data handling requirements. The database manager consists of a complete run-time database management system, and a number of components associated with the support environment (described in section 5.2) such as a data definition language, data dictionary, and data mapping language.

The engineering database manager is basically a relational system. However, it has a number of extensions which are considered significant for the engineering application. These include: (1) extended data types, (2) context and scope, and (3) data tracking.

Data Types: The existing relational systems are business oriented, and have a limited number of data types, typically names, character strings, booleans, and integers. For use in engineering applications, these data types need to be extended to include additional basic data types and other data aggregates. The additional basic types would include (but not be limited to) reals (various precisions), complex numbers, and enumeration set types. Data aggregates would include the traditional vector and array structures, but these would be extended to include other structures such as tensors, networks, and trees. These and similar types of logical data groupings exist in engineering, and a mechanism needs to exist in the database management system to handle such data aggregates (in the whole as well as the individual components).

Since it is impossible to predetermine the complete set of all possible data types, appropriate mechanisms must exist to augment and extend the base types as needed, and to provide more abstract types (higher level types) based on the supplied primitive types — to provide a "data abstraction" capability.

Data types can not be considered just to represent a collection of bits or words. Many of the various data items used in engineering have some physical significance, and the database manager must be able to deal with the attributes which represent the physical characteristics of the data items. Typically, these attributes include: (1) the units of the data, (2) default values for the data, and (3) constraints on valid data values. Capabilities would exist in the database manager to automatically deal with the associated attributes while manipulating the actual data.

Context and Scope: The features of context and scope described in section 4.2 need to be built into the database manager. This permits all programmed data references to be context independent. Context and scope are declared externally to the

data references, either through explicit statements in the procedural language used or through user level commands. The context and scope declarations are then used by the database manager to augment each data reference to determine the actual data which will fulfill a request.

The various data primitives can have context and scope dependent information associated with them. This information is used to invoke transformations, converting data from one form to another slightly different form automatically.

Data Tracking: Data tracking is accomplished by associating "ingredients" and "dependents" attributes with each data item. These attributes declare what other data items and processes are used to create a given piece of data — its ingredients, and what other data items are computed from an item — its dependents (one set can be determined from the other). Such declarations may be either static or dynamic. The information is used by the database manager to determine the effects of changing a data item, and to maintain the consistency of the data. The database manager can determine the set of data which needs to be updated due to a change, and will invoke all of the necessary processing to complete the update.

Data tracking attributes can be used to form the basis for a data flow driven architecture. The database manager will determine and compute all of the ingredients of any data item and cause the data item to be derived automatically whenever it is needed. Programs only need to request the desired results, the database manager will provide all of the control used to compute the results.

In addition to the database manager to support the applications, the complete database management system must include an information storage and retrieval component. The information storage and retrieval system allows end-users to query, create, and update information in the various databases, without the need to write applications programs. This component utilizes the capabilities of the interface system to provide input, and report generation facilities to provide output.

5.1.2 Knowledge Based System Kernel

The knowledge based system kernel provides the mechanisms for controlling the use of expert knowledge. It consists of a control processor and an explanation system. Learning and knowledge integration, components of a complete knowledge based system, are part of the support environment.

Control Processor: The control processor is the basic operational component of a knowledge based system. Through the use of the knowledge sources, it determines which rules to invoke, and monitors the various actions that result. For its operation, the control processor must be able to access all of the knowledge sources, and determine which are applicable in any situation. Additionally, there must be a database access mechanism to permit the control processor to monitor and query the database, and to provide the knowledge sources with the mechanisms to access the data they require for problem solving. The data is accessed through the common database manager, but the encoding of the knowledge sources may not contain explicit data access statements (this type of explicit coupling reduces the system flexibility and adaptability). All linkages of data to knowledge sources is encoded by generic name, and the actual binding is deferred until execution-time.

Explanation System: Since the problem solving behavior of the knowledge based system is not determined until execution-time, it is necessary for the system to explain what problem solving strategy it is using, and why it selected that particular strategy. In this way, an engineer can monitor the performance of the system, and redirect its behavior when needed. This redirection can take the form of changes to the knowledge sources, or it can consist of providing additional data which will cause the controller to select a different problem solving behavior. Once a solution is determined, the explanation system will inform the engineer as to how the decisions were reached, and he can then use this information to determine the next step to be taken. Additionally, such information can be used to determine the effectiveness of the various knowledge sources and rules, and can provide information which is used to improve the problem solving behavior of an application.

5.1.3 Standards Processing System

Standards provisions are very similar to the rules in expert systems. Thus, the structure and form of the standards processing system will be quite similar to that of the knowledge processor. Since the standards typically have been represented as decision tables in computer based processing, it may be logical to continue to use this representation. This representational difference in knowledge will be the major difference between standards and knowledge sources, and it may dictate that there be two different processing systems: one for general knowledge, and one for standards. However, it may be the case that both processors are instances of the same system.

The standards processing system will consist of a control processor used to govern the execution and interpretation of the standards. The control processor must have the mechanisms to select the applicable provisions of a standard, just as the knowledge based system must be able to access and select the knowledge sources. Similarly, the standards processing system will contain a database link to allow the various provisions of the standards to obtain their data from an application's databases. An explanation system also will be included to provide a mechanism to inform the engineer as to how-and-why the standards processing decisions were made.

The standards processing system is designed to be independent of any particular standard. The standards exist as a collection of data which is accessed and processed only by the standards processor. Since explicitly coded linkages between processes, standards, and data do not exist, the standards can be changed as needed, with minimal impact. This ability to change standards will require the existence of support tools, in the support environment, to build the computer processable form of a standard from its normal textual description.

5.1.4 Interface System

The interface system provides all of the mechanisms through which users access and communicate with CAESE and the applications. It includes facilities for language input, tabular and report output, graphical input, and graphical output. There are a number of individual interface components which can be used by the applications and other system components. They include: (1) an input language translation system, (2) a report generator, (3) a graphics core, (4) an error handler, and (5) a communications mechanism.

Input Language Translation System: The input language system provides the capabilities to support the translation of the variety of command and data languages which would be used by CAESE and the applications. The system consists of three distinct levels of software. The lowest level interfaces to the physical devices, and produces a stream of input characters which is device independent. The second level takes this input stream and converts it into a variety of basic tokens (words, numbers, delimiters, etc.). The highest level combines the basic tokens into higher level language constructs. All of the languages are described in terms of these constructs, and the applications interface to the system to parse this level of language input.

Only artificial (as opposed to natural) languages are being considered. Natural language translation is still beyond the scope of a production system. However, some of the inferencing and implicit context techniques used in natural language systems to produce more fluent and natural input may be of value [WaltD78].

Report Generator: The computational procedures of the applications can produce output which may be represented in either tabular or graphical form. The report generator provides the software tools which are used to produce tabular output without the explicit coding of output producing programs. The user (either end-user or applications programmer) can use the report generator to describe the content of a report along with the physical layout and organization of the tables. The report generator will access the requested information from the databases and produce a report in the prescribed format.

Graphics Core: The graphics core provides basic software support system for all graphical interactions, both input and output. Patterned after the proposed standards [GKS79, GSPC79], the graphics core provides a programmer interface which is both system hardware and graphical device independent. To provide this independence, there are two levels of software: a device dependent processing level, and a device independent level. Programs are written using the capabilities provided by the

device independent level. The graphics system converats the requested graphical opertions into a lower level set of basic device independent primitive operations. The device dependent level converts these primitivies into the actual instructions used to drive the graphics devices. It is advantageous to construct another level on top of the device independent level. The various graphics constructs provided by the proposed standard core systems represent primitive operations, and applications require considerable programming to provide usable interfaces. A higher level provides graphical primitives which are more useful in engineering. Essentially, it provides a virtual engineering graphics machine. Thus, the common capabilities and features required by the applications need not be repeatedly developed for each application.

Error Handler: Errors occur throughout the execution of the applications. Errors can be due to either incorrect data or due to a program detecting faults and inconsistencies in its operation. The concept of the error handler is to provide a single system component which is invoked whenever any error occurs. In this way, all errors are routed through a common error handler and treated in a consistent manner. Features such as run-time errors from batch execution being routed back to an interactive terminal initiating the task and logging of errors can all be isolated and programmed at the system level. The applications need only raise error conditions; the system is responsible for all further interactions and processing.

Communications and Access Mechanism: The communications and access mechanisms provides the lowest level of user interface and communications support. CAESE is designed to be used simultaneously by a variety of users, each accessing the system through different types of devices. In addition to accessing the system, it is necessary for the users to communicate among themselves. The communications and access mechanism provides the support software for these features, eliminating the need for any application to deal with device dependent issues or multi-user communications.

5.1.5 Project Manager

CAESE is envisioned as being a project oriented system. Each application is developed to be a design system for a certain class of engineering project, and each individual project is handled separately, via its own databases. The project manager is used to instantiate and supervise any project. It is a general purpose application, independent of any individual type of design application. Each application system will share a number of databases, standards, knowledge sources, and computational processes. The project manager is used to configure the exact set of such components for any application. Additionally, it is used to establish what users have access to a project and what are the rights of the individual users.

In addition to establishing configuration and user control, the project manager acts as an overall run-time project supervisor. The project manager maintains the status of the project, monitors all work, and is used to produce the reports describing project work. It also enforces security, verifying all users' rights and privleges.

5.1.6 Design Processor

The design processor is the highest level of any engineering application system, yet it is application independent. The project monitor is used to create and monitor a project. The design processor is used to control all of the engineers' work on the project. It is the mechanism which the engineer uses to communicate with the various application components. Through it he invokes processes to establish goals and direct computations. The function of the design processor is similar to the operating system on a computer. It establishes and directs the various tasks to be performed, allocates resources to the tasks, and oversees the routing of user input to the task and the routing of task output back to the user.

5.1.7 Overall Organization

The various components described above, along with the application modules are combined into a single system to form a computer aided engineering application system. All of the system environment components are designed to be autonomous. Just as explicit coupling between data items and processes is not specified, there is a similar desire to uncouple the individual system environment components to the greatest possible extent. Some coupling will

exist. Various components, such as the standards processor and the knowledge processor, must be able to access information from the databases, and they will use the same database management system as used by any application. To access the data, some interfaces between the components will exist, but these interfaces will not be tightly coupled links.

Based on the success of the virtual computer model in earlier large systems, it seems reasonable to select the virtual machine model as a suitable structure for the overall CAESE organization. In the virtual machine model, each of the system environment components are considered to be individual virtual processors, each tailored to the specific tasks being performed. All of the virtual processors which represent the system environment components are implemented using a common kernel of support functions specifically designed for a multiple processor virtual computer model. The support functions provide the overall system control and the mechanisms for the system environment components to communicate and interface with one another. The design processor described above is at the highest level in the virtual system, and is used to control and coordinate all of the other processes; it is the operating system for the virtual machines.

The CAESE system environment can not do any engineering without the addition of application modules. Applications are designed and implemented as separate components using the facilities provided by the components of the system environment. Each of the applications are modeled as one or more virtual attached processors; each processor implements a particular set of operators and performs a specific task, but all rely on the system environment to perform common system functions. Thus, the overall virtual computer model can control and support the various application components, just as it controls the system components.

5.2 The Support Environment

Any large-scale software system can not simply be designed and then coded in a programming language. There are a number of problems associated with managing the development of the various software components. Managing and maintaining the symbolic form of the software (the source code representation), and developing procedures for modifying operational programs are not straightforward tasks. Failure to properly deal with these issues has adverse effects on software development and maintenance efforts and costs, and can also influence the availability and reliability of systems. In a system

such as CAESE, these problems are compounded by the existence of: (1) databases, (2) standards, and (3) knowledge rules.

The support environment is a collection of software tools [KernB76] used to assist in the software development and maintenance effort. It includes several components: (1) software to assist in preparing standards for processing, (2) software to assist in developing the various knowledge rules and to incorporate the rules into the applications, (3) programming languages used to develop the various system and application components, (4) software used to maintain the symbolic and execution forms of the system, and (5) an overall framework for all of these components. Each of the support environment components are discussed below.

5.2.1 Standards Support

The information which represents standards is not coded into any part of an application which uses a standard, but rather, it is the data to the standards processor. As such, there must be a mechanism to enter this data, and to organize and structure the internal representation of the standards. The simplest means of providing standards support consists of a language and input system used to describe and input the various components of the standard, and a database used to store the standard. The standards administrator (the individual system level user who has the responsibility for maintaining and managing the standards) would convert the textual form of a given standard into its language description, and use the standards support system to enter the data and prepare the data structures used in standards processing. It also will be necessary to supply an output system. The output system provides a means to display the internal representation of the standard as maintained by the system. Such a display could be used to verify that the system internal representation is consistent with the desired form, and that errors of misinterpretation have not occurred.

There are a number of problems associated with developing a representation for any given standard. Conversion of the textual form into the decision tables, networks, and outlines, which have been used to represent standards, is a difficult process made more complex by the inherent ambiguity and inconsistency in the standards. Significant research has been conducted which addresses these problems, and a variety of techniques and prototype tools have been developed. The development of a second generation of standards development and analysis aids is now underway [FenvS79a, FenvS79b].

These aids are designed to "provide a comprehensive, general set of computer aids for the analysis and synthesis of standards." As such, these aids are the integration of a number of tools, (tools which are used to convert from the textual representation of a given standard to an internal representation) into a complete standards development system. It is logical to consider that such a standards development system be included in the CAESE support environment. This software would integrate the work of developing standards and their textual representations with the system for standards processing. A processable form of the standard would be developed in parallel with the textual representation. Thus, many potential problems resulting from the misinterpretation of the standard would be eliminated.

Standards are dynamic, constantly being revised. An important function of the standards support software is to aid in the changing and updating of standards. Since the standards are data, and are separated from the run-time standards processor, changes are made by replacing standards or individual provisions of standards. A major problem lies in the linkage of a standard to the remainder of the system. The standards are implicitly linked to the database and to the knowledge rules and applications programs. Any change of any of these components may result in problems if the actual linkages can no longer be resolved at run-time. Thus, it is important that the standards support system have a mechanism for determining which linkages exist, and to provide information to the system standards administrator concerning the potential impact of a change on the remainder of the system (not only a standards change, but also a database change or a knowledge rule change).

5.2.2 Knowledge Integration

Knowledge is represented as data in the knowledge sources, and it is separate from the knowledge processing of the kernel of the knowledge based system. As with standards, a mechanism must exist for codifying and presenting to the system the expert knowledge used by the applications. The knowledge support software will consist of a language to describe the knowledge and knowledge sources, a database to store the knowledge, an input processor to enter the knowledge into the database, an output processor to display the knowledge which had already been entered into the system, and a mechanism to determine how the knowledge rules relate and are linked to the applications which use the knowledge.

Knowledge will change and must be updated, and the system must be able to "learn", either through explicit instruction or through the automatic accumulation and modification of knowledge based on experience [LatoJ77]. Learning through experience is performed in conjunction with the actual processing of the knowledge. Explicit instruction may be performed along with problem solving, or this may be a segregated activity. Thus, a complete set of learning features will be available for use in both the support environment and in the system environment. Similarly, explanation features will exists in the system environment for run-time use, and in the support environment to assist in developing and maintaining knowledge.

5.2.3 Development Tools

Standards support and knowledge integration are two specific examples of the capabilities in the support environment. There are a large number of similar types of components and tools useful in other phases of the software development process. For example, it can be envisioned that CAESE uses a number of languages developed for the specific needs of the various system environment components. In addition to the languages for the standards and knowledge representation, there would be database definition languages, data mapping languages, languages to describe graphics operations, a language to describe the physical hardware configuration, one to describe the software configuration of an application (the databases, standards, knowledge sources, etc.), and a language in which the applications are written. Additionally, many of the application systems will have their own end-user languages. A separate language would be used to describe these individual application languages. As part of the support environment, each of these languages require a compiler and data or file structures for maintaining source and object forms of the programs written in these languages.

The proliferation of all of these tools implies that the total system will be quite large and encompass many lines of code. Maintaining such a volume of code will be quite difficult. This problem is complicated by the desire to maintain the code in a form which is compact (duplicate code, such as occurs with COMMON in FORTRAN, being stored only once) and independent of the particular hardware and software system used for execution. Software tools will be required to assist in this code management problem. Ada has attacked this problem by providing a language environment of tools which are used to support the development of programs, an editor tuned for the language,

a debugger, and a database, all grouped into a single operational system [FairR80]. These tools serve the sole purpose of easing the software development burden by providing needed capabilities which are tuned to the language. The set of development tools present in the CAESE support environment form a similar software development environment.

5.2.4 Operational Tools

The operational tools are used to assist in the actual operation and use of CAESE and its application systems, whereas the development tools are designed specifically for system and application creation. A variety of operational tools are needed. They include utilities to dump databases for display, utilities for archiving databases, and a utility used to reconfigure and remap databases if a change in the physical or logical organization invalidates the current form.

A log reporter would also be a useful tool. It is often desirable to maintain a running log of who performed what operation or who is responsible for what change. The system environment interface component contains the capabilities to create such a log. The log reporter would be used to prepare reports and answer queries about the information in the log. A variety of similar tools would be used to assist in the operation of CAESE and the applications.

5.2.5 Overall Organization

The various support environment components are designed to be application independent. They exist as system wide capabilities used to support all or any of the individual applications which require such features. The components may be built into the applications or they may be used as stand-alone systems.

Each of the tools is to be considered an application system of CAESE, but an application which does no engineering or design. The various tools are all built and configured in a manner similar to the applications (as a virtual attached processor). They all perform some specific task, are built from the various components provided at the system level, and are integrated into the entire system. There is a fine line of distinction between what is a support environment application and what is a design and engineering application.

As an example, consider both the standards support and knowledge integration systems. Both of these components require databases to store the knowledge rules or to store the standard provisions. Each of the databases will be standard CAESE databases. The databases are defined using the database languages of the support environment, and the system database manager is used to provide all database functions. All of the input and output components of the standards and knowledge support systems can use the various interface features available in the system environment. In effect, the standards and knowledge support components are completely dependent on the system environment of CAESE for their operation, and they are identical to engineering applications in their overall structure, their utilization of system environment facilities, and their operational appearance to the end-user. The only difference between support and applications is that the support applications and the support environment are developing data and programs which are used in the application environment to support the engineering applications, whereas actual applications are performing engineering and design for the end-user. Through the utilization of the various components of the system environment in providing software support for other components of the system environment, the support environment, and the application environment, and the similar use of the support environment in developing applications, the total system is used to develop and support itself.

5.3 The Application Environment

All of the project engineering and design is done using the application environments of CAESE. The typical applications for which CAESE is designed are each considered to encompass a large, multi-disciplinary problem domain, rather than being a larger number of smaller, more specific applications. The following are all potential application domains and some of the major subsystems of each:

Nuclear Power Plants: Reactor, pressure vessel, and containment structures, cooling systems, control systems, electrical generation, auxiliary structures, etc.

Off Shore Platforms: Platform structural analysis and design, fabrication, exploration components, production systems, etc.

High-rise Office Buildings: Foundations, space layout, structural system, vertical transportation, electrical distribution, plumbing, environmental and energy systems, construction management, etc.

Aircraft: Airframe, avionics, propulsion, flight dynamics, navigation, etc., for some class of aircraft.

Ship Building: Hull structure, propulsion, control systems, navigation, cargo handling, etc.

Bridges: Substructure, superstructure, construction management, site layout, etc.

Software Development: Design of large-scale software is similar to the engineering and design of any physical system. In this application, the components being designed are the software subsystems, and the design and engineering problems are due to managing the interrelations of the components.

The above are typical of the types of applications for which CAESE is intended. The applications are typified as being: (1) large-scale projects, (2) complete engineered systems (rather than components of systems), (3) the integration of multiple subsystems from different engineering disciplines, (4) ill-structured problems, and (5) governed by a variety of standards.

It is not to be construed that CAESE only will be used for applications similar to those listed above. The applications listed where all chosen because they represent the types of large, multi-disciplinary problem domains for which CAESE is specifically designed. Other applications, such as finite element analysis, structural optimization, construction management, or network planning and modeling are equally well supported by CAESE. The single discipline or analysis oriented activities may not require all of the facilities provided by CAESE, but there are many capabilities that will be beneficial in developing software systems for any type of engineering problem domain.

Each of the specific engineering problem domains which are processed by CAESE exist as individual application environments (i.e., CAESE — Bridges, CAESE — Power Plants, CAESE — Office Buildings, etc.). Each application environment consists of a number of individual subsystems which are integrated to form a complete engineering design system. Various utility systems, such as finite element analysis, which are components of many different application environments are developed individually, but linked together with other

components to form a complete application package (this linkage need only be done at a logical level). Thus, each of the applications appear to be whole in-and-unto themselves, and each may be used individually without knowledge of any other application environment.

There are several components in each application environment. These components exist for one or more of the subsystems. They include: (1) database descriptions, (2) descriptions of standards, (3) knowledge rules and procedures, and (4) computational and analytic procedures. The first three of these are processed directly by the support environment software to form an information base for the application. The various computational procedures are integrated with the system environment to form the complete set of application environment software. This software requires the data from the information base for its operation. This integration, performed with the assistance of the tools of the support environment, results in a complete application environment. After all of the components have been integrated and linked into a complete application environment, the resulting application system is then ready to be used for the domain specific design and engineering problem for which it was created.

5.4 The Software Environment

Based on the preceding description of the individual environments which comprise CAESE, the following presents more detail on the interrelations of the environments and system components.

As stated above, the system environment is the lowest level of software in CAESE. It is built using the capabilities of the host computer hardware and system software. The support environment is also at the lowest level, and it is similarly built on the facilities of the host machine. Both of these environments are dependent of each other for some functions, such as the support environment providing source code maintenance for the system environment, and the system environment providing database management for the knowledge integration and standards support of the support environment. The individual application environments are built using the facilities provided by both the system and support environment in addition to the facilities of the host. Figure 5.1 shows the interrelation of the levels of the three environments which comprise the total system.

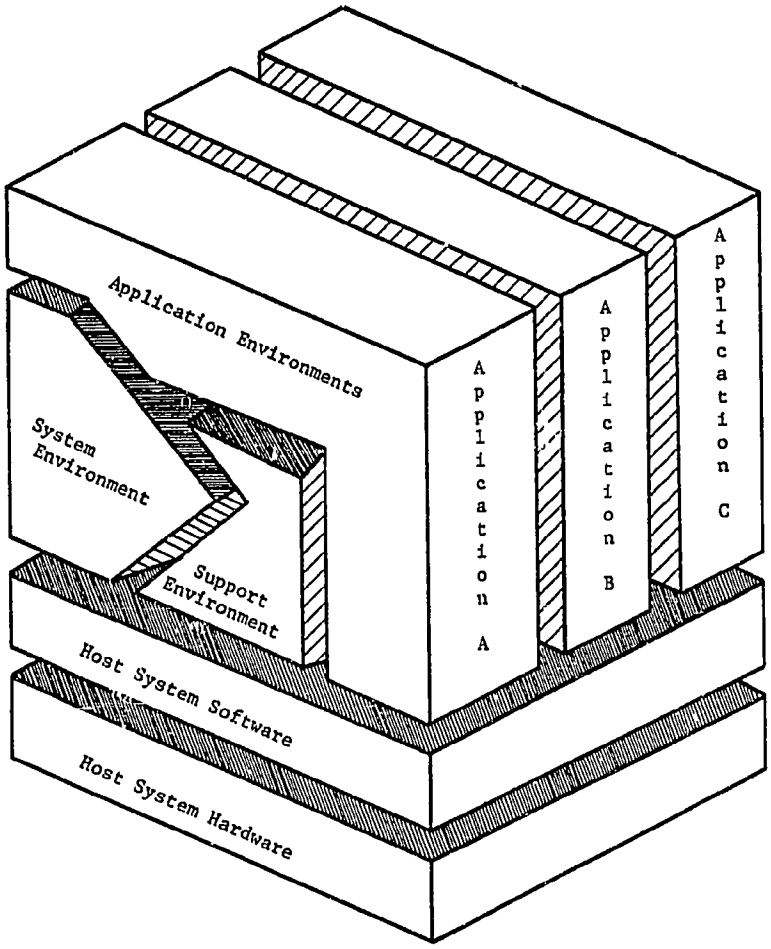


Figure 5.1. CAESE Configuration

The relationships between the components of CAESE used in all aspects of standards processing are shown in figure 5.2. There are two sets of standard CAESE databases: one set is for the application's data, and the other set is for storing the standards. The application and the standards processor both access the databases through the database management system for all their data needs. Similiarly, the application links to the standards processor for all of the application's standards processing requirements. The standards support software also links directly to the database management system. The relationship between the components of the knowledge processing software is similar, and it is depicted in figure 5.3. Similar in structure, the interface component relationships are shown in figure 5.4

Figure 5.5 shows a simplified view of a complete application system. The information base for the system consists of databases for standards, knowledge sources, project data, and application data. All databases are accessed through the database management system. The remaining system environment components (standards, interfaces, and knowledge processing) comprise the next level of software (the internal structure of these systems has been eliminated from this figure). Support environment components are not shown since they do not contribute to the run-time structure of the system. The application software modules are then built on the top of the system level. The user then accesses the applications, which remain under the control of the design supervisor and are monitored by the project management system.

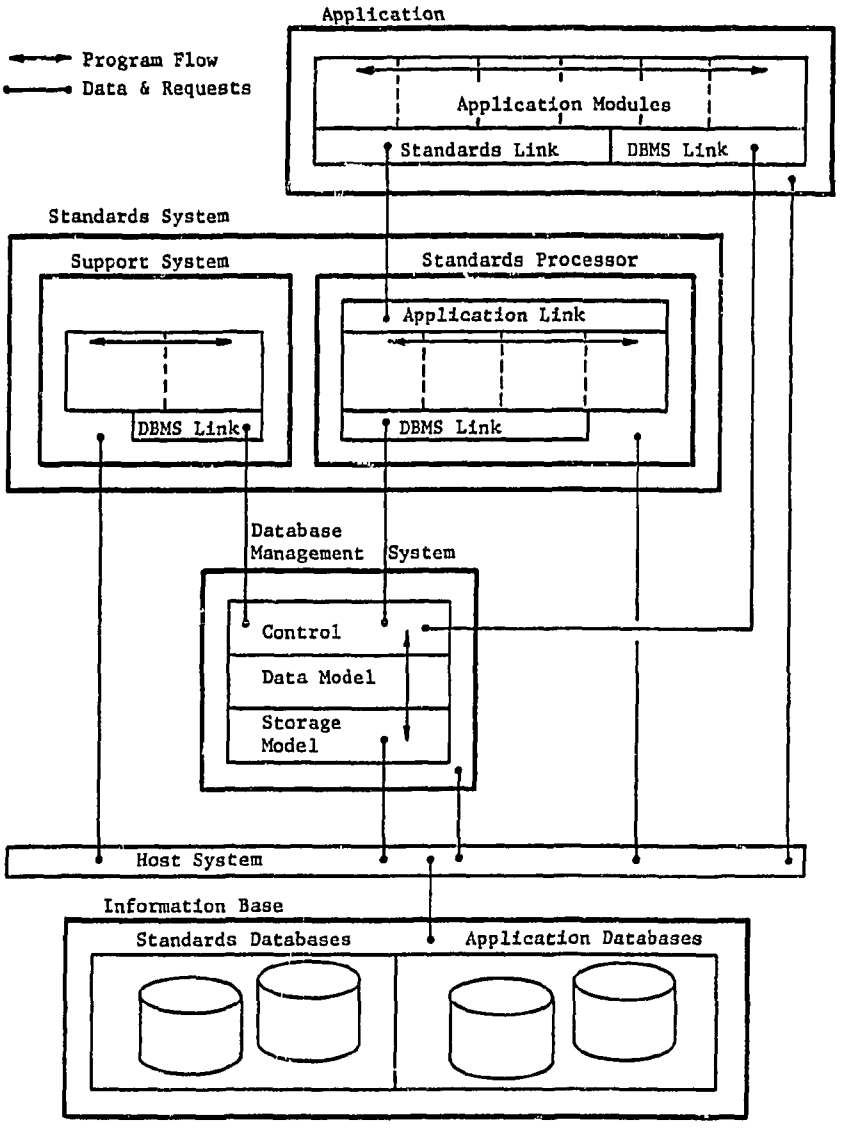


Figure 5.2. Standards Processing System

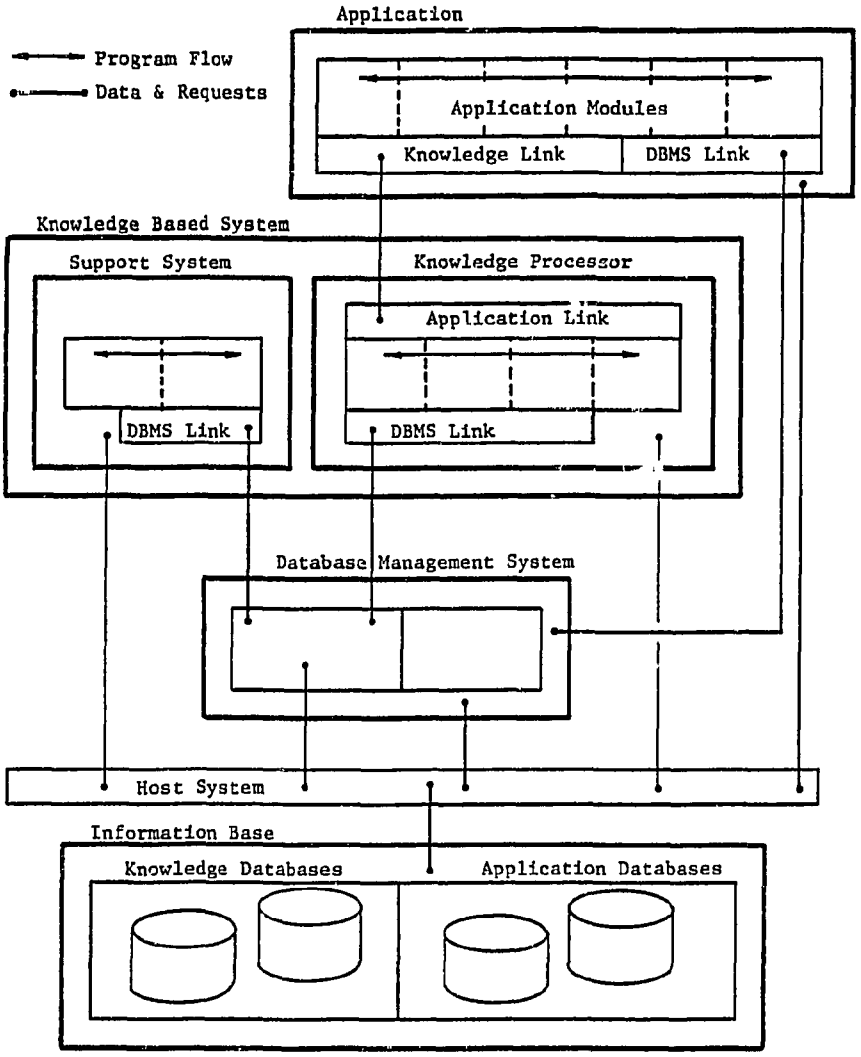


Figure 5.3. Knowledge Processing System

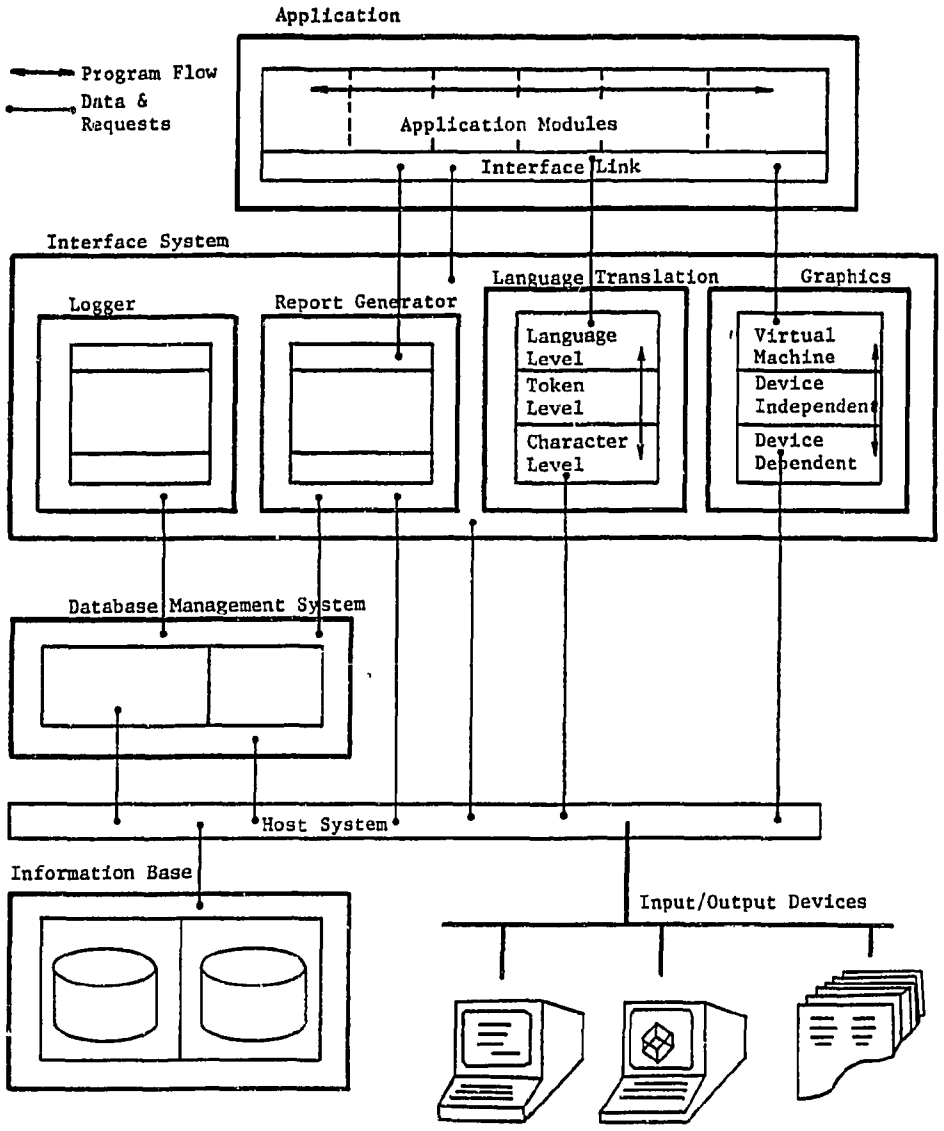


Figure 5.4. Interface System

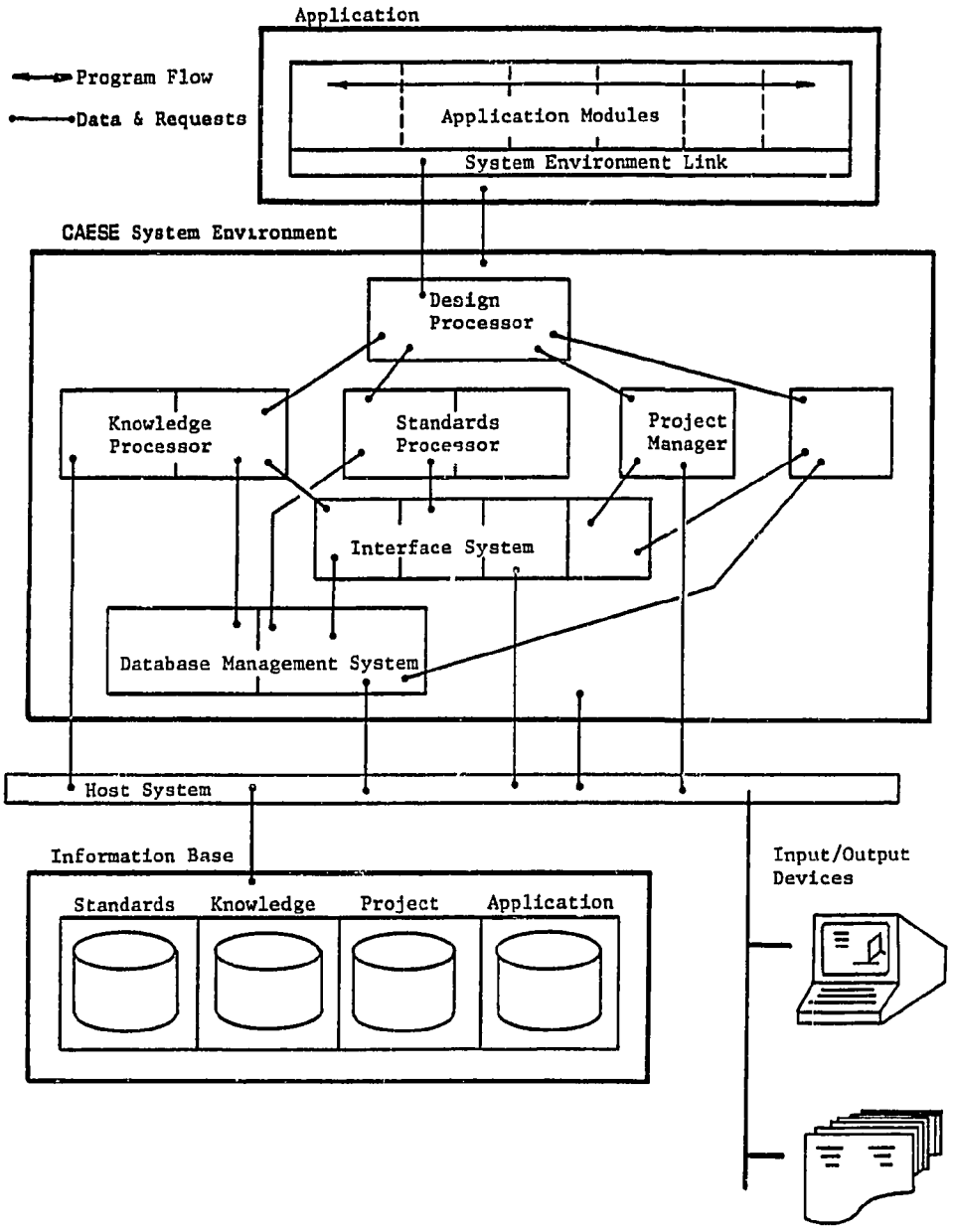


Figure 5.5. Application System

6. DISCUSSION

In the preceding chapters, a number of problems limiting the development of advanced engineering software, and a number of potential solution techniques for these problems were presented. A description of CAESE, a proposed prototype for the next generation of engineering software systems was also presented. The following is a discussion of the proposed solution approach, what prospects there are for the implementation of a system like CAESE, and what problems still remain to be solved.

6.1 Why the Problems are Currently Unsolvable

Each of the problem domains described in chapter 2 has a relatively simple and straightforward description. The descriptions are intentionally vague and rather general; they are first level descriptions of very general, open-ended applications. The purpose of the generality was to insure that the solutions would not be over-constrained. The solutions should reflect the generality and open-ended nature of the problem domains. In this way, they will be adaptable and applicable in both current and future design and engineering problem solving environments.

In addition to the two problem domains, a number of technical problems were described in chapter 3. These problems result from the scope and the generality of features desired in engineering software systems (including those which implement solution systems for the problem domains). Complete solution systems for the two problem domains would represent state-of-the-art engineering software systems. If such solution systems were implemented, they would contain features which are not available in current applications. Due to the generality of the solutions and the current state of software technology for engineering systems, it does not seem to be feasible to develop acceptable solutions for the two problem domains without addressing the technical problem areas described in chapter 3.

The basic technology to provide the solutions to these problem areas is available, either as techniques which are currently used in engineering software systems, or as techniques which can be taken from computer science research. Even though the technology and some prototype tools exist, nothing is available for direct use in, and application to, the problem domains.

Adaptations will take time; the techniques must be tailored to the engineering environment and converted into production software tools.

Basically, there is no framework for developing software for general purpose, open-ended problem domains similar to those described in chapter 2. The current software technology has been applied only in a limited number of areas. Integrated, multi-disciplinary, engineering design software systems do not exist. There are no large-scale production engineering software systems using techniques such as knowledge based systems or relational databases. Most applications of these technologies are still in computer science research.

There appear to be two potential solution approaches for developing advanced engineering applications and computer based design systems:

Brute Force: In this approach, software systems are developed to solve the specific problems at hand. Such systems would be conceived to solve only these problems, and they would be based on the direct applications of current tools and techniques. These systems will work; they will solve the problems described earlier; but they will do no more. Such solution will tend to be unresponsive, cumbersome, and complex. The brute force approach would be a continuation of what the profession is currently doing — developing ad hoc programs. This approach has not solved the problems, nor has it overcome the difficulties associated with developing general design and engineering systems (some of the various issues such as standards processing have been known for several years, and production systems have not yet been developed). There is no reason to believe that a continuation of this approach will be successful in the future. Attempts at solutions based on the brute force approach have produced more problems rather than solutions (that is how the work described herein evolved). These attempts resulted in a better understanding of the problems, and this has led to a new set of issues to be resolved.

A major part of the problem of developing advanced engineering software is not with the variety of technical issues, but rather, it is with the solution approach. Ad hoc, rigid solutions do not work for general, ill-defined, open-

ended problems. The current solutions are rigid, unadaptable, and inflexible because they are based on technologies which are rigid, inflexible, and do not provide the means to address open-ended, ill-defined problems.

Sophisticated Software: This approach is based on the concept of developing a new software technology base which is responsive and addresses the specific problem areas which limit the development of computer applications for engineering. This approach is based on the application of sophisticated, state-of-the-art software techniques. The goal is to produce general, open-ended, extensible, responsive solutions. With such a system, it should be possible to address the open-ended, ill-structured problems currently limiting the development of engineering applications. CAESE is designed to be such a system.

The approach of extending the software technology base, and providing a more sophisticated software environment, is identical to what was done in the development of the support-supervisory systems. These systems were developed because the then current brute force approach to software development did not successfully meet the needs of engineering applications.

The use of a sophisticated software technology has been successful in the past. FINITE provides an example of the usefulness of such an approach. Software complexity measures [WaltC77, SchnV78] indicate that a system like FINITE (120000 lines of code, 1500 subroutines) should require 406-413 man-months of development, with a project duration of 20-23 months (these values are based on conventional programming practice, i.e., the brute force approach, and may have a margin of error of 40%). This estimate does not account for the fact that FINITE would be significantly larger (2-3 times) if developed without the use of POLO, using the brute force approach. This size increase implies a development effort of 768-1236 man-months. The actual development effort was approximately 100-150 man-months (accurate data is not available, but the development team consisted of 4 individuals each contributing 2-3 man-years). This is effectively an order of magnitude reduction. A major portion of this reduction can be attributed to the use of the advanced software technology provided by POLO.

The use of the appropriate technology serves to reduce the complexity of the software product, and it permits software with advanced capabilities and features to be more readily developed. The continuation of the development of advanced software support technologies appears to be a viable approach to solving the current problems.

6.2 Application to the Problem Domains

The various solution techniques discussed in chapter 4, and CAESE, as described in chapter 5, are designed to address the various aspects of the two problem domains of chapter 2. The following is a short description of how these techniques and CAESE will help in the development of computer applications for these two problem domains.

6.2.1 Problem A — A Computer Aided Design System

CAESE is designed to meet the needs, and to respond to the problems, described in section 2.1, and it contains many of the features and capabilities outlined in section 2.1.3. It is directly applicable to the computer aided design system problem domain. If CAESE existed, it could be used to develop and support the advanced design and engineering software which is needed by our profession. The significant features of CAESE, relative to this problem domain, are the use of knowledge based systems and relational database management.

The use of a knowledge based system permits the problems associated with developing a solution to the ill-structured design problem to be addressed. Knowledge based systems provide a mechanism: (1) to represent design algorithms, and (2) to perform standards processing including access to a standard's provisions and feedback from computations. A knowledge based system approach to engineering software provides the flexibility and structure to develop a system which is adaptable. Since a knowledge based system will determine its own problem solving strategy, and since the linkages between the various problem solving components and data items are weak, the use of knowledge based systems yields the types of adaptable, flexible, and extensible systems which are needed for a computer aided design application.

The use of an extended engineering relational database management system provides the mechanisms to address all of the various problems associated with data handling and data integration. Since the data is content addressed, and

accesses from the application to the database are weakly coupled, the resulting engineering software system is flexible and extensible.

Besides having the basic components, form, and structure to address the needs of design and engineering applications, CAESE has a number of specific features which are useful for this problem domain. The various interface features, project management system, and software development and support environment all assist in developing advanced applications with less work. These, and other features such as the information storage and retrieval component of the data manager, provide a total system which is well suited to the needs of the engineer, and which has a number of components which need not be developed for every application.

6.2.2 Problem B — User Interfaces for Finite Element Systems

CAESE is not directly applicable to the finite element interface problem domain described in section 2.2. One of the requirements for the interfaces was that the kernel finite element system be FINITE. FINITE relies on POLO for its support, and a change to a different base system would be equivalent to redeveloping the application. In fact, the features of CAESE are such that a simple, direct conversion would not be appropriate. However, the development of a finite element application based on CAESE would be significantly simpler, the resulting code would be cleaner and less cluttered, and it would require less effort than was spent in the development of FINITE. The development of FINITE was aided by the existence of POLO and the features it provides. CAESE may be considered to be a successor to POLO; it provides features which would further simplify the development of a finite element application.

In CAESE, there are a variety of features to support user interfaces. The basic graphics components and an extensible graphics core would simplify the graphics programming task. An extended set of input language translation features would also ease the development of the user interfaces, and would permit more work to be performed by the system supplied software. Other features, such as a single error handler and the logger built into the system, would fulfill needs and provide a more usable system. CAESE provides the features needed to develop a finite element system which will have the capabilities, and which will respond to the needs, described in section 2.2.

In addition to the interface features, the other capabilities of CAESE are potentially useful, and may lead to a finite element system with a rather different structure. Consider the use of the data tracking features in the data manager. This capability, combined with a goal directed, data flow architecture system design could be used to eliminate all of the program development associated with controlling the computational process. Associated with individual processes would be declarations of data requirements and data products. A program goal of a set of final results, as requested by the user, could be established. The system would then automatically determine, based on the relationships between data items, which data items need to be computed in what order to arrive at the final, requested results. The majority of the conventional programming for implementing the problem solving strategy is of the form "do this, then do this, then this, etc." All of the overall programming strategy of this type would be eliminated. Changing the relationships between data items would change the program flow without requiring the reprogramming of the algorithms. This is extremely useful and powerful, since it permits complex processing to occur without the direct programming of any of the complex linkages.

Other features of CAESE, such as the relational form of the database would eliminate much detailed programming. Much of the complex code used to transform one data representation to another representation would not be needed. Other capabilities, such as operator overloading could also reduce development effort and code complexity. Simple, direct encoding of matrix, tensor, and other types of engineering operations in a programming language permits it to regain some of the elegance and conciseness of our mathematical forms.

6.3 Unresolved Issues

A system like CAESE is not a cure-all. There are a number of issues which have not been resolved. The two most important unresolved items appear to be: (1) the selection of a computer technology base, and (2) social and legal acceptance problems. The following discusses these issues, but it does not provide any solutions.

6.3.1 Computer Technology Base

The problems due to the rapidly changing computer technology base were presented in section 3.5. Section 4.5 and the glossary (section 4 and 5) presented a number of computer languages and language techniques which could assist in producing better software, but the prototype design of CAESE does not address any of these issues. Nothing in the design of CAESE is oriented towards a particular language, a particular hardware configuration, or a particular systems approach. The only requirement is that the system be oriented towards interactive usage.

The system, indeed any new application, should be designed to function in a variety of hardware and systems environments. This is necessary for it to gain widespread acceptance and use, to be adaptable, and to be long-lived. Machine and operating system dependencies are inevitable. The objective is to minimize these dependencies, and more importantly, to recognize what types of machine dependent features are needed, and to isolate these. Isolation does not eliminate such problems, it only localizes them, and reduces their impact on the remainder of the total system.

Potentially, a more important issue is the selection of a programming language. Each of the different languages have a number of features which are potentially beneficial and others which may be detrimental. It is desirable to develop the complete, detailed system design without being concerned with an implementation. In this way, biases towards a particular language can not manifest themselves in the final system structure. Once the complete design is prepared, an evaluation of the then current, applicable languages can be made, based on the actual needs.

6.3.2 Social and Legal Issues

A system like CAESE provides the engineer with an approach to computer based problem solving which is quite different from that commonly in use today. As a result, it is expected that there will be considerable resistance from the engineering community to the acceptance and use of any system like CAESE. The system presents a radical change (that of a totally integrated, computer based, engineering environment), and organizations resist change. The various political, organizational, and social problems [KlinR80, KeenP81] all present serious questions about the attempts to improve computer usage. An engineering computer system, and the resulting improvements in engineering, can be readily justified in terms of their savings and their producing better

designs, but this does not insure acceptance. Engineers are accustomed to their current practice. They traditionally have not been responsive to innovations in the design process. Introduction of techniques and changes in procedures and standards have been slow to be accepted. There is no reason to expect that a new approach to computer applications should be received differently.

The use of a computer based design system also poses serious legal questions. When design work is performed by a computer, who will take legal responsibility for the design. Engineers may be reluctant to approve work which they did not personally perform. It will be impossible for the engineer to verify all computations and results. It will be equally difficult to verify that the software is error free, and the host hardware is performing without errors. The software developers will be reluctant to accept legal responsibility for their systems (currently software is released with a disclaimer absolving the software developer from all responsibility and placing this responsibility on the user). This problem is complicated by the inclusion of standards processing. The computer implementation of a standard is a representation of the legal requirements for design and engineering. The interpretation problem of expressing the machine processable form of the standard now has legal implications. All of these legal questions regarding liability due to the use of a computer based design system will affect the acceptance of such systems.

It is important that such problems are recognized, and if possible, prepared for. These issues should not deter the development of a new approach to engineering computer applications. The various technical problems continue to exist, and there is the need to anticipate the future needs of computer usage within the engineering profession, regardless of professional acceptance.

6.4 Conclusions

The computer is a powerful engineering tool. However, as discussed, its utilization is well below its potential. This underutilization is not due to a neglect of its power, nor is it due to any explicit desire not to have the computer do more. The applications which the profession is now trying to computerize are much more complex and ill-structured than any attempted in the past. In attempting to develop these new applications, it is necessary to push the technology which represents how engineering processes are computerized to its limits. The technological limits of the current

generation of software used to support engineering applications are now being reached. New engineering application systems will exceed the capacity of the current software tools and support-supervisory systems, and will require capabilities which are not present in these systems. Pushing the applications beyond the capabilities of the technology only results in serious problems. Current problems result from trying to develop computer systems for the eighties and beyond based on the technology developed and used in the sixties. This is a hopeless situation. Indeed, attempts at developing advanced features in current computer applications based on the current technology have not been successful.

This lack of success in developing advanced engineering applications is based on the lack of a suitable technological base for engineering software systems. Computer science research has developed a number of new techniques and concepts which can be utilized in engineering applications. Engineers have done little to incorporate these ideas into their work. Computer science researchers have done nothing to address the engineer's problems. The gap between engineering problems and the current technology used to solve these problems, and between this technology and the state-of-the-art technology increases. These relations are depicted in figure 6.1 (revised from figure 1.1).

Knowledge based systems, relational database management and other topics from computer science appear to be beneficial to solving the types of technological problems which are appearing in the attempts to develop advanced engineering software systems. It is time that the engineering profession take these techniques and convert them into a set of software tools which are applicable to engineering and engineering problems. Software tools designed for dealing with loosely structured, ill-defined problems appear to be a viable approach to solving the current engineering software development problems. These tools can form the basis for the next generation of engineering software system.

Two choices exist for the profession; one is to neglect advances in computer technology, developing ad hoc engineering applications as in the past. The other is to try to select what is useful from the computer science research community, and adapt it to the needs of the engineering profession. The desire for advanced features and capabilities in engineering applications will increase, and without changing the current approach to engineering software development, there is no way to meet these desires and to fulfill the future software needs of the profession.

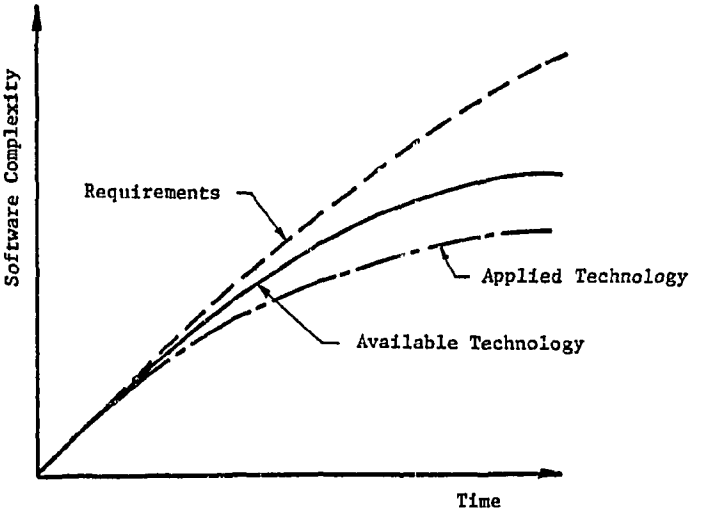


Figure 6.1. Software Technology

6.5 The Next Step

The various technologies and the preliminary design of CAESE are just that, technologies and a preliminary design. They represent only the first step along the road to changing the current approach to the development of engineering computer applications. The ideas presented herein form the starting point for the development of the next generation of engineering computer systems. A logical next step would be to proceed with the development of this next generation of software.

This step is going to be long and difficult, consisting of overlapping a number of phases, as described below. A work schedule for an implementation of CAESE and a first application is shown in figure 6.2. The bars on the graph are in correct proportions to each other, but an absolute time scale has been specifically excluded. It is too early to accurately estimate the total effort involved, but it can be expected that the resulting system will be on the order of many tens of thousands of lines of code (50000-500000) and a total effort being measured in tens of man-years.

The first phase will be to review the preliminary design, and to obtain more information on the details of current relational database management and knowledge based systems, since these two areas are changing rapidly. Then it will be necessary to select aspects of all of the technologies which are most applicable to the engineering problem domains, and to proceed with a complete, detailed design of the prototype version of CAESE. The second phase will be an implementation of the prototype system. With such a system, the actual viability of the approach can be tested. The prototype must then be evaluated. Minor changes can be made as the next version of the system is developed. If major problems are encountered they must be resolved and further testing done. The third phase will be the development of a complete version of CAESE, with all the "bells and whistles." This version will be used to support the first application.

Once the complete, detailed design of the production version of the system software is available, the design and implementation of the applications can proceed. An appropriate first application must be selected. The problem domain must be sufficiently large to exhibit all of the various problems described earlier, yet it must not be so large that the scope will be beyond what can be handled successfully in the first test. The significance of this first application can not be underestimated. The technical acceptance of the system will not come from the design of the base system. The true

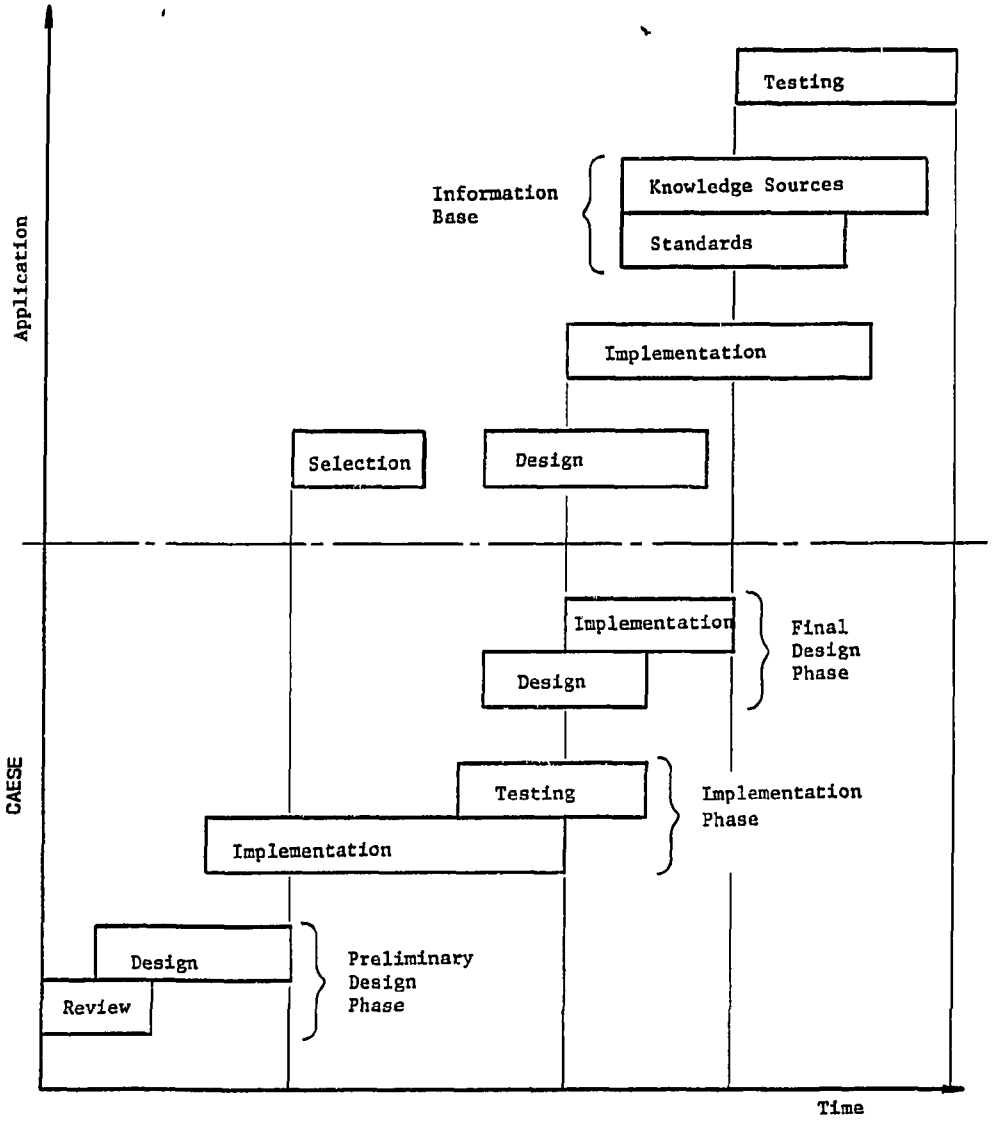


Figure 6.2. CAESE Implementation Schedule

acceptance will come only from the application (the previous support-supervisory systems are best known for their major applications, not for the systems themselves).

Once an application has been selected, it must be implemented. This will require the design of the application system and its subsystems. Along with the development of code of the application will be the development of the processable forms of the various standards which will be used. Similarly, the various knowledge sources and the rules which determine how the system will operate must be developed. All these pieces can then be integrated to complete the application. At this point the application will be ready for full scale testing. Actual engineering problems, those for which existing solutions are known, must be redesigned using the application system. Comparisons with the existing solutions will determine how well CAESE and the application perform, if they are usable, if they have technical problems (either in the application or in the base system), or if they are too costly and unresponsive. Then will be time to step back and analyze what has been created, and to determine what the future might be.

6.6 Epilogue

We keep talking about it.
We say we want it.
We say we are going to do it.

But we never make any real progress.

Maybe it is hard.
Maybe we are afraid of it.

REFERENCES

ACI71 ————, Building Code Requirements for Reinforced Concrete, (ACI 318–71), American Concrete Institute (ACI), Detroit, Michigan, 1971.

AISC70 ————, "Specification for the Design, Fabrication and Erection of Structural Steel for Buildings," Manual of Steel Construction, Seventh Edition, American Institute of Steel Construction (AISC), New York, 1970.

AISC80 ————, "Specification for the Design, Fabrication and Erection of Structural Steel for Buildings," Manual of Steel Construction, Eight Edition, American Institute of Steel Construction (AISC), Chicago, 1980.

AlwoR72 Alwood, R. J, and Maxwell, T. O'N., "GENESYS — A Machine Independent System," Proceedings, Colloque International sur les Systèmes Intégrés en Génie Civil, Principes et Description Générale des Systèmes Intégrés, [International Colloquium on Integrated Systems in Civil Engineering, Principles and General Description of Integrated Systems], Editeur G. Deprez, Centre d'Etudes pour la Promotion des Ordinateurs dans la Construction [Center of Studies for the Promotion of Computers in Construction (CEPOC)], Université de Liège, Liège, Belgium, Vol. 1, No. 1.1, April, 1974.

AstrM76 Astrahan, M. M., et al., "System R: Relational Approach to Database Management," ACM Transactions on Database Systems (TODS), Association for Computing Machinery (ACM), Vol. 1, No. 2, pp. 97–137, June, 1976.

BackJ78a Backus, J., "The History of FORTRAN I, II, and III," Preprints, ACM-SIGPLAN, History of Programming Languages Conference, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 13, No. 8, pp. 165–180, August, 1978.

BackJ78b Backus, J., "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs," Communications of the ACM (CACM), Association for Computing Machinery (ACM), Vol. 21, No. 8, pp. 613–641, August, 1978.

BaerA79 Baer, A., Eastman, C., and Henrion, M., "Geometric Modeling: A Survey," Computer Aided Design, Vol. 11, No. 5, pp. 253–272, September, 1979.

BellK73 Bell, K., Hatlestad, B., Hansteen, O. E., and Araldsen, P. O., NORSAM; A Programming System For the Finite Element Method, User's Manual, Part 1, General Description, Selskapet for industriell og teknisk forskning ved Norges Tekniske Hogskole [The Engineering Research Foundation at The Norwegian Institute of Technology], Trondheim, Norway, February, 1973.

BogeR75 Bogen, R., et al., MACSYMA Reference Manual, Laboratory of Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1975.

BuchB69 Buchanan, B., Sutherland, G., and Feigenbaum, E. A., "Heuristic DENDRAL: A Program for Generating Explanatory Hypotheses in Organic Chemistry," In Machine Intelligence 4, American Elsevier, New York, 1969.

BurnB78 Burner, B., Ives, F., Lixvar, J., and Shovlin, D., "The Design, Evaluation, and Implementation of the IPAD Distributed Computing System," Proceedings, American Society of Civil Engineers Conference on Computers in Civil Engineering, American Society of Civil Engineers (ASCE), Atlanta, Georgia, pp. 126–144, June, 1978.

CanoM80 Canon, M. D., et al., "A Virtual Machine Emulator for Performance Evaluation," Communications of the ACM (CACM), Association for Computing Machinery (ACM), Vol. 23, No. 2, pp. 71–80, February, 1980.

ChamD76 Chamberlin, D. D., "Relational Data-Base Management Systems," Computing Surveys, Association for Computing Machinery (ACM), Vol. 8, No. 1, pp. 43–66, March, 1976.

DateC75 Date, C. J., An Introduction to Database Systems, Addison Wesley, Reading, Massachusetts, 1975.

DaviR77 Davis, R., "Generalized Procedure Calling and Content-Directed Invocation," SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 12, No. 8, pp. 45–54, August, 1977.

DeMiR79 DeMillo, R. A., Lipton, R. J., and Perlis, A. J., "Social Processes and Proofs of Theorems and Programs," Communications of the ACM (CACM), Association for Computing Machinery (ACM), Vol. 22, No. 5, pp. 271–280, May, 1979.

DoD80 ————, Reference Manual for the Ada Programming Language, Defense Advanced Research Projects Agency (DARPA), United States Department of Defense (DoD), Washington, D.C., June, 1980.

DoddR78 Dodds, R. H., Jr., Lopez, L. A., and Pecknold, D. A., Numerical and Software Requirements for General Nonlinear Finite Element Analysis, UILU–ENG–78–2020, Civil Engineering Studies, Structural Research Series, No. 454, Department of Civil Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois, September, 1978.

DoddR80 Dodds, R. H., Jr., and Lopez, L. A., "A Generalized Software System for Nonlinear Analysis," Advanced Engineering Software, Vol. 2, No. 4, pp. 161–168, 1980.

EastC76 Eastman, C., Lividini, J., and Stoker, D., "Database for Designing Large Physical Systems," Workshop on Computer Representation of Physical Systems, Carnegie-Mellon University, Pittsburgh, Pennsylvania, August, 1976.

EastC77 Eastman, C., and Henrion, M., "GLIDE: A Language for Design Information Systems," Computer Graphics, Special Interest Group on Computer Graphics of the Association for Computing Machinery (SIGGRAPH-ACM), Vol. 11, No. 2, pp. 24–33, Summer, 1977.

EastC80 Eastman, C., GLIDE2 User's Manual, Institute of Building Sciences, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1980.

Egel074 Egeland, O., and Araldsen, P., "SESAM-69, A General Purpose Finite Element Program," Computers and Structures, Vol. 4, No. 1, pp. 41–68, January, 1974.

ErmaL80 Erman, L. D., Hayes-Roth, F., Lesser, V. R., and Reddy, D. R., "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty," Computing Surveys, Association for Computing Machinery (ACM), Vol. 12, No. 2, pp. 213–253, February, 1980.

FairR80 Fairley, R. E., "Ada Debugging and Testing Support Environments," Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 15, No. 11, pp. 16–25, November, 1980.

FalkA73 Falkoff, A. D., and Iverson, K. E., "The Design of APL," IBM Systems Journal, International Business Machines (IBM), Vol. 17, No. 4, pp. 324–334, July, 1973.

FalkA78 Falkoff, A. D., and Iverson, K. E., "The Evolution of APL," Preprints, ACM-SIGPLAN, History of Programming Languages Conference, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 13, No. 8, pp. 47–57, August, 1978.

FenvS64 Fenves, S. J., Logcher, R. D., and Mauch, S. P., STRESS — A User's Manual, MIT Press, Cambridge, Massachusetts, 1964.

FenvS66 Fenves, S. J., "Tabular Decision Logic for Structural Design," Journal of the Structural Division, American Society of Civil Engineers (ASCE), Vol. 92, No. ST6, pp. 473–490, December, 1966.

FenvS69 Fenves, S. J., Gaylord, E. H., and Goel, S. K., Decision Table Formulation of the 1969 AISC Specification, Civil Engineering Studies, Structural Research Series, No 347, Department of Civil Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois, August, 1969.

FenvS73 Fenves, S. J., "Representation of the Computer-Aided Design Process by a Network of Decision Tables," Computers and Structures, Vol. 3, No. 5, pp. 1099–1107, September, 1973.

FenvS79a Fenves, S. J., Performance Requirements for Standards Processing Software, R–79–111, Department of Civil Engineering, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Also available from National Bureau of Standards (NBS), NBS GCR 80–257, United States Department of Commerce, Washington, D.C., April, 1979.

FenvS79b Fenves, S. J., Functional Specifications for Standards Processing Software, R–120–679, Department of Civil Engineering, Carnegie-Mellon University, Pittsburgh, Pennsylvania, Also available from National Bureau of Standards (NBS), NBS GCR 80–258, United States Department of Commerce, Washington, D.C., June, 1979.

FryJ76 Fry, J. P., and Sibley, E. H., "Evolution of Data-Base Management Systems," Computing Surveys, Association for Computing Machinery (ACM), Vol. 8, No. 1, pp. 7–42, March, 1976.

GA080 ————, Use of Computers by Firms Providing Architect-Engineer Services to Federal Agencies, LCD–81–2, United States General Accounting Office (GAO), Washington, D.C., October, 1980.

GarrC74 Garrocq, C. A., and Hurley, M. J., "The IPAD System: A Future Management / Engineering / Design Environment," Proceedings of the Design Automation Workshop, Institute of Electrical and Electronics Engineers (IEEE), Vol. 11, pp. 327–334, June, 1974.

GaylE72 Gaylord, E. H., and Gaylord, C. N., Design of Steel Structures, Second Edition, McGraw-Hill, New York, 1972.

GKS79 ————, Information Processing Graphical Kernel System (GKS) Functional Description, Proposal of Standard DIN 00 66 252, 1979.

GoelS71 Goel, S. K., and Fenves, S. J., "Computer-Aided Processing of Design Specifications," Journal of the Structural Division, American Society of Civil Engineers (ASCE), Vol. 97, No. ST1, pp. 463–479, January, 1971.

GrovL80 Groves, L. J., and Rogers, W. J., "The Design of a Virtual Machine for Ada," Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 15, No. 11, pp. 223–234, November, 1980.

GSPC79 ————, "Status Report of the Graphic Standards Planning Committee, Part II: General Methodology and the Proposed Core System Standard (Revised)," Computer Graphics, Special Interest Group on Computer Graphics of the Association for Computing Machinery (SIGGRAPH-ACM), Vol. 13, No. 3, August, 1979.

HarrJ75a Harris, J. R., Melin, J. W., Tavis, R. L., and Wright, R. N., Technology for the Formulation and Expression of Specifications, Volume I: Final Report, UILU-ENG-75-2029, Civil Engineering Studies, Structural Research Series, No. 423, Department of Civil Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois, December, 1975.

HarrJ75b Harris, J. R., Melin, J. W., and Albarran, C., Technology for the Formulation and Expression of Specifications, Volume II: Program User's Manual, UILU-ENG-75-2030, Civil Engineering Studies, Structural Research Series, No. 424, Department of Civil Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois, December, 1975.

HarrJ80 Harris, J. R., Organization of Building Standards: Systematic Techniques for Scope and Arrangement, Unpublished doctoral thesis, Department of Civil Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1980.

HernE74 Herness, E. D., and Tocher, J. L., "Design of Pre- and Postprocessors," In Structural Mechanics Computer Programs, Surveys, Assessments, and Availability, Ed. W. Pilkey, K. Saczalski, and H. Schaeffer, University of Virginia Press, Charlottesville, Virginia, pp. 887-898, 1974.

IBM81a ————, SQl/Data System General Information, GH24-5012-0, International Business Machines (IBM), White Plains, N.Y., 1981.

IBM81b ————, SQL/Data System Concepts and Facilities, GH24-5013-0, International Business Machines (IBM), White Plains, N.Y., 1981.

IPAD80 ————, IPAD: Integrated Programs for Aerospace-Vehicle Design, NASA Conference Publication 2143 (CP-2143), National Aeronautics and Space Administration (NASA), Washington, D.C., 1980.

IverK62 Iverson, K. E., A Programming Language, John Wiley and Sons, New York, 1962.

JensK76 Jensen, K., and Wirth, N., PASCAL User Manual and Report, Second Edition, Springer-Verlag, New York, 1976.

JensR79 Jensen, R. W., and Tonies, C. C., Software Engineering, Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

KeenP81 Keen, P. G. W., "Information Systems and Organizational Change," Communications of the ACM (CACM), Association for Computing Machinery (ACM), Vol. 24, No. 1, pp. 24-33, January, 1981.

KernB76 Kernighan, B. W., and Plauger, P. J., Software Tools, Addison Wesley, Reading, Massachusetts, 1976.

KimW79 Kim, W., "Relational Database Systems," Computing Surveys, Association for Computing Machinery (ACM), Vol. 11, No. 3, pp. 185–211, September, 1979.

KlinR80 Kling, R., "Social Analyses of Computing; Theoretical Perspectives in Recent Empirical Research," Computing Surveys, Association for Computing Machinery (ACM), Vol. 12, No. 1, pp. 61–110, January, 1980.

LatoJ77 Latombe, J-C., "Artificial Intelligence in Computer-Aided Design," In CAD Systems, Proceedings of the IFIP Working Conference on Computer-Aided Design Systems, Ed. J. J. Allan, III., North-Holland, pp. 61–170, 1977.

LevyD80 Levy, D., Mittman, B., and Newborn, M., "3rd World Computer Chess Championship," Communications of the ACM (CACM), Association for Computing Machinery (ACM), Vol. 23, No. 11, pp. 661–664, November, 1980.

LogcR67 Logcher, R. D., et al., ICES STRUDL-I, The Structural Design Language, Engineering User's Manual, R67–56, Civil Engineering Systems Laboratory, Department of Civil Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1967.

LopeL72a Lopez, L. A., "POLO — A Supervisor for Integrated Systems Development," Proceedings, Colloque International sur les Systèmes Intégrés en Génie Civil, Principes et Description Générale des Systèmes Intégrés, [International Colloquium on Integrated Systems in Civil Engineering, Principles and General Description of Integrated Systems], Editeur G. Deprez, Centre d'Etudes pour la Promotion des Ordinateurs dans la Construction [Center of Studies for the Promotion of Computers in Construction (CEPOC)], Université de Liège, Liège, Belgium, Vol. 1, No. J.6, April, 1974.

LopeL72b Lopez, L. A., "POLO — Problem Oriented Language Organizer," Computers and Structures, Vol. 2, No. 4, pp. 555–572, September, 1972.

LopeL77a Lopez, L. A., "FINITE: An Approach to Structural Mechanics Systems," International Journal for Numerical Methods in Engineering, Vol. 11, No. 5, pp. 851–866, 1977.

LopeL77b ————, Report on the Workshop for Software Coordination within the University Environment, National Science Foundation Project Report (NSF), Prepared by L. A. Lopez, Available from National Technical Information Service (NTIS), PB 273686/AS, United States Department of Commerce, Springfield, Virginia, October, 1977.

LopeL77c ————————, Appendices to the Report on the Workshop for Software Coordination within the University Environment, National Science Foundation Project Report (NSF), Prepared by L. A. Lopez, Available from National Technical Information Service (NTIS), PB 273687/AS, United States Department of Commerce, Springfield, Virginia, October, 1977.

LopeL79a Lopez, L. A., Dodds, R. H., Rehak, D. R., and Urzua, J., POLO-FINITE, A Structural Mechanics System, User's Manual, Civil Engineering Systems Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois, Department of Civil Engineering and the Academic Computer Center, University of Kansas, Lawrence, Kansas, 1979.

LopeL79b Lopez, L. A., "Software Problems in the University Environment," Journal of the Technical Councils, American Society of Civil Engineers (ASCE), Vol. 105, No. TC2, pp. 385–399, December, 1979.

LopeL80 Lopez, L. A., Dodds, R. H., Rehak, D. R., and Urzua, J., POLO-FINITE, A Structural Mechanics System, Example Solutions Manual, Civil Engineering Systems Laboratory, University of Illinois at Urbana-Champaign, Urbana, Illinois, Department of Civil Engineering and the Academic Computer Center, University of Kansas, Lawrence, Kansas, 1980.

LoveL42 Lovelace, Lady A. A., Notes upon the Memoir "Sketch of the Analytical Engine Invented by Charles Babbage," By L. F. Menabrea (Geneva, 1842), Reprinted in Charles Babbage and His Calculating Engines, Ed. P. Morrison and E. Morrison, pp, 248–249, 284. Dover Publications, New York, 1961.

MacNR71 MacNeal, R., and McCormick, C. W., "The NASTRAN Computer Program for Structural Analysis," Computers and Structures, Vol. 1, No. 3, pp. 389–412, October, 1971.

McCaJ62 McCarthy, J., et al., Lisp 1.5 Programmer's Manual, MIT Press, Cambridge, Massachusetts, 1962.

McCaJ78 McCarthy, J., "History of LISP," Preprints, ACM-SIGPLAN, History of Programming Languages Conference, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 13, No. 8, pp. 217–223, August, 1978.

McCoC72 McCormick, C. W., The NASTRAN User's Manual, NASA Specialty Publication 222(01) (SP–222(01)), National Aeronautics and Space Administration (NASA), Washington, D.C., June, 1972.

MeloR78 Melosh, R. J., Marcal, P. V., and Berke, L., "Structural Analysis Consultation Using Artificial Intelligence," In Research in Computerized Structural Analysis and Synthesis, NASA Conference Publication 2059 (CP–2059), National Aeronautics and Space Administration (NASA), Washington, D.C., October, 1978.

MichA76 Michaels, A. S., Mittman, B., and Carlson, C. R., "A Comparison of Relational and CODASYL Approaches to Data-Base Management," Computing Surveys, Association for Computing Machinery (ACM), Vol. 8, No. 1, pp. 125–151, March, 1976.

MichJ78 Michener, J. C., and van Dam, A., "A Functional Overview of the Core System with Glossary," Computing Surveys, Association for Computing Machinery (ACM), Vol. 10, No. 4, pp. 381–387, December, 1978.

MillC61 Miller, C. L., COGO — A Computer Programming System for Civil Engineering Problems, Department of Civil Engineering, Massachusetts Institute of Technology, Cambridge, Massachusetts, August, 1961.

MillR74 Miller, R. E., et al., "Feasibility Study of an Integrated Program for Aerospace Vehicle Design (IPAD)," Proceedings of the Design Automation Workshop, Institute of Electrical and Electronics Engineers (IEEE), Vol. 11, pp. 335–346, June, 1974.

Mo078 Mo, O., Klein, H. F., Pahle, E., and Harwiss, T., "Finite Element Programs Based on General Programming Systems," Computers and Structures, Vol. 8, No. 6, pp. 703–715, June, 1978.

NASA72a ————, The NASTRAN Theoretical Manual, Ed. R. H. MacNeal, NASA Specialty Publication (SP–221(01)), National Aeronautics and Space Administration (NASA), Washington, D.C., April, 1972.

NASA72b ————, The NASTRAN Programmer's Manual, NASA Specialty Publication (SP–223(01)), National Aeronautics and Space Administration (NASA), Washington, D.C., September, 1972.

NaurP60 Naur, P., et al., "Report on the Algorithmic Language Algol 60," Communications of the ACM (CACM), Association for Computing Machinery (ACM), Vol. 3, No. 5, pp. 299–314, May, 1960.

NeweA72 Newell, A., and Simon, H. A., Human Problem Solving, Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

PerlA78 Perlis, A. J., "The American Side of the Development of Algol," Preprints, ACM-SIGPLAN, History of Programming Languages Conference, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 13, No. 8, pp. 3–14, August, 1978.

RadiG78 Radin, G., "The Early History and Characteristics of PL/I," Preprints, ACM-SIGPLAN, History of Programming Languages Conference, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 13, No. 8, pp. 227–241, August, 1978.

RehaD79 Rehak, D. R., and Lopez, L. A., "A Tool for Translating Problem Oriented Languages," Journal of the Technical Councils, American Society of Civil Engineers (ASCE), Vol. 105, No. TC1, pp. 33–42, April, 1979.

RequA80 Requicha, A. A. G., "Representations for Rigid Solids: Theory, Methods, and Systems," Computing Surveys, Association for Computing Machinery (ACM), Vol. 12, No. 4, pp. 437–464, December, 1980.

RoosD66 Roos, D., ICES System Design, MIT Press, Cambridge, Massachusetts, 1966.

RossD59 Ross, D. T., APT System Documentation, General Description of the APT System, MIT Servo Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, Vol. I, June, 1959.

RossD78 Ross, D. T., "Origins of the APT Language for Automatically Programmed Tools," Preprints, ACM-SIGPLAN, History of Programming Languages Conference, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 13, No. 8, pp. 61–99, August, 1978.

SammJ78 Sammet, J. E., "The Early History of COBOL," Preprints, ACM-SIGPLAN, History of Programming Languages Conference, SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 13, No. 8, pp. 121–161, August, 1978.

SchaH78 Schaeffer, H. G., "A Review of the International Symposium on Structural Mechanics Software," Computers and Structures, Vol. 8, No. 5, pp. 589–598, May, 1978.

SchnV78 Schneider, V., "Prediction of Software Effort and Project Duration — Four New Formulas," SIGPLAN Notices, Special Interest Group on Programming Languages of the Association for Computing Machinery (SIGPLAN-ACM), Vol. 13, No. 6, pp. 49–59, June, 1978.

SchrE74 Schrem, E., "Development and Maintenance of Large Finite Element Systems," In Structural Mechanics Computer Programs, Surveys, Assessments, and Availability, Ed. W. Pilkey, K. Saczalski, and H. Schaeffer, University of Virginia Press, Charlottesville, Virginia, pp. 669–685, 1974.

SchrE77 Schrem, E., "From Program Systems to Programming Systems for Finite Element Analysis," In Formulations and Computational Algorithms in Finite Element Analysis: U.S.-German Symposium, Ed. K-J. Bathe, J. T. Oden, and W. Wunderlich, MIT Press, Cambridge, Massachusetts, pp. 163–190, 1977.

SchrE78 Schrem, E., "Functional Software Design and its Graphics Representation," Computers and Structures, Vol. 8, No. 3/4, pp. 491–502, May, 1978.

SchrE79 Schrem, E., "Trends and Aspects of the Development of Large Finite Element Software Systems," Computers and Structures, Vol. 10, No. 1/2, pp. 419–425, April, 1979.

ShorE76 Shortliffe, E. H., Computer-Based Medical Consultations: MYCIN, American Elsevier, New York, 1976.

SimoH73 Simon, H. A., "The Structure of Ill Structured Problems," Second Edition, Artificial Intelligence, Vol. 4, pp. 181--201, 1973.

TaneA76 Tanenbaum, A. S., "A Tutorial on ALGOL 68," Computing Surveys, Association for Computing Machinery (ACM), Vol. 8, No. 2, pp. 155—190, June, 1976.

TaylR76 Taylor, R. W., and Frank, R. L., "CODASYL Data-Base Management Systems," Computing Surveys, Association for Computing Machinery (ACM), Vol. 8, No. 1, pp. 67—103, March, 1976.

TricD76 Trichritizis, D. C., and Lochovsky, F. H., "Hierarchical Data-Base Management," Computing Surveys, Association for Computing Machinery (ACM), Vol. 8, No. 1, pp. 105—123, March, 1976.

TuriA50 Turing, A. M., "Computing Machinery and Intelligence," Mind, Vol. LIX, No. 236, 1950, Reprinted in Computers and Thought, Ed. E. A. Feigenbaum, and J. Feldman, McGraw-Hill, New York, pp. 11—38, 1963.

UBC76 ————, Uniform Building Code, International Conference of Building Officials, Whittier, California, 1976.

WaltC77 Waltson, C. E., and Felix, C. P., "A Method of Programming Measurement and Estimation," IBM Systems Journal, International Business Machines (IBM), Vol. 16, No. 1, pp. 54—73, 1977.

WaltD78 Waltz, D. L., "An English Language Question Answering System for a Large Relational Database," Communications of the ACM (CACM), Association for Computing Machinery (ACM), Vol. 21, No. 7, pp. 526—539, July, 1978.

WilsJ76 Wilson, J. L., and Lansberry, C. R., "Interactive Computer Graphics for Computer Aided Design in Civil Engineering," Computer Graphics, Special Interest Group on Computer Graphics of the Association for Computing Machinery (SIGGRAPH-ACM), Vol. 10, No. 2, pp. 89—96, Summer, 1976.

WinoT71 Winograd, T., Procedures as a Representation for Data in a Computer Program for Understanding Natural Languages, Ph.D. Thesis, MAC TR—84, Massachusetts Institute of Technology, Cambridge, Massachusetts, Reproduced by National Technical Information Service (NTIS), AD 721—399, United States Department of Commerce, Springfield, Virginia, February, 1971.

WinsP77 Winston, P. H., Artificial Intelligence, Addison Wesley, Reading, Massachusetts, 1977.

WrigR75 Wright, R. N., Harris, J. R., Melin, J. W., and Albarran, C., Technology for the Formulation and Expression of Specifications, Volume III: Technical Reference Manual, UILU—ENG—75—2031, Civil Engineering Studies, Structural Research Series, No. 425, Department of Civil Engineering, University of Illinois at Urbana-Champaign, Urbana, Illinois, December, 1975.

APPENDIX A. FINITE USER'S WISH LIST

The following is a short description of the features which are needed, or have been requested as modifications and extensions to FINITE. Familiarity with the details of the capabilities currently present in FINITE will be useful in understanding these requests [DoddR78, LopeL79a, LopeL80]. This discussion is based on the model of the system presented in section 2.2.3. There is no significance to the ordering of the items within the list.

Sizes not specified: The first data item required to describe any substructure is its size, in terms of the number of elements and nodes in the substructure. Providing these values is sometimes inconvenient for the user. This data is required by the system, but the model input processor could determine values based on the remaining data entered by the user. The lack of an exact value during data entry would create difficulties in strict error checking used to guarantee that all of a substructure's data has been provided.

Alphanumeric node and element labels: All node and element labels are currently integer quantities. Users would like the ability to use descriptive names for these items. The integer representation could be maintained within the computational kernel. The model input and output processors would need to deal with the alphanumeric labels and the translation of these to the internal integer form. Data structures to store the translation to the internal form also would be needed in the mathematical model.

Noncontiguous node and element numbers: The element and node number labels must be contiguous to aid in checking. However, it is often inconvenient to renumber the entire mesh when deleting a portion of a substructure. The solution to this problem (providing an external form, and internal form, and a translation) is the same as the solution for the alphanumeric node and element label problem described above.

Nonrectangular constraints: Constraints must be entered in a rectangular coordinate system. This is often inconvenient for curved structures. The model input processor could be used to translate the data from a nonrectangular system to the common internal form, as is currently done for coordinates. A more advanced approach would be to provide coordinate systems as a basic part of the input language. Any type of coordinate translation would then be performed by the language support software.

Default element loads: There are no default forms for element loads. The element load input lists are often long, with only minor variations between items. The change would require extensions to the input language definition and the associated language translation facilities in the model input processor.

Nodal coordinate system: All primary results are computed in the local structural coordinate system. Nodal coordinate systems would be convenient in many cases; shells are a typical example. The model input and output processors would be affected, and an additional model data structure would be required to support this extension.

Nodal temperature gradients: Temperature gradients may only be applied at the element level, via element loads. This often requires large amounts of duplicate element load data when a gradient field is distributed over several elements. Changes to the model data structures, the model input processor, and the computational kernel would be required.

Nodal materials: For certain types of directional materials, the ability to specify material data based on nodes is more convenient than associating the data with the elements. Changes to the data structures, the model input processor, and computational kernel would be required.

Improved lists: The ability of describe data through lists such as "1-35 BY 3" is useful, but the capability could be extended, "ALL BUT 10-17 for example. Such a change would be limited to the modification of the support language translation routine.

Units: All input and output quantities must be given in a fixed set of units, and the system assumes the user is consistent. This is bothersome and error prone. The change would permit units to be associated with any data item. Such a change would require that parameters have units, and that all of the model input and output processors handle units. This change is potentially complex due to the nature of degree of freedom assignments. Any element may use a certain degree of freedom to represent a quantity with certain units. The system needs to be able to combine this element with any other element which many use the same degree of freedom to represent a quantity with different units.

Parameter input: Parametric models are often useful in research. Parameters could be provided by extensions to the language support software. This software would provide the capabilities needed to translate the parametric model input into a conventional problem description.

Expression input: Simple quantities are often computed from complex expressions before being input. Direct expression input is more useful and less error prone. Expressions could be provided at the language level, by the language support software. The input language definition would be modified so that an expression would be acceptable wherever a number is now required.

Natural language: Natural language input is the most flexible input form. Language processing is independent of any underlying support system. Thus, the change could be isolated to the language support software. However, natural language input translation is very complex.

Renumbering: Requiring the user to properly number the mesh is often a complex and error prone user task. Mesh renumbering algorithms are useful in providing economical solutions by reducing the bandwidth of the equations. Implementing renumbering algorithms requires support data structure to provide the translation between the user numbering and the internal numbering, the renumbering process, and a numbering translation process in the model output processor.

Top down structural models: Structural models are defined from the bottom up; the lowest level substructures must be defined first, and higher level structures are defined based on these lower level substructures. Top down models, recursively subdividing the structure, are sometimes more natural. This change is compatible with the current modeling process. An alternative model input processor would be required.

Material models at strain points: Each nonlinear element has one common material model for all strain points. This is restrictive for some types of problems. A change to permit different material models at each strain point would be basic and affect much of the system. Data structures for both the computational and mathematical models would be changed, and all processes which deal with stresses, strains, or materials would be affected.

Material models for all elements: Material models are currently used only for nonlinear elements. Linear elements take material data from element parameters. To make the process of using material models consistent would also be a basic change, similar to the one described above.

Nonvector degrees of freedom: The system supports ten groups of degrees of freedom at each node. Each group consists of three components which transform as a vector quantity. Certain derivative quantities, such as twist, transform as tensors, and elements using such degrees of freedom do not function properly when used out-of-plane. This change would be basic and impact the library and the computational kernel.

Multiple sets of constraints: Any change to a constraint invalidates the entire constraint set and all computed results. It is often valuable to combine results from different constraint conditions. The ability to maintain multiple constraints sets and solutions for each set would be better than the current "FOURIER" approach. The effects of this change would be wide spread, affecting both the computational model and computational kernel.

Linear analysis as a subset of nonlinear: Internally, linear analysis and nonlinear analysis are handled as separate cases, although from the user's viewpoint there are no major differences between such analyses. For the system to treat linear analysis as a subset of nonlinear analysis would require changes to the computational kernel.

Global versus local stiffness: All stiffnesses are computed in the global coordinate system. For certain types of elements and problems this is not convenient. This change would require a modification of the computational kernel.

Dynamics: Dynamics is a major extension, and it impacts all aspects of the system.

Tables used with any element: The use of tables to provide element parameters is restrictive, in that the table must be compatible with the element. This compatibility is the responsibility of the table and element implementors. A dynamic linkage mechanism is needed to allow any table to be used with any element. Additionally, the use of units has possible side effects, since the default table units may not be the same as the current problem units. This change would affect only the library and the model input processors.

Multi-valued parameters: All element parameters are scalars. For materials the parameters may be multi-valued. Elements are required to use many single parameters for items such as nodal thickness, which requires a vector with a value for each node. Providing multi-valued parameters would be a minor language change and would affect the library and the model input processor.

Bounds on properties: Element parameters must be bounds-checked by the element modules (i.e., $E > 0.0$, $0.0 \leq NU \leq 0.5$, etc.). Associating bounds with the parameters in the library would permit the system to uniformly enforce bounds-checking. This change would require modifications to the library and the model input processor.

Number of element parameters: The maximum number of parameters and results for each element is fixed. The current value of this limit has been found to be restrictive. Changes to eliminate

this restriction would be required in the library and the computational model creation and processing components.

Change element parameters at run-time: Once defined, any change to an element parameter causes the system to invalidate all results for the substructure in which it appears, and for all higher level substructures. For some cases, this is not appropriate. For example, it is not possible to change a parameter which would affect output computation without requiring all data to be recomputed. This capability could be provided by designating the action which is to occur when a parameter is changed. The change would affect the library and the model input processor.

Element parameters grouped by type: Element parameters are untyped. Providing types (i.e., as control, geometric, display, etc.) may be more convenient for processing and modifications of parameters as described above. It would require changes to the library, the model input processor, and all element modules.

Element geometry: The complete geometry of an element is scattered throughout the coordinates and parameters. It is not known until stiffness generation time, and is often recomputed by every element module. A single element process to create and store the element geometry may be appropriate. Placing this process in the model checking process would also provide better model diagnostics. This change would require modifications to the library and model data structures, as well as to the model input processor, the computational kernel, and the element modules. This would be a new feature with wide spread effects on the entire system.

System control by elements: The various element modules are automatically and unconditionally invoked. In some cases (such as stress computations for nonlinear elements) the element module's function may be performed directly by the system without the need of invoking the element module. The element could provide information which would direct the system's operations. This would require changes to the library and the computational kernel.

Improved data generators: There are a number of problems with the data generators; polar generation does not function properly at all times, and triangular elements can not be generated. Improvements would be isolated to the generators.

Ability to generate any type of data: Generation of incidences and coordinates may eliminate a large portion of the input data. Constraints and loads often exhibit patterns which could be generated in a similar manner. Extending the generator to compute spatial variations of any component would require an enhanced generator, and its incorporation into the model input processor.

User data generators: For complex problems, such as shell intersections, a specific problem oriented user data generator would often be useful. There is a need to interface such a generator to the system without the generator producing POL input to be processed as normal input. This would require the ability to support user written modeling processors in the model input processor. Such a generator would consist of data generation routines and a description of the POL input language used by the generator. The generator input would be translated by the system, and the generator would be invoked to directly build the mathematical model.

Data extraction: Data extraction is the opposite of user data generators. It is used to provide specific data for special post-processors. It would require the modification of the model output processor to support user written output processors. The data extractor would consist of a set of routines which would access the model results and produce output, and a POL description of the language statements used to drive the data extractor. The system would provide language translation and invoke the data extractor.

Solution status: The system does not record the status (i.e., triangulated, solved, etc.) of a solution for any problem. It must be externally recorded by the user. The change would require the system maintain the status information, and permit user inquiry. Modifications of the computational database and computational kernel are required to support this change.

Solution log: A log would record all steps in the solution process. This would permit inquiry to determine the actions taken to reach the current problem state. Logging would require the addition of the log data structure and a logger in the computational kernel.

Improved error recovery: The system was designed for the batch environment. Errors usually cause eventual abnormal system termination. The error action should depend on the operating environment, permitting the user to regain control if possible and take corrective action. This change would influence all processors.

No fatal errors: Many errors terminate the system in a manner such that problem restart is not possible. No fatal errors should exist, unless caused by a fault in the underlying system. This change would influence all processors.

Model output: There is currently no mechanism to output the current structural model. Capabilities are needed to output any portion of the model. Such a process would require major additions to the model output processor, and possibly the addition of element output modules. This would be a valuable user feature.

Computational results output: Only the unassembled stiffness matrix may be output from the computational model. Maintenance and element testing would be improved by the ability to output any component of the computational model. This would require the creation of the computational results output processor.

Nonnumeric output quantities: All element stress and strain output quantities are restricted to be real numbers. State quantities (i.e., element loading, element unloading, or strain point yielded) are best expressed as nonnumeric values. This change would require modifications to the library and the model output processor.

Material output: Material models have no formal mechanism for providing output. Currently they may augment element output, but this requires explicit element and material compatibility. Formal material model output is needed, and would require modifications to the library, the computational kernel, and model output processor.

Combined compute and output requests: The compute and output requests must be given separately for each step, if output is to be obtained as computed. This often leads to long sequences of requests, especially for nonlinear problems. More complex request forms would be useful and eliminate user input.

Stress averaging: Stress averaging is a complex problem when different types of elements (with different types of stress resultants) are combined. Stress averaging would be performed by the model output processor, and the library must contain data describing the types of stress quantities computed by each element.

Stress interpolation: Stress interpolations are needed for display and averaging. This would be performed by the model output processor, and the library must contain data describing the interpolation functions.

Maximum and minimum stresses: Stress limits could be computed by the model output processor.

Subsets and ordered results: Only a limited capability exists for providing a subset of the results computed by an element. Complete control over the number and order of all output quantities is more useful. This change would affect the model output processor and the element output modules.

Model graphics: An integrated model display capability is lacking. This facility would permit any portion of the model to be displayed. This would be a major addition. The additions include library descriptions of how to process an element, element display modules, and the display components of the model output processor.

Result graphics: Integrated results display is also lacking and would be a major addition. This provides the capability of displaying any computed results. It would require similar extensions to the library and the model output processor as described above.

Graphics transformations: Arbitrary graphics viewing capabilities (i.e., clipping, perspective, rotations, hidden surface elimination, etc.) are essential to provide useful displays. These features would be provided by the model output processor and the graphics support system.

Function plotting: In addition to display of the results in terms of the model, direct plotting of graphs, such as load versus displacement are useful. This feature would require a function plotting capability in the model output processor and in the graphics support system.

Digitizer input: Translation of structural models from drawings may be best accomplished through digitizer input. This extension would be isolated to the model input processor.

APPENDIX B. DATABASE MANAGEMENT

A database is a collection of data items, used in an organization's data processing applications. This collection is stored on some type of secondary storage medium, typically disk. The database exists independently of all the processing applications which use it. The contents of the database are created, used, and maintained by the various applications. As such, the database integrates the applications. The database management system is the collection of software which lies at the interface between the physical device access procedures and the applications. It supports all of the operations on the data and all accesses to the database.

B.1 The Evolution of Database Management Systems

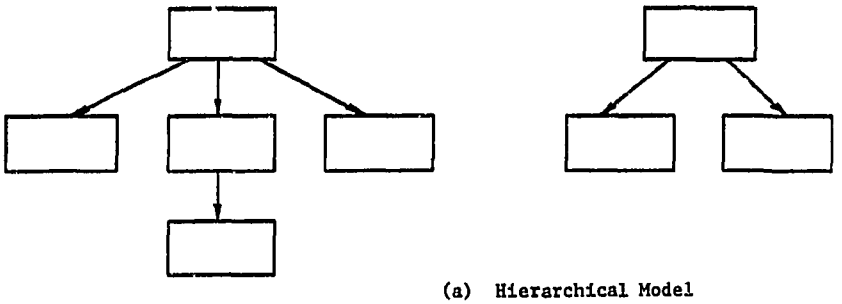
Database management has resulted from attempts by commercial computer users to improve their data processing capabilities. Originally, programs input data, processed it and created files of information (usually on tape) to be used in subsequent processing. The file creation program "decided" what data would be kept, and how it would be stored. The file creation program was, in effect, the "owner" of the data, and was responsible for making all decisions regarding data storage and retention. The data produced by one program was soon needed by other applications. The needed data was quite often difficult to obtain (missing data items, wrong format, format unknown, data order made it difficult to process, etc.). This led to attempts to integrate the data from all the applications. Databases and database management systems (DBMS) are now used to store and maintain this integrated, centralized form of the data.

There are a number of advantages to the centralized approach: (1) reducing redundancy: duplicate or similar data stored in different files for the use of different applications can be combined and stored only once; (2) improved availability: data can be shared and made available to any application independent of other applications; (3) reducing inconsistency: redundant data can be different, once the data is combined these inconsistencies can not occur; (4) enforcing standards: data can be represented in a standardized form which simplifies use and maintenance for all applications; (5) enforcing data security: authorization and data access

come from a single point, and this part of the system enforces all access procedures; and (6) balancing conflicting requirements: each data user has his own best form for the data, and a centralized system permits a form which is most appropriate for all users to be selected. The problems with the database approach are related to the same aspects which are its advantages. It is not a simple task to create such a system, to insure security and integrity, and to determine what forms are most appropriate for all data users.

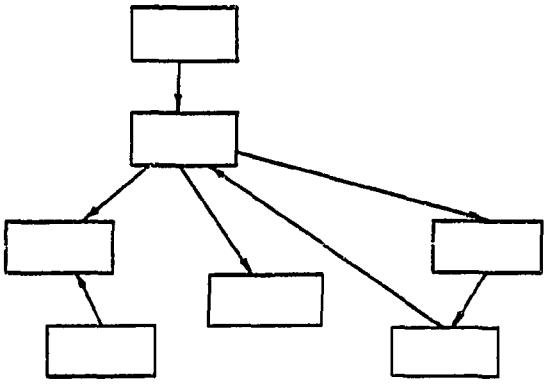
Through many years of development, the file based systems led to database systems [FryJ76]. The various systems which evolved all have two basic components: (1) a data model used to define the organization and structure of the data items in the database, and (2) a data language used by the application to access the data. There are a variety of forms which are possible for these components. However, the data language is highly dependent upon the type of data model. The three data models which have evolved are: (1) hierarchical [TricD76], (2) network [TaylR76], and (3) relational [ChamD76, MichA76, KimW79].

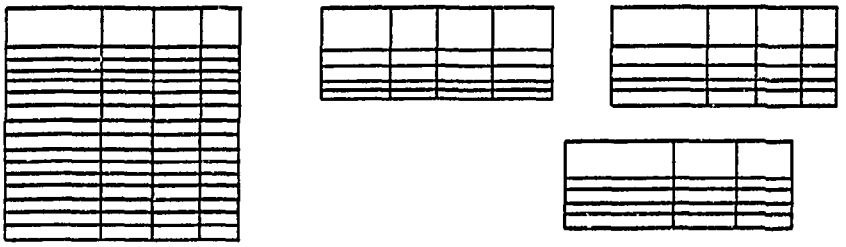
Hierarchical: The hierarchical model is based on the organization of data into a hierarchical or tree structured form. This data model is often chosen because tree structures are natural in many applications and organizations. Data frequently occurs in a 1:N relationship. There are many (N) occurances of a data item which are all subordinate to a single item. A typical example might be a company which is divided into many departments with a group of employees associated with each department. The employees are below the departments in the hierarchy, and similarly, the departments are below the company. An example of a hierarchical model is shown in figure B.1.a. To access the data, it is necessary to know the physical organization of the database. The data is "organizationally addressed." Traversal of the tree structures is required to obtain any data item. The hierarchical organization is useful and simple, especially if the applications are naturally hierarchical. However, if the data structure or the access paths do not conform to the hierarchical form, severe problems can result due to the complex processes needed to access the data.



(a) Hierarchical Model

(b) Network Model





(c) Relational Model

Figure B.1. Data Models

Network: The network model is also known as CODASYL or DBTG (Data Base Task Group). The 1:N relationship used in the hierarchical model can not be used to represent all forms of data. The network approach relaxes the strict tree form, and allows data items to be interrelated as are the nodes in a network. Data relationships are N:M (M items are subordinate to N items). An example of a network model is shown in figure B.1.b. Again, the data is accessed by traversing the various components of the data structure.

Relational: The relational model is based on the mathematical theory of relations. The data is stored in sets of relations, typically represented as tables of tuples. A typical relation is shown in figure B.1.c. The major difference with the other models is that all data in the relational model exists in a single form, and all of the data accesses are made by logical content rather than by the physical data organization. In the relational model, the data is "content addressed." Additional information on the relational model is presented below.

B.2 Database Management Systems Structure

A database management system consists of a number of different components, and provides a number of distinct features. The major components of a typical system are:

Data Definition: The data definition facilities are used to describe the data items and data structures used in any database. There typically are two components. The first is a data definition language (DDL). This language is used to describe the logical organization of the database: its components, their organization, and their relationships. This description is sometimes denoted the "schema." There exists one global schema for the entire database. Individual users may need only portions of the database, and these subsets of the database are described in a similar manner and are denoted "subschema" or "views." The data definition does not specify how the data is physically stored (the size of fields, record formats, etc.). A data mapping language (DML), the second component of the data definition facilities, is used to describe this physical data representation.

Access: The access facilities are used to enter, query, update, and manipulate the data contained in the database. There are often two sets of access facilities. One is provided for programmers to use in application programs. The other is for end-users, to support simple generalized queries, without the need to develop special application programs.

File Structure: The file structures are used to store the contents of the database.

Data Dictionary: This is a component of the database which is used to maintain the descriptions of the items in the database. Thus, the database is self-documenting.

Integrity Control: Various constraints on items in the database must be maintained (i.e., SALARY > 0). The integrity software is used to verify all data manipulation requests to insure they do not violate any constraints.

Concurrency Control: Databases are used in a multi-user environment, and this component of the database management system insures that all data updates are synchronized and that deadlocks do not occur.

Access Control: This feature provides the authorization of a user's privileges for data query and modification.

Recovery: Failures, due to software or hardware, can invalidate portions of the database. Recovery features are used to maintain sufficient information so that the database can automatically be rebuilt after a failure.

Report Generator: This software is used to produce generalized, tabular output from the database, without the need to program special applications.

B.3 The Relational Approach

A relation is defined as [DateC75]:

> Given sets D_1, D_2, ..., D_n (not necessarily distinct), R is a relation on these n sets if it is a set of ordered n-tuples $< d_1, d_2, \ldots, d_n >$ such that d_1 belongs to D_1, d_2 belongs to D_2, ..., d_n belongs to D_n. Sets D_1, D_2, ..., D_n are called the <u>domains</u> of R. The value n is the <u>degree</u> of R.

The tabular representation of relations is the most common form utilized. It has the following properties:

(1) no two rows (tuples) are identical;

(2) the ordering of the rows (tuples) is insignificant; and

(3) the ordering of the columns is significant unless the columns are referred to by their domain name, rather than by position.

The rules stated above are all that is known about data organization in the relational model. Data accesses are made by specifying which values of which domains of a given relation are desired. From the base relation, a new relation is formed which contains only the requested data. The entire set of data is then returned to the application as a set of tuples, or the tuples are returned individually. The database management system is responsible for determining how the data is actually stored (its physical organization).

The terminology used in the relational model can be compared to that of the more conventional file structures. A relation corresponds to a file. A tuple corresponds to one record in a file (all records have the same format). A domain is equivalent to a given field within the records. A set corresponds to all possible values of a field (domain).

As an example of a relation and its use, consider the structural steel properties from the AISC manual [AISC70]. A portion of a relation W_SHAPES_PROPERTIES might be as shown below. In this example the domains are the designation of the shape, its weight, area, principal moment of inertia (IXX), etc.

W_SHAPES_PROPERTIES				
DESIGNATION	WEIGHT	AREA	IXX	IYY
W14x136	136	40.0	1590	568
W14x127	127	37.3	1480	528
W14x119	119	35.0	1370	492
W14x111	111	32.7	1270	455
W14x103	103	30.3	1170	420
W14x95	95	27.9	1060	384
W14x87	87	25.6	967	350

A typical query on this relation may take the following form (syntax based on System R [AstrM76]). The query will find the WEIGHT and DESIGNATION of all members with IXX greater than 1000 and AREA greater than 35.

```
SELECT  WEIGHT, DESIGNATION
  FROM    W_SHAPES_PROPERTIES
    WHERE   IXX > 1000
     AND      AREA > 35
```

The result would be a relation (unnamed) with two domains (weight and designation) and two tuples (those which satisfy the conditions). The resulting relation is shown below:

WEIGHT	DESIGNATION
136	W14x136
127	W14x127

As can be seen from the above example, the relational approach is conceptually quite simple. A single common form is used for all data items, and the physical data organization is never utilized.

APPENDIX C. ARTIFICIAL INTELLIGENCE

"Can machines think?" The question was posed in 1950 by Alan Turing [TuriA50], but the controversy over the potential of machine intelligence has existed for 150 years. Lady Ada Lovelace (Lord Byron's daughter), the "programmer" of Charles Babbage's Analytical Engine wrote, "The Analytical Engine has no pretensions to originate anything. It can do whatever we know how to order it to perform" (her italics) [LoveL42]. The study of this intriguing problem has evolved into the discipline of artificial intelligence (AI). A simplistic definition of AI is: creating a nonhuman system capable of intelligent thought. The entire human thought process (cognition, knowledge representation, learning, reasoning, perception and communication) is so complex and ill-defined that the characterization of what constitutes an intelligent human process is quite impossible. As a result, no attempt will be made to give a formal definition of AI.

Artificial intelligence deals with the computer implementation of those tasks which require (or are currently limited to) human problem solvers. Some typical problem domains which are considered to typify human intelligent processes (and are subjects of AI research) include: language translation, game playing (bridge, poker, chess), theorem proving, symbolic manipulation, natural language understanding and discourse, speech understanding, and expert problem solving. Of course, once a machine is able to solve any of these problems with the efficiency and skill of a human, there is the feeling that the problem does not require real intelligence.

C.1 Artificial Intelligence Concepts and Research

Artificial intelligence is a new field. No formal methodology exists for converting an intelligent problem solving task into a program. Rather the field consists of the status of the solution to a number of problem domains, and a number of concepts upon which the solutions are based.

C.1.1 Concepts

A common set of basic concepts and ideas are present in the programs which implement solutions to the intelligent problem solving tasks. These basic concepts are search, control, and representation.

Search: This is the most basic tool used in all AI systems. Solutions to the types of problems for which AI is used are nondeterministic (for a deterministic problem an explicit solution could be developed). Search provides a systematic method of exploring (searching) a variety of alternatives in a solution space.

Control: An AI system often consists of a number of individual processes, each with a limited behavior. Control determines how the various procedures are selected, and how they interact with the information in the problem space. Control and search are interrelated. Control selects the problem solving mechanism; search orders the evaluation and invocation of the control processes while traversing the solution space.

Representation: Knowledge and data must be translated into some internal representation to be used in processing and problem solving. Additionally, all of the concepts and processes used to solve the problem must be converted into some symbolic form which can be processed by the machine.

A variety of control, search, and representation techniques exist. Artificial intelligence research involves finding the appropriate combinations of these basic concepts which yield effective problem solvers for particular problem solving domains.

C.1.2 Problem Solving Domains

No complete solutions to any of the various problem solving domains exists. However, considerable progress has been made, and in a variety of areas the computer shows respectable behavior [WinsP77].

Chess: Due to the complexity of the problem, chess is the subject of much work. Solutions typically involve search with heuristics to reduce the search space. Recent systems rely on special computer hardware to improve performance [LevyD80]. Game playing quality improves with the depth of the search, but the problem solving time required grows exponentially with search depth. It does not appear that computers and humans play chess in the same manner. Humans appear to use abstract pattern recognition, viewing the board as a whole, while the machine treats each piece individually. The best machines can

now approach the level of play of masters. They have defeated masters in individual games, but never in a complete tournament.

Natural Language Processing: The best example of natural language understanding and discourse is the work of Winograd [WinoT71]. He presents a detailed example of discourse with his robot (called SHRDLU) conversing about a specific domain — the blocks world. The system shows the complexity of dealing with such ill-defined problems as language. Additionally, it introduces a number of concepts (such as the procedural representation of knowledge), and provides the basis for later work in many areas.

Symbolic Manipulation: Part of Project MAC was the development of the MATHLIB system. Part of MATHLIB is MACSYMA (MAC Symbolic Manipulation System) [BogeR75]. One of the most interesting components of MACSYMA is its symbolic integration capability, which is regarded as superior to all human problems solvers for this task.

Medical Diagnoses: MYCIN is an "expert system" used to help physicians diagnose and treat bacterial infections [ShorE76]. It is based on rules provided by experts, and has a special subsystem which allows modifications of the expert knowledge. MYCIN operates in the domain of uncertainty. All the data presented to the program may have a margin of error, and all of the knowledge rules are based on operations on uncertain data. The system shows good performance, approaching the level of a human specialist.

Mass Spectrogram Analysis: DENDRAL is one of the first true expert systems [BuchB69]. It is used to analyze organic chemistry mass spectrograms. Given the spectrogram and the chemical formula, the system will deduce the structural arrangement. DENDRAL is capable of operating at the level of an expert graduate student.

Speech Understanding: The most successful system to date has been HARPY [ErmaL80]. In a specific task domain, it can understand a vocabulary of 1000 words, with an error rate of 5%, in real-time. HARPY provided the basis for Hearsay-II, an advanced

knowledge based system for speech understanding. The problem solving model used in Hearsay-II is applicable to other problem domains.

C.2 Production Systems

The production system [NeweA72] represents one of the techniques used in AI problem solving, and is the basis for the types of expert systems discussed in the text. Production systems are deductive problem solvers. Such a system consists of four basic items: (1) the description of an initial problem state which contains a number of entities and facts about the problem, (2) a goal state, (3) a set of productions, and (4) a controller. Each production is a rule, consisting of a predicate and an action. The predicate states: if some condition about an entity is known to be a fact, then the corresponding action is to be performed which will modify the problem state. The controller is responsible for determining which productions are to be applied, and the order of application of the productions.

One possible operational procedure is as follows. The controller loops, selecting and applying productions until no productions are applicable, or until the goal state has been reached. Production selection strategies include: (1) apply the first production applicable, (2) find all productions applicable, and select one based on a predefined priority, or (3) find all applicable productions, and apply the most recently used. If the goal state is reached, the system has deduced the goal by transforming the knowledge, and adding new knowledge from the operation of the productions. If the system runs out of productions, it has deduced all possible knowledge, and the goal is unreachable. Either the goal is incompatible with the knowledge, or insufficient productions exist. The process of working from known-to-new facts is called forward chaining.

The alternative of backward chaining is sometimes more appropriate. In such a system, the goal is hypothesized to be true. Productions which produce the goal are found, and the new goal becomes all of the knowledge needed to make the predicates of these productions true. The process is applied recursively until no other productions are found. If the unresolved goals are consistent with the knowledge in current problem state, the hypothesis is true. Otherwise the hypothesis fails; either it is incorrect, or insufficient productions exist.

In the recursive process, a number of search strategies are possible. The two simplest are: (1) depth first — select one alternative production, generate one new goal, and move forward. When blocked, move back and try another goal at the last decision point. (2) breadth first — generate the goals for all productions at each level and move forward in parallel, one level at a time. Again, when blocked, backtrack. Both procedures have advantages and drawbacks, based on the nature of the search space. Other procedures, such as best first, hill climbing, or branch and bound, all attempt to minimize the total work done in searching, but no procedure is optimal in all cases. The concept of backup is one of the the most important components of controlling any search. Backup permits the system to recover from a failure state, and to examine other alternatives.

Production systems may operate as control systems. They can continually monitor the problem state and perform actions based on state changes, to control some object. Execution continues until a production explicitly terminates the system. Similarly, systems may be connected to an external information source from which they may request information when knowledge is lacking or can not be derived.

The following is an example of a simple production system for a thermostat [NeweA72]. Control starts with the first line of the list of productions and continues until a "true" predicate is found. Then the corresponding action is performed and execution resumes with the first production.

THERMOSTAT

TEMPERATURE > 70° AND TEMPERATURE < 72° ⟹
STOP.

TEMPERATURE < 32° ⟹
CALL-REPAIR-MAN; TURN-ON [ELECTRIC-HEATER].

TEMPERATURE < 70° AND FURNANCE-STATE = OFF ⟹
TURN-ON [FURNANCE].

TEMPERATURE > 72° AND FURNANCE-STATE = ON ⟹
TURN-OFF [FURNANCE].

Production systems are valuable because the problem solving knowledge is modular. It is possible to change or augment the knowledge in the

productions, and thereby change the behavior of the problem solver, since the knowledge is simply data to the controller. Since the controller is knowledge independent, the interactions of the various productions need not be specified. This eliminates the combinatorial increase in the number of interaction of items, and it also allows the controller to generate all possible interactions, some of which may have been overlooked if they were explicitly programmed.

All of this flexibility does lead to a major drawback. Such systems are known as "weak" problem solvers. They operate in a blind fashion. They may overlook obvious solution paths and produce circuitous ones, or they may require much knowledge and do much problem solving which is not pertinent to obtaining the goal. As the number of productions increases, the resulting interactions may not be readily predicted, and control is effectively lost.

C.3 Knowledge Based Systems

Search is the basis of many of the problem solving methods; formulate a set of alternative solutions and search that solution space. Increasing problem complexity leads to larger search spaces. An effective problem solver must search efficiently. To do so, it must determine the solution by examining as small a portion of the solution space as feasible. A weak solver has no guidelines to assist in searching. Knowledge helps: knowledge about the problem domain, or knowledge about effective problem solving strategies in the problem domain. This knowledge is the expertise of problem solving. Expert or knowledge based systems have been developed to use such information in providing effective problem solvers. Such systems are known as "strong" solvers. MYCIN, DENDRAL, and Hearsay-II are all examples of knowledge based systems.

Knowledge may be used in a number of ways. One method is to use "meta rules" [DaviR77]. Meta rules are used to describe which rules are appropriate in a given situation. Thus, the search becomes a two level process. At the lowest level a solution is found. At the higher level a similar problem solving strategy is used to determine the process for the selection of the actual rules used to solve the real problem. Of course, such a system may be extended to many levels; meta meta rules describe which meta rules are applicable and determine how to select the meta rules used to select the problem solving strategy, etc. The ability of the system to direct the problem solving strategy is one difference between the weak, general solvers

and the knowledge based systems. Thus, knowledge serves a key role in selecting the knowledge sources (rules).

Another use of knowledge is in the description of the problem domain and the problems solving rules. Knowledge based systems are types of production systems. Weak production systems are based on simple axiomatic rules. They consist of a large number of simple rules, and problem solving involves deduction through simple transformations. The large number of rules and lack of direction contribute to ineffective solvers. In the knowledge based systems, the rules are more complex. In MYCIN, for example, rules consist of large predicates each with several premises each involving a number of parameters. Actions may affect multiple parameters, and a parameter description may require several lines of definition. In Hearsay-II, rules are denoted knowledge sources, and these knowledge sources are encoded as procedures. Such knowledge ranges from a hundred to several thousand lines of algorithmic language code. Encoding problem domain knowledge in higher level units provides more efficient solvers, since the number of rules and the number of interactions are reduced. This reduces the search space.

Knowledge also helps to control uncertainty. Complex problems often do not have an exact solution, or the data present is incomplete or uncertain. Knowledge of the problem domain, combined with complex rules based on knowledge uncertainty allows the knowledge based systems to operate in the domain of inexact problem solving.

The following is an example of an expert rule taken from MYCIN [ShorE76].

RULE200

IF: 1) THE SITE OF THE CULTURE IS BLOOD, AND
IF: 2) THE STRAIN OF THE ORGANISM IS GRAMNEG, AND
IF: 3) THE MORPHOLOGY OF THE ORGANISM IS ROD, AND
IF: 4) THE AEROBICITY OF THE ORGANISM IS ANAEROBIC, AND
IF: 5) THE PORTAL OF ENTRY OF THE ORGANISM IS GI
THEN: THERE IS STRONGLY SUGGESTIVE EVIDENCE (.9) THAT THE IDENTITY OF THE ORGANISM IS BACTEROIDES

The rule shown above deals with a number of parameters such as SITE, STRAIN, AEROBICITY, etc. The description of a simple MYCIN parameter is given below [ShorE76].

YES-NO PARAMETER

```
      FEBRILE: <FEBRILE is an attribute of a patient and
               therefore a member of the list PROP-PT>
       EXPECT: (YN)
   LOOKAHEAD: (RULE149  RULE109  RULE045)
       PROMPT: (Is * febrile?)
        TRANS: (* IS FEBRILE)
```

Expert systems use knowledge to assist in problem solving. Rather than attempting to be general systems capable of solving any type of problem, they use the same basic AI concepts to attack specific problems which require knowledge to represent complex problem solving strategies. The systems do retain the advantage of the original production systems by maintaining knowledge independently of the problem solver. The knowledge based systems present a problem solving paradigm which may be applied to other problem domains by changing the knowledge sources.

Problems still exist. Complex knowledge sources perform complex tasks with limited interaction with the rest of the system, due to the reduced number of components which can interact. The result is a limit to the interaction of the knowledge, and a resulting limit on system performance, since potentially useful interactions do not occur. Also, as the amount of knowledge increases, the problem of determining the appropriate knowledge becomes more important and more costly. Once acceptable processes exist, algorithmic encoding can improve efficiency and effectiveness. Despite their drawbacks, knowledge based systems appear to be the best technique currently available for performing complex ill-structured problem solving tasks.

GLOSSARY

There are a variety of terms and phrases used throughout the text with which the reader may not be familiar. The following contains a short definition of these terms. For the readers convenience, the terms are grouped by subject.

1. General

The following are general computer science terms which are used throughout the text.

Applications: Computer software which is applied to, or used for, some particular task.

Artificial Intelligence: A discipline of computer science dealing with the development of computer based systems for intelligent problem solving behavior (see appendix C).

Back-End Database Management Machine: A dedicated computer performing all database management functions. A back-end database machine is logically located between the main processor (which requests all database processing) and the secondary storage system.

Configuring: The process of selecting the components, and the arrangement of these components into a system.

Data Abstraction: The process of defining new data types (abstract types) based on a set of existing data types.

Database: A logical collection of data maintained in a single organizational unit on some secondary storage devices (see appendix B).

Database Administrator: The individual who is responsible for supervising a database management system.

Database Manager: A database management system. The run-time data handling component of a database management system (see appendix B).

Database Management: The process of managing data through the use of a database and database management system (see appendix B).

Database Management System: The set of computer software used to control and support a database (see appendix B).

Data Model: The type of basic logical organizational structure of items in a database.

Data Types: The generic data quantities which have a particular representation and behavior (e.g., REAL and DOUBLE PRECISION in FORTRAN are both floating point types).

Expert Systems: A type of artificial intelligence system which uses expert knowledge to control and direct problem solving in a knowledge based system (see appendix C.3).

Hierarchical Model: A data model used in database management systems which is based on a hierarchical data organization (see appendix B.1).

Information Flow: The process through which data and information moves between the various individuals and processes that create, use, and manipulate the information.

Kernel: The basic core of software and operational capabilities in a system.

Knowledge Based System: Any type of artificial intelligence system which uses domain specific knowledge to control and direct problem solving behavior (see appendix C.3).

Knowledge Source: A single logical unit of expert knowledge used in a knowledge based system (see appendix C.3).

Language Extensibility: Computer language facilities which allow the language definition to be extended (see section 4 below).

Network Model: A data model used in database management systems which is based on a network data organization.

Operators: The basic primitive functions and operations implemented directly by the hardware of a computer system (i.e., add, multiply, load, store, read, write, etc.).

Natural Language: The normal (unrestricted) form of spoken and written language.

Packages: A complete set of computer code and associated data structures (organized into a single logical unit) design to perform some particular function.

Production: A premise-action rule of a production system (see appendix C.2).

Production System: A type of artificial intelligence system based on representing problem solving behavior in the form of productions (see appendix C.2).

Problem Oriented Language: POL. An artificial computer language subset of natural language (with restricted syntax and vocabulary) used for some particular problem area.

Relational Model: A data model used in database management systems which is based on a relation form of data organization (see appendix B.1 and B.3).

Schema: The definition of the logical structure and content of a database (see appendix C.2).

Software Engineering: The process of "engineering" a piece of software. A discipline of computer science dealing with the application of engineering principles to the development of software.

Software Tools: General purpose utility programs used to assist in developing software. Utility components of a complete system.

Strong Solver: Any type of artificial intelligence system which uses domain specific knowledge in problem solving (see appendix C.2).

Token Scanner: A program which converts (parses) a stream of input characters into a set of basic symbols (i.e., numbers, names, punctuation, etc.).

Tuning: The process of adjusting software to improve its performance.

Virtual Back-End Database Machine: A virtual computer implementation of a back-end database machine.

View: The description of the organization of a subset of the data in a database.

Virtual Machine: The implementation of a complete computer execution environment in software. The software which implements the functions of a real piece of hardware.

Weak Solver: Any type of artificial intelligence system which operates without the use of domain specific knowledge to assist in problem solving (see appendix C.3).

Writable Control Store: Memory which can be loaded under program control, and which contains the microcode definitions of operators which can be executed by the processor.

2. Computer Aided Design Applications

The following are all types of applications of computers to engineering and design. Computer aided design applications are discussed in section 1.2.

Computer Aided Design (CAD): The acronym CAD usually denotes this application area. A definition of CAD is: the use of computers anywhere in the design process. As such, any of the following applications fall within the scope of CAD.

Computer Aided Drafting: This application is sometimes denoted CAD. It is the application of computer graphics to the production of drawings through assisting draftsmen.

Computer Aided Manufacturing (CAM): Computer aided manufacturing is the combination of geometric modeling and numerical control. It permits a description of an object to be created within the computer and automatically converted into manufacturing instructions.

Computer Graphics (CG): Computer graphics usually refers to the software tools and techniques for graphics, and to the development of innovative graphics applications. Any software which uses the computer to produce graphical output applied to engineering or design falls within this application area.

Design Automation (DA): Design automation is used to denote the application of design and analysis software to the layout, routing, and mask artwork of printed circuit boards and integrated circuits.

Geometric Modeling: The geometric modeling application deals with the development of mathematical models for the geometry of physical objects. It usually consists of procedures to create, manipulate, perform processing on (such as volume computations), and display the descriptions of objects [BaerA79, RequA80].

Numerical Control (NC): Numerical control is the application of computers to provide control mechanisms for automated milling machines. NC is one of the oldest CAD application areas. The major application program is APT [RossD59. RossD78], APT provides a mechanism to convert user commands describing the part to be machined into the control tapes used to operate a NC milling machine.

3. Programming Languages

The following are some of the major computer languages which might be used to support engineering applications, or which have unique features which may be of value in the computerization of engineering problems. Programming languages are discussed in section 3.5.2 and 4.5.

Ada: The Department of Defense [DoD] has found that defense contractors use a variety of languages to implement software. To attempt to regain control and provide some standardization, Ada [DoD80] has been designed. Ada is based on Pascal (see below) with many of Pascal's problems removed. It has a number of additional features for use in real-time and multi-tasking problems. Ada is a DoD standard, is being considered as a national standard, and its future is uncertain.

Algol: Algol 60 [NaurP60, PerlA78] is the parent to the Algol family of languages. Although not widely used, its block structuring and control structure concepts are now features of the majority of new algorithmic languages.

Algol 68: Algol 68 is a revision of Algol, and it was designed to overcome a number of difficulties in its predecessor. It introduced a number of concepts, including preludes and operator overloading [TaneA76].

APL: APL [FalkA73, FalkA78] was introduced as a theoretical language [IverK62], and implementations which are different from the original language design have been introduced. APL operates at a higher level than common procedural languages, has a number of unique operators, and has a distinct style. The data components are scalars, vectors, and matrices of arbitrary dimensions. All operators are equally applicable to all data items (A + B can represent the addition of scalars, vectors, or matrices). A complex procedure in a conventional programming language can often be coded in a simple APL statement. However, the lack of common control structures, such as loops, results in programs which solve problems in a manner quite different from traditional languages.

LISP: LISP [McCaJ62, McCaJ78] and its variants are the de facto standard for artificial intelligence programs. The language treats all programs and data items equally, as elements of

linked lists. This linked list form is the basic data item supported by LISP. LISP is an extremely expressive language. Programs which write and execute other programs by producing the linked list representation of the program may easily be developed. However, LISP is usually an interpreted language, and it is very costly in terms of machine resource utilization.

Pascal: Pascal [JensK76] was designed as a student teaching language, and is a successor to Algol. Because of the rational basis for the language design, it has become quite popular for many general applications. It is block structured and contains all of the basic control and data structuring facilities. Due to its original teaching nature, it contains a number of serious deficiencies (particularly in I/O and compilation facilities) which do not make it acceptable for large-scale engineering applications.

PL/I: PL/I [RadiG78] was introduced by IBM as a general purpose replacement for both FORTRAN and COBOL, and to provide a system implementation language. It is block structured and has an extensive set of features and data structuring facilities.

4. Programming Language Features

The following are various features of programming languages. Language features are discussed in section 4.5.

Control Structures: Control structures provide the mechanisms to control the execution flow of a program. The various looping, selection, and iteration constructs simplify the details of programming. Resulting programs look cleaner and resemble the desired processes rather than obscuring the process amid the language statements required to produce the needed flow control.

Data Structures: Data structures permit individual data items, which are logically related, to be grouped into single organizational and processing units developed to meet the representational needs of a program. The data structures can then be dealt with as an aggregate, through formal programming language mechanisms, rather than through ad hoc constructs.

Data Flow Architecture: Data flow architecture is a new approach to both hardware and programming languages. The classical programming languages are control structure driven. The user explicitly states the control paths used to transform the data (Do X to datum Y to produce datum Z). In data flow, the program consists of descriptions of operators and data items, without explicit flow statements. Associated with the data items are the operators which transform the data. The programs are data driven. Whenever the input data for an operator is present (1) the operator is invoked, (2) the data is transformed by the operator, and (3) the process continues (When datum Y becomes present, perform X, yielding datum Z). Such program forms simplify program development. Programs transform data; programming a statement of the transformation which can take place at any step is simpler that explicitly coding all the actions which need be performed and the interrelations and sequencing of control of these actions.

Environments: Language environments are sets of tools, oriented towards a particular language, and used to assist in developing programs in that language [FairR80]. The development of complex systems requires more than a computer and a compiler. Language environments are design to help in such cases. They provide the tools to help maintain, edit, and debug the applications, as well as the ability to integrate applications and support packages. Since these tools are oriented towards a single particular language, they are more beneficial than generic tools because the tools have a built-in knowledge of the problem domain in which they operate.

Extensibility: Language extensibility is the capability of a language to support the definition of extensions to the language without modifying the language compiler. Many languages have a fixed set of features (data types, control statements, data structures, etc.). Extensible languages have a number of particular features which allow the details of the language to be extended, to permit the language to be tuned to an application, and to provide a more abstract set of features. Operator overloading (see below) is an example of language extensibility.

Operator Overloading: Operator overloading is the ability to define or redefine how an operator is applied. In languages such as Algol 68 and Ada, facilities exist to define how the various operators act, based on the type of operand. For examples, the "*" is defined to perform multiplication on INTEGER, REAL, COMPLEX, and DOUBLE PRECISION types in FORTRAN. Newer languages allow procedures to be written determining how such an operator will perform for any type of operand. Thus, "*" could be extended to vectors or matrices. The compiler can determine the appropriate operator action (i.e., if A * B is scalar or vector addition based on A and B), and can even handle the necessary coercion (converting data types such as integer to real) to provide the correct data items. The ability to "overload" the operators permits the data structures to be changed, the definitions of the operators updated, and the program recompiled without dealing with the actual code which uses the operators and which is used to perform the computations.

5. Computer Operating Systems

The following are various type of computer operating systems and system configurations in use today. Computer systems are discussed in section 3.5.3.

Batch: Batch is the classical type of system where all processing is done utilizing bulk input and output systems with no interaction with the user from the time the job is submitted until it is completed.

Distributed: Distributed systems consist of multiple linked machines (usually at different sites). Data is available for sharing among the components of the system, and the actual processing of tasks is also shared (distributed) across the entire system (sometimes automatically).

Networks: Networks are created through the linking of multiple systems to permit the sharing of system resources, and to permit the transfer of data and programs between the machines at the various nodes of the network. In a network, only the data is shared. The processing of tasks is explicitly assigned to a particular machine.

Satellite: Satellite systems are types of distributed systems. They consist of a large general purpose computer at a central site and one or more subordinate satellite processors with lesser capabilities. Data and processing is shared between the central system and the individual satellites. A typical use is to provide a satellite processor to drive a graphics display subsystem, off-loading the graphics tasks which require a dedicated system to obtain acceptable response time.

Time-Sharing: Time-sharing is the classical interactive system where each user accesses the computing resource through a terminal, and each user appears to be using a dedicated system. All processing is done immediately after the user makes a request, and all input and output is directed to the user's terminal.

Transaction Processing: Transaction processing is the use of on-line terminals for simple data entry and inquiry. This is typical of the activities done in banking and airline reservation systems. A simple request (transaction) or piece of data is entered and completely processed by the transaction processing application as a single unit.

VITA

Daniel Robert Rehak is a native of Leechburg, Pennsylvania. He was born on May 11, 1951 in Wilkensburg, Pennsylvania. In 1969 he graduated from Leechburg Union High School, Leechburg, Pennsylvania, and Lenape Area Vocational Technical School, Ford City, Pennsylvania. Attending Carnegie-Mellon University, in Pittsburgh, Pennsylvania, he received a Bachelor of Science Degree in Civil Engineering in May 1973. He served as an undergraduate research assistant, and was awarded fourth place in the Lincoln Arc Welding Foundation Student Design Competition in 1972.

Mr. Rehak began his graduate studies at Carnegie-Mellon University, and received a Master of Science Degree in Civil Engineering in November 1975. He served as a research assistant in the Department of Civil Engineering participating in a project to develop a pilot version of a national engineering software center.

From September 1974 to May 1975, Mr. Rehak attended George Washington University, Washington, D.C. He served as a research assistant in the JIAFS program at NASA Langley Research Center, Hampton, Virginia.

Since 1975, Mr. Rehak has been a graduate student at the University of Illinois, where he has been a research assistant in the Department of Civil Engineering. During this time he also has held minor appointments with the U.S. Army Corps of Engineers, Construction Engineering Research Laboratory (CERL), Champaign, Illinois, and the Coordinated Science Laboratory of the University of Illinois. His research has been in the fields of computer applications to engineering problems, development of engineering computer systems, computer graphics, and finite element systems. From 1976 through 1978 he held a University of Illinois Fellowship.

Mr. Rehak has coauthored several technical reports and papers on the subjects of the research conducted at Carnegie-Mellon University and the University of Illinois. He has been employed as a private consultant for the development and utilization of engineering software. He is a member of Phi Kappa Phi and Sigma Xi honorary fraternities, and is a member of the Association for Computing Machinery.