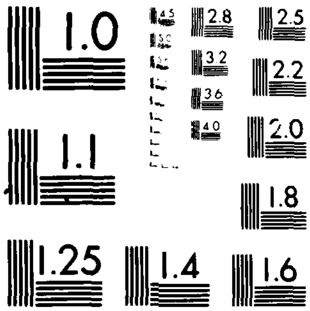
AD-A108 104 TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES F/G 9/2
DECISION SUPPORT SYSTEMS: A PRELIMINARY STUDY.(U)
SEP 77 R T YEH, W W BLEDSOE, M CHANDY N00039-77-C-0254
UNCLASSIFIED TR-74 NL

1 of 2
AD
A108104


1.0
4.5
5.0
5.6
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

(1)

SOFTWARE AND DATA BASE ENGINEERING GROUP
DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT AUSTIN

DECISION SUPPORT SYSTEMS:
A PRELIMINARY STUDY*

by

Raymond T. Yeh, Woodrow W. Bledsoe, Mani
Chandy, Philip Chang, Daniel Chester, Jack
Lipovski, Jayadev Misra, Laurant Siklossy,
and Robert Simmons

Sep[redacted]977 TR-74

* This research is supported by ARPA under contract N00039-77-C-0254

DEPARTMENT OF COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN

FORWARD

This report is the result of a study on Decision Support System sponsored by ARPA. A seminar, three times a week, was conducted during the summer of 1977 for the study. Participants of the seminar include my colleagues: W. W. Bledsoe, M. Chandy, P. Chang, D. Chester, T. Kunii, J. Lipovski, J. Misra, L. Siklossy, and R. Simmons; and my students: A. Araya, J. Baker, M. Conner, C. Reynolds, H. Kunii, S. Lee, T. Mao, and T. Tien.

The core of this report is based on written materials provided to me by the participants of the seminar. I am, of course, responsible for all the mistakes that may occur.

Raymond T. Yeh
Austin, Texas
September, 1977

Table of Contents

1. Introduction..1

2. Characteristic and Requirements for Future Decision Support Systems....................................2

3. An Example..4

4. Problem Domain..8

5. Technical Findings..11

   a. System Structure...11

   b. Semantic Representation......................................12

   c. Performance Modeling...14

   d. Software Design: Development of a Comprehensive Methodology and Tools....................................18

   e. Computer Architecture For Decision Support Systems........42

   f. Reliability Issues in the Design of Distributed Data Bases...46

6. Appendices

   Appendix 1: Semantic Representations for an Integrated Data System -- R. F. Simmons

   Appendix 2: Toward a Design Methodology for DBMS: A Software Engineering Approach -- R. T. Yeh and J. Baker

   Appendix 3: Software Design Tools -- D. Chester

   Appendix 4: A Method for Control of the Interaction of Concurrent Processes -- M. H. Conner

   Appendix 5: Some Thoughts on Automatic Theorem Proving in Data Base Design and Use -- W. W. Bledsoe

   Appendix 6: A Computer Architecture for a FDSS

1. Introduction

Few people knowledgeable in computer science would deny the assertion that we are in the midst of a revolution caused by the increased availability and power of computers. Yet, few can predict what lies ahead just two decades from now based purely on what modern electronic computers have accomplished over the last 25 years. But one thing is sure, the information explosion will continue and at an ever increasing rate. This, coupled with the continued declining cost in computing, will make data processing our number one national asset in the management and organization of our resources. We predict that the need for computer-based decision support system (DSS) will increase dramatically.

During the 60's, much hope has been placed on the so called "Information Management Systems" to support humans in various stages of decision making. While many systems have been developed, the dream was never quite realized due to the fact that technology was not there. If history is any guide in computing, we can safely say that demand on the sophistication and usage of DSS will exceed its actual capability by a wide margin. In light of the trend that this nation will become so dependent on such systems that mistakes can have serious economic, political or environmental impacts, it is important that resources be devoted now toward the understanding and construction of such systems.

This report contains findings of a summer study, supported by ARPA, on the requirements and design of future decision support systems (FDSS). It is our opinion that computer science as a discipline has finally reached sufficient maturity to provide a technological basis for realizing much of the goal of "information management systems" of the last decade.

2. Characteristics and Requirements for Future Decision Support Systems

There are many decision support systems (DSS) which work well today. However, most of these with any sophistication are specialized small systems. Large systems are usually badly designed with ad hoc techniques. As a consequence, it is usually extremely costly to modify such systems. The study here is not intended to come up with piecemeal solutions to the existing problems, but rather to investigate the feasibility of developing a methodology for constructing decision support systems which can meet the demands of the next decade. We think that through this more systematic approach to the problem as a whole, many of the existing problems will also be solved. We shall first outline some of the important features of FDSS.

a. Physical Characteristics of FDSS.

i) System is geographically distributed.

We expect most large systems to consist of a number of smaller DSS, organized in parallel or in hierarchies, and communicating with each other (see example in section 3).

ii) System is large.

It usually contains vast amounts of data of different types, and many processors.

iii) System will be used by many different kinds of users for decision support.

iv) Dynamic environment.

Data is constantly added or deleted from the system, and requirements are changing (due to new applications or new machines, etc.).

b. Requirements for FDSS

i) Adaptability and Modifiability

DSS should be adaptable to a wide variety of problem domains.

In particular, the system should be able to evolve to improve the quality of its support. This can be done by adjustments made either on the information content or structure of the system.

The design for the system should be such that usually small changes in the environment of the system should cause correspondingly small changes in the system.

ii) Intelligence

We expect that more sophisticated DSS should be capable of engaging in complex dialogues with users, and is capable of providing fast response to complex queries most of the time. We shall make explicit two features that DSS should have to achieve this goal.

ii.a) Domain and goal knowledge

Besides the obvious knowledge of the domain, the DSS should have general knowledge of the types of goals of interest to the user.

Why a knowledge of goals? Since the data base is assumed very large, we must assume that the user does not know all the implications. Hence, it is the system's responsibility to point out to the user relevant data of which the user may not be aware, and which he may not have thought to ask, but which would help his goals. For example, a system to support software design should be able to evaluate and respond to a query such as "I intend to make such-and-such changes in the design, what are the effects of these changes? Why?" The system must therefore maintain a general awareness of the domain as it is being queried and modified by the user. This knowledge about goals represents the knowledge of (human) experts in different problem situations.

The DSS's answers should be relevant, i.e., be directed toward the known general goals of the user. They should also be explicit and given at the level of thought that is familiar and comfortable to the user.

ii.b) Generality

DSS should handle unexpected situations, by asking questions of the user if necessary. We cannot anticipate every type of query by having a specific program to answer it.

iii) Trustworthiness

We must have a high degree of confidence in such a system. The notion of trustworthiness implies that the system should be reliable, robust, available on demand, and secure.

3. An Example. A scenario is described in the following to illustrate some of the desirable properties of FDSS and the environments that surround it.

Dramatis Personae

A client interested in ordering oil. Several DSS: local, regional, nataional. A salesman (we may assume that the salesman converses with the DSS), translating the client's requests into a language understood by the DSS. (By DSS we shall mean the local DSS, and generally assume a combination Salesman/DSS to avoid problems of translating client's requests to the DSS, or interpreting output from the DSS.)

C = client.

DSS = local DSS with salesman's help.

S = Salesman as distinguished for the DSS (i.e., the human being).

Scenario: (parenthesized statements are explanatory comments about the DSS).

C: I need 1,000 units of oil, quality premium, at Harlingen depot.

DSS: when?

(an order needs a date of delivery)

C: within ten days.

DSS: 1,000 units will be available at $14/unit within the specified time period. 800 units will be shipped from A and 200 from B.

(System uses operations research tools. For example, it may utilize a "Linear Programming" program to determine an optimal solution. This implies that the DSS knows about the existence of this program, and the kind of inputs and outputs associated with it.)

S: Consider shipping all 1,000 units from A to save extra shipping cost.

(Salesman intervenes to try to improve the solution from the DSS.)

DSS: Sending the additional 200 units from A would increase overall cost.

S: why?

DSS: An additional truck from A would be required carrying only 200 units. In the proposed solution, the truck from A is full, as it is delivering oil to other customer.

(DSS explains answer by comparing the solutions. The "discontinuity" in the truck capacities is noticed.)

C: Okay.

DSS: Before confirming, let me mention that if you are satisfied with only 800 units, the cost per unit will be only $13.40/unit.

(The dimensions of an order are: size, cost,delivery date, delivery delay, security of the delivery, price guarantee, etc., the system is now looking for discontinuities near the order as given by the customer.)

C: No, 800 is not enough.

DSS: I can ship up to 1,500 units at almost no additional shipping cost.

(First, a discontinuity down from the size of the order was noticed. It was found by considering tables of prices and amounts of oil available at the price. Now the system has found a discontinuity up from the order size.)

C: OK, ship 1,200

DSS: If you can wait 4-6 extra days, I should be able to ship these amounts for less.

(The system now looks for discontinuities in the price of the oil if the delivery date is changed. It consults tables of expected deliveries of oil in times close to the order.)

C: How come?

DSS: There is a scheduled ship arrival with cheaper oil.

(DSS explains, using some inferencing.)

C: What will be the cost then?

(The original DSS answer was qualitative: "for less". The customer wants something more precise.)

DSS: I'll need 20 minutes before I can compute that exactly. Approximate price: $12.80 - $13.35. If you call back after 5:00 p.m. I can compute the price immediately.

(The system can give approximate answers, and knows about its capabilities, and typical system loads.)

C: I can't wait 4-6 days for the shipment, and I am willing to pay up to $14.50/unit. Please confirm.

S. I'll try to confirm the order now.

(DSS has access to summary information about available oil at various sites. This information is updated periodically, say every half day.)

DSS2 (Regional DSS): There are only 400 units left at A.

(In the meantime some capacity at A has been utilized. DSS did not know about it. At confirmation time, DSS2 is interrogated.)

DSS: I can confirm shipment of 400 units from A and 600 from B, average cost, $14.25.

DSS3 (National DSS): Stock at A keeps running out. We must increase shipments to A. I suggest a shipment increase of 25%.

(The National DSS keeps dynamic information on transactions, and attempts to improve oil delivery performance.)

C: I want to plan next month's delivery schedule. I want 800 units on the 1st, 7th, 18th and 30th. How much would it cost me?

DSS: I cannot confirm delivery on the 1st, closest date would be the 3rd.

C: Okay.

DSS: Shall I confirm the following delivery schedule and cost?

Location	Date	Cost/Unit
A	3rd	$12
B	10th	12.50
C	19th	11
D	30th	13

C: Okay.

DSS: All confirmed except delivery on 30th. Change to delivery from location A at $13.10/unit. Okay?

C: Okay.

(An illustration of unexpected events.)

DSS3 to DSS: Bad weather at sea. Arrival of ship at D delayed 8 days approximately.

DSS to S: Customers C2 and C3 have confirmed orders from D. They must be contacted to check whether they can accept the delay. If not, we need to find other sources to fill their order.

(DSS keeps a watch on the weather to the extent that it can influence such dimensions of an order as: time of arrival, quantity of arrival, possible loss at sea, increased cost due to delays, strikes, etc.)

4. Problem Domain

In the previous e:.ample, we try to exemplify several concepts which we shall discuss in this section and point out general problems to be encountered in FDSS research.

a. Discontinuities

The decision support system (DSS) provides information to the user about discontinuities near the area presently being considered by the user. (A discontinuity exists if a small change in one parameter of the domain results in a large change in another parameter of the domain.) Although the use of "discontinuity" concept for the design of DSS is new, we have quite a bit of experience in the design of an airline reservation system for western Europe utilizing this idea.

b. Distributed data

Data is distributed at different sites. Some sites may have only summary or probabilistic data. In our scenario, there is a hierarchy of authority, with higher level DSS having only the summary information. Some problems encountered here include the consistency of information at different sites, data and process migration, access rights, performance issues, etc.

c. Knowledge bases

i) Decision support capabilities - the system can help the user make knowledgeable decisions. This not only implies that the system has specific knowledge bases, but also must contain general knowledge about the world. (For example, simulation models.)

ii) Self-knowledge - the DSS has information about its own capabilities such as expected costs of running its programs.

d. Inference capabilities

This aspect is of course something that artificial intelligence (A.I.) has been concerned with for sometime. However, most of the existing A.I. systems are small by comparison, and it is not clear that techniques and principles used in constructing small systems can scale up. To overcome this barrier of size, it seems that certain conceptual tools are necessary. We list a few here:

i) Ready accesss to data and procedures

The designer must be able to think of his large data and large procedure base as readily accessable, so that he does not get sidetracked in data access issues.

ii) Conceptual neighborhoods

This idea is used during retrieval (relevant information is information in the same neighborhood) and while searching a problem space (nearness to a discontinuity or approximation of a solution).

It is an obvious extension of focused access to data. We not only wish to access specific data, but also data "close" to these specific data. Implementation would depend on the metrics or topology of the data.

iii) Problem-solving tools

The designer should have a set of formalized concepts or techniques such as planning (subgoaling, problem reduction), backtracking, plan execution (simulation), etc., at his disposal as tools for general problem solving.

iv) Concurrency

For large systems, it must be the case that a designer can think in terms of many processes running concurrently.

v) Information types

The information in a DSS must be of many types. We can distinguish at least:

- environment knowledge (position and status of troops, transport capabilities, etc.).
- user models. In the past they were often implicit. They must be made explicit. (For the same questions, the answers given to the Secretary of State or a colonel stationed in Turkey will usually differ.)
- system self-knowledge. The DSS should be able to describe its capabilities, explain its logical organization and the methods it used to answer questions, etc.

We think that adequate engineering support for these conceptual tools can overcome the barrier of size. We also believe that new computer hardware is crucial for providing the necessary engineering support. In particular, support for content-addressibility (parallel access to data and procedures), context-addressibility)"semantic paging" for retrieving relevant information), and concurrent evaluation of conditions (hardware implementations of demons.) More detailed discussions of the hardware support will be provided in the next section.

e. Dynamic environment

The example illustrated that data changes through time. However, in reality, the whole environment; requirements, processors, data, etc., changes through time.

When constructing very large systems with dynamic environment, the designer is forced to consider evolving systems rather than fixed systems. One may approach this problem by designing a flexible system structure so that small changes in system environment will impact a correspondingly small change of the system. Or, one may predicate the system's usage and its environmental changes in the near future so that contengencies for growth can be provided in the design of the system. In both approaches, a methodology for design and modeling is needed. While it is possible to borrow from existing software design methodology and performance modeling techniques, much more is needed. For example, in current design methodology, design documentation is almost totally ignored. Similarly, performance evaluation is usually done too late - after the system has been constructed. How can performance modeling be incorporated into the design phase is an important problem in the design of evolving systems!

In order to develop a methodology for the realization of FDSS, we conclude that knowledge in many diverse disciplines of computer science including software engineering, artificial intelligence, modeling, data management, and computer architecture, must be brought to bear on these problems. We shall present our technical findings in the next section.

5. Technical Findings

a. System Structure

It is a well accepted principle today that large software systems should be structured hierarchically with each level in the hierarchy described by an abstract machine which is implemented by the machine at the next lower level.

In Figure 1, we propose a hierarchical organization of language interpreters, memory management systems, and hardware that we believe can provide an integrated data system for decision support in the near future.

The proposed system can be accessed by the user via many languages; a subset of English, a Formal Data language, and Predicate Logic, etc. Other languages are implied by the various support systems s.ich as statistical and mathematical packages, graphics, and various models; economic, political, etc. Much complexity is implied for understanding statements, questions and commands in the several languages that have been mentioned. Each language requires an interpreter that embodies a description of the language it can accept and a set of transformations to produce representations of its input in the common language of semantic relations. The prevalence of inference rules introduces virtual data paths of potentially infinite length and questions requiring many inference rules for computing their answers may greatly multiply the number of data accesses in the system.

Effective computation of inferences will require improved architecture with parallel processing capability among shared fast memories as well as disc processors such as the proposed CASSM system that can provide parallel disc searching capability. (see Appendix 6).

It should be pointed out that levels in the proposed system are not fixed, but is rather flexible depending on the specific system (see part d below). In the next few subsections, we shall focus our attention on various problems and issues associated with such a proposed system.

b. Semantic Representation

One goal for data management research is an integrated data system

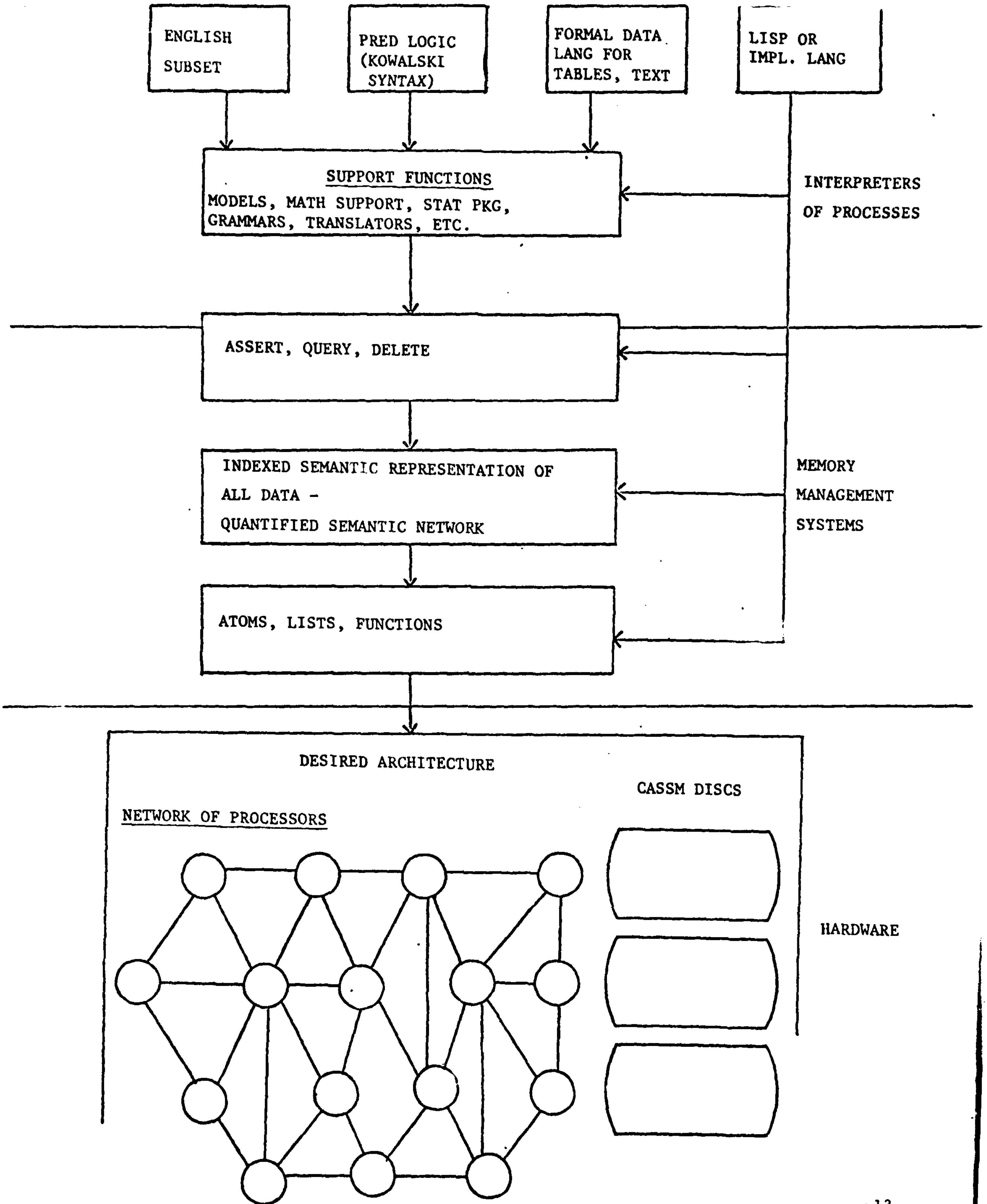

ENGLISH SUBSET
PRED LOGIC (KOWALSKI SYNTAX)
FORMAL DATA LANG FOR TABLES, TEXT
LISP OR IMPL. LANG
SUPPORT FUNCTIONS
MODELS, MATH SUPPORT, STAT PKG, GRAMMARS, TRANSLATORS, ETC.
INTERPRETERS OF PROCESSES
ASSERT, QUERY, DELETE
INDEXED SEMANTIC REPRESENTATION OF ALL DATA -
QUANTIFIED SEMANTIC NETWORK
MEMORY MANAGEMENT SYSTEMS
ATOMS, LISTS, FUNCTIONS
DESIRED ARCHITECTURE
CASSM DISCS
NETWORK OF PROCESSORS
HARDWARE

FIGURE 1. - A proposed organization for an Integrated Data System

that uses a common representation for tables, logical assertions, and text. Tabular information is the stock in trade of current data management systems and its usefulness is well established. Text is a term that describes the content of general files of symbolic material such as programs, unprocessed data, and natural language. Logical assertions include simple propositions, inference rules, and systems of assertions that are in fact predicate logic programs to accomplish certain computations, e.g., proofs of programs, grammatical analysis and problem solving.

A unified representation for all these materials is required to minimize the complexity of the system. A possible common representation formalism into which logical assertions (tuples), tables, and text may be transformed is Quantified Semantic Networks. The networks provide indexing to any extent desired and a classification system for all elements of vocabulary used. They are generally operated on by three operators, ASSERT, DELETE, and QUERY, and include full logical inference capability. In Appendix 1, the power of quantified semantic network for the proposed integrated data system will be discussed in detail.

c. Performance Modeling

Modeling will play a curcial role in the development of a design methodlolgy for FDSS. We shall identify a few areas involving modeling.

i) One of the contrasts of existing data base management systems and AI systems is that in DBMS design extreme care is used to minimize the storage, whereas full indexing is usually employed for AI systems for flexibility. It is clear that flexibility is necessary for FDSS and must be paid for. The question is, how much? In our proposed system for an integrated data system, we advocate the use of quantified semantic networks for representing all kinds of data. It is necessary to know the tradeoff between flexibility and storage inefficiency at each level of the system hierarchy! We believe that performance models should be set up for such a system, with levels of abstraction clearly identified, so that such tradeoffs can be measured in a relatively precise manner.

ii) Subsystem Performance: Competition and Interference in Distributed Systems

Performance modeling of program subsystems within a larger geographically distribured hardware system configuration have not been fully accomplished. The transition to a distributed environment with network interconnections and hetergeneous host computers adds an extra dimension of complexity. Performance characteristics of the computer network and the associated host computers must be estimated under varying workload conditions ans the effective resource availability to the given subsystem determined under varying load conditions. The performance of the specific subsystem in question can then be predicted through analysis of competition with other programs for the effective amounts of system resources. This characterization will require, however, the determination of the performance of the program as a function of the competing programs, the system configuration and the effective resource levels available under varying workloads. The National Software Works (NSW) is a prime example of a program which operates in such a competitive distributed environment. NSW competes with the other processes extant on the ARPANET, both for network resources and for resources with the host computers.

We can use NSW as an example of the types of factors which are involved in subsystem performance analysis. Questions that we are interested in (in WW terminology) include:

- How does MSG response time vary with the TENEX "pie-slice (fraction of CPU time dedicated to NSW)?
- How does the non-NSW workload on the TENEX PDP-10's impact NSW performance?
- How will changing the hardware configuration (for instance, increasing the amount of main memory) impact performance?
- Can similar performance tools be used to analyze MSG running on other machines, such as an IBM 370?
- How will network load impact performance of NSW functions operating on geographically distributed machines?

iii) Reliability Models

Reliability plans for distributed data base systems are complex because of the number of factors that need to be considered. Enhanced reliability is achieved at the expense of additional hardware and increased processing and communication requirements. It is very important to estimate the overhead in enhanced reliability protocols. It is, therefore, necessary to have modleing tools to predict the impact of performance of different reliability plans. Our overall goal is to model the interrelationships between reliability and performance. For instance, from the point of view of rapid recovery it is helpful to have two copies of a file stored in proximate locations in a network (RECOVERY ISSUE). Proximate copies also reduce the overhead of maintaining consistent copies (CONSISTENCY ISSUE). However, from the viewpoint of btaining rapid responses to queries it may be preferable to have copies placed in widely separated locations (PERFORMANCE ISSUE). We propose to build performance models to help resolve these tradeoffs.

iv) Design of Evolving Systems

We cannot afford systems which require drastic expenditures to adapt to changing environments. It is generally accepted that rapidly changing environments are a fact of life in the computing area and especially so in decision support systems. There are two ways of designing systems to handle the costs required to adapt systems to constantly changing user requirements. One approach is to design systems to meet all eventualities without attempting to specify what contingencies are likely to arise in each specific case. The second approach is to require planners to consider possible contingencies, evaluate (rough) probability estimates of different scenarios, and then plan systems to adapt gracefully to probable contingencies. Scenarios may be specified in terms of pessimistic, average and optimistic estimates. The process of gauging future contingencies must proceed periodically, as the system evolves. A static design is concerned with how to distribute data and processors, select communication line topologies, and so on. However, a contingency plan must include a complete design for the current period and then specify appropriate actions for probable contingencies in future periods; for instance, IF after two years, the level of activity in the Gulf region develops as expected, THEN increase the processing capability in that region as planned; HOWEVER, IF the level of activity is much less than expected, THEN shift processing capability to headquarters..... It is important that such performance models be part of the overall design model so that performance of the system can be controlled at the design level.

d. Software Design: Development of a Comprehensive Methodology and Tools

i) Design Philosophy

We advocate a design approach that is somewhat like the process of sculpting a block of stone; this is done by chipping it away gradually as the finalized sculpture takes shape. In order for the software designers to do their refinement steps effectively, the designers need guidance as to where to chip next, and tools for measuring how close they are getting to their goal.

Formally, we propose to characterize the design process by means of three interacting models: a model of the system structure, a model for system (performance) evaluation, and a model for design structure documentation. These three models will be refined simultaneously during the design. Furthermore, in order to allow a designer to "tinker" with his design, we propose a computer processable specification language and tools so that early feedback can be provided to both the deisgner for the quality of his design, or to the user for the inadequacy (if any) of his requirements.

In the next few subsections, we shall describe briefly the progress we have made toward the development of a comprehensive design methodology with the aforementioned philosophy in mind, and identify the problems that remain to be tackled.

ii) Design Process

Our concept of the design process is that it consists of many stages, each of which has a model that satisfies some of the constraints on the design and a set of constraints that have yet to be satisfied. Figure 2 shows the different paths that the process can follow from the original constraints (requirements) and the null model to the final model and null constraints.

As can be seen from the figure, each step along a path can be expanded in several different directions to reach different final designs. Thus, each model represents a family of designs. By providing suitable means for documenting models, we make it easier for the designers to back up and try another member of the family when one path leads to a bad design. We also make it possible to consider other designs in the family when system requirements are changed, either during the design process or after the system has been in service for some time.

Each model along the design path is a refinement of the previous one. The first models only exhibit the gross behavior of the desired system without consideration of performance and hardware requirements. This is an especially important phase in the development of the decision support systems because it clarifies the purpose of the system by requiring the designers to state precisely what it is they want the system to do. At the same time they are able to simulate the system at this early stage and modify it until it appears to be what they really want. Later models begin to reflect the efficiency and hardware considerations as the designers begin to outline the algorithms that will actually be run on the target machine. Eventually through this process, the original constraints get satisfied and the design is ready for transfer to the hardware of the actual system.

iii) Three Models of Design

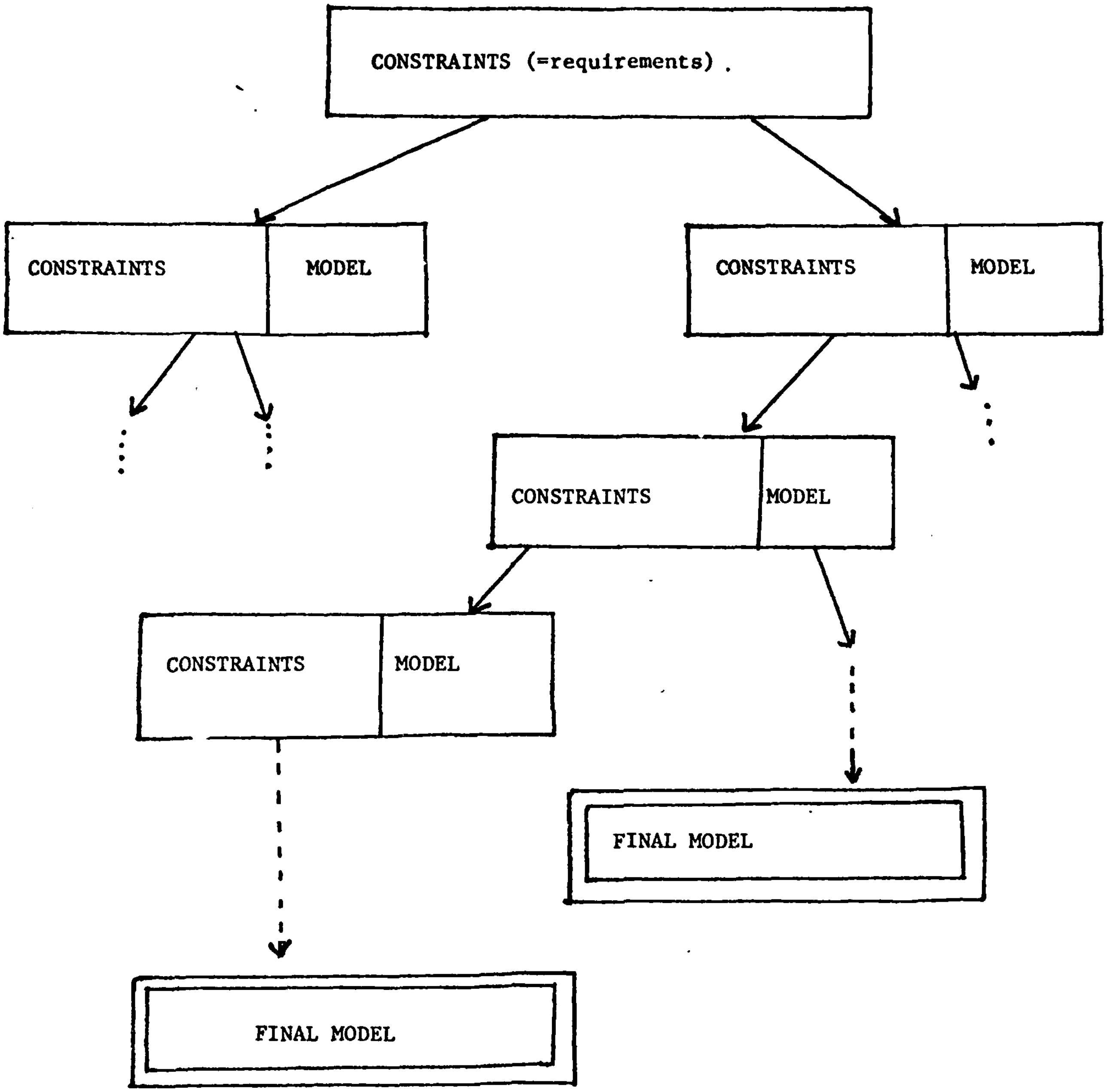


FIGURE 2. - Possible paths in the design process.

iii.a) System Structure

We envision the design of a DSS as a stepwise refinement process of functional abstraction which begins with the construction of a "top-level" abstract machine, M_n, satisfying the functional requirements of some high level requirements specification. This machine consists of a set of data abstractions represented by formal module specifications. Each module specification is self-contained in the sense that it specifies the complete set of operations which define the nature of the data abstraction. Collectively, these data abstractions define the data model which is visible to the user of the machine.

In the next step of the process, another abstract machine, M_{n-1}, representing a "refinement" of M_n is designed. Its data abstractions are chosen in such a way that they can "implement" those of M_n. Basically, this implementation consists of a set of abstract programs each of which defines an operation of M_n in terms accesses to functions of machine M_{n-1}. A verification process can then be used to ensure that the implementation is consistent with the specification of both machines.

This stepwise process of machine specification, implementation, and verification proceeds until, at some point, the data abstractions of the lowest level machine can be easily implemented on a specified "target" machine, which may be the data abstractions of some programming language, a low-level file management system, or the operations of some appropriate hardware configuration. This design process results in a structure consisting of a hierarchy of abstract machines, or levels, $M_n, M_{n-1}, \ldots M_0$

connected by a set of n programs I_n, I_{n-1},...I_1. Each machine M_i in the hierarchy represents a complete "view" of the system at a $I_i (1 \le i \le n)$ represents the implementation of that view upon the next level machine M_{i-1}.

We observe that the notion of levels of abstraction translates to a natural interpretation within the context of decision support systems. That is, we can expect that any integrated data system will have a wide variety of users whose views of the system and access requirements will be quite different. Through the hierarchical design approach different levels of design may be constructed to accommodate this bariety of views and access requirements. A specific view representing one path of Figure 1 is shown in Figure 3.

It is observed that through hierarchical design, many different users may be accommodated, and that reliability and understandability of the system is enhanced. Furthermore, such a system is machine and application independent and hence can evolve with its environment. More detailed discussion of this model is contained in Appendix 2.

iii.b) Design Structure Documentation

The role of specifications in the development of large software systems is quite important. Specifications are used not only as a means of communication between members of the design team, but also serve to enhance the understandability of the system. This is important both for users of the system and for future design teams which must perform modifications.

In order to understand a system as a whole or for explaining why a particular design was developed, there exists the need to document the system design and the design process.

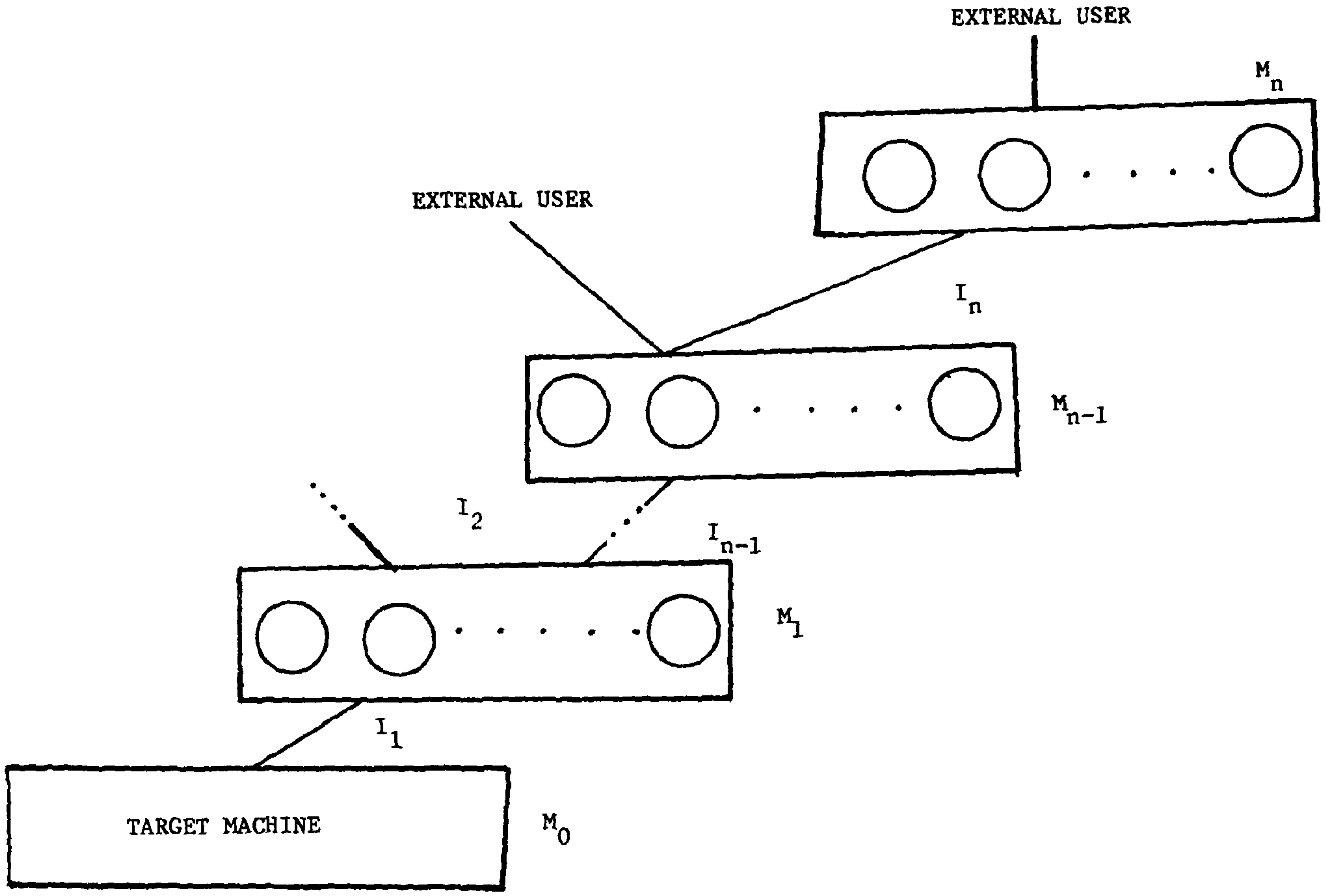


Fig. 3. A hierarchy of formally specified machines showing modularity. Levels may be constructed to accommodate the different views required by various users.

Such documentation would suppress details - concentrating rather on the global properties of the system design and the design structure.

We have introduced a System Design Langauge (SDL) which can be used to document the design process and record information about the decision-making processes that occur during it. The features of the SDL include methods for:

1. specifying the design alternatives at each level,
2. specifying the hierarchical relationships between system modules, and
3. specifying the structure of each system level.

More detailed information of this language is contained in Appendix 2. However, much more development is needed in order for the language to accommodate the design structure of concurrent, multiple user programs.

iii.c) Hierarchical Performance Evaluation

The success or failure of any DSS, of course, depends greatly upon the level of performance which the system achieves during actual operation. Based upon the results of current research efforts, however, it would seem that our approaches to performance evaluation are somewhat less than satisfactory. This section contains a very general description of a performance evaluation technique which can be used with the hierarchical design approach and which seems to have several advantages over current performance evaluation procedures. This technique involves the construction of a hierarchical performance evaluation model. The purpose of this model is two-fold:

1. to provide the designer with feedback at each step of the design process as to the performance characteristics of his design,
2. to provide part of a basis for choosing between alternative designs at each level.

In this approach the designer develops the system design and evaluation model in parallel - the evaluation model being constructed so that it represents the relevant performance aspects of the current system design. The evaluation model provides constant feedback to the deisgner at all levels of design as to the performance characteristics of the system. Through constant interaction between designer, the system design, and the evaluation model, it is hoped that a reasonably efficient system can be developed with a minimum of backtracking and redesign.

Evaluation Model Structure

The structure of a hierarchical evaluation model reflects that of the system design itself. Corresponding to the ith level is a set of performance parameters, P_i, which represents the relevant performance aspects of the machine at each level. Data structure parameters represent information about the abstract data objects of the level (e.g., number of relations, average number of records per block, etc.). While function parameters characterize the operations of M_i in terms of expected execution speed and expected frequency or probability of access. Parameters may also be classified as design parameters or scenario parameters. Deisgn parameters are variables whose values may be changed by the designer to determine the effects of various database designs and implementations upon the performance of the system. Scenario parameters, however, represent an expecteu usage of *he system in terms of the operations and data objects of level i. Their values are determined by the values of parameters of P_{i+1} according to a performance parameter mapping set T_{i+1}. Each mapping in this set defines a performance parameter of P_i as a function of the parameters of P_{i+1}. A set of values for the scenario parameters of level i is called a scenario for level i.

The values of scenario parameters of P_n are determined by an application scenario supplied as part of the high level requirements specifications. The application scenario is a statement of the expected use of the system in terms of the operations and structures of machine M_n. The requirements specification also contains a performance assertion which specifies the level of performance expected from the system for the given scenario. This performance assertion, by its structure, will indicate the measure to be used in analyzing system performance. Various performance measures might include:

1. mean response time for a given load,
2. expected total execution time for a specified mix of operations,
3. total storage requirements, or
4. a suitably weighted mixture of the above.

The specification of this performance assertion enables the designer to construct a cost function, C_n, for M_n using the parameters of P_n. This cost function may be used by the designer to estimate the performance characteristics of M_n.

It should be noted that this model is only a proposal, experiments are needed to evaluate the adequacy of this model, particularly, in the multiple user environment.

iv) Design of Concurrent Systems

The progress made so far is primarily in sequential systems and must be extended to concurrent systems to be viable for DSS. We discuss some of the issues that are peculiar to concurrent systems here. In addition to the usual problems encountered in sequential programs, the two most important problems are

(i) managing the interaction between processes;

(ii) supporting multiple views of the system (simultaneously) for multiple users.

In recent years, a number of techniques have been advocated for dealing with the design issues of concurrent systems. These may be summarized as the following:

(i) *Hierarchical Decomposition:* This technique has been used with great success for sequential programs. For concurrent programs, it has so far been much less successful, since the decomposition of a part needs to take into account the interaction of that part with several other parts.

We propose a methodology for decomposing a cluster of functions simultaneoulsy, where the cluster members greatly interact with each other, and interact only slightly with functions outside the cluster.

(ii) Notion of information hiding: A way to enforce the module independence is to place a discipline for limiting the interactions among them. Furthermore, the modules do not exhibit their internal details, thus enforcing a discipline in their invokation. These ideas are applicable to concurrent processes; we propose a view of process interaction which takes this into account.

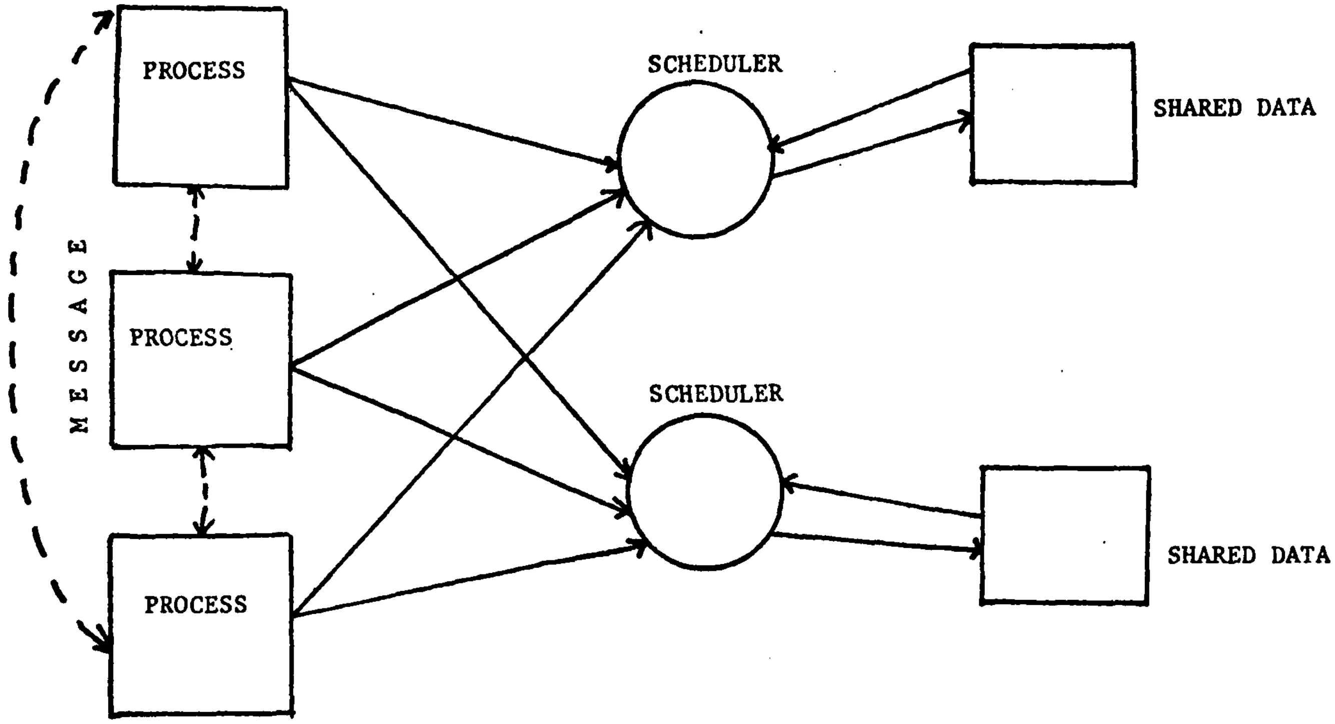
(iii) Enforcement of Coordination: Coordination of the interactions among processes has been studied at great depth, since the pioneering work by Dijkstra. On cooperating Sequential Processes [Dijkstra, 1968], solutions using P,V semaphores dealt with machine level concepts. Ultimately synchronizing mechanisms have to rely on such low level concepts for their implementations. However, it is counterproductive to study a complicated system synchronization problem in terms of these primitives. Many different high level constructs have been proposed for synchronization; each of these can be viewed as a means of event driven coordination.

"Demons" have been used in A.I. work to trigger processes whenever an associated condition arises. Thus, some processes are driven by events rather than through explicit invocations. Current attempt is to implement demons efficiently.

Another method of synchronization is through explicit transfer of messages between processes. It is usually implemented through a central "post office" with "mail boxes" which actually are message buffers. This method has been found to be useful in communicating with processes whose identities are known to the communicating process.

A notion of "monitor" has been advocated by Brinch-Hansen [1973] and Hoare [1974]. Monitors are attached to shared global data, through which processes may interact. Monitors enforce mutual exclusion in access to shared data. They also implement a scheduling policy for access to that data (first come, first serve, for instance). Thus, the monitor acts as a central scheduler for access to the data.

For performance, as well as the information hiding point of view, the following process interaction figure illustrates a number of ideas related to process coordination ideas:



Processes interact explicitly with messages sent through implicitly shared data. The process scheduler is transparent both to the process and to the data. A process is not aware of other processes when accessing shared data. Hence, it may be designed and verified as a sequential program, given only the semantics of operations on data.

Similarly, the data object is not aware of multiple simultaneous accesses to it. Hence, it may be designed and verified independent of the invokation sequence.

The scheduler handles the various aspects related to process synchronization in accessing shared data. Each request for access to a shared data is routed to the proper scheduler who decides whether to grant access or not. If a process is granted access to shared data, it returns to the scheduler on completion. If a process is denied access to shared data, the scheduler may put the process in a wait sequence. The scheduler, in fact, implements the scheduling policy. It may grant multiple processes to access the same data simultaneously (as in the reader/writer problem). It may furthermore enforce security constraints.

This decomposition of the problem into its three essential components results in a decomposition in design and verification. Essentially different properties may be proven corresponding to each part.

(i) Process: Correctness of computation. This may use traditional techniques in program verification.

(ii) Scheduler: Absence of deadlock; fair scheduling; absence of indefinite postponement of processes; correctness of access sequences to data.

(iii) Data: Correctness of implementation; integrity. Data verification techniques for sequential programs are applicable here.

An open problem is how to partition data base so that different schedulers are assigned to different portions of the data base while activities are still coordinated.

We propose to study the verification issues in the scheduler, particularly the problem of verifying each property independently. The basic idea is to verify each property based on certain axioms so that the verification of another property does not nullify the axioms. A formalism for studying such a partitioned environment has been developed and is discussed in detail in Appendix 4.

v) Data Base Design

Data bases form an integral part of any DSS. However, the systematic design of data bases has eluded researchers in this area. In this section, we shall describe how automatic theorem proving can be used in data base design, and how system design methodology might be applied to data base design.

We are concerned with a data base system which consists of a very large memory and mechanisms for processing ana answering queries. Also mechanism should be available for processing and storing information in the memory.

Some queries would require the finding of one or more items in the memory on the basis of a given KEY, while others would require calculations and inference on the information im memory.

We envision a hierarchical system whereby (in some cases) a query causes the fetching of selected items from the large memory, and putting them into an auxiliary memory (e.g., high speed core) for further processing in order to answer the query. For example, we might fetch a part of a semantic net from the large memory, and bring it into auxiliary memory for further processing.

The fetching operation itself may require "intelligent" mechanisms, such as simple inferencing (e.g., and-gates, or-gates, matching, table lookup, etc.), calculations (counting, averaging, weighted sums, etc.), and various other methods.

Also, within the auxiliary memory, more complex mechanisms would be used to complete the answer to the query. Since the amount of material being processed in the auxiliary memory is drastically reduced (from the amount in the large memory) we could afford to employ much more sophisticated inferencing programs and calculations.

The large memory might be "distributed" over a large number of sites, with different formats for data in each site, so the hierarchical system might be required to employ different local mechanisms for different sites. And these might be more than two levels in the hierarchy, thereby processing a query in a number of stages.

The large memory might employ new and/or novel concepts in hardware design, including parallel searching ability, content addressability, and ability to do minimal inferences and calculations. The design and implementation of these concepts should be correlated closely with the design of the overall system.

v.a) Automatic Theorem Proving as an aid to Data Base Design and use.

It is highly desirable to have data base systems which can give answers which are not explicitly stored in their memory. For example, a data base which contained only the two entries (A is an ancestor of B) and (B is an ancestor of C), should be able to answer "yes" to the question: (A is an ancestor of C), even though that entry is not explicitly stored in the data base, (provided that it was given an additional inference rule on the transitivity of "ancestor-of").

Much more complicated examples than this can be handled using inferencing mechanisms, but the problem gets more difficult as the size of the data base memory and the complexity (or depth) of the inference is increased. It depends of course on how the entries are stored (as relations, semantic nets, etc.) and what inferencing mechanisms are used. But it is clear that automatic theorem proving (ATP) plays a central role here. It is not that we can use our existing provers as off the shelf items to be "plugged" into this new application, but rather we expect to use the concepts and experience with provers. This situation is similar to that of Program Verification where existing theorem proving pr›grams were heavily modified before they were inserted as modules in several program verification systems.

A good deal of research has already been conducted on inferential data bases. For example, the rather large effort in natural language understanding [Chester & Simmons, 1977] falls in the category as well as many others. Some of these workers have had considerable experience in automatic theorem proving. But, their efforts have left much left to do, especially for large scale systems. Also, it is important that in designing and building new large data base systems (or in developing general procedures for large data base design), that inferencing mechanisms properly interface with the rest of the system. It is important that ATP people work as part of the larger team.

As mentioned earlier, the inferencing mechanisms might be minimal at the fetching point in the large memory. It would probably not be feasible to carry out there more than simple and-or gates, and matches. A possibility would be to retrieve a subset of the data base which is clearly relevant, and to perform inferences and calculations on it in the fast auxiliary memory. Such an interaction might require several references to the large memory, when and if the processing uncovered the need for further data from the large memory.

Even in the fast auxiliary memory we do not expect the inferencing to be very deep (like, for example, the proving of a difficult mathematical theorem).

A more detailed explanation of ATP in data base design is provided in Appendix 5.

v.b) Hierarchical Design of Data Bases

Because of the dynamic environment faced by DSS, the data bases supporting a DSS must be able to adapt frequent changes without extensive reorganization. The top-down system design methodology can be applied in the design of data bases to improve their adaptability. The data bases designed with such methodology will also provide automatic linkages between decision models and have self-organizing capabilities.

The data base design process starts with a high level description of the universe of discourse (UOD) - the part of reality that is of interest to the users. A top level data base schema is just designed to represent this high level abstraction of data. Then the stepwise refinement process begins; at each step of refinement, a new data base schema is formed with more details of the UOD and/or more details of how the data base is actually stored. A hierarchy of data base schemata is thus generated. The schema at the lowest level of the hierarchy contains the storage structure of the whole data base. By using this approach, related data can be "clustered" together and small changes of the environment will only induce small changes in the data base.

Note that in the top-down data base design process, there is no distinction between "logical data base design" and "physical data base design". Traditionally, "logical designs" only consider the user convenience and data semantics in constructing data base schemata; the "physical designs" only consider the efficiency factors in designing storage structures. However, the convenience factors and efficiency factors should not be considered separately as they can influence each other. Our top-down methodology will design data bases by evaluating different factors in their order of importance without a rigid separation between "logical" and "physical" factors. The data bases thus designed should have a better overall performance than those designed using traditional methods.

When the details of the UOD are added to a data base schema, some data abstraction techniques (such as the ones developed by Smith and Smith [ ]) can be used as a guide for refinement. We will develop more "abstraction operators" as the two operators developed in [ ], aggregation and generalization, are not sufficient for the construction of the schema hierarchy. For example, at one level the schema may contain a field total sale per year, and at a lower level the other schema may contain the field total sale per month; the "abstraction operator" we need in this case is a summation operator. This concept of "abstraction operator" can be generalized to contain a whole decision model: the schema in a higher level contains the output of a decision model which uses the data in a lower level schema as its input. The schema hierarchy constructed by using such operators can provide each decision mcdel the required data and can support automatic linkage between different decision models. A strategic model (e.g., a cooporation model) may need some data from the outputs of different tactical models (e.g., financial planning models) or operational models (e.g., payroll model and marketing model). Upon the activation of the strategic model, the data base system can automatically activate the tactical and operational models that are needed. Sprague [ ] noted that the successfulness of a DSS largely depends on the system's ability to link different decision models together. Our design methodology provides a solution to the linkage problem.

The characteristics of the multi-level virtual machines (fig. 2) designed with top-down DBMS design methodology can also be utilized in the data base design process. If a data base schema is based on a level of virtual machines in the system hierarchy, the performance of the schema can be predicted by using the performance evaluation functions developed for the virtual machines. Such performance evaluation can also be applied to guide the self-organizing activities of the data base, the data base schemata can evolve with changing environment in order to optimize the performance.

v.c) Design of distributed data base which uses summary information

The design problem is related to responses based on incomplete or partial information. An example of a flight reservation will illustrate the idea. Consider a primary data base (central computer) which has (all) the information regarding a flight booking. There are several secondaries (mini-computers with slight memory) which can be used to make a reservation. Each secondary holds 1 bit of information, which denotes whether the number of vacant seats in the flight exceeds 10% of the flight capacity. The secondary uses the following logic to book a seat or deny a request.

If the bit shows availability of vacant seats (more than 10% of flight capacity) then a seat is booked on request and the primary is informed of the booking. Otherwise, the request is denied.

Periodically, the secondary might receive messages from primary to turn the bit off (vacant seats less than 10% of flight capacity due to a number of bookings) or on (following cancellations).

Primary uses the information received from the secondaries to decide whether the bit should be off or on; it transmits any change in the status of the bit to all the secondaries.

The point of this example was to show that rapid response to queries can be provided based on incomplete information. However, the danger in the above example is that of overbooking (too many secondaries book simultaneously) and underbooking (all secondaries were instructed to cease booking while there were a number of vacant seats). It seems that this method can be used to keep summary information to serve several sites most of the time; however, some time (with low probability) all the current information may be needed. We propose to study the use of summary information in several real life problems and to generalize the idea. Furthermore, the effectiveness of such strategies have to be studied with probabilities of erroneous response and probability of querying the primary data base.

vi) Tools

A set of software tools must be developed along with the methodology in our project to aid the construction of decision support systems. We should include four classes of tools: languages for communication, modeling system for testing out our ideas, reasoning systems for exploring the consequences of our ideas at a general level, and knowledge systems for gaining from our past experiences. It is envisioned that such an integrated set of tools is itself a decision support system. With such tools, a designer can tinker with his designs by executing and testing specifications, or ask the system questions such as, "I intend to make such a change to my design, what will the consequences be, and why?" We will explain in the following an initial set of such tools.

a) Languages

i) A requirements language. This shruld be a restricted form of English and/or graphics for stating the problem initially. It is characterized by its vagueness and high level of abstraction.

ii) A specifications language. This should be a formal, non-procedural language for stating the problem after it has been clarified. A specification in this language fixes a particular representation of the problem so that finitary procedures can be applied to obtain a solution. This language is considerably less vague than the previous language. It may even be precise. A computer executable specification language is developed, a sample is given in Appendix 3.

iii) A programmming language. This is the procedural language that we use to state our proposed solutions. It may be at the level of a modern computer language like Pascal, or it might be higher.

iv) A meta-language. This is the language that we use to talk to each other (and to the computer) about our engineering efforts, that is, about requirements, specifications, programs, assertions, documentation, models, simulations, testing, debugging, problem solving, reporting, etc. This may just be English, but we should try to formalize at least some of it so that we can get help from the computer.

We may in fact have several examples of each of the above languages to serve special purposes. In any case, each language consists of a vocabulary of concepts that are "natural" for the intended application; this means that they are as close to common sense concepts as possible.

b) Modeling systems

i) An interpreter and/or compiler for the specifications and programming languages. This allows us to test our evolving software to see whether it does what we expect. This kind of testing will catch many of the simpler errors and will help us to see whether we really want the properties given in the requirements or specifications. We need an interpreter for the specifications language because a precise statement of the problem is in a sense a high level solution to the vague problem posed by the requirements.

ii) Special simulation packages. These are used to model only part of the behavior of a system. Queuing models, for example, simulate the interactions between processes while ignoring most of the details of the processes. We may have a different package for each major performance parameter that interests us.

iii) Hierarchical performance evaluator. This will be the tool to support the hierarchical performance modeling discussed in iii.c). We envision that such a tool has some gross similarity to current program verifiers in that inference capability is needed, and hierarchical performance requirements (analogous to the verification conditions) will need to be generated. The development of this tool will be a major undertaking.

c) Reasoning Systems

i) An interacting theorem prover. It can be used to verify conjectures about the developing software. The most important kind of conjecture will probably be that one system design is a refinement of another design or of the specifications.

ii) An inference engine. This is different from a theorem prover in that it is not given a conjecture to prove. Instead it derives "interesting" generalizations about a program or pair of programs. It will need some guidance to know what "interesting" means. This is the system that has the ability to discover important facts as an active agent for the engineering team. It can also be used to determine the consequences of a proposed program modification will be.

iii) A monitor. This is the agent that uses the inference engine (and perhaps the theorem prover) to detect violations of project standards and undesirable interactions between different programs. It can inform a designer that what he is doing conflicts with what someone else has done, or that someone else has already done something similar.

iv) A symbolic debugging aid. There will be debugging aids for use with the modeling systems, but this aid helps the designer locate a bug by looking at the code with him. It will make heavy use of the theorem prover and inference engine.

v) A code analysis system. This is mcre general than the debugger in that it helps the designer find the relevant code that produced some effect.

d) Knowledge Systems

i) An advice-giver. This can help the designer clarify his problem. It is a data base of knowledge about high level concepts, algorithms, heuristics for solving special problems and for general problem solving, and the technical literature. It will be especially helpful when the designer is trying to clarify his problem.

ii) A project library. Here is where all written material concerning the design effort is stored and made readily accessible. It will have an extensive association network so that specific information can be found with a minimum of keyword guessing. It will thus give some question answering ability like the advice-giver.

iii) A knowledge acquisition system. This is the system that we need to put all the detailed knowledge into the other systems. Instead of one system it might be a separate component of each of the other systems. The success of the overall system depends directly on the ease with which its subsystems can be brought up to a satisfactory level of performance.

e) Computer Architecture for Decision Support Systems

The DSS computer architecture will use recent hardware advances (especially LSI) technology to facilitate the development of the very large distributed and intelligent data base management system. We sketch here architectural features of a planned system and some problems for architecture research and development.

Three major computing systems are to be accommodated. Firstly, users interface with the data base system through a network of intelligent terminals. Secondly, intelligent discs are located at various nodes in this network and are powerful enough to search the data where it is stored to avoid shipping large quantities of data through the network. Thirdly, an array computer will use parallelism to extend the analytical capacity of artificially intelligent software. We submit that these three major systems have to be accommodated because none of them alone, nor any pair of them, are adequate to support the envisioned software.

In the following paragraph, we shall give a brief description of the intelligent Disc Architecture since it is used to support the important conceptual tools by providing both content- and context-addressability. It also can be designed to support distributed queries in the network and to support deep theorem proving in the array computer. Other system architecture as well as details of how an intelligent disc can achieve content and context searching is described in Appendix 6.

i) Intelligent Disc Architecture

From our earlier work on the CASSM system at the University of Florida and from related work on the RAP system at the University of Toronto, we have established techniques which will efficiently store relational data bases and semantic networks on a disc. The logic associated with the disc makes it sufficiently intelligent to resolve almost all typical relational queries and sufficiently intelligent to greatly assist extracting useful data from a semantic network for artificial intelligence programs.

The disc architecture will consist of multiple moving head discs (we are looking at IBM 3330 or equivalent stores of about 109 bits per removable disc pack) in which all heads are on a common frame, and there is one head on each disc surface. By moving the frame, the heads are located over a given "cylinder". One or more such discs will be operated together so that their "cylinders" form a larger cylinder; the data on this larger cylinder we call a file. Each head will have a "microprocessor" similar in complexity to current popular microprocessors but having quite different organization and instruction set. It will be attractive to put each "microprocessor" in an LSI chip. The disc track and "microprocessor" we call here a cell. The logic looks like a chain of identical cells. In one revolution of the discs, an "instruction" is executed on the entire file. The file consists of records of a variable number of words, and the words are fixed length. Records correspond to tuples in the relational data base system and to nodes in semantic networks. The first word of each record stores a bit stack. Other words appear to store domain names and items in the tuple, or arcs incident from the node in the network. A typical "instruction" pushes a bit in the bit stack of every record in the file, which is the result of a search for a domain name and item in the tuple, or the result of transfering from one node to another node through an arc in the network. Alternatively, one can AND or OR the result of the search or transfer onto the top bit of the bit stack in each record. These operations are accomplished by means of a one bit wide random access memory, with as many bits as there are records in a cell, in each cell. Significantly, as the data base size increases, it is possible to add more discs, so that retrieval time is relatively independent of the size of the data base. (If tertiary memory is used, as will be necessary for 10^{12}-10^{15} bit data bases, this feature will be harder to maintain but is still possible). Furthermore, both tuples of relational data bases and nodes of semantic networks can be efficiently stored in the same record, and that record can be accessed by two users who are working in either semantic networks or relations.

ii) Problems for Research and Development

Since the intelligent disc is common to studies in networks, relational queries and artificial intelligence it is necessary to build a prototype disc system and make it available to the other researchers. Since research on the architecture of an intelligent disc has been essentially completed in the design of the CASSM machine, this aspect of the work is more like development of a tool based on that research. However, the added requirements imposed by network and artificial intelligence pose some new research problems. Significant among these are the techniques to lock out records on the file and to regenerate a query from one file that is to be sent to another file.

In the network architecture we expect the usual problems of deadlock, routing, and protection. Considerable research has to be carried out to evaluate how to take advantage of intelligent discs that permit locking of records. Performance studies will be required to determine the effect of strategies to search multiple files on traffic through the network.

In the array architecture, further studies are indicated to determine if cannonical forms can be used to make vector operations out of operations like COND (from LISP). Studies of the utilization of memory by concurrent vector techniques will indicate how successful the cannonical forms may be.

Other research questions interrelate with other areas and will be described in other sections.

f. Reliability Issues in the Design of Distributed Data Bases

A distributed data base has a number of different specifications associated with it. Broadly, we may divide them into two categories: those dealing with the user and those pertaining to the functioning of the system.

The specifications associated with uses include (a) specification of query language through which the user communicates with the system, (b) specification of the (user) view of data that the system supports, (c) response time and other performance specifications. System specifications may include those aspects dealing with integrity, consistency, absence of deadlock, specification of a fair scheduling policy, etc.

A number of new problems arise in dealing with system function specification. In particular, a language formalizing such specifications is a must; however, very little work has been done in formal specifications of properties of concurrent system. The problem can be explained informally in terms of a simple reader-writer problem. Readers access a data base in query mode; writers perform updates on the data base. For performance reasons, it is desired that

(i) a number of readers may simultaneously access the data base.

In order to avoid unpredictable modification, it is required that

(ii) no more than one wirter may access the data base at any time. Furthermore, no reader may access the data base if a writer has been granted access.

A fair scheduling policy must also ensure that no process is indefinitely postponed. Hence, it is required that

(iii) no reader is granted access to data base if there is a writer previously waiting. Similarly, no writer is granted access if there is a reader previously waiting before it.

Finally,

(iv) a reader or a writer may be granted access if no other process has been currently granted access to the data base. A reader must be granted access if only readers are currently accessing data base and no writer is waiting.

This problem, though simple in nature, results in a number of distinct solutions of varying complexity. In order to verify that a solution meets the requirements, we need to state the requirements in a formal manner, independent of any specific solution. This small problem highlights some of the difficulties. For larger problems, specifications are required not only for verification, but also to check for the consistency of the requirements.

A number of other forms of assertions, to be called "soft assertions" [Saltzer, 1977], seem to arise in distributed data base specifications. Soft assertions involve the notion of time and probabilities. While probabilistic assertions have been found useful in other areas (operating systems in particular, where one may assert that the probability of system deadlock is less than 10^{-5}, etc.), "time" has not been used as a parameter in specifications of systems. The reason for this is simple: normally we deal with algorithms or processes which do not exist for extended periods of time or which model a part of a real system evolving in time. Data bases exist for years and hence must include "time" as an improtant parameter in the system specifications.

The time dependent assertions can have a variety of types, as illustrated below:

(i) Copy at a location A is consistent with copy at another location B, to within one day.

(ii) Every March 31, every copy is current.

(iii) On the 1st of every month, automatic transfer of a certain amount takes place from one account to another.

At present, no formal technique exists for succinctly stating such assertions or verifying a system with respect to these assertions.

Probabilistic assertions deal with probabilities of events. An event, such as total system deadlock, may not be preventable in any reasonable manner. However, it may be asserted that the probability of such an event is negligibly small. A number of efficient solutions to several system problems may be designed, if one is willing to risk an undesirable event; however, it must then be shown that the event is highly unlikely. For instance, an airline might follow a booking policy where the probability of overbooking by x seats does not exceed $10^{-(x+1)}$. Probabilistic assertions may also relate the software's ability to deal with physical component failures, given the probability of such a failure.

A number of research issues arise in dealing with such assertions.

(i) formal specification technique for soft assertions,

(ii) identification of reasonable (tractable) classes of assertions which are pertinent to distributed data bases,

(iii) design and verification of system based on such assertions.

A system constraint of special importance is that of _integrity_. It is a constraint either dictated by the application or enforced by the data base administrator. _An integrity constraint is an assertion about the data base which holds following every transaction_. Hence, it must be verified that every transaction maintains integrity. An example is a constraint such as "no employee earns more than his manager", or "no manager manages less than 3 persons or more than 20 persons", etc. However, it is expensive to verify through run time checks that integrity is preserved. Fortunately, we have found that most of the integrity constraints deal with the _structure_ of the data rather than the _value_ of the data. For instance, social security number is an integer with 9 digits; no employee belongs to more than one department, etc. Such constraints are routinely handled by compilers through type checking.

This idea can be exploited by preprocessing the transaction structure to determine whether it would violate the structure constraints. However, most run time checks are usually limited to a single tuple or a small number of them. ("Salary of no employee below the rank of a manager may exceed $20,000-" can be checked whenever a tuple is updated). This type of integrity constraint does not require us to go over the entire data base.

Another commonly occurring form of constraint dealing with an entire data base can be checked _incrementally_. For instance, a constraint might require that the average salaries for males and females must be within 10% of each other. Normally, it would be required to verify this following addition of every new employee and change in salary of any employee. This constraint involves the entire data base. However, the relevant quantities can be computed incrementally, if we keep track of total number of male and female employees and their total respective salaries.

Most integrity constraints dealing with an entire data base exhibit this property of incremental computation.

A problem studied by Eswaran, et al, [1976] is the sequence in which multiple transactions may interact to destroy integrity, though each transaction preserves integrity when executed above. They showed that integrity is preserved if and only if every transaction locks all pieces of data used by it prior to any unlock. This has the interesting property that a system wide requirement is unnecessary, so long as every transaction meets this requirement. However, the proposed method also implies a specific order for locking data items in order to avoid potential deadlocks. This, in turn, implies that dynamic decisions which items should be locked during a transaction, are dangerous. Furthermore, their solution is based around a central scheduler which grants (or denies) locking privilieges based on the entries in a lock table, which shows the items currently under lock. Several problems arise in connection with multiple copies of the same data base, location of the lock table and recovery problems when the scheduler (or the lock table) site fails.

A number of issues arise in handling multiple copies. The central problem is that of recovery from a faulty transaction or hardware failure. In the latter case, it may be necessary to suspend all operations on all copies; otherwise, some queries may receive incorrect responses. A statistical approach is needed. Certain other problems dealing with multiple copies are the following:

(i) How consistent do the copies need to be? Absolutely consistent, within 1 day of each other, etc.?

(ii) Given sufficient time and no further updates, do all the copies converge to the same state?

(iii) How can the lock tables and file directories be maintained absolutely consistently?

(iv) How are updates broadcast so that older updates do not overwrite the newer updates? This problem has been addressed by Bunch [1977] with time-stamping and Allsberg [1977], for inventory type data bases.

A further area of research is transaction preprocessing. If the transaction is not dynamic, i.e., decisions about data accessing etc., are not made based on the outcomes of responses in the same transaction, then it is possible to preprocess the transaction to guarantee certain properties. As we have mentioned earlier, this can be used to eliminate checks on the resulting data base for integrity constraints. It can be used to guarantee legality and authorization of access.

REFERENCES

1. Ballantyne, A.M. and Bledsoe, W.W., (1977), "Automatic Proofs of Theorems in Analysis Using Non-Standard Techniques", J.ACM, vol. 24, pp. 353-374.

2. Baker, J. and Yeh, R.T., (1977), "A Hierarchical Design Methodology for Data Base System", TR-70, Dept. of Computer Sciences, University of Texas at Austin, Austin, Texas.

3. Belady, L.A. and Lehman, M.M., (1976), "A Model of Large Program Development", IBM Systems Journal, vol. 15, no. 3, pp. 225-251.

4. Belady, L.A. and Merlin, P.M., (1977), "Evolving Parts and Relations - A Model of System Families", IBM Research Reports RC6677.

5. Berild, S. and Nachmens, Sam,(1977), "CS4-A Tool For Database Design by InFOLOGICAL SIMULATION", Proc. 3rd. Int. Conf. on Very Large Data Bases, Tokyo, Japan, pp. 85-94.

6. Blagen, M. and Eswaren, K., (1976), "A Comparison of Four Methods for the Evaluation of Queries in a Relational Data Base System", IBM Research Report RJ1726, IBM Research Center, San Jose, Calif.

7. Brinch-Hansen, P., (1973), "Concurrent Programming Concepts", ACM Computing Surveys, vol. 4, no. 4, pp. 223-245.

8. Bledsoe, W.W., (1971), "Splitting and Reduction Heuristics in Automatic Theorem Proving", A.I. Jour., 2, pp. 55-71.

9. Bledsoe, W.W., (1975), "Non-Resolution Theorem Proving", Automatic Theorem Proving Project Report #29, Department of Mathematics, University of Texas at Austin, Austin, Texas.

10. Bledsoe, W.W., "A Maximal Method for Set Variables in Automatic Theorem Proving", University of Texas Math Department Memo ATP-33A, July, 1977. To be presented at IJCAI-77, MIT, Aug., 1977.

11. Bledsoe, W.W., Boyer, Robert S., and Henneman, William H., (1972), "Computer Proofs of Limits Theorems", A.I. Jour., 3, pp. 27-60.

12. Bledsoe, W.W. and Bruell, P., "A Man-Machine Theorem-Proving System", A.I. Jour., 5, pp. 51-72.

13. Bledsoe, W.W. and Tyson, Mabry, (1975), "The UT Interactive Theorem Prover", University of Texas at Austin, Math Department Memo ATP-17.

14. Bledsoe, W.W. and Tyson, Mabry, "Typing and Proof by Cases in Program Verification", Machine Intelligence 8, Donald Michie and E.W. Elcock (eds.), Ellis Horwood Limited, Chichester, pp. 30-51.

15. Carlson, W., (1976), "Software Research in the Department of Defense", Proc. 2nd Int. Conf. Soft. Eng., San Francisco, pp. 379-382.

16. Chester, D. and Simmons, R.F., (1977), "Influences in Quantified Semantic Networks", Proc. 4th Int. Conf. on Artificial Intelligence, Boston.

17. Chester, D. and Yeh, R.T., (1977), "Software Development by Module Evaluation", Proc. Int. Conf. on Software and Applications, Chicago.

18. Codd, E.G., (1970), "A Relational Model of Data Fpr Large Shared Data Banks", CACM, vol. 13, no. 6, pp. 377-387.

19. Dahl, O-J., Dijkstra, E.W. and Hoare, C.A.R., (1972), Structured Programming, Academic Press, New York, N.Y.

20. Davis, R., Buchanan, B. and Shortliffe, E., (1977), "Production Rules as a Representation for a Knowledge-Based Consultation Program", Artificial Intelligence 8, pp. 15-45.

21. DeRemer, F. and Kron, H.H., (1975), "Programming-in-the-Large Versus Programming-in-the-Small", IEEE Trans. Soft. Eng., vol. SE-2, no. 2, pp. 87-96.

22. Dijkstra, E.W., (1968), "Cooperating Sequential Processes", Programming Languages, F. Gennys (ed.), Academic Press, New York, N.Y.

23. Dijkstra, E.W., (1971), "Hierarchical Ordering of Sequential Processes", Acta Informatica, vol. 1, no. 2, pp. 115-138.

24. Dolotta, T.A. and Mashey, J.R., (1976), "An Introduction to the Programmer's Workbench", Proc. 2nd Int. Conf. on Soft. Eng., San Francisco, pp. 164-168.

25. Donovan, J., (1976), "Data Base System Approach to Management Decision Support", ACM TODS.

26. Donovan, J.J. and Madnick, S.E., (1977), "Institutional and AD HOC DSS and Their Effective Use", ACM SIG DG.

27. Goodenough, J.B., (1975), "Exception Handling: Issues and a Proposed Notation", CACM, vol. 18, no. 12, pp. 683-696.

28. Hoare, C.A.R., (1970), "An Axiomatic Approach to Computer Programming", CACM, vol. 12, no. 5, pp. 76-80,83.

29. Hoare, C.A.R., (1974), "Monitors: An Operating System Structuring Concept", CACM, vol. 17, no. 10, pp. 549-557.

30. Lipovski, J.G., and Su, S.Y.W., On Non-Numeric Architecture, Dept. of Electrical Engineering, University of Florida

31. Madnick, S.E. and Alsop, J.W., (1969), "A Modular Approach to File System Design", Proc. AFIPS, vol. 34, pp. 1-12.

32. Misuri, G., (1976), "Survey of Existing Programming Aids", ACM SIGPLAN Notices, pp. 38-41.

33. Moriconi, Mark, "An Interactive System for Incremental Program Design and Verification".

34. Parnas, D.L., (1972), "A Technique for Software Module Specification with Examples", CACM, vol. 15, no. 5, pp. 330-336.

35. Parnas, D.L., (1976a), "On the Criteria to Be Used in Decomposing Systems Into Modules", CACM, vol. 15, no. 12, pp. 1053-1058.

36. Parnas, D.L., (1976b), "On a Buzzword: Hierarchical Structures", IFIP Proc.

37. Randolph, J.A., (1972), "A Production Implementation of an Associative Array Processor: STARAN." Proc. AFIPS 1972 Fall Int. Computer Conf., vol. 41, pt. 1, pp. 229-241.

38. Robinson, L. and Levitt, K.N., (1977), "Proof Techniques for Hierarchically Structured Programs", to appear - Current Trends in Programming Methodology, Vol. 2, R.T. Yeh (ed.), Prentice-Hall, Englewood Cliffs, N.J.

39. Sandewall, E., (1971), "Formal Methods in the Design of Question-Answering Systems", Artificial Intelligence, vol. 2, no. 2.

40. Senko, M.W., (1976), "DIAM II and Levels of Abstraction", Proc. Conf. on DATA: Abstraction, Definition and Structure, pp. 121-140.

41. Simon, H.A., (1973), "The Structure of Ill Structured Programs", Artificial Intelligence 4, pp. 181-201.

42. Simmons, R.F., (1973), "Semantic Networks: Their Computation and Use for Understanding English Sentences", Computer Models of Thought and Languages, (Schank & Colby, eds.), pp. 63-113.

43. Smith, J. and Chang, P., (1976), Optimizing the Performance of a Relational Algebra Interface", CACM.

44. Smith, J. and Smith, D., (1973), "Data Abstraction: Aggregation and Generalization", ACM TODS

45. Sprague, R. and Watson, H., (1975), "MIS Concepts", Journal of Systems Management.

46. Sungren, B., (1976), &Data Base Theory/& , Patrocelle-Charter.

47. Sussman, G., Winograd, T., and Charniak, E., (1970), "Micro-Planner Reference Manual", AI. Memo 203, Artificial Intelligence Laboratory, MIT.

48. Weber, H., (1976), "The D-graph Model of Large Shared Data Bases: A Representation of Integrity Constraints and Views of Abstract Data Types", IBM TC (San Jose).

49. Wegbreit, B., (1976), "Verifying Program Performance", JACM, vol. 23, no. 4, pp. 691-699.

50. Winston, P., (1977), Artificial Intelligence , Addison-Wesley, New York.

51. Yeh, R.T. (ed.), (1977), Current Trends in Programming Methodology: Vol. 1 Software Specification and Design, Prentice-Hall, Englewood Cliffs, N.J.

52. Yeh, R.T. (ed.), (1977), Current Trends in Programming Methodology: Vol. 2 Program Validation, Prentice-Hall, Englewood Cliffs, N.J.

53. Yeh, R.T., (1977), "Program Verification by Predicate Transformation", Current Trends in Programming Methodology: Vol. 2, Program Validation, Yeh (ed.), pp. 228-247, Prentice-Hall, Englewood Cliffs, N.J.

54. Yeh, R.T. and Baker, J.,(1977), "Towards a Design Methodology of DBMS: A Software Engineering Approach", Proc. 3rd Int. Conf. on Very Large Data Bases, Tokyo, Japan

APPENDIX 1

SEMANTIC REPRESENTATIONS FOR AN INTEGRATED DATA SYSTEM

R. F. Simmons

I. Languages

Semantic Networks evolved primarily to represent the deep logical semantics of natural language discourse. Consequently communication in English is the *raison d'etre* of the system and we have previously described interpreters and grammars that we have developed to translate sentences and queries in English into the networks and from network structures back into English (see Simmons, 1977).

The language of semantic relations and predicates evolved as a linear expression of the networks, and statements in it may be used as arguments of the functions, ASSERT, QUERY and DELETE to communicate directly with the system. This language is an alternate notation for predicate logic and it is fu-ly quantified and includes logical functions -- AND, OR NOT, and IMPLIES -- and can include general functions (see Simmons and Chester, 1977).

The user may prefer for some purposes to use a simpler language of tuples. A predicate logic in this form was introduced to computation by F. Black (1964) and has been further developed by Kowalski (1974). A simple assertion such as: "the pencil is in the desk", is represented in Kowalski's notation as: (IN PENCIL DESK)←. The transitivity of "in" is expressed as: (IN X Z)←(IN X Y) (IN Y Z), i.e. if X is in Y and Y is in Z, then X is in Z, where X, Y, and Z are free variables. The tuples to the left of the arrow are consequents, to the right are antecedents. A query has the form, ←(IN PENCIL Y). Both Black and Kowalski show that this is a complete logical system. This language translates easily into semantic networks. An example will illustrate:

```
 (IN PENCIL DESK)←  ⟹(ASSERT(IN R1 PENCIL R2 DESK))
 (IN X Z)←(IN X Y) (IN Y Z)⟹
 (ASSERT(IN R1 X R2 Z ANTE ((IN R1 X R2 Y)(IN R1 Y R2 Z))))
←(IN PENCIL X) ⟹ (QUERY (IN R1 PENCIL R2 X))
```

The logic of answering questions in semantic networks is similar to the logic of Kowalski's system which he has shown is very powerful for solving problems and even for evaluating programs.

Of primary interest to data management, special functions are introduced for asserting, querying and deleting tables. ASSERTAB as exemplified in a following section, takes a tablename, a list of headings, and a list of tuples as arguments:

```
(ASSERTAB TABLE COURSE*TAB
          FORM (COURSE STUDENTS)
          DATA ((CS343 23)(CS375 37)...(CS399 9)) )
```

The result of ASSERTAB is to construct a network representing the table. DELETAB is provided to delete all or parts of a table. Although the ordinary ASSERT, DELETE and QUERY functions work on tables, a special quantified function is provided in the following form:

```
(FOR QFY CLASS PARTITION OPERATION),
```

an example call might be:

```
(FOR SOME STUDENTS COURSE*TAB (IF (GR STUDENTS 5)
                                  (PRINT COURSE STUDENTS)))
```

The operation can be any program in a language decided to be suitable for the user. It is of particular importance that the operations include the capability of constructing new tables, e.g.

```
(FOR SOME NAMES EMPLOYEE*TAB (IF (GR SALARY 20000)
                                 (ASSERTAB NAME TEMP*TAB
                                    FORM (NAME SALARY DEPT)
                                    DATA (NAMES SALARY DEPT))))
```

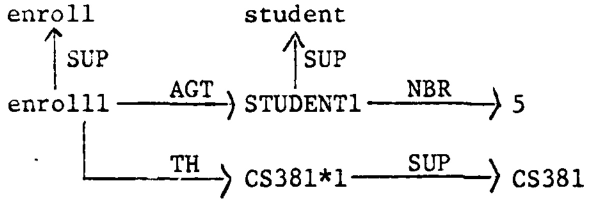
This will construct the new table, TEMP*TAB selecting NAMES, SALARY and DEPT from the old one when entries have a salary greater than 20000.

Additional functions ASSERTEXT, DELETEXT and KEY* are provided for introducing text to the semantic networks and for retrieving bestmatching strings from it.

II. Basic Structures

Semantic networks can be viewed as a representation that reduces all data to sets of binary relations. A semantic network can be drawn as a directed graph in which each arc represents a relation term and the two nodes* which it connects are the arguments. Since a node can participate in many binary relations, a node, its arcs and the nodes to which they directly connect it comprise a set of binary relations. A simple example follows:

"Five students enrolled in CS381."



The subscripted terms, e.g. enroll1, student1, etc. can be seen as a special encoding for instances of the concept to which they are in a SUP relation. The arcs are relation names which in this example are derived from the names, Agent, Theme, Number and Superclass. Each arc is understood to have an inverse as follows:

SUPerclass--INSTance
AGT--AGT*
TH-- TH*
NBR--NBR*

*In fact, in our implementations, an arc connects a node to a set of nodes, e.g. stately and graceful ~~coconut~~ palm is represented as (PALM $\xrightarrow{MOD}$ (STATELY GRACEFUL)). This proves most economical for representing tables and texts.

The unsubscripted concept is also in relation to other concepts; e.g. ENROLL SUP JOIN, STUDENT SUP PERSON, CS381 SUP COURSE, etc., thus classifying the vocabulary of the system.

The above graph can be represented also as triples which is an attractive form at the machine implementation level.

```
(ENROLL1 SUP ENROLL)                      (CS381* SUP CS381)
(ENROLL1 AGT STUDENT1)                    (CS381*1 TH* ENROLL)
(ENROLL1 TH CS381*1)                      (5 NBR* STUDENT1)
(STUDENT1 SUP STUDENT)
(STUDENT1 NBR 5)
(STUDENT1 AGT* ENROLL1)
```

The triples facilitate implementation in that they reduce any form of data to a fixed dimension array. Their use of indirect reference is advantageous for defining recursive and iterative inference procedures but results in significant difficulties in terms of the number of auxiliary storage accesses that they may imply.

Most of our work locally has been accomplished in a LISP 1.5 environment in which the semantic networks are conveniently represented as property list structures. A property list can be viewed as a node associat[illegible] set of pairs. The first member of a pair is the name of an arc or relation and the second is the name of the node that it connects. For the above graph or set of triples a property list structure appears as follows:

```
(ENROLL1 SUP ENROLL AGT STUDENT1 TH CS381*1)
(STUDENT1 SUP STUDENT AGT* ENROLL1 NBR 5)
(CS381*1 SUP CS381 TH* ENROLL1)
(ENROLL SUP JOIN INSTANS ENROLL1)
```

The LISP environment is additionally helpful in providing a transformation from linear machine organization of memory to a logically organized memory in which the names of atoms and lists point to their addresses. This list organization represents a difficult problem when auxiliary storage is required as in a large data management system.

III. Tuples and Tables

In ordinary conventions of mathematical notation a statement such as our example, "Five students enrolled in CS381", might be represented as the following data 2-tuple:

(CS381 5).

The author of such a tuple would remember that he is talking about a course and its enrollment of students. His understanding of the tuple can be represented by a corresponding form 2-tuple, (COURSE STUDENTS). These two forms may be combined into a semantic network representation: (COURSE 381 STUDENTS 5) thus making explicit the form that is required to understand the original 2-tuple.

If he wishes to organize or present data for several courses, he may construct a table such as the following:

TABLENAME	COURSE*TAB	
HEADINGS	COURSE	STUDENTS
DATA	CS381	5
	CS382	7

Alternatively, he can present the same information in semantic network form.

(COURSE*TAB INSTANS (COURSE*TAB1, COURSE*TAB2))

(COURSE*TAB1 SUP COURSE*TAB COURSE CS381 STUDENTS 5)

(COURSE*TAB2 SUP COURSE*TAB COURSE CS382 STUDENTS 7)

The tablename, COURSE*TAB, is a partitioning of the semantic network that organizes assertions about courses and students into a subnetwork that is easily accessible by the name, COURSE*TAB. Additional specifications of the kind of data in the partition can be associated with the tablename to facilitate retrieval or to insure accuracy of its entries as in the following example:

```
(COURSE*TAB ARCS(COURSE STUDENTS)  ACCESS UNRESTRICTED
            ENTRIES 2
            COURSE NAME
            STUDENTS(NBR LS 500))
```

So the node, COURSE*TAB specifies that its headings are COURSE and STUDENT, its access is unrestricted, it has 2 entries, COURSE is a NAME and STUDENT a NUMBER less than 500. The INSTANS arc as seen earlier indexes the entries. Various ordering arcs can be provided to subset large tables into alphabetic or numerical categories.

Interpreters for two forms of query can be provided. The first is the standard form called a Case Relation query:

```
(QUERY (Y COURSE CS381 STUDENTS X))
```

where the argument of ASK is a partial specification of a case relation and X is a variable that matches the value associated with the matching case relation. The value returned by ASK is (COURSE*TAB1 COURSE CS381 STUDENTS 5). The partial specification succeeds by finding the instances of CS381 and discovering if any have the arc, STUDENTS. If we knew that courses were partitioned in COURSE*TAB, we might have asked:

```
(QUERY (COURSE*TAB COURSE CS381 STUDENTS X))
```

and retrieved the same answer by examining the instances of COURSE*TAB. The value returned in this case would be (COURSE*TAB1 COURSE CS381 STUDENTS 5).

The second general type of query is a quantified form, similar to formal data management languages;

```
(FOR QFY CLASS PARTITION OPERATION)
```

FOR establishes an iteration where QFY specifies the number of instances desired, e.g. 1,17, some, all; CLASS specifies an arc name or a function on its values, PARTITION specifies a partition or table if any and OPERATION is a set of procedures to be accomplished on the set that has been specified. We might wish to ask COURSE*TAB for all courses with more than 5 students;

```
(FOR SOME STUDENTS COURSE*TAB (IF STUDENTS GR 5
                               (PRINT COURSE STUDENTS)) )
```

The OPERATION argument accepts a program of procedures in a data language convenient for the user.

The interpreter must also accept Assertions and Deletions. A set of predicates may be asserted to the network with the following command:

```
(ASSERT ((COURSE*TAB COURSE 375 STUDENTS 7)
         (COURSE*TAB COURSE 343 STUDENTS 23)) )
```

The result of the ASSERT is to create INST and SUP arcs from COURSE*TAB to COURSE*TAB3 and COURSE*TAB4, and to create the arcs COURSE, COURSE*, STUDENTS and STUDENTS* between the data items. Convenient brief forms such as ASSERTAB can also be provided, e.g. (ASSERTAB TABLE COURSE*TAB

```
                                          FORM (COURSE STUDENTS)
                                          DATA ((CS375 37)(343 23)) )
```

DELETE can accept the same forms as ASSERT and delete them from the network.

In some applications where many small tables characterize the data, it may prove desirable (in order to save memory) to avoid indexing the values of data in the tables. In this event the semantic network form of the table is exactly the same as the argument form of ASSERTAB. For example the unindexed form for COURSE*TAB appears as follows:

```
(COURSE*TAB  FORM (COURSE STUDENTS)
             DATA ((CS380 5)(CS381 7)) )
```

Since this is a well-formed semantic network, it may be directly Asserted. A variation of the quantified FOR statement, FOR*, can be provided to query unindexed tables.

IV. Text

In its printed form a text is an ordered set of word symbols. For retrieval purposes it is best represented as an index of Word-types and a list of occurrances of Tokens. Consider the sentences, "Big fish eat little fish. Little fish eat littler fish." The representation as types and tokens is shown below.

TYPES	INDEX	TOKENS
1 Big	(1)	(1 2 3 4 2 4 2 3 5 2)
2 Fish	(2,5,7,10)	
3 Eat	(3,8)	
4 Little	(4,6)	
5 Littler	(9)	

The tokens are references to entries in the type list which for each word-type shows a list of its occurrences as sequence numbers referring to the *string of tokens.*

For retrieval from such a structure, any list of words may be taken as a request and the Token substrings containing hits can be returned as answers ordered by the number of hits in each substring. This is the general approach to keyword retrieval as used in many kinds of system.

This approach is adapted easily to representation in semantic networks almost literally as shown below:

```
(BIG NBR 1 TEXT1 (1))              (TEXT1 SEQ (1 2 3 4 2 4 2 3 5 2))
(FISH NBR 2 TEXT1 (2 5 7 10))
(EAT NBR 3 TEXT1 (3 8))
(LITTLE NBR 4 TEXT1 (4 6))
(LITTLER NBR 5 TEXT1 (9))
```

If we query with the procedure KEY* to retrieve what it said about "little fish";

(KEY* (LITTLE FISH))

under the requirement of returning sentences as answers, both sentences would be returned. In the process the tokens would be translated back to words. The procedure for retrieval operates wholly on the index to determine an ordering of the sentences in the text, then reconstructs those sentences from the token list.

Many heuristics have been developed as variations on this simple retrieval scheme to improve the ordering of answers. If the text is large its token list can be subcategorized by volume, chapter, paragraph and sentence so that the index numbers each become tuples and the search through the text string is shortened to any extent desirable. For example, if we partition the text by sentences marking each sentence in TEXT1 with parenthesis,

1. BIG $\underrightarrow{\text{TEXT1}}$ (1.1)
2. FISH $\underrightarrow{\text{TEXT1}}$ (1.2, 1.5 2.2 2.5)
3. EAT $\underrightarrow{\text{TEXT1}}$ (1.3, 2.3)
4. LITTLE $\underrightarrow{\text{TEXT1}}$ (1.4, 2.1)
5. LITTLER $\underrightarrow{\text{TEXT1}}$ (2.4)

TEXT1 $\underrightarrow{\text{SEQ}}$ ((1 2 3 4 2)(4 2 3 5 2))

we then use 2-tuples 1X, 1Y as indexing numbers. If we wished to further partition the text to chapters and paragraphs we would use a 4-tuple as an index number:

chapter·paragraph·sentence·sequence·

The network representation for text is designed to minimize storage requirements by representing each text as a vector of tokens where each word-type occurring in the text references the vector locations of its occurrences.

As with tuples and tables, the procedures ASSERTEXT and DELETEXT can be defined.

V. Discussion

A core-limited prototype of the proposed system exists in LISP 1.5 on both the CDC and DEC10 systems. As it stands it can translate English statements and questions into semantic network forms. A translator is provided to enable a user to use Kowalski's form of predicate logic notation. Our experimentation with this system has been primarily oriented toward insuring that the semantic network representation is logically complete and that its proof procedures for answering questions are adequate. Tables can be directly asserted to this system as it is and their contents can be queried.

Procedures for interpreting the quantified FOR statement are not yet developed. Additional procedures are needed to provide for storing and querying unprocessed text.

If a large INTERLISP system were available, with its paging control as a disc memory manager, the prototype could deal successfully with several million words of data. In the local environment it is limited to about 300K words for system and data and is expected to be useful primarily for developing data structures and language interpreters.

REFERENCES

Black, Fischer, A Deductive Question Answering System, in Minsky, M., (ed.) Semantic Information Processing, MIT Press, Boston, 1969.

Kowalski, Robert, "Predicate Logic as Programming Language", IFIP Congress, Stockholm, 1974.

Simmons, Robert F., Rule-Based Computations on English, in Hayes-Roth, R. and Waterman, D., (eds.) Pattern-Directed Inference Systems, Academic Press, N.Y., 1977 (in press). Also, University of Texas, Department of Computer Science, Technical Report, NL31, Austin, 1977.

Simmons, R.F. and Chester, D., Inferences in Quantified Semantic Networks, IJCAI77 (in press), and University of Texas Department of Computer Science, Technical Report NL32, Austin, 1977.

APPENDIX 2

TOWARD A DESIGN METHODOLOGY FOR DBMS:
A SOFTWARE ENGINEERING APPROACH

by

Raymond T. Yeh and Jerry W. Baker

A design methodology for DBMS is presented. The methodology consists of three interacting models: a model for the system structure, a hierarchical performance evaluation model, and a model for design structure documentation, which are developed concurrently through a top-down design process. Thus, using this methodology, the design is evaluated and its consistency checked during each phase of the design process. It is shown that systems designed using this methodology are reasonably independent of their environments, reliable, and can be easily modified. A modest example is used to illustrate the methodology.

TOWARD A DESIGN METHODOLOGY FOR DBMS: A SOFTWARE ENGINEERING APPROACH*

Raymond T. Yeh and Jerry W. Baker

Department of Computer Science
University of Texas
Austin, Texas U.S.A.

A design methodology for DBMS is presented. The methodology consists of three interacting models: a model for the system structure, a hierarchical performance evaluation model, and a model for design structure documentation, which are developed concurrently through a top-down design process. Thus, using this methodology, the design is evaluated and its consistency checked during each phase of the design process. It is shown that systems designed using this methodology are reasonably independent of their environments, reliable, and can be easily modified. A modest example is used to illustrate the methodology.

INTRODUCTION

The environment of a DBMS can be partitioned into three categories of things: machines, data, and applications (or users). Furthermore, they are dynamic and constantly changing. Thus, it seems reasonable to require that the design of a DBMS be such that the resulting system is as independent of its environment as possible so that it can evolve along with its environment.

Although a significant amount of research has been dedicated to specific aspects of data base systems (data models, query languages, performance modeling, etc.), relatively little has been accomplished in the way of integrating these ideas into a design methodology which can be used to systematically construct data base systems for large classes of applications. Part of the reason for the lack of a design methodology for DBMS is, we believe, due to the complexity of its environment. For example, the environment of an operating system only consists of machines (processes) and data (resources). Since the design problems for a DBMS are diverse, we believe that appropriate knowledge from other disciplines, especially in software engineering, can contribute toward a unified design methodology for DBMS.

In this paper we shall describe a design methodology for DBMS. Our basic philosophy is that the design process can be grossly described by three models: a model for the system being designed, a model for system design evaluation, and a model for design structure documentation.

The system structure is modeled by a set of n+1 abstract machines, M_n, M_{n-1},...M_0 connected by a set of n implementation programs, I_n, I_{n-1},..., I_1. Each machine in the hierarchy represents a "view" of the system at a particular level of abstraction and, moreover, constitutes a refinement of the previous (higher) level in the sense that its data abstractions are used to "implement" those of the previous machine.

In order to minimize the system redesign effort, we believe that design must be evaluated during the design process. To do so, we propose a hierarchical performance evaluation model which is to be developed top-down alongside the development of system structure. Its main function is to provide feedback to the designer as to which alternatives at the current level can satisfy the performance constraints. However, even with performance evaluation provided, backtracking is inevitable. It would be very desirable to know during backtracking why some of the previous alternatives were not chosen. Thus, a language for documenting the design structure is desirable and will be discussed in a later section.

In summary, we will introduce a design methodology for DBMS which allows constant evaluation of the system as the design unfolds.

DESIGN OF HIERARCHICALLY STRUCTURED DBMS

In this section we present a methodology in which we borrow heavily from software engineering. This methodology provides for the systematic design, specification, and implementation of a reliable DBMS such that integrity and security constraints can be automatically included and that correctness proofs can be established for the resulting system. Using this methodology, a DBMS can be described and structured in a hierarchical fashion. The design is top-down and the resulting system will consist of multiple levels - each level being described by a self contained specification.

Abstraction, Stepwise Refinement, and DBMS Design

One of the most powerful tools in software development is abstraction. The use of abstraction allows a designer to initially express his solution to a problem in a very general term and

* This research is supported by AFSOR under contract AFOSR-77-3409 and by ARPA under contract N00039-77-C-0254 and by an IBM Pre-Doctoral Fellowship to the second author.

**This paper was invited for the VLDB held in Tokyo, Japan, 1977.

with very little regard for the details of implementation. This initial solution may be refined in a step by step manner by gradually introducing more and more details of implementation. The process continues until the solution is finally expressed within the framework of some appropriate "target" language. This combination of abstraction and stepwise refinement enables the designer to overcome the problem of complexity inherent in the construction of systems by allowing him to concentrate on the relevant aspects of his design, at any given time, without worrying about other details. An important result of this approach is the development of a hierarchically structured system (function abstraction) such that each level consists of a number of modules (data abstractions). Thus, the system is both horizontally and vertically modular.

The notion of abstraction is also important from the standpoint of protection. Through data abstraction a designer may limit the access to a data object through a specified set of well-defined operations. Likewise, by hiding the implementation of a data abstraction from its users the designer protects them from any changes which might occur in that implementation.

We envision the design of a DBMS as a stepwise refinement process of functional abstraction which begins with the construction of a "top-level" abstract machine, M_n, satisfying the functional requirements of some high level requirements specification. This machine consists of a set data abstractions represented by formal module specifications. Each module specification is self-contained in the sense that it specifies the complete set of operations which define the nature of the data abstraction. Collectively, these data abstractions define the data model which is visible to the user of the machine.

In the next step of the process, another abstract machine, M_{n-1}, representing a "refinement" of M_n is designed. Its data abstractions are chosen in such a way that they can "implement" those of M_n. Basically, this implementation consists of a set of abstract programs each of which defines an operation of M_n in terms of accesses to functions of machine M_{n-1}. A verification process can then be used to ensure that the implementation is consistent with the specification of both machines (the implementation and verification processes are described in more detail in a later section).

This stepwise process of machine specification, implementation, and verification proceeds until, at some point, the data abstractions of the lowest level machine can be easily implemented on a specified "target" machine, which may be the data abstractions of some programming language, a low-level file management system, or the operations of some appropriate hardware configuration. This design process results in a DBMS structure consisting of a hierarchy of abstract machines, or levels, $M_n, M_{n-1}, \ldots, M_0$ connected by a set of n programs $I_n, I_{n-1}, \ldots, I_1$. Each machine M_i in the hierarchy represents a complete "view" of the DBMS at a particular level of abstraction while the corresponding program $I_i (1 \le i \le n)$ represents the implementation of that view upon the next level machine M_{i-1}.

Module Specification

The method of module specification used in this hierarchical approach is based upon the work of Parnas [1972] and Robinson and Levitt [1977] with slight modifications (see Baker & Yeh,[1977]). The specification of each module defines two types of access functions, SV and ST. SV functions return values and the set of all SV functions of a machine is said to characterize the machine's abstract "state". ST functions, on the other hand, produce a state change in a machine. The state change which a function produces in a machine is defined in an EFFECTS section of the module specification. Each "effect" is an assertion defining the change in the value of an SV function of the machine when the ST function is successfully invoked. The only observable change in the state of the machine produced by the execution of the ST function is that defined in the EFFECTS section.

The specification of each module also includes a set of exception conditions each of which defines a condition about which the invoker of an operation must be notified. An exception condition definition consists of a name with a formal parameter list and a predicate using the SV functions of the module and the formal parameters. The specification of each function in the module contains a list of exception conditions with the parameters of the function call appropriately substituted for the formal parameters of the exception condition list. If any predicate defining an exception condition in the list is true when the function is invoked, then a specified action is taken by the system. If the exception condition is "fatal" and the function is of type ST, then the effects specified in the function definition will not be observed and the user is appropriately notified. If the function is of type SV, then the value(s) returned is (are) undefined. For a "non-fatal" exception condition a simple warning message is issued.

Implementation and Verification

The implementation between two adjacent machines M_i and M_{i-1} is the process by which the data abstractions of M_i are defined in terms of the data abstractions of M_{i-1}. More formally, if $F_i = \{f_i^1, f_i^2, \ldots, f_i^k\}$ is the set of module functions for M_i then the implementation of M_i by M_{i-1} is defined by

$$I_i = \{\emptyset_i, p_i^1, p_i^2, \ldots, p_i^k\}$$

where $\emptyset_i$ is a mapping from the states of M_{i-1} to the states of M_i and p_i^j is an abstract program which implements the function f_i^j on machine M_{i-1}. The mapping function $\emptyset_i$ has the effect of "binding" each state of M_i to a state or set of states of M_{i-1}. That is, if S_i and S_{i-1} are the state sets

Yeh - 2

of M_i and M_{i-1}, respectively, then the mapping $\emptyset_i$ is defined such that for every state $s_i \in S_i$ we have $s_i = \emptyset_i (s_{i-1})$ for some state s_{i-1} of S_{i-1}. The mapping function is actually constructed by expressing each SV function of M_i as an expression containing the SV functions of M_{i-1}. Each such expression is referred to as a partial mapping function and the set of all partial mapping functions for M_i comprises the mapping $\emptyset_i$.

The purpose of abstract program p_i^j is to express the function f_i^j of M_i in terms of the functions of M_{i-1}. Thus, the program is constructed using well-defined control constructs and the function set F_{i-1}. This implementation process must be consistent with the formal specifications of M_i and M_{i-1}. That is, the following commutative diagram must be satisfied:

$$\begin{tikzcd}
s_i \arrow[r, "f_i^j"] & s'_i \\
s_{i-1} \arrow[u, "\emptyset_i"] \arrow[r, "p_i^j"'] & s'_{i-1} \arrow[u, "\emptyset_i"']
\end{tikzcd}$$

Fig. 1

where s_i and s'_i are states of M_i and s_{i-1} and s'_{i-1} are states of M_{i-1}.

The verification of the implementation I_i requires a formal proof that the commutative diagram of Fig. 1 is satisfied for every abstract program p_i^j. This verification process is basically a standard inductive assertion proof (Hoare,[1970]) on p_i^j and we, therefore, only give a brief description of it. However, the reader is referred to Robinson and Levitt [1977] which contains a detailed discussion of the hierarchical proof techniques used in the methodology.

In general, the precondition for each abstract program p_i^j is true because the program contains its own mechanisms for exception handling. The output assertions for p_i^j are derived from the assertions in the EFFECTS section of the specification for function f_i^j and from the mapping function $\emptyset_i$. Each output assertion is obtained by taking an EFFECTS assertion and replacing each reference to an SV function by the instantiation of the appropriate partial mapping function of $\emptyset_i$.

Inductive assertions for p_i^j can be taken directly from the EFFECTS sections of the ST operations used to construct the program. Verification conditions can then be derived and used to establish the validity of these assertions. The verification of the output assertions then follows.

Design and Specification - An Example

The concepts discussed in the previous section can perhaps be best understood by looking at a DBMS designed using this hierarchical approach. A partial outline of such a system is shown in Tables 1 and 2. Table 1 contains a brief description of the nature of each system module, while Table 2 outlines the basic properties of the different level machines.

Table 1. A description of the system modules. Only a partial list is given for each level.

Level 5

UNIV - Defines operations for recording and accessing information about university departments and professors.

Level 4

REL - Defines the concept of a relation through relational algebraic operations.

INT - Specifies operations for creating and enforcing "integrity assertions" which specify allowable data values for relations.

AUTH - Defines operations for creating and enforcing "authorizations" which specify allowable interactions for users.

Level 3

RT - Defines operations for creating, updating, and accessing logical "record tables".

RDIR - Represents a directory of existing record tables.

FNT - Defines tables containing information about field values for each existing record table.

TDS - Specifies operations for creating and accessing sets of record identifiers. Used to implement the concept of a cursor (Astrahan [1976]).

IMAGE - Represents logical reorderings of records (Astrahan [1976]).

IMCAT - Defines a catalog of existing images.

SEL - Specifies operations for creating, maintaining, and accessing partial indexes to record tables.

Table 1 (Cont'd.)

SLCAT - Represents a catalog of existing partial indexes.

LNK - Defines operations for creating, maintaining and using logical associations between records of different record tables.

LCAT - Represents a catalog of associations.

<u>Level 2</u>

BTR - Defines the concept of a B-tree. Used to implement the IMAGE module of Level 3.

RBLK - Represents fixed-length blocks of records. Used to implement the FT and LNK modules of Level 3.

Level 2 (Cont'd.)

RBDX - Specifies operations for creating and using directories to RBLK structures.

RIDX - Represents fixed-length blocks of record pointers. Used to implement the TDS, SEL, and LNK modules of Level 3.

<u>Level 1</u>

VP - Defines the concept of a virtual page space.

<u>Level 0</u>

Machine hardware

Table 2. A brief description of six levels in a hierarchically structured DBMS. The actual system contains eight levels. However, for purposes of presentation, several levels were combined.

Level	Visible Concepts	Operations	Concepts Hidden By Level
5	entities (university departments, professors, etc.), and their attributes	operations corresponding to real-world transitions ("hire", "terminate", etc) and queries ("get_salary", "get_age", etc.)	logical structure of data
4	relations, tuples, cursors, authorization and integrity assertions	algebraic relational operations, creation and enforcement of authorization and integrity assertions, cursor creation and sequencing operations	access paths, record table structure, record identifiers
3	record tables, records, images, partial indexes, record table associations, record identifier sets	creation, access, and maintenance of record tables, access paths, and record identifier sets	record block structure, implementation of access paths
2	fixed-length record and pointer blocks, B-trees, links between record blocks	record block access, B-tree operations	bit representation of information, distribution of record block and B-tree nodes on virtual memory pages
1	virtual page space	bit and byte extraction and encoding	distribution of pages in memory devices
0	primary and secondary memory devices	paging operations	

The top-level machine, M_5, represents an application view of the system. The UNIV module provides operations for recording and accessing information about university departments and professors. Specifically, the information represented includes the following:

1. the name, social security number, age salary, rank, and department of all professors employed by the university,and
2. the chairman, number of professors, and average salary for each university department.

The ST operations of the module are semantically meaningful - each corresponding to a real world transition. They include "hire", "terminate", "promote", "raise_salary", and "change_chairman". The SV functions of the module include "get_salary" "get_chairman", and "get_rank". At this level of interaction a user is well-protected from organizational changes in the database system because no physical (access paths, storage structures, etc.) or logical (relations, etc.) structures are visible. Rather, the user is aware only of very abstract relationships and transitions which may occur in his application.

The operations of the UNIV module are implemented on the next level machine, M_4, which represents a relational algebra view of the database system. The REL module, for example, defines the concept of a relation in terms of relational algebraic operations while the RDIR module represents a relation directory which contains information about all existing relations. The two other modules shown, INT and AUTH, relate to the concepts of integrity and authorization and are described in more detail in a later section. We note that at this level of interaction the concept of an access path is completely hidden from the user. That is, the operations at this level provide no mechanisms for defining, deleting, or using any type of access path.

At the level of machine M_3 the DBMS represents a somewhat different view. A user of this level can create and manipulate logical record tables (RT module) and a directory (RDIR module) to record information about existing record tables. Also, several modules - IMAGE, LINK, and SELECTOR - make it possible to create fast access paths to records of existing record tables. The implementation of M_4 by M_3, of course, consists of programs which implement the module functions of M_4 in terms of the module functions of M_3. Thus, for example, the relational algebraic operations of the REL module are implemented in terms of record table operations and calls to the appropriate functions of the fast access path modules.

As the DBMS is viewed at lower levels the data abstractions become more "physically" oriented until the level of the machine hardware is reached. Missing is the sharp transition from logical to physical representation found in many systems. Rather, there is a gradual progression from a very abstract view to machine hardware occurring in a sequence of discrete steps.

Levels of Abstraction and DBMS Design

We observe that the notion of levels of abstraction translates to a natural interpretation within the context of database systems. That is, it can be expected that any integrated data base will have a wide variety of users whose views of the system and access requirements will be quite different. Through the hierarchical design approach different levels of design may be constructed to accommodate this variety of views and access requirements (Fig. 2).

The design of the system shown in Tables 1 and 2 illustrates how different users may be accommodated through hierarchical design. At the highest level of abstraction, for example, is the casual user who is concerned primarily with accessing the information relevant to his application with as little trouble as possible. He is unconcerned about efficiency and organizational properties of the data and, therefore, is provided with a set of high-level, semantically meaningful operations which hide such details.

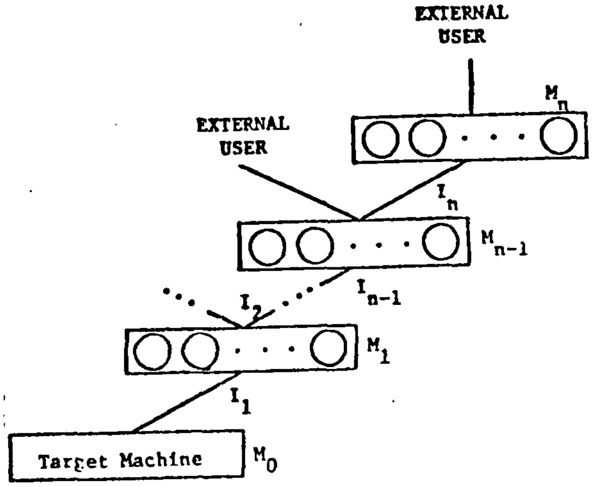


Fig. 2. A hierarchy of formally specified machines showing modularity. Levels may be constructed to accommodate the different views required by various users.

The privileged programmer, while still being concerned with the information relevant to his application, is also concerned with the efficiency of his interactions with the system. Therefore, he may be willing to sacrifice a certain amount of data independence for increased efficiency. A privileged programmer may therefore require access to levels 3 or 4.

The application programmer's job is to create interfaces for new applications when they arise. This may require a modification to the top-level machine or possibly the specification of new machines to be implemented on existing levels. The application programmer would most likely require interaction with levels 3, 4 and 5.

The access path programmer has the task of creating fast access paths for the system. Like the application programmer he is not interested in the information content of the system, but rather in defining access paths which enhance the efficiency of other users. The access path programmer would thus interact at level 3.

Finally, the storage structure development programmer interacts with the system at level 2. His task is to ensure that logical access paths are implemented as efficiently as possible.

Our mention of the different levels of users is neither intended to be exhaustive or even the best possible. We merely wish to emphasize that the hierarchical design approach can be used to construct levels which correspond directly to the views of the system desired by different types of users and that this is a useful way of partitioning the different interfaces required. We do not mean to imply, however, that every level in a hierarchically structured system will correspond to a type of user. Different levels may in fact be introduced during the design process merely as an aid to the designer himself.

Design of Authorization and Integrity Mechanisms

Protecting a data base from semantic errors and from use by unauthorized persons is, of course, an important function of any DBMS. The development of integrity and authorization subsystems, then, is an integral part of the DBMS design process. Through the use of exception conditions the hierarchical design approach provides a reliable mechanism for handling such problems. Exception conditions provide a means by which the designer can specify that a function cannot be successfully invoked when certain integrity or authorization conditions are not satisfied.

Consider, for example, the "hire" function of the UNIV module of level 5. This function requires, among other things, the specification of values for the parameters rank and salary. The function has a fatal exception condition

BAD_SALARY(salary,rank)

which is defined as

```
BAD_SALARY(s,r):
    case r of
       "assistant professor": s>18000;
       "associate professor": s>24000;
       "professor": s>40000;
    end.
```

Therefore, if the "hire" function were invoked with salary=19500 and rank="assistant professor" then the effects of the function (as stated in the module specifications) would not be observed. That is, the function would have no effect on the state of machine M_5.

The approach is similar at lower levels of the design. The design of the INT module, for example, provides for operations which enable the creation of "integrity assertions" which define the semantic correctness of existing relations. Moreover, the module contains certain SV functions which can be used to determine if a particular update operation would violate defined integrity assertions. This module combined with the appropriate exception conditions in the REL module can be used to ensure that any update function which would violate defined integrity assertions cannot be executed. For example, the function

insert_tuple(r,R)

of the REL module has the effect of inserting tuple r into relation R. One fatal exception condition for this function is

BAD_TVAL(r,R)

which is defined as

```
BAD_TVAL(t,T):  ∃i(1≤i≤ncomp(t))
                    [check_val(domain(i,T),T,t(i))=
                                                 false]
```

where ncomp(t) returns the number of components of tuple t, t(i) is the ith component of tuple t, and domain(i,T) returns the name of the ith domain of T. Also, check_val(d,S,v) is a boolean function of INT which returns true if v is an acceptable value for domain d of relation S and false otherwise. This specification indicates that the operation insert_tuple(r,R) cannot be executed if the tuple r contains data values which are non-allowed by any defined integrity assertions. Moreover, the verification process ensures that the abstract program implementing the insert_tuple function satisfies this specification.

Protecting data objects from unauthorized use can be handled in a similar manner. For example, the AUTH module enables the creation of "authorizations" which define the allowed accessed to level 4 data objects. Also, an SV function can be used to check if a user has a certain access to a data object. Each module function of level 4 contains an exception condition which prevents unauthorized access from occurring. For example, the insert_tuple(r,R) function has the exception condition

NO_AUTH(uid,R,'INSERT')

which is defined as

NO_AUTH(id,S,op): check_auth(id,S,op)=false

where uid is the identification number of the user invoking the function and check_auth(id,S,op) is a boolean function of AUTH which returns true if user "id" has "op" access to relation S. Again, the verification process can be used to ensure that the implementation of insert_tuple satisfies this specification.

An Assessment of the Methodology

The methodology presented in this section is but a small step in the development of a design theory for DBMS. This approach has several advantages over ad hoc methods currently used. We summarize a few of the most important ones here.

1. Reliability of Design

   The multi-level design process enables the designer to concentrate on the relevant aspects of each level without worrying about implementation details. Also, because the implementation occurs in small steps the probability of design errors is reduced.

2. Machine, Application, and Data Independence

   The horizontal and vertical modularity provided by this approach to DBMS design enhances machine, application, and data independence of a system. Machine independence is enhanced because the information hiding properties of each level limit the effects of modifications to hardware architecture. Vertical modularity provides a degree of application independence because the addition and deletion of applications can be accommodated through changes in columns (modules and their vertical refinements) but not the whole system.

   It should also be clear that each level of a hierarchically structured system provides a measure of data independence. That is, each level tends to hide from its users the organizational properties of lower levels. Providing a hierarchical structure can thus be useful in protecting the system itself from the effects of internal modifications.

3. Formal Consistency Proofs

   The hierarchical nature of the implementation reduces the verification of the entire system into a sequence of the hierarchical proofs designed to insure the consistency of the specification and implementation of adjacent levels. Because the verification proceeds in sequence with the design process, implementation errors can be detected at the same level in which they are introduced.

4. Localized Effects of Modification

   A database system is a dynamic entity which requires constant modification and maintenance. Even after the system is installed and operating, frequent modifications may be required to correct programming errors or to increase system efficiency. Likewise, design changes may be necessary to adapt the system to changing user requirements or to a new operating environment. If the system is poorly designed then the impact of such modifications may be so great that maintenance is a significant part of the overall development cost. At each level of a hierarchically structured, modular system, an abstract concept is realized by a formally specified module. Because the module structures hide all aspects of the implementation, modifying a machine design or implementation requires only localized changes in the system.

5. Understandability

   The hierarchical design process allows the designer to understand the operation of the system at each level of abstraction before proceeding with the implementation.

6. Formal Specification of Exception Conditions

   The hierarchical nature of the system structure enables the specification of exception conditions at the most appropriate level of abstraction. As a result, integrity and security checks can be easily specified.

There are, of course, many difficult problems remaining to be tackled in order for the methodology to be effective. We will point out a few here.

1. The methodology needs to be extended to incorporate the concept of multiple users and concurrent access.

2. There needs to be additional design tools for testing formal specification so that a designer is reassured that a lengthy formal statement is "consistent" with his intuition.

3. Development of hierarchical performance models for design evaluation. The performance modeling subsystem not only should be able to predict the gross system performance characteristic at each level, but should also be able to provide guidelines for structuring data bases which can best fit the system. An informal approach will be presented in a later section.

4. There is a great need for methods and automatic aids to document the design structure. This is important for generation and evaluation of alternative designs. We will present an approach in the next section.

DESIGN STRUCTURE DOCUMENTATION

The role of specifications in the development of large software systems is certainly an important one. Specifications are used not only as a means of communication between members of the design team, but also serve to enhance the understandability of the system. This is important both for users of the system and for future design teams which must perform modifications.

The previous sections have described certain "local" specifications which are required in the hierarchical design approach - module specification, abstract programs, and mapping functions. Each such specification describes in detail the nature of a very small part of the total system. Yet these specifications are inadequate for purposes of understanding the system as a whole or for explaining why a particular design was developed.

There exists the need, then, to document the system design and the design process at a much higher level of abstraction. Such documentation would suppress details - concentrating rather on the global properties of the system design and the design structure.

The following sections briefly describe a System Design Language (SDL) which can be used to document the design process and record information about the decision-making processes that occur during it. The features of the SDL described in the following sections include methods for:

1. specifying the design alternatives at each level,
2. specifying the hierarchical relationships between system modules, and
3. specifying the structure of each system level.

Specification of Alternative Designs

One aspect of the hierarchical design approach which has yet to be emphasized is that of developing alternative designs at each level. In general a module at level i may be implemented in many different ways and, therefore, at level i-1 the designer may specify various alternative modules to accomplish this task. There exists the need, then, to document exactly how the various alternative modules for implementing the data abstractions of level i may be combined to form designs for level i-1. The designer may then choose the most appropriate alternative design as part of the system (based perhaps upon expected performance).

Using the SDL the designer may accomplish this task of specifying the various alternatives through a process of constructing level components. The syntax of component specification is defined formally in the following BNF grammar:

```
<compname> ::= C<integer>
<modlist> ::= <modname> | <modname>,<modlist>
<complist> ::= <compname> |
               <compname>,<complist>
<compdef> ::= <modlist> | <complist> |
              <compdef>,<modlist> |
              <compdef>,<complist>
<ctype> ::= REQ | ALT | OPT
<cspec> ::= <compname>: (<ctype>,{<compdef>})
```

The simplest type of level component is a single module. However, more complex components can be constructed by combining modules or previously defined components.

Associated with each component constructed is a component type specification (<ctype>) which indicates how "members" of the component may be combined or used in any alternative design. The meanings of the three component types are as follows:

1. REQ - each member of the component must be included in any design.
2. ALT - exactly one member of the component must be i luded in any design.
3. OPT - exactly one subset of the members of the component must be present in any design (this includes the null set).

Formation of alternative designs begins when the designer has developed all alternative modules for implementing each data abstraction of level i. The designer then begins to construct a hierarchy of components - each component in the hierarchy being a composition of lower level components. This process of composition continues until a single component has been constructed which encompasses, directly or indirectly, every module of the initial set. This final component specification is then the starting point for the development of possible alternatives for level i-1.

The process of component construction and alternative design formation for level 3 of Table 1 can be illustrated by the following example.

C1: (REQ,{IMAGE,IMCAT})
C2: (REQ,{LNK,LCAT})
C3: (REQ,{SEL,SLCAT})
C4: (REQ,{INDEX,INDCAT})
C5: (ALT,{C1,C4})
C6: (OP,{C2,C3,C5})
C7: (REQ,{RDIR,FNT,RT,TDS,C6})

This specification indicates, among other things, that

1. components RDIR, FNT, RT, TDS, and C6 must be in every alternative design for level 3,
2. any subset of {C2,C3,C5} may be present in a design for level 3 (because C6 is of type "OP"),
3. if C5 is chosen to be in an alternative design then exactly one of C1 or C4 is to be in the design, and
4. if C1 is chosen to be in the design then both IMAGE and IMCAT must be in the design.

Each component of type "OP" or type "ALT" represents a decision for the designer regarding the structure of the alternative design. Different alternative designs may thus be formed by following different decision pathways.

Specification of Hierarchical Relationships

The next important aspect of the SDL is that of specifying capability relationships between modules of adjacent levels. These capability relationships define the hierarchy which exists between the different modules of the system. Three types of relationships are of interest.

The has access relationship indicates the ways in which a module m can obtain access to instances of a module m'. We distinguish between three different types of allowable access:

1. Creation access (C) - m obtains access to instances of m' by virtue of its ability to invoke operations to create such instances.
2. Indirect access (I) - m obtains access to instances of m' indirectly by using another module m''.
3. Global access (G) - m is "aware" of every instance of m' or is provided with information from a higher level module which enables it to access instances of m' without the need to use other modules.

The uses relationship indicates the means by which a module m may use instances of a module m' to which it has access. We also distinguish between three different types of usage:

1. Read (R) - m can invoke the SV operations of m'.
2. Write (W) - m can invoke the ST operations of m' to modify instances in some way.
3. Create (C) - m can use ST operations to create instances of m'.

The provides relationship indicates what types of module instances a module m may obtain by accessing another module m'.

Formally, a capability set for levels i and i-1 is defined as a triple (A,U,P) where A, U, and P are sets of triples defineu as follows:

A: $\{a | a \in M_i \text{ X } \{C,G,I\} \text{ X } M_{i-1}\}$

U: $\{u | u \in M_i \text{ X } \{R,W,C\} \text{ X } M_{i-1}\}$

P: $\{p | p \in M_i \text{ X } M_{i-1} \text{ X } M_{i-1}\}$

Fig. 3 illustrates the capability relationships which exist between some modules of levels 3, 4 and 5 of the system design of Table 1.

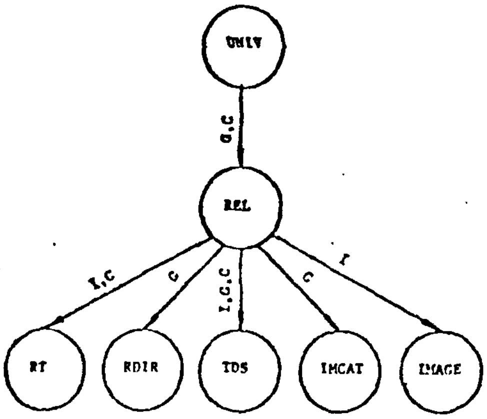


Fig. 3a. The has access relationship between several modules of Table 1. The types of access are Global (G), Indirect (I), and Creation (C) .

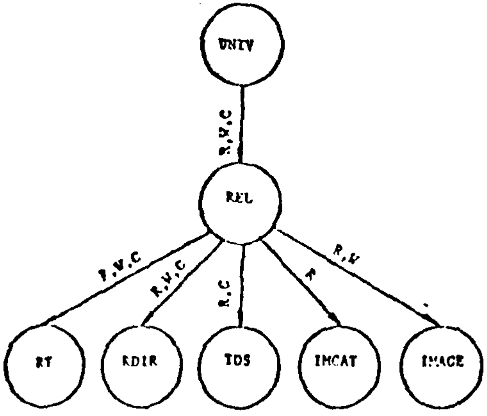


Fig. 3b. The uses relationship between several modules of Table 1. The types of usage are Read (R), Write (W), and Create (C).

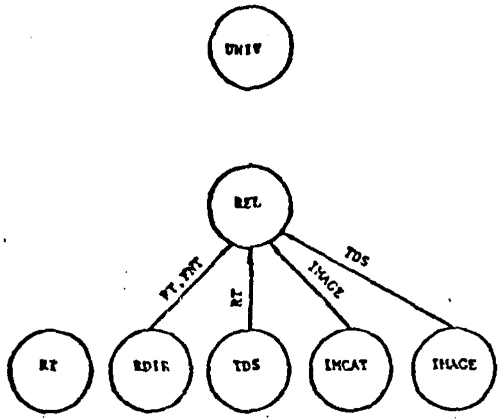


Fig. 3c. The provides relationship between some modules of Table 1.

A specification of capability relationships can be useful in enforcing restrictions on communication between modules. It can also aid the designer in assessing the impact of modifications to system design.

Specification of Level Structure

The final aspect of the SDL which we wish to mention is that of specifying machine structure. It may be useful to allow a limited hierarchy within a particular level and hence the SDL enables the designer to specify the global properties of such a hierarchy. The level structure specification of the SDL indicates, for any level design, the modules which form the level interface (those visible to users of the level), those modules which are hidden (from users of the level), and those modules which must use the interface of the next level (i.e., those modules which are not completely implemented within the level). The level structure specification also defines the hierarchical relationships which exist between modules of the level.

Assessment

The development of SDL presented here is motivated by the need of providing a tool to designers to specify global or macro properties of various system designs. It should be emphasized however, that SDL is meant to be an integral part of the design process, and not merely a specification tool to be used "after the fact". While much of our motivation for developing the SDL is the same as that behind the Module Interconnection Language (MIL) of DeRemer and Kron [1976], there are some fundamental differences:

1. The MIL is concerned with documenting system designs but not the whole design structure (or process). Thus, it does not support the notions of alternative designs, backtracking, etc.

2. In MIL, a module is a small program. In SDL, we consider a module to be the functional specification of a resource type or abstract data type.

3. Our module interconnections are based strictly upon the "uses" concept of Parnas [1974] while this is not the case in MIL.

Much, of course, needs to be done in order for SDL to be a truly useful tool. Extension to include various concepts, such as concurrency, locking, backtracking, etc., is necessary. Automatic aids will be needed for this tool to be practical.

HIERARCHICAL PERFORMANCE EVALUATION

The success or failure of any DBMS, of course, depends greatly upon the level of performance which the system achieves during actual operation. Based upon the results of current research efforts, however, it would seem that our approaches to performance evaluation are somewhat less than satisfactory. This section contains a very general description of a performance evaluation technique which can be used with the hierarchical design approach and which seems to have several advantages over current performance evaluation procedures. This technique involves the construction of a hierarchical performance evaluation model. The purpose of this model is two-fold:

1. to provide the designer with feedback at each step of the design process as to the performance characteristics of his design, and

2. to provide a basis for choosing between alternative designs at each level.

In this approach the designer develops the DBMS design and evaluation model in parallel - the evaluation model being constructed so that it represents the relevant performance aspects of the current DBMS design. The evaluation model provides constant feedback to the designer at all levels of design as to the performance characteristics of the system. Through constant interaction between designer, the DBMS design, and the evaluation model, it is hoped that a reasonably efficient system can be developed with a minimum of backtracking and redesign.

Evaluation Model Structure

The structure of a hierarchical evaluation model reflects that of the DBMS design itself. Corresponding to the ith level is a set of performance parameters, P_i, which represents the relevant performance aspects of the machine at that level. Data structure parameters represent information about the abstract data objects of the level (e.g., number of relations, average number of records per block, etc.). While function parameters characterize the operations of M_i in terms of expected execution speed and expected frequency or probability of access. Parameters may also be classified as design parameters or scenario parameters. Design parameters are variables whose values may be changed by the designer to determine the effects of various database designs and implementations upon the performance of the system. Scenario parameters, however, represent an expected usage of the system in terms of the operations and data objects of level i. Their values are determined by the values of parameters of P_{i+1} according to a performance parameter mapping set T_{i+1}. Each mapping in this set defines a performance parameter of P_i as a function of the parameters of P_{i+1}. A set of values for the scenario parameters of level i is called a scenario for level i.

The values of scenario parameters of P_n are determined by an application scenario supplied as part of the high level requirements specifications. The application scenario is a statement of the expected use of the DBMS in terms of the operations and structures of machine M_n. The requirements specification also contains a performance assertion which specifies the level of performance expected from the system for the given scenario. This performance assertion, by its structure, will indicate the measure to be used in analyzing system performance. Various performance measures might include:

1. mean response time for a given load,

2. expected total execution time for a specified mix of operations,

3. total storage requirements, or

4. a suitably weighted mixture of the above.

The specification of this performance assertion enables the designer to construct a cost function, C_n, for M_n using the parameters of P_n. This cost function may be used by the designer to estimate the performance characteristics of M_n.

Construction of the Evaluation Model

The construction of the evaluation model proceeds top-down with the design of the DBMS. After the design of a machine at level n-1 and the corresponding evaluation model parameter set P_{n-1}, it is necessary to construct the mapping set T_n. Those mappings of T_n which correspond to parameters defining abstract data structure characteristics can be easily constructed from the mapping function of the implementation I_n. However, T_n must also contain mappings which define the probability (or frequency) of access of the operations of M_{n-1} as a function of the probability (or frequency) of access of operations of M_n.

These mappings can be constructed using a technique for the formal verification of performance properties of programs which is based on the method of inductive assertions (Wegbreit [1976]. In this approach an input assertion defines the probability distribution of the input data to a program. From this input assertion various inductive assertions describing the distribution of data at various points in the program are derived. Verification conditions are then constructed which enable the proof of the inductive assertions. It is then possible to derive branching probabilities of various program statements and the expected mean and maximum number of loop iterations for all loops in the program. This, in turn, yields the expected mean and maximum number of executions of each operation in the program text given that the input data is correctly described by the input assertion.

Applying this technique to the abstract programs of I_n enables the derivation of the necessary parameter mappings of T_n. The input assertions for these programs can be derived from the application scenario of the requirements specification. It is then possible to compute the expected mean or maximum number of calls to each operation of M_{n-1} for each call to a given operation of M_n. A set of equations can then be derived, each of which expresses the expected probability (or frequency) of access of each operation of M_{n-1} as a function of the expected probability (or frequency) of access to the operations of M_n.

The application scenario, which is defined in terms of level n structures and operations, can thus be "mapped down" to level n-1 via the mapping set T_n to provide a scenario for the system in terms of level n-1 structures and operations. The designer may then construct a cost function, C_{n-1}, for this level to obtain a more accurate estimate of system performance. By varying the design parameters of P_{n-1} the designer may derive a system configuration which yields a reasonable cost function value and thus determine if the design is capable of satisfying the performance assertion.

Alternative designs at level n-1 may be treated the same way. That is, cost functions may be constructed and evaluated for each alternative. This information may then be used by the designer as a basis for deciding which design path(s) to follow.

The process of evaluation is repeated at each level of the design with the uncertainty of the evaluation model results diminishing at lower levels. The designer may use the information from the evaluation model at any level as a basis for backtracking to a previous level and following a new design path. Likewise, the information may allow the designer to choose one (or more) design paths to follow from a set of alternatives. The end result of this design/evaluation process is a tree-like structure of machine designs and a correspondingly structured hierarchical evaluation model (Fig. 4).

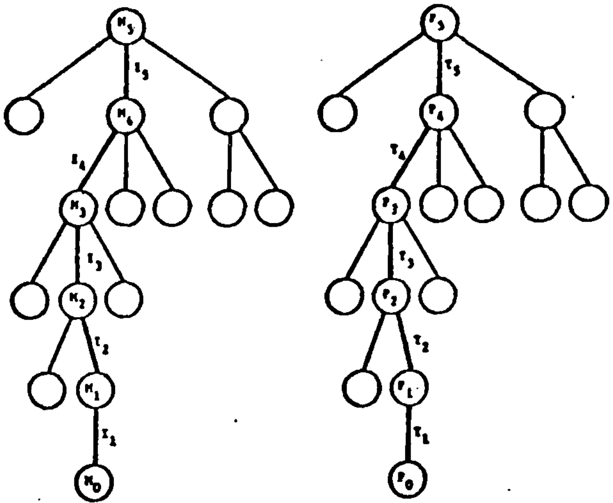


Fig. 4. A hierarchical DBMS design and the correspondingly structured performance evaluation model. Unlabeled nodes represent unused alternative designs.

Assessment

The proposed method of performance evaluation seems to have several advantages over current approaches:

1. Understandability

   Performance related issues are distributed over many levels. Hence the designer may deal with these issues as they occur in the natural hierarchy of design. The hierarchical structure of the model should thus facilitate its use and understanding.

2. Flexibility

   The designer can model each level design in as much detail as desired. Moreover, the approach does not limit the designer to models of specific architectures - models for any alternative design may be developed.

3. Immediate Feedback

   At each level the designer receives feedback from the evaluation model. This, hopefully, can limit the amount of redesign and backtracking which is necessary.

4. Data Base Design

   The evaluation model used for DBMS design may be used to facilitate the data base design. The process would be top-down. At each level the cost function would be used to determine a performance-effective data base structure for that level.

CONCLUDING REMARKS

The methodology presented in the previous sections is, of course, a first attempt toward a comprehensive approach to design problems. We have assessed the three models in the methodology at the end of appropriate sections. However, one point that should be stressed is that the methodology provides for the development of a family of designs rather than a single design. Such a documentation will certainly be of immense help to an evolving system.

The methodology still lacks engineering flavor. To make it complete, additional tools will be necessary. In this aspect, we would like to mention that the notion of a "mock-up" model should be part of this design methodology. We think that in this context we should develop computer processable specification so that performance evaluation not only can be done by mathematical modeling as we have discussed here, but also by actual or symbolic execution of the specification (of the mock-up model). Such a tool woul allow a designer to tinker with his design (e.g., to make sure that formal specification is consistent with the more informal requirements) until he is satisfied. Furthermore, this would provide users with earlier warnings if any inadequacies were discovered in the requirements. At the University of Texas at Austin, we are in the process of developing such tools.

REFERENCES

1. Astrahan, M. M., et al, [1976], "System R: Relational Approach to Database Management," ACM TODS, vol. 1, no. 2, pp. 97-137.

2. Aurdal, E. and Solberg, A., [1975], "A Multiple Process for Desgin of File Organization," CASCSDE Working Paper No. 39, Royal Norwegian Council for Scientific and Industrial Research.

3. Baker, J. and Yeh, R. T., [1977], "A Hierarchical Design Methodology for Data Base System,", TR-70, Dept. of Computer Sciences, University of Texas at Austin, Austin, Texas.

4. Bayer, R. and McCreight, E., [1972], "Organization and Maintenance of Ordered Indexes," Acta Informatica, vol. 1, no. 3, pp. 173-189.

5. Chen, Peter P.S., [1975], "The Entity-Relationship Model - Toward a Unified View of Data," Rech. Report, Center for Information System Research, Sloan School of Management, M.I.T.

6. Codd, E. G., [1970], "A Relational Model of Data for Large Shared Data Banks," CACM, vol. 13, no. 6, pp. 377-387.

7. DeRemer, F. and Kron, H. H., [1976], "Programming-in-the-Large Versus Programming-in-the-Small," IEEE Trans. Soft. Eng., vol. SE-2, no. 2, pp. 87-96.

8. Goodenough, John B., [1975], "Exception Handling: Issues and a Proposed Notation," CACM, vol. 18, no. 12, pp. 683-696.

9. Hoare, C.A.R., [1970], "An Axiomatic Approach to Computer Programming," CACM, vol.12, no. 5, pp. 76-80,83.

10. Kraegeloh, Klaus-Dieter and Lockemann, Peter C., [1975], "Hierarchies of Data Base Languages: An Example," Information Systems, vol. 1.

11. McKeeman, W., [1975], "On Preventing Programming Languages for Interfering with Programs, IEEE Trans. on Soft. Eng., vol. 1, no. 1, pp. 19-25.

12. Madnick, S. E. and Alsop, J. W., [1969], "A Modular Approach to File System Design," Proc. AFIPS, vol. 34, pp. 1-12.

13. Parnas, D. L., [1972], "A Technique for Software Module Specification with Examples," CACM, vol. 15, no. 5, pp. 330-336.

14. Parnas D. L., [1976a], "On the Criteria to Be Used in Decomposing Systems Into Modules," CACM, vol. 15, no. 12, pp. 1053-1058.

15. Parnas, D. L., [1976b], "On A Buzzword: Hierarchical Structures," IFIP Proc.

16. Robinson, L. and Levitt, K. N.,[1977], "Proof Techniques for Hierarchically Structured Programs," to appear - Current Trends in Programming Methodology, Vol. 2, (Yeh, ed.), Prentice-Hall.

17. Senko, M. W., [1976], "DIAM II and Levels of Abstraction," Proc. Conf. on DATA: Abstraction, Definition and Structure, pp. 121-140.

18. Smith, J. M. and Smith, D. C. P., [1977], CACM.

19. Weber, H., [1976], "The D-graph Model of Large Shared Data Bases: A Representation of Integrity Constraints and Views of Abstract Data Types, IBM TC (San Jose).

20. Wegbreit, B., [1976], "Verifying Program Performance," JACM, vol. 23, no. 4, pp. 691-699.

21. Yeh, R. T, (ed.) [1977], Current Trends in Programming Methodology: Vol. 1. Software Specification and Design, Prentice-Hall, Inc, Englewood Cliffs, N.J.

APPENDIX 3

Dan Chester

The specifications in this appendix are for a relational data base system that stores explicit relation on sequential files such as tapes. The time to retrieve the n-tuples in an implicit relation is expected to grow at a rate that is much less than N^2, where N is the number of n-tuples that can be formed from the individuals named in the data base.

The first specification is a function module modelling the whole data base system. It exhibits the basic behavior of the system without making commitments to performance aspects. Each function is defined by an expression in the following format:

```
function: <function name> <argument pattern> = <value pattern>
effects:
        <statement>
            :
        <statement>
```

The effect statements are optional. When present the function is computed by making the statements true and returning the value indicated by the <value pattern>.

```
function:  data(X) = Y


function:  define(R(X(1),...,X(N)),Y) = nil
effect:

  definition(R(X(1),...,X(N)),Y) = true.

  defined(R) = true.


function:  defined(X) = Y


function:  definition(X,Y) = Z


function:  insert(R(X(1),...,X(N))) = nil
effect:

  data(R(X(1),...,X(N))) = true.

  for all I such that 1 <= I <= N: universe(X(I)) = true.


function:  list(R) = (X(1),...,X(M))
effect:

  for all I,J such that 1 <= I,J <= M:
    if not I = J then not X(I) = X(J).

  for all I such that 1 <= I <= M:
    for some Y(1),...,Y(N):
      X(I) = R(Y(1),...,Y(N)) and tempdata(X(I)) = true.

  for all Y(1),...,Y(N) such that tempdata(R(Y(1),...,Y(N))) = true:
    for some I: X(I) = R(Y(1),...,Y(N)).

  for all R,X(1),...,X(N) such that 'defined'(R) = nil:
    tempdata(R(X(1),...,X(N))) = 'data'(R(X(1),...,X(N))).

  for all R,X(1),...,X(I),S,Y(1),...,Y(J) such that
      'definition'(R(X(1),...,X(I)),S(Y(1),...,Y(J))) = true:
    for all Z(1),...,Z(I): tempdata(R(Z(1),...,Z(I))) = true iff
      for some U(1),...,U(J) such that
          for all K,L: if Y(K) = Y(L) then U(K) = U(L), and
            if Y(K) = X(L) then U(K) = Z(L);:
        tempdata(S(U(1),...,U(J))) = true.

  for all R,X(1),...,X(I),S,Y(1),...,Y(J) such that
      'definition'(R(X(1),...,X(I)),not S(Y(1),...,Y(J))) = true:
    for all Z(1),...,Z(I):
      tempdata(R(Z(1),...,Z(I))) = true iff
      for all K such that 1 <= K <= I: 'universe'(Z(K)) = true;
      for all U(1),...,U(J) such that
          for all K,L: if Y(K) = Y(L) then U(K) = U(L) and
            if Y(K) = X(L) then U(K) = Z(L);:
```

```
      tempdata([illegible](1),...,V(J))) = nil.

  for all R,X(1),...,X(I),S,Y(1),...,Y(J),T,Z(1),...,Z(K) such that
       'definition'(R(X(1),...,X(I)),S(Y(1),...,Y(J)) and
          T(Z(1),...,Z(K))) = true:
    for all U(1),...,U(I):
      tempdata(R(U(1),...,U(I))) = true iff
        for some V(1),...,V(J),W(1),...,W(K) such that
            for all M,N:
              if Y(M) = Y(N) then V(M) = V(N) and
              if Z(M) = Z(N) then W(M) = W(N) and
              if Y(M) = X(N) then V(M) = U(N) and
              if Z(M) = X(N) then W(M) = U(N) and
              if Y(M) = Z(N) then V(M) = U(N);:
          tempdata(S(V(1),...,V(J))) = true and
          tempdata(T(W(1),...,W(K))) = true.


function:  remove(R(X(1),...,X(N))) = nil
effect:

  data(R(X(1),...,X(N))) = nil.

  for all I such that 1 <= I <= N:
    if for all S,Y(1),...,Y(M) such that
          data(S(Y(1),...,Y(M))) = true:
        for all J such that 1 <= J <= M: not X(I) = Y(J);;
    then universe(X(I)) = nil.


function:  tempdata(X) = Y


function:  undefine(R) = nil
effect:

          [illegible]R) = nil.

  for all X(1),...,X(N),Y: definition(R(X(1),...,X(N)),Y) = nil.


function:  universe(X) = Y
```

```
module: DBS

procedure: insert (R(X(1),...,X(N)))
definition:
     include (R,(X(1),...X(N)))
     for I = 1 to N step 1 do
     increment ("universe", X(I))

procedure: remove (R(X(1),...,X(N)))
definition:
     exclude (R,(X(1),...,X(N)))

procedure: define (R(X(1),...X(N)),Y)
definition:
     include ("definitions:,(R,(X(1),...X(N)),Y))
     include ("defined",R)

procedure: undefine (R)
definition:
     let X = find ("definition",1,R)
     exclude ("definition",X)
     exclude ("defined",R)

procedure: list (R)
definition:
     let X = relations (R)
     while X ≠ nil do
       begin
       makefile (head(X))
       let X = tail (X)
       end
     print (R)

procedure: relations (R)
definition:
     if find ("defined",0,R) then
       begin
       let Z = find ("definition",1,R)
       let (R,X,Y)=Z
       if let S(W(1),...W(N)) = Y then
        return append (relations (S),(R))
       else
       if let not S(W(1),...,W(N)) = Y then
        return appenc (relations (S),(R))
       else
       if let S(W(1),...,W(N)) and T(V(1),...,V(M)) = Y
       then
        return append (relations(S), append (relations(R),(Y))
     end
```

```
procedure:  makefile (R)
definition:
     if find ("defined",OR) then
       begin
       let Z = find ("definition",1,R)
       let (R,X,Y) = Z
       if let S(W(1),...,W(M)) = Y then
       begin
       erase (R)
       project (R,X,S,(W(1),...W(M)))
     end
     else if let not (S(W(1),...,W(M)) = Y then
       begin
       erase (R)
       complement (R,X,S,(W(1),...,W(M)))
     end
     else if let S(W(1),...,W(M)) and T(V(1),...,V(N)) = Y
     then
       begin
       erase (R)
       join (R,X,S,(W(1),...,W(N)),T,(V(1),...,V(N)))
     end

procedure:  find (X,I,Y)
definition:
     rewind (X)
     repeat
     let Z = next (X)
     until
     Z = nil or
     (I = 0 AND Z = Y) or
     return Z

procedure:  increment (X,Y)
definition:
     let Z = find (X,1,Y)
     if Z = nil then begin include (X,(Y,1))
     else
       begin
       let (Y,M) = Z
       let N = M + 1
       replace (X,(Y,N))
     end

procedure:  decrement (X,Y)
definition:
     let Z = find (X,1,Y)
     if Z ≠ nil then
       begin
       let (Y,M) = Z
       let N = M - 1
       if N = 0 then begin exclude (X,(Y,M))
       else replace (X,(Y,N))
     end
```

```
procedure:  include (X,Y`
definition:
     rewind (X)
     repeat
     let Z = next (X)
     until
     Z = nil or
     Z = Y
     if Z = nil then extend (X,Y)

procedure:  exclude (X,Y)
definition:
     let Z = "time"
     erase (Z)
     rewind (Z)
     rewind (X)
     repeat
     include (Z,next (X))
     until pointer (X) = nil
     erase (X)
     rename (X,Z)

procedure:  project (R,X,S,W)
definition:
     rewind (S)
     repeat
     let Z = next (S)
     if Z ≠ nil then
     include (R,bind (Z,W,X))
     until Z = nil
     sort (R,X,X)

procedure:  complement (R,X,S,W)
definition:
     let V = "time"
     erase (V)
     project (V,X,S,W)
     rewind (V)
     startgen (X)
     repeat
     let Z = next (V)
     repeat
     let U = nextgen (X)
     if U ≠ nil and (Z = nil or U = Z )
     then include (R,U)
     until U = nil or U = Z
     until Z = nil
```

```
procedure:  join (R,X,S,W,T,V)
definition:
    let Z = common (W,V)
    sort (X,W,Z)
    sort (T,V,Z)
    rewind (S)
    rewind (T)
    erase (R)
    let S1 = next (S)
    let T1 = next (T)
    repeat
    if less (bind (S1,W,Z),bind (T1,V,Z))
    then let S1 = next (S)
    else if bind (S1,W,Z) = bind (T1,V,Z)
    then
      begin
      erase(S2)
      let S3 = S1
      erase (T2)
      let T3 = T1
      repeat
      include (S2,S3)
      let S3 = next (S)
      until S3 = nil or (bind (S1,W,Z) bind (S3,W,Z))
      repeat
      include (T2,T3)
      let T3 = next (T)
      until T3 = nil or bind (T1,V,Z)  bind (T3,V,Z)
      let S1 = S3
      let T1 = T3
      rewind (S2)
      repeat
      let S3 = next (S2)
      rewind(T3)
      repeat
      let T3 = next (T2)
      include (R, bind (append (S3,T3), append (W,V),X))
      until T3 = nil
      until S3 = nil
    end
    until S1 = nil or T1 = nil
    sort (R,X,X)
```

Fig. 6 (cont'd.)

```
procedure:  sort (R,X,Y)
definition:
     let S = "temp"
     let T = "temp2"
     let N = 1
     repeat
       rewind (r)
       erase (S)
       repeat
     let J = 1
     erase T
     repeat
     include (T,next (R))
     J = J + 1
     until J > N or pointer (R) = nil
     if J >N then begin
     rewind (T)
     let I = 1
     repeat
     let W = next (T)
     let V = next (R)
     repeat
     if W = V then let W = next (T)
     else if bind (W,X,Y) < bind (V,X,Y)
     then begin
     include (S,W,)
     W = next (T)
     end
     else begin
     include (S,V,)
     V = next (R)
     end
     I = I + 1
     until V = nil or W = nil
     if W ≠ then
     repeat
     include (S,W)
     W = next (T)
     I = I + 1
     until W = nil
     if I ≤ 2N and V ≠ nil then
     repeat
     include (S,V)
     V = next (R)
     I = I + 1
     until I > 2N or V = nil
     until V = nil
     rename (R,S)
     let N = 2N
     end
     until J≤ N
```

```
module:  files

function:  file (X) = Y

function:  pointer (X) = Y

function:  rewint (X) = nil
effect:  pointer (X) = 'file'(X).

function:  next (X) = y
effect:
     for some Z(1),...,Z(N) such that
     'pointer'(X) = (Y,Z(1),...Z(N)):
     pointer (X) = (Z(1),...,Z(N)).

function:  erase (R) = nil
effect:
     file (R) = nil.
     pointer (R) = nil.

function:  replace (X,Y,) = nil
effect:
     for some Z(1),...,Z(M),I such that
     'file'(X) = (Z(1),...,Z(M)) and
     'pointer' (X) = (Z(I),...Z(M)):
     file (X) = (Z(1),...,Z(I-L),Y,Z(I+1),...Z(M)) and
     pointer (X)=(Y,Z(I+1),...Z(M)).

function:  extend (X,Y,)= nil
effect:
     if 'pointer'(X) = nil then
     for some Z(1),...,Z(M) such that
     'file'(Y) = (Z(1),...,Z(M)):
     file (X) = (Z(1),...,Z(M),Y).

function:  rename (X,Y) = nil
effect:
     file(X) = 'file'(Y).
     file (Y) = nil.
```

Module: Records

```
function:  current (X) = Y

function:  append ((X(1),...,X(M)),Y(1),...,Y(N)))=
           (X(1),...,X(,),Y(1),...,(Y)N))

function:  head ((X(1),...,X(N))) = X(1)

function:  tail ((X(1),...,X(N)) = (X(2),...X(N))
```

AD-A108 104 TEXAS UNIV AT AUSTIN DEPT OF COMPUTER SCIENCES F/G 9/2
DECISION SUPPORT SYSTEMS: A PRELIMINARY STUDY,(U)
SEP 77 R T YEH, W W BLEDSOE, M CHANDY N00039-77-C-0254
UNCLASSIFIED TR-74 NL

2 OF 2
AD
A108104

END
DATE
FILMED
1-82
DTIC

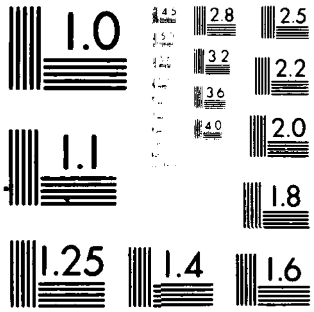

1.0
4.5
5.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

Fig 6 (Cont'd.)

```
function:  bind ((X(1),... X(M)),
                 (Y(1),...Y(M)),
                 (Z(1),...,Z(N))) = (U(1),...,U(N))
effect:
     for all I,J such that I ≤ I, J ≤ N:
     if Y(I) = Z (T) then X(I) = U(J) and
     if Y(I) = Y(J) then X(I) = X (J).

function:  startgen ((X(1),...,X(N))) = nil
effect:
     for some Y(1),...,Y(N) such that
     for all I such that 1 ≤ I <N:
     Y(I) = Y(I + 1);:
     for some Z(1),...,Z(M) such that
     'oblist' = (Z(1),...,Z(M)):
     for all I such that 1 ≤ I ≤ M:
     Y (1) ≤ Z (I);;
     current (X(1),...,X(N)) = (Y(1),...,Y(N))

function:  nextgen ((X(1),...,X(N))) = (Y(1),...,Y(N))
effect:
     for some Z(1),...,Z(N) such that
     'current' (X(1),...,X(N)) = (Z(1),...,Z(N)):
     current (X(1),...,(Y(1),...,Y(N)) and
     for some I such that 1 ≤ I ≤ N:
     for all J such that 1 ≤ J <I:
     A(J) = Y(J);
     for some W(1),...,W(M) such that
     oblist - (W(1),...,W(M)):
     for all J such that I <J ≤ N:
     for all 1  such  1 ≤ 1 ≤M:
     Y(J)≤W(K);
     for some 1  such that 1 ≤ 1< ≤ M:
     Y(J) = W(K);
     for all J such that 1 ≤ J ≤ M:
     Y(I) ≤ W(J);
     Z(I) <Y(I).

function:  common ((X(1),...,X(M)),
                   (Y(1),...Y(N)))= (Z(1),...,Z(1<))
effect:
     for all I such that i ≤ I ≤ M:
     if for some J such that 1 ≤ J ≤ M;
     X(I) = Y(J); then
     for some J such that 1 ≤ j ≤<1 :
     X(I) = Z(J);
     for all I such that 1 ≤ I ≤ 1<:
     for some J,H such that 1 ≤ J ≤ M and
     1 ≤ H ≤ N: Z(I)=X(J) and Z(I)=Y(H);
     for all J such that 1 ≤ J ≤ L* and I ≠ J:
     Z(I)≠Z(J).
```

```
function:  oblist = X

function:  tooblist (X) = nil
effect:
     for some Y(1),...,Y(N) such that
     'oblist' = (Y(1),...,Y(N)):
     oblist = (X,Y(1),...,Y(N)).

function:  fromoblist (X) = nil
effect:
     for some Y(1),...,Y(N),I such that
     'oblist' = (Y(1),...,Y(I),X,Y,(I+1),...,Y(N)):
     oblist = (Y(1),...,Y(N)).
```

APPENDIX 4

A METHOD FOR CONTROL OF THE INTERACTION OF CONCURRENT PROCESSES

by

M. H. Conner

It is the objective of this research to explore a method for controlling the interaction of concurrently executing processes. The nature of my approach is to observe that processes exhibit an external behavior in the form of calls to operations to shared data objects. My basic premise is that by placing various external controls on this behavior one can usefully control the interaction of concurrent processes. I examine this premise by giving a model of computation in which the external behavior of processes is well defined. I then introduce the notion of behavior controllers to constrain the external behavior of processes.

In the following, I present a model of computation which I call the structured environment. I chose this name since it reflects my desire to define a model which is both sufficiently and appropriately structured for rigorous identification of the interaction between control and data. As the name "structured environment" connotes, it is my intention to incorporate several of the notions associated with "structured" programming. Namely, the model incorporates the notions of one entry/one exit control structures and abstract data objects.

In order to motivate some of the concepts used in the structured environment model, I present the following informal analysis of a Turing machine.

Even the most casual analysis of a Turing machine must note its decomposition into two primary parts. Namely, a Turing machine consists of a finite state control (or control part) and a tape (or data part). As soon as this decomposition is noted, it is reasonable to consider how these parts interact. At first glance one might say that the parts interact via the positioning and writing operations which the finite state control causes to be performed on the tape. In fact, this is sufficient to describe the mechanism by which the tape is modified. However, these operations do not describe the mechanism by which the finite state control receives information from the tape. Typically, this interaction is described by specifying that the domain of the finite state control's state transition function includes the value of the symbol currently under that tape head. Let me propose a slightly different view. Suppose one associates two "local" data objects with the finite state control: a current state data object and a current symbol data object. Further, suppose that one adds to the operational repertoir of the Turing machine an operation which transfers the value of the finite state control's current symbol data object to the position on the tape which is currently under the tape head. Also, add an operation that does the inverse. It is now possible to restrict the domain of the state transition function entirely to the values of the finite state control's two local data objects if one assumes that each step of the computation proceeds as follows:

1) Transfer symbol under tape head to current symbol data object.
2) Compute new value for current state data object and for current symbol data object based on the present values of these two object.
3) Write value of current symbol data object to the tape.
4) Perform desired operation to reposition to the tape head (e.g., Move left, No move, or Move right).

Clearly, these modifications to the traditional notion of a Turing machine have no effect on its computational power. In fact, in most formal definitions of a Turing machine it would not be necessary to make any change in the tuple which describes a particular Turing machine. One would only have to change the definition of the configuration of the Turing machine to incorporate the value of the current symbol data object and then make the obvious change to the relation between two configurations (i.e., redefine a computational step as specified above). However, these changes do have one very important effect. They demonstrate that one can view a Turing machine as composed of two separate parts, a control part and a data part, and that interaction between these parts can be defined to occur only through an identifiable set of operations. Thus, these operations precisely define the interface between the control part and the data part of the Turing machine.

This precisely known interface is very important for at least the following two reasons:

1) Since the only means of information flow between the process and

data parts is some known set of operations, each part is effectively insulated from the representation (or implementation) details of the other. This property is of course quite unimportant in the normal context of Turing machines, but is very important in the normal context of programming. In fact, this property forms the basis of the information hiding that is so important in the work on modules and abstract data types.

2) It is frequently valuable to constrain the access a process may have to data objects. If the only access a process has to some data object is through some set of operations, then there are many constraints that may be converted into simple restrictions on the set of sequence of operations the process may perform on the data object. This is certainly the underlying notion in the work concerning capabilities, monitors, path expressions, etc.

I have presented this example to illustrate the relation between control and data that underlies the structured environment model. Namely, I maintain that there must be some small amount of data which is actually a part of the control in some intuitive sense. I will refer to such data as local data. However, it seems that there exists a natural decomposition between the control and a large portion of the data. I will refer to such data as external data. In fact, this example and our intuition suggest that one can reduce the local data to an almost arbitrarily small amount. This then is an intuitive justification for only constraining the interaction between control and external data.

I am now prepared to introduce the structured environment model. My presentation will be heirarchical and I will only present a very abstract view to begin with.

The first three components that I wish to discuss are:

1) Processes

2) Operations

3) Data Objects.

Abstractly, a data object is an entity with an associated property usually referred to as a value. But a value is just a property, it is derived by the interpretation of a representation. Thus, a data object is really an incapsulation of a representation which if interpreted properly yields meaningful information. It is the representation incapsulated in the data object that must be manipulated to extract or change the information contained in the data object. Since a data object is just the incapsulation of a representation of information it is necessarily a static object. That is, a data object does not change in any way unless its representation is manipulated by some other object. In this model there are two classes of objects that may manipulate the representation of a data object. These are the operations and processes mentioned above. However, I will consider that data objects are divided into two classes: local data objects and external data objects. Processes may directly manipulate local data objects only, while operations may directly manipulate data objects of both classes. The reason for this distinction will be brought out when processes are discussed below.

At this point, I wish to be somewhat vague concerning operations. I will simply say that operations are <u>performed</u> on data objects. The effect of performing an operation on a data object is to manipulate directly the representation of the data object's associated value, possible causing some change in the information contained in the data object. For any given data object only one operation may be performed on it at a time. That is, as far as data objects are concerned, the performance of an operation is an indivisible operation. An operation may only manipulate the representations of the data objects on which it is performed. Since operations are the only class of objects allowed to manipulate the representation of external data objects and since the only way to extract or change the information in an external data object is to manipulate its representation, it follows that the only way to extract or change the information is an external data object is to perform an operation on it. (The above discussion makes more sense if the reader considers that the data objects on which an operation is performed may be a subset of the data objects which one would normally refer to as the parameters of the operations. I will discuss this much more fully later.)

So far I have described data objects for storage of information and operations for the transformation of information stored in data objects. All that remains in order to have a complete computational model is some way to meaningfully sequence the performance of operations on data objects. This is precisely the role of processes. That is, processes are the control units of the model; they each cause a sequential sequence of actions to take place in order to effect some computation. There are precisely two types of actions a process may cause:

1) A process may directly manipulate the representation contained in a local data object.

2) A process may sequentially perform operations on both local and external data objects.

In particular, no process may directly affect another process. Thus, two processes can only communicate through data objects. I will say that data acted on by more than one process are shared by all those processes that act on them. (By "act", I am referring to the two types of actions allowed to processes as described above.) I will also make the restriction that no local data object may be shared. This has a very important implication: two processes may communicate only by sequentially performing operations on shared external data objects. This is the result that I believe justifies the structured environment model as presented so far.

In summary, I have started to present a model of computation that allows multiple interacting processes but restricts their interaction to the performance of operations on shared data objects. I have given an intuitive argument for the feasibility of such a restriction by examining a Turing machine and showing that one can take the view that the finite state control only interacts with the tape by the performance of certain operations. In Figure 1, I present a decomposition of a Turing machine along the lines of the structured environment as presented so far.

I would like to use this figure to review several important points:

- The process component, which I called FINITE_STATE_CONTROL in the figure, is strictly sequential in its interaction with the external data objects (in this case there is only one, TAPE). I.E., It may perform exactly one operation at a time.
- No restrictions are placed on the interaction between the FINITE_STATE_CONTROL and its local data objects, CURRENT SYMBOL and CURRENT_STATE. Nor is anything said about how FINITE_STATE_CONTROL is implemented, except that it is sequential in its interaction with data objects.
- No restrictions are placed on the operations except to say on which objects they are "performed", i.e, which objects they may manipulate directly. In fact, I have not prohibited operations from performing other operations (this topic will be dealt with later).

Process: FINITE_STATE_CONTROL

Data Objects:

Local: CURRENT_SYMBOL, CURRENT_STATE

External: TAPE

Operations:

WRITE (CURRENT_SYMBOL,TAPE): Copies the symbol contained in CURRENT_SYMBOL to position on the TAPE which is currently under the tape head.

READ (CURRENT_SYMBOL,TAPE): Copies the symbol currently under the tape head on the TAPE into CURRENT_SYMBOL.

MOVE_LEFT (TAPE): Moves the TAPE's tape head left.

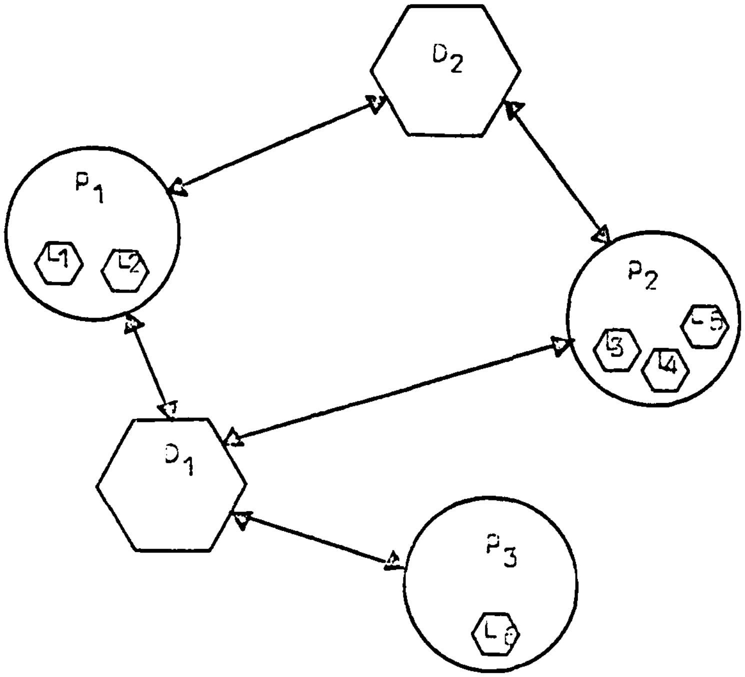
MOVE_RIGHT (TAPE): Moves the TAPE's tape head right.

FIGURE 1. Structured environment model of a Turing machine.

In Figure 2, I graphically depi-t the communication allowed in the structured environment model. In order to illustrate some of the communication problems that arise in such an environment, I would like to consider an example.

Suppose one had a system consisting of several processes and a shared output device which I shall model as a data object. Now suppose that one wished to insure that the following two properties held in this system:

1) Proper use: Before actually sending data to be output to the device it must be readied for use. (Consider a printer where certain forms control and heading information might need to precede the actual text to be printed.)



Key:

○ Processes ⬡ Data Objects

⟷ Joins processes with data objects on which they perform operations.

Data objects drawn inside of processes are local data objects belonging to the process.

FIGURE 2. Communication in the structured environment model.

2) Proper synchronization: Only one process should be using the device at a time. I.E., after setting up the device for use, the same process should retain control of the device until it has completed its output task.

How can these properties be insured? First of all, one might note that these are properties concerning the interaction between the processes and the output device (an external data object).

In the structured environment model there is only one way a process may interact with an external data object. This requires that such actions as setting up the output device, writing to it, etc., must be incapsulated in operations to be performed on the device. But then it should be possible to translate the above properties into properties concerning the sequence in which operations are performed. First, I will propose a set of operations that may be performed on the output device. The following three operations seem to sufficient.

1) OPEN - Prepares the output device for the next output task

2) WRITE - Causes one unit of data to be output

3) CLOSE - Signals the completion of an output task.

The above properties can now be restated in terms of the operations as follows:

1) Proper use: Each process will always perform operations on the output device in the order: OPEN, any number of WRITEs, CLOSE. This sequence may be repeated any number of times. No process will perform any other operations on the output device.

2) Proper Synchronization: Once one process has performed an OPEN no other process will perform any operation on the output device until the first process performs a CLOSE.

Consider Figure 3, depicting the communication paths in the structured environment model for a two process version of this example.

Now consider how one might insure that the restated properties hold.

The proper use property could be insured by examining each process in the system and verifying that each process would only perform the allowed set of operations and then only in the allowed sequence. This method has two outstanding drawbacks.

First of all it can be very difficult. In fact, it is clear that the rigorous verification of this property could be as hard as the rigorous verification of any other property of a process. A very difficult task indeed!

Secondly, this method requires that the definition of all processes (current and _future_) be available for examination. However, it is frequently desirable in a multiprocess environment to be creating new processes some of which may have been unavailable for examination. (Consider an operating system running user processes.)

The only general solution to both these drawbacks seems to require some sort of external constraint on the operations a process may perform on shared data objects.

In fact, the notion of capabilities can be viewed as a very limited form of such a constraint. A capability for a data object defines the _set_ of operations a process may perform on a data object. This, of course, still leaves the very difficult problem of insuring that processes perform the proper sequence of operations. I suggest that one needs a general mechanism to constrain the sequence of operations performed by a process on a data object. I therefore add to the structured environment model a component which I call a _rights controller_.

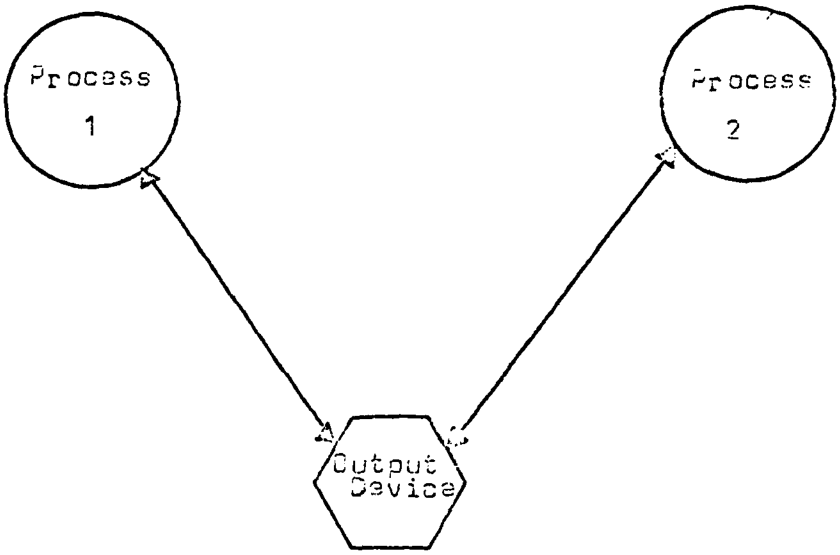
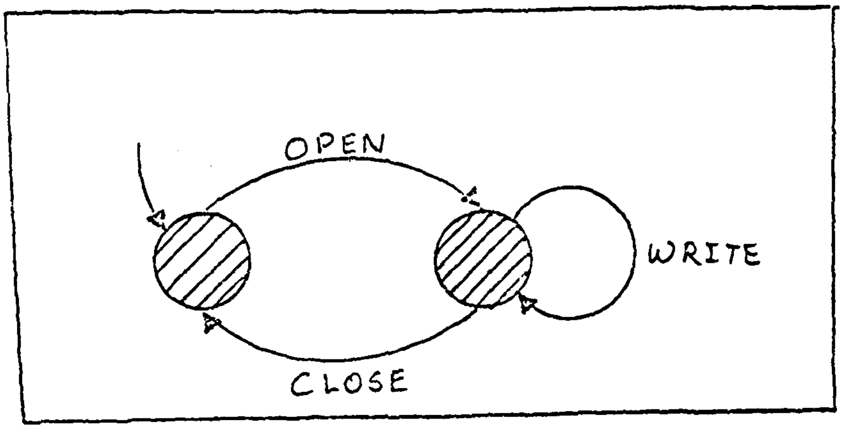


FIGURE 3. Structured environment model of a two process version of the output device example.

A rights controller is simply a finite state acceptor over the sequences of operations that may be performed by a particular process on a data object. That is, each rights controller defines a set of sequences of operations that may be performed on a particular data object.

In order to make the rights controllers effective, there must be some way in the structured environment model to require the process to observe the constraints of the appropriate rights controllers. I achieve this through the notion of an environment, where an environment is defined to be a sequence of rights controllers with the constraint that there cannot be two rights controllers in the same environment controlling the performance of operations on the same data object. I then specify that there be associated with each process a single unique environment and that a process may only perform an operation on an external data object if it is allowed by the appropriate rights controller in the process's environment.

Since a rights controller is a finite state acceptor of the sequences of operations that a process may perform on a data object, one obvious way of describing a rights controller is a state graph with arcs labeled by operations. Figure 4 describes an appropriate rights controller for the processes in the output device example. Figure 4 also shows how this rights controller might be described by a regular expression over operations. The specification of the structured environment model says nothing about how rights controllers are to be inplemented, but it does say that a process may only perform the operations allowed by its rights controllers. Therefore, it would seem that a very reasonable way to achieve this effect would be through a runtime monitor (i.e., an active finite state acceptor). Thus, I prefer the state graph description for its dynamic connotation.



(OPEN, WRITE*, CLOSE)*

FIGURE 4. Two descriptions of a rights controller.

Returning to the output device example, consider the situation of each process that shares the output device having a copy of the rights controller described in Figure 4 as an element of its environment. Figure 5 depicts a two process version of such a situation. (Note that any other elements in the process's environment cannot directly affect its interaction with the output device because of the requirement that only one rights controller constrains access to the same data object in any one environment.) In Figure 5, I have interrupted the lines connecting the processes with the output device to indicate that the only interaction each process may have with the output device is the performance of the operations allowed by the rights controllers. This will be the normal way I indicate a process's environment in subsequent figures. Thus, Figure 5 indicates that each process can only interact with the output device in precisely the manner required by the proper use property given above. However, it should be clear that even though each process is trying to make proper use of the output device, there is no guarantee that the processes will synchronize their performance of operations properly to achieve the proper synchronization property given above. For example. Process 1 might perform an OPEN followed by several WRITEs and then Process 2 might perform an OPEN which clearly violates the proper synchronization property. Clearly, the notions of environments and rights controllers are not enough to directly handle the problem of process synchronization.

Consider, for a moment, the structured environment model as it stands so far. I have constrained the interactions of processes to a single mechanism, namely the performance of operations on shared data objects.

Suppose I refer to the performance of operations on external data objects as the behavior of a process. Then one can think of a rights controller as defining allowable behavior. It follows then that a process's environment

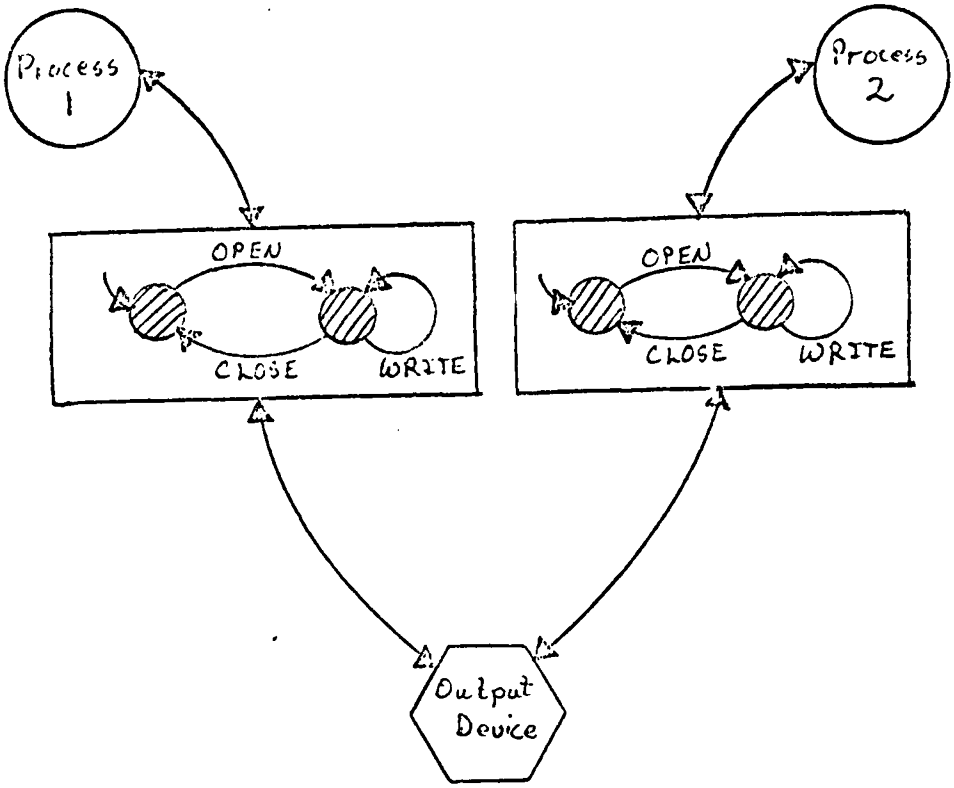


FIGURE 5. Two process version of output device example including rights contollers.

defines the totality of a process's allowable behavior. However, there are two possible ways to control behavior:

1) At its source, the process
2) At its destination, the data object.

Thus, I suggest that the problem of synchronization be dealt with as the behavior arrives at a data object. To this end, I add to the structured environment model a class of components I call synchronizing controllers. A synchronizing controller will synchronize the operations that may be performed on a data object in order to achieve a particular sequence of operations. Thus, the description of a synchronizing controller is very similar to that of a rights controller. Namely, it consists of a specification of the sequence of operations that it allows to be performed on its associated data object. Note, however, that there is a considerable difference of interpretation. A rights controller defines the allowable behavior for a process. If the process violates its allowable behavior then it is outside of the structured environment model, i.e., it is in error and must be aborted or something. However, a synchronizing controller will actively attempt to achieve its required sequence of operations by delaying processes.

I have referred to the synchronizing and delaying of processes above without describing how this is done. Let me do so now.

Recall that the primary defining characteristic of a process is that it performs a <u>sequential</u> sequence of actions. Thus, once a process begins to perform an operation the process is essentially inactive (it cannot interact with any data object) until the operation is completed. With this in mind I will decompose the performance of an operation into three phases:

1) scheduling
2) execution
3) completion.

These phases must occur in the order shown above. The scheduling phase consists of the operation being scheduled by the synchronizing controller associated with each of the data objects on which it is to be performed. The execution phase occurs after the scheduling phase has completed and consists of the actual transformation on the data objects. The completion phase occurs after the execution phase has completed. This phase marks the completion of the operation. That is, the process that performed the operation becomes active again at the completion of the completion phase and is only then able to cause more actions.

This decomposition allows me to fully explain the action of a synchronizing controller as follows.

The synchronizing controller has one active function: it schedules operations to be performed on its associated data object. The synchronizing controller is an event driven component, with the following two significant events:

1) An operation to be performed on the synchronizing controller associated data object entering its scheduling phase,

2) An operation that is being performed on the synchronizing controllers associated data object entering its completion phase.

In the first event the operation will be immediately scheduled if and only if no other operation is currently scheduled or executing on the synchronizing controllers associated data object and the performance of the operation would not violate the sequence of operations the synchronizing controller is trying to achieve.

In the second event the synchronizing controller will schedule one of the operations pending on its associated data object that is currently allowed in the synchronizing controllers prescribed sequence of operations, if there are any such operations.

Note that only one operation will be scheduled or executing at a time under the above rules.

Now let me return to the output device example and show how a synchronizing controller can be used to insure the specified synchronization property.

Figure 6a shows the two process version of this example which retains the rights controllers (drawn in rectangles) developed earlier plus a synchronizing controller (drawn in a triangle).

Consider how this system would work. Initially I assume there are no operations pending (waiting to be scheduled), scheduled or executing on the output device (as indicated in Figure 6a). Now suppose Process 1 attempts to perform an OPEN operation. Since there are no other operations scheduled or executing on the output device and since OPEN is currently allowed by the synchronizing controller, the OPEN operation would be immediately scheduled, thus allowing it to execute and complete. This results in the situation shown in Figure 6b. In this situation Process 1 can perform either a WRITE operation or a CLOSE operation, either of which would be immediately scheduled and allowed to execute and complete. However, Process 2 can only perforn an OPEN operation which would not be scheduled since OPEN is not currently allowed in the synchronizing controller's prescribed sequence of operations. Thus, if Process 2 performs an OPEN operation, it (the process) will be suspended until a CLOSE operation is performed by Process 1. Figure 6c shows the semetric situation where Process 2 has gained control of the output device. In fact, Figures 6a, 6b and 6c show the only three situations that are possible in this simple example. Thus, it is quite clear that no matter how many processes shared the output device, the proper synchronization property would hold as long as each process had a rights controller equivalent to the ones described in these figures.

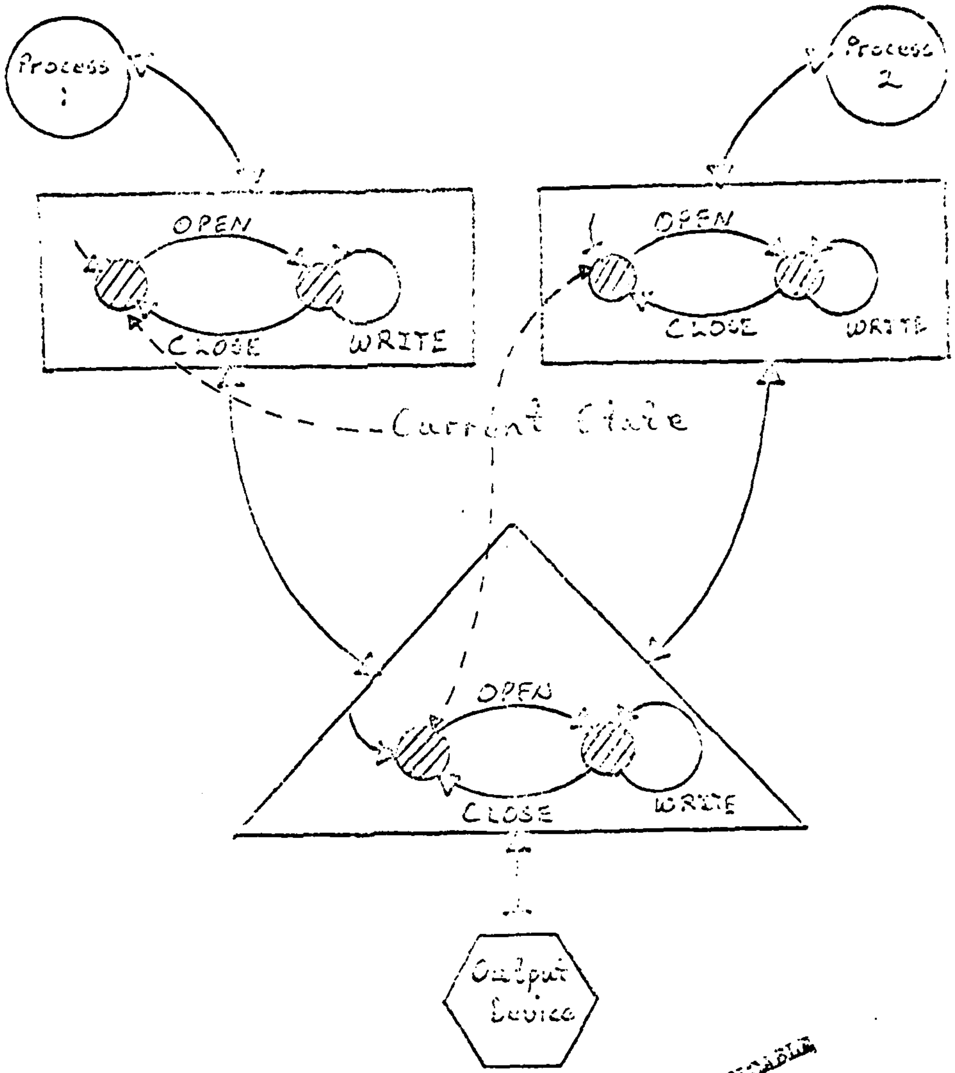

Process 1
Process 2
OPEN
CLOSE
WRITE
OPEN
CLOSE
WRITE
Current State
OPEN
CLOSE
WRITE
Output Device

FIGURE 6a

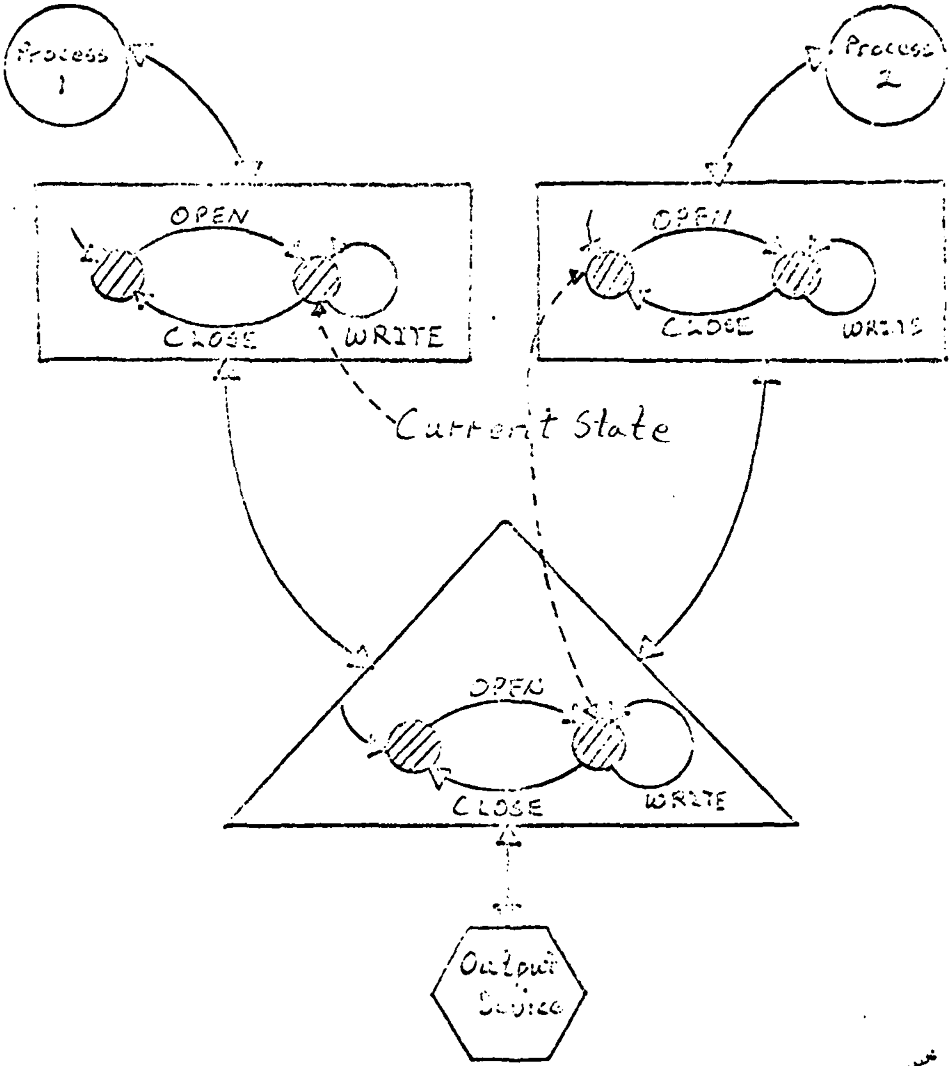

Process 1
Process 2
OPEN
CLOSE
WRITE
OPEN
CLOSE
WRITE
Current State
OPEN
CLOSE
WRITE
Output Device

FIGURE 6b

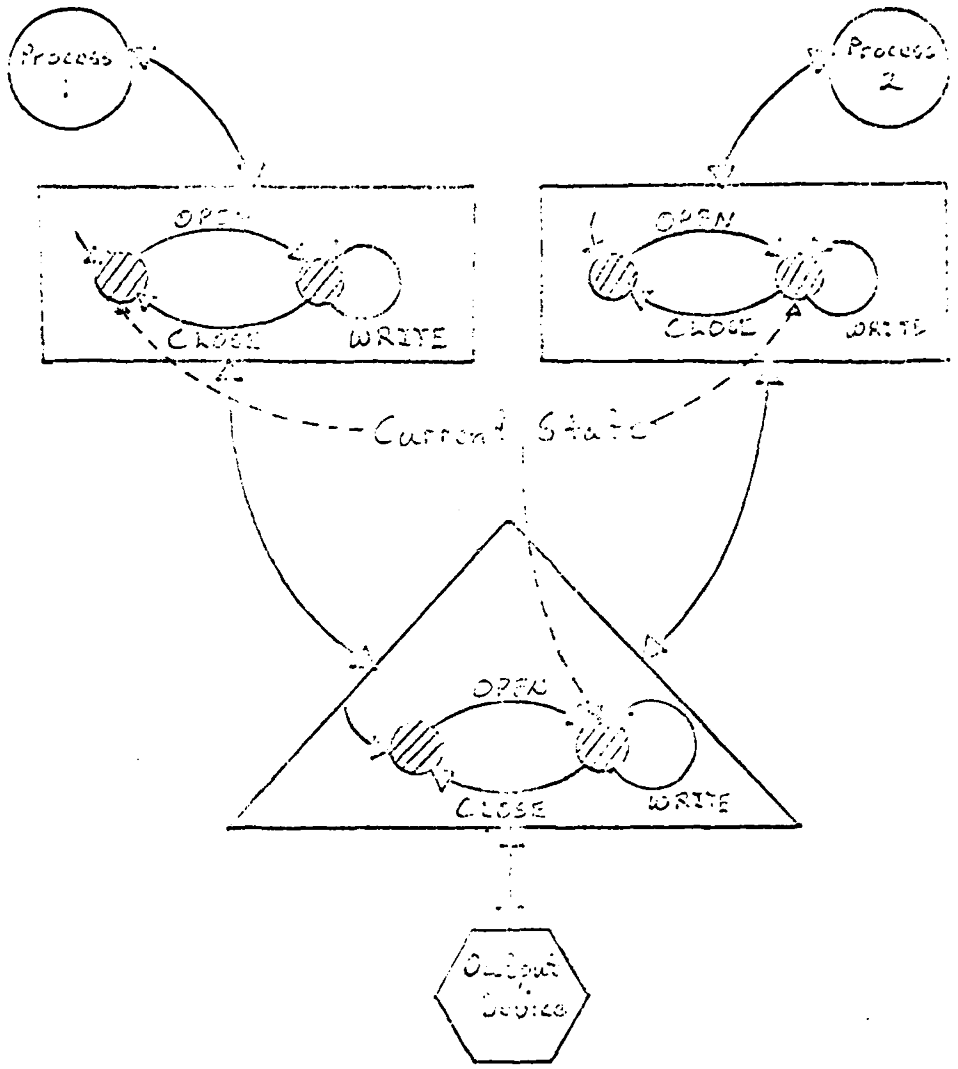

Process 1
Process 2
OPEN
CLOSE
WRITE
OPEN
CLOSE
WRITE
Current State
OPEN
CLOSE
WRITE
Output Device

FIGURE 6c

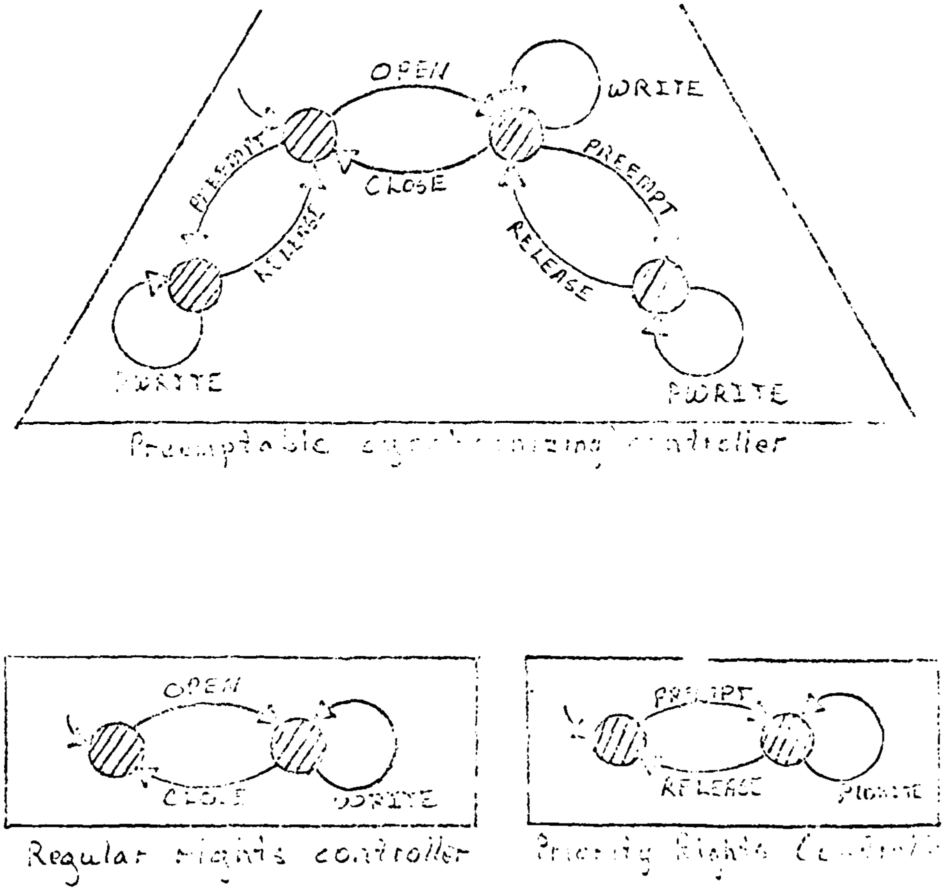
Thus, by the combination of rights controllers and synchronizing controllers, I am able to insure both of the properties concerning the sharing of the output device. Note that the synchronizing controller by itself would not have insured the proper synchronization property. For example, if the process were able to perform the operations in any sequence, then Process 1 might have performed an OPEN operation, after which any process in the system could perform WRITE or CLOSE operations because the synchronizing controller is not concerned with which process is performing the operations.

I would now like to consider some extensions to the output example which I believe will help to show how truely flexible these behavior controllers are.

Let us suppose that our output device is used for messages to the machine operator as well as user output. Now, suppose some operator messages need to be output immediately, i.e., before the end of some user output task. Figure 7 shows how the synchronizing controller for the output device might be modified to allow processes that have the proper "rights" to "preempt" the output device from another process. In Figure 7, I also describe the two reasonable rights controllers to go along with the amended synchronizing controller. Figure 8 shows how these rights controllers might be distributed in a three process system. In a system with such controllers, no matter what state the synchronizing controller is in due to a process with "regular rights", a process with "priority rights" can perform a PREEMPT operation. This will put the synchronizing controller in a state where only PWRITE and RELEASE operations may be scheduled, thus effectively preempting the output device. However, among processes having the "priority rights" preemption cannot occur.

Note that the change to the synchronizing controller and the addition of the new rights controller would not require any changes to the processes that continued to use the "regular" rights controller.

Let me continue to add complexity to this example by suggesting that



Preemptable synchronizing controller

Regular rights controller

Priority Rights Controller

FIGURE 7

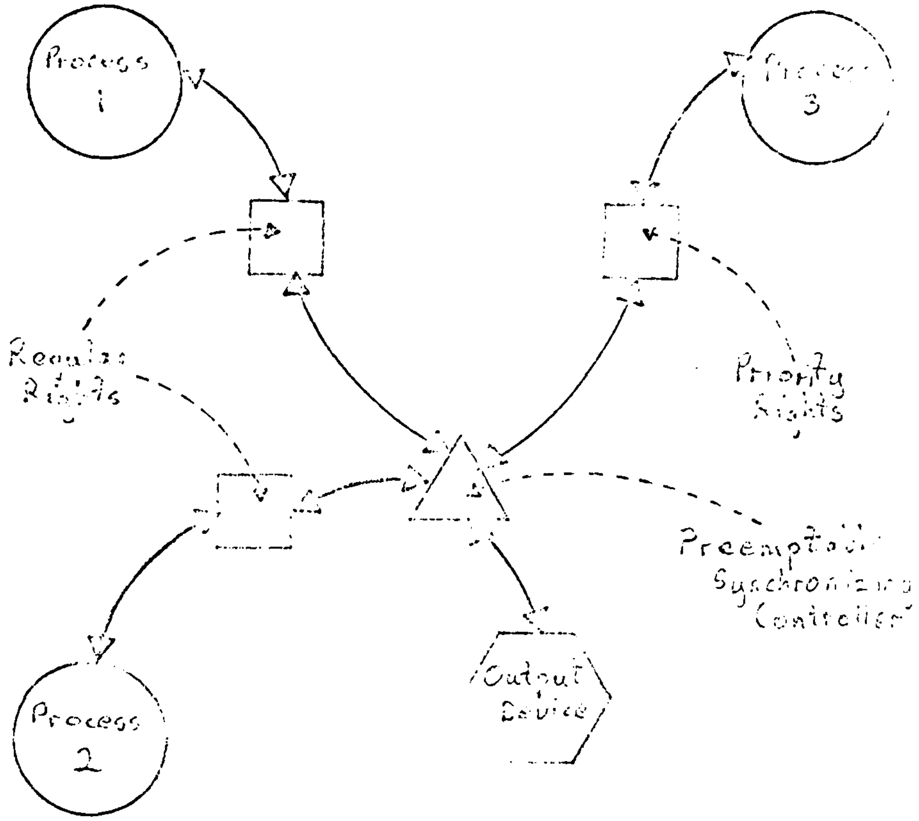

Process 1
Process 3
Regular Rights
Priority Rights
Preemptable Synchronizing Controller
Output Device
Process 2

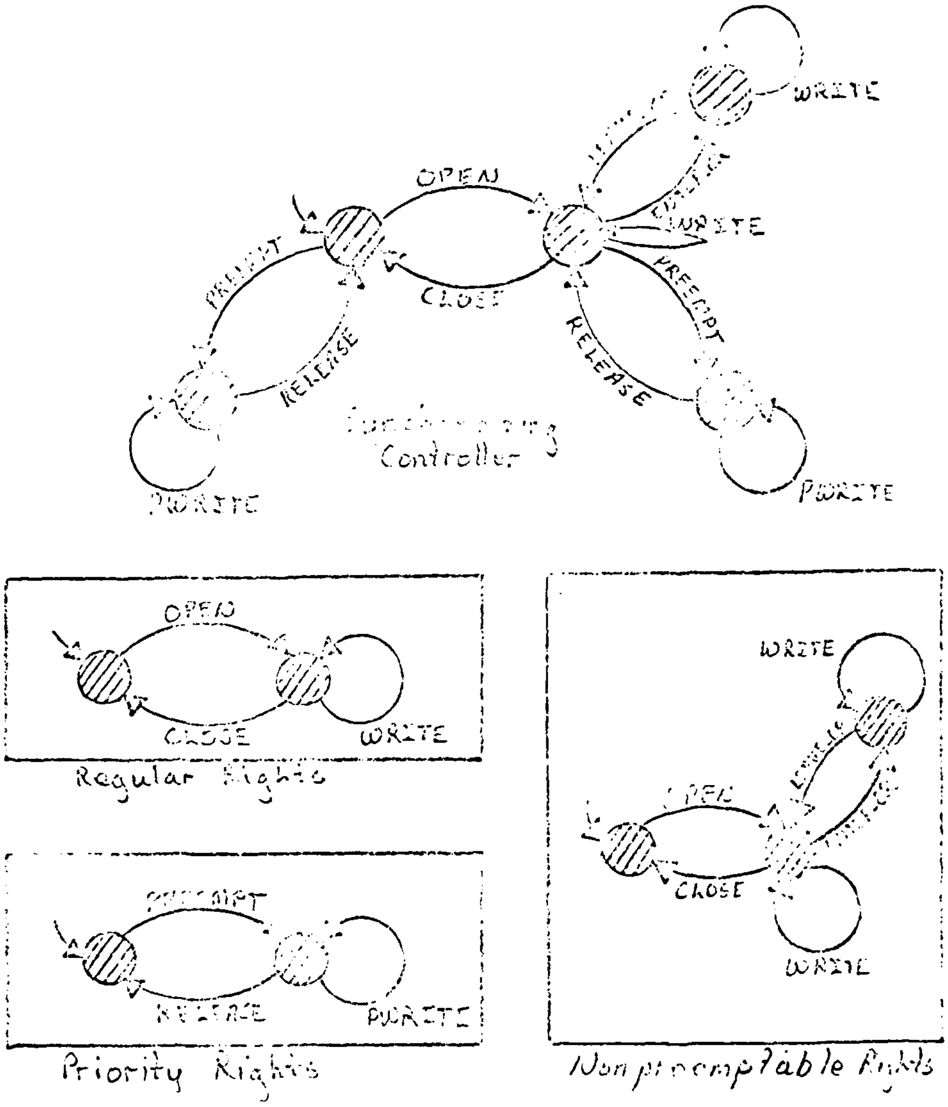
FIGURE 8

after our hypothetical system has been in use for some time, one of the system users might come in with the complaint that his output has operator messages in it. Now suppose that this user's output involves the use of expensive registered forms (e.g., payroll checks) and the system manager decides to protect the user from preemption.

Figure 9 shows the set of controllers that could be used to effect this change. Note that the "regular" and "priority" rights controllers are unchanged, thus no changes would be required in the processes which continued to use them. The change is simply to add a nonpreemptable state to the synchronizing controller along with operations to effect the transition into and out of this state. Note that the "nonpreemptable" rights controller still requires the OPEN operation first. Thus, the processes with this rights controller must still wait their turn for initial access to the output device. That is, it was not necessary to give these processes any special rights except the ability to prevent preemption during critical parts of their output.

I think that this solution compares very favorably t. a more traditional solution involving conventions over semaphores or such. I find especially impressive the way one is able to modify the constraints concerning the sharing of a data object without affecting those processes which do not wish to take advantage of the new features.

In summary, I have presented a model which I called the structured environment. In this model processes may only interact via the performance of operations of shared data objects. I refer to this interaction as the behavior of the processes and have shown that two types of behavioral constraints, rights controllers and synchronizing controllers, can be used to usefully control the interaction of the processes in a system. Some of the benefits that I feel arise from this approach to concurrent process control are listed below:



Synchronizing Controller

Regular Rights

Priority Rights

Nonpreemptable Rights

Figure III-9

- Simpler context for verification: Certainly the restrictions on process interaction along with the external behavior controllers makes the verification of certain properties much simpler than it would be in a model that required one to examine the definition of each process.
- Localized scheduling of process: All scheduling in this model occurs in the event driven synchronizing controllers. This seems to be a much simpler concept to implement than say a system involving conditional critical regions or predicate locks.
- Greater reliability through external constraints: Since the constraint placed on a process by its rights controllers is independent of the definition of the process, it should be straightforward to implement a run-time check to enforce the rights controllers. Thus, this insures that even in a system with incorrect processes, errors would not propagate.

APPENDIX 5

SOME THOUGHTS ON AUTOMATIC THEOREM PROVING IN DATA BASE DESIGN AND USE

by

W. W. Bledsoe

The paper sketches some of the ways in which research in Automatic Theorem Proving (ATP) can support the interdisciplinary project on Data Base Methodology being conducted at The University of Texas.

DATA BASES

Here we treat a data base as a list of facts and information (which might be distributed over several geographic locations), along with a set of rules of inference for using these facts. (Figure 1)

LIST OF FACTS
(DISTRIBUTED)
INFERENCE RULES

Figure 1
A Data Base

Querries to this data base are processed by

a) Direct lookup

b) By Inference

Also the data base must be tested somehow for internal consistency.

For example, if we have the statements

1) John is older than Mary

2) Mary is 15 years old

in the base, we want to answer querries such as

a) Is Mary 15 years old?

b) Is Mary older than 25?

c) Is John older than 12?

The last two, of course, would require simple inference. Much more complicated cases are desirable and, to some extent, possible.

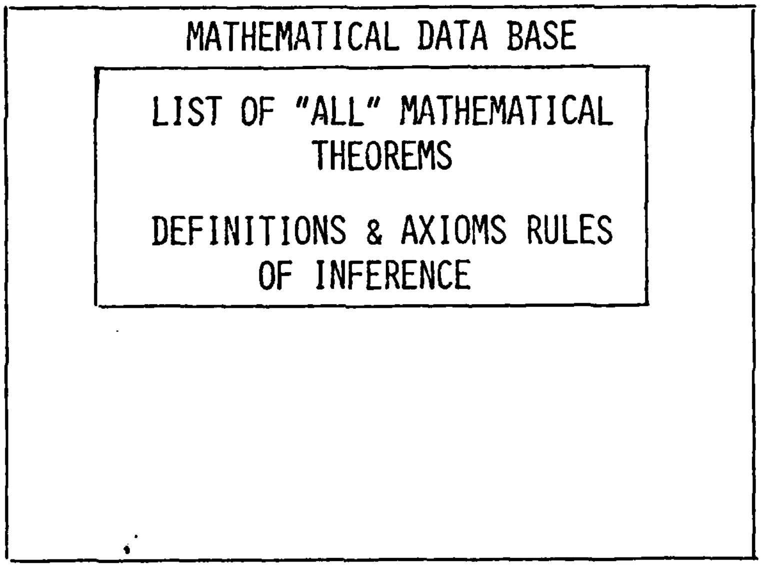
If we add the entry

3) Mary is younger than 4 years,

what does the whole thing mean? What, if anything, can it be use for?

EXAMPLES

For example we might have a mathematical Data base (Figure 2)



MATHEMATICAL DATA BASE

LIST OF "ALL" MATHEMATICAL THEOREMS

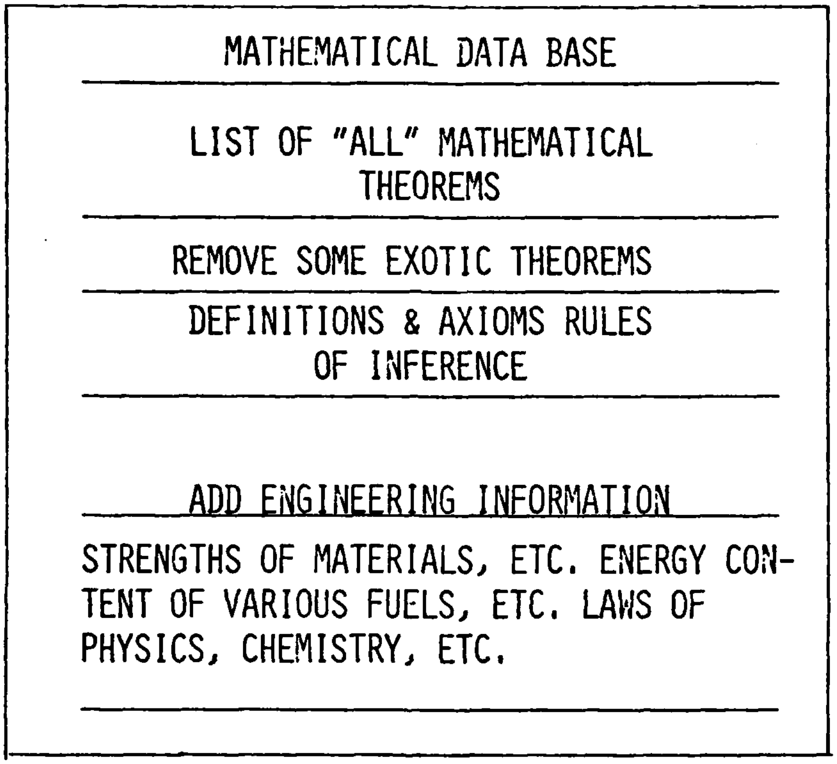
DEFINITIONS & AXIOMS RULES OF INFERENCE

A. PROVE A THEOREM BY FINDING IT ALREADY IN THE DATA BASE.

B. PROVE A THEOREM BY INFERRING IT FROM THEOREMS IN THE DATA BASE.

Figure 2

We can extend this to an engineering data base (Figure 3).



MATHEMATICAL DATA BASE

LIST OF "ALL" MATHEMATICAL THEOREMS

REMOVE SOME EXOTIC THEOREMS

DEFINITIONS & AXIOMS RULES OF INFERENCE

ADD ENGINEERING INFORMATION

STRENGTHS OF MATERIALS, ETC. ENERGY CONTENT OF VARIOUS FUELS, ETC. LAWS OF PHYSICS, CHEMISTRY, ETC.

A. PROVE A THEOREM BY FINDING IT ALREADY IN THE DATA BASE.

B. PROVE A THEOREM BY INFERRING IT FROM THEOREMS IN THE DATA BASE.

C. ANSWER QUESTION ABOUT THE DESIGN OF A BRIDGE OR THE FEASIBILITY OF A SPACE PROBE.

Figure 3

Or extend it to a military data base (Figure 4).

MATHEMATICAL DATA BASE

LIST OF "ALL" MATHEMATICAL
THEOREMS

DEFINITIONS & AXIOMS
RULES OF INFERENCE

ADD MILITARY INFORMATION
DISPOSITION OF OUR FORCES AND EQUIPMENT
STATE OF READINESS, ETC.
DISPOSITION OF ENEMY FORCES
CONTINGENCY OF INFERENCING

A. Prove a theorem by finding it already in the Data Base.

B. Prove a theorem by inferring it from theorems in the Data Base.

C. Answer a question on our ability to repulse a conjectured attack.

Figure 4

Many other examples easily come to mind.

SOME OBSERVATIONS

Several points can be made

- AUTOMATIC INFERENCING IS CLEARLY DESIRABLE IN ALL SUCH EXAMPLES.
- IF WE HAD TRULY POWERFUL AUTOMATIC THEOREM PROVERS, IT WOULD CHANGE OUR CURRENT PROPOSAL FOR DATA BASE DESIGN.
- FOR THE NEXT 10-30 YEARS WE MUST SETTLE FOR A "MODERATE" ABILITY OF ATPS, BUT EVENTUALLY ATP WILL DOMINATE.
- THE PRESENT PROJECT SHOULD USE THIS MODERATE CAPACITY, USING AUTOMATIC INFERENCING BUT NOT EXPECTING TOO MUCH.

ATP AT THE UNIVERSITY OF TEXAS

The University of Texas has been one of the leading centers for ATP since 1968, and is the most successful in actually carrying out proofs of moderately difficult theorems on the computer. Our provers have been LISP programs for the CDC 6600 computer and the DEC 10. In addition to a number of theorems proved in set theory[1], calculus[2], analysis[3,7,9], and topology[3,9], we have seen our program and ideas successfully used in program verification systems [5,6], and in incremental design of programs with documentation and verification [10]. Also we have been early propouents of new directions [9] now finding their way in ATP research.

Others at UT (Skilossy, Simmons, Chester, and other students) have been or are now engaged in some form of ATP research. The research on inference in semantic net [11] seems especially pertinent here.

ROLE IN DATA BASE DESIGN

We would expect to use the concepts from ATP, not the actual programs in data base design. A team effort would insure that ATP ideas would be integrated into the project in an effective way.

The research would blend nicely with a larger effort here, funded by NSF, on general ATP.

At this time we feel that inferencing (in Data bases) can best be done at two levels

1) At the hardware level (simple inferences)

2) At the software level.

In 2) certain "pertinent" information is retrieved from the data base ("semantic paguig") to be used in core memory for deeper inferencing.

We believe that only a moderate capability in ATP can be depended upon during the next 10-30 years. However in the long run ATP will be the dominant factor in Data Base design. It is crucial that ATP research, geared to that application, (here and elsewhere) be supported in the interim.

Two important factors in data base design are

a) conflicting data

b) changing data

These seem to point more toward

1) automatic inferencing, and

2) man-machine cooperation.

LONG-TERM RESEARCH INTEREST

Our group here has a long-term interest in deep inference in data bases, where a sizable ATP capacity is required. We will be pursuing this interest independent of this project.

Included in our concerns are

- uncertain and conflicting knowledge
- predicting with probabilities
- (limited) natural language input and output
- man-machine interaction

EXAMPLE

DATA BASE

ALL THE NEWS PAPER STORIES ON THE MIDDLE EAST FILED BY THE MAJOR NEWS AGENCIES DURING THE LAST 10 YEARS. (WITH OLDER STORIES CAREFULLY CULLED)

+ RULES FOR INFERENCING

QUERRY:

WHAT IS THE LIKELIHOOD OF SYRIA ATTACKING ISRAEL WITHIN THE NEXT TWO DAYS?

TASK:

DETERMINE THE SOURCE OF THE MAJOR INCONSISTENCIES IN THE DATA BASE.

We could add to this

INTELLIGENCE INFORMATION ON

- TROOP STRENGTHS AND DEPLOYMENT
- RECENT MOVEMENT

ETC.

MAN-MACHINE INTERACTION

1. THE USER WOULD ADD, SUBTRACT, OR CHANGE, DATA AND INFERENCING RULES.

2. THE USER COULD HELP WITH THE INFERENCING ON DIFFICULT PROBLEMS (E.G., SUGGESTING RELEVANT FACTS).

REFERENCES

1. W. W. Bledsoe, Splitting and reduction heuristics in automatic theorem proving. A. I. Jour. 2(1971), 55-71.

2. W. W. Bledsoe, Robert S. Boyer and William H. Henneman, Computer Proofs of Limits Theorems. A. I. Jour. 3(1972), 27-60.

3. W. W. Bledsoe and P. Bruell, A man-machine theorem-proving system. A. I. Jour. 5(1974), 51-72.

4. W. W. Bledsoe and Mabry Tyson, The UT Interactive Theorem Prover. The University of Texas at Austin, Math. Department Memo ATP-17, May 1975.

5. D. I. Good, R. L. London and W. W. Bledsoe, A Complete Method for Higher Order Logic. Ph.D. thesis, Case Western Reserve Univ. Jennings Computer Center Report 1117.

6. W. W. Bledsoe and Mabry Tyson, Typing and Proof by Cases in Program Verification. Machine Intelligence 8, Donald Michie and E. W. Elcock (Eds.), Ellis Horwood Limited, Chichester, pp. 30-51.

7. A. Michael Ballantyne and W. W. Bledsoe, Automatic Proofs of Theorems in Analysis Using non-standard Techniques. J. ACM, Vol. 24(1977) pp. 353-374.

8. W. W. Bledsoe, Non-resolution Theorem Proving. University of Texas Math. Department Memo ATP-29, Sept. 1975. To appear in the A. I. Jour.

9. W. W. Bledsoe, A Maximal Method for Set Variables in Automatic Theorem Proving. University of Texas Math. Department Memo ATP-33A, July 1977. To be presented at IJCAI-77, MIT, Aug. 1977.

10. Mark Moriconi, An Interactive System for Incremental Program Design and Verification

11. D. Chester and R. F. Simmons, Influences in Quantified Semantic Networks.

Appendix 6

A COMPUTER ARCHITECTURE FOR A FDSS

Jack Lipovski

Computer architecture aims to make recent advances in hardware technology (especially LSI) useful to the new and demanding software envisioned for a very large distributed and intelligent data base management system. Some preliminary architectural features of a planned system are herein sketched and some problems for research and development are delineated.

Three major computing systems are to be accomodated. Firstly, users interface with the data base system through a network of intelligent terminals. Secondly, intelligent discs are located at various nodes in this network and are powerful enough to search the data where it is stored to avoid shipping large quantities of data through the network. Thirdly, an array computer will use parallelism to extend the analytical capacity of artificially intelligent software. We submit that these three major systems have to be accomodated because none of them alone, nor any pair of them, are adequate to support the envisioned software. However, each system can be effectively and economically built with LSI modules, so the total system will take advantage of LSI economics. We aim to design each system so that it will interface well with the other systems and, conversely, we are relieved of the need to perform each function in any one system alone. The object of studying the three systems together is to develop each one of them to fit together later on in an integral system. While we do not propose to build all of them in this project, but only the intelligent disc, we will design the disc to support distributed queries in the network and to support deep theorem proving in the array computer. Other proposals have been or will be submitted to study the other systems. If the other proposals are funded, they will be used in research conducted in this proposal. Otherwise, they will be simulated in this proposed research effort.

Moreover, each system will be designed to work as I/O devices with existing computers to provide considerable improvement in their performance, even though the greatest improvement in performance can only be expected from using all three computing systems together in a total system.

In the following paragraphs, we outline the three systems. The intelligent disc, which we plan to build, will be discussed in more detail. The other two will be sketched for completeness.

1. Other Systems Architectures

1.1 The Network

The network will consist of small microcomputers in intelligent terminals and intelligent secondary memories and communication will be accomplished by packet switching in the network. Although the terminals deserve some study, we need not specify them at this stage except to say that they have to be able to maintain the user's schema, a compiler for the data base language, and means to direct packets, embodying the query, through the network to the intelligent discs. Upon sending out a query from an intelligent terminal, the object of a packet will generally be a file on an intelligent disc. The file will be explained in the next section. A group of files will be at one physical node of the network of different cylinders of the disc, or even in tertiary memory in that node. There may be several physical nodes distributed through the network. In processing a complex query, references from one file to another will require that packets be sent from files to files as well. In retrieving the answer to a query, packets will be sent out from files to intelligent terminals.

This architecture requires that the intelligent disc node be able to examine an incoming packet to determine which file is to be operated on. A queue of incoming packets will be buffered and scheduled by a conventional microcomputer associated with the disc at the node. Records will be checked for locking to prevent interference among queues. Once a file is in position to be searched by the logic in the intelligent disc, and all required records have been locked to the user, the file will be entirely searched in each disc revolution, as discussed in the next section.

Of significance, this network architecture combines the problem of accessing file data from intelligent terminals with the problem of solving complex queries where one file has to be linked up with other files. It offers hope in simplifying problems of protection, lockout and deadlock by locking records within the intelligent disc.

1.2 The Array Processor

The array processor will be used to support artificially inteligent software by means of parallelism. Two forms of parallelism are useful. In vector parallelism, a very wide word width processor is created by work on vector operands. This is commonly referred to as single instruction stream multiple data stream (SIMD) processing. In concurrent parallelism, small independent processors simultaneously but independently operate on separate pieces of data. This is referred to as multiple instruction stream multiple data stream (MIMD) processing.

In artificial intelligence programs using LISP, lists can be "vectorized" by writing them as paranthesized strings. Operations like EQUAL can be executed on two strings as though they were vectors. Operations like CDR or CAR can be done in a parallel machine as simply as in a conventional machine, but no better. However COND and MAP do not take much advantage from vector parallelism.

In a concurrent machine, each independent processor can evaluate different lists using standard LISP techniques. Potentially, all

LISP primitives can be executed faster through parallelism. In order not to have to store the entire LISP interpreter in each memory, a set of common memories store fragments of the interpreter. Each processor can fetch instructions from one of the common memories, but all processors accessing any one common memory must be accessing exactly the same word in it. With a fixed prοgram like a LISP interpreter, we believe it will be possible to carefully schedule fragments into common memories to use this technique. Then very small, cheap processors with a small amount of local memory should be able to efficiently execute concurrent LISP programs.

The key to both vector and concurrent parallelism is the design of a powerful but inexpensive computer switching array. We have submitted a proposal to NSF to build a prototype computer using such a switch. This computer can be used to experiment with concurrent and vector parallelism in executing artificial intelligence programs.

2. Intelligent Disc Architecture

From our earlier work on the CASSM system at the University of Florida and from related work on the RAP system at the University of Toronto, we have established techniques which will efficiently store relational data bases and semantic networks on a disc. The logic associated with the disc makes it sufficiently intelligent to resolve almost all typical relational queries and sufficiently intelligent to greatly assist extracting useful data from a semantic network for artificial intelligence programs.

2.1 Physical Description of Disc Hardware

The disc architecture will consist of multiple moving head discs, in which all heads are on a common frame, and there is one head on each disc surface. (We are looking at IBM 3330 or equivalent discs that store about 109 bits per removable disc pack). By moving the frame, the heads are located over a given "cylinder". One or more such discs will be operated together so that their "cylinders" form a larger cylinder. The data on this larger cylinder is called here a file. For instance, if three IBM 3330 discs are operated together, a larger cylinder would have 60 heads on 60 surfaces. Two hundred such files are stored on the 200 cylinders of an IBM 3330 disc. More files may be stored in tertiary memory, and paged into cylinders of the disc. Exactly one file will be under the moving disc heads at any time.

In one revolution of the discs, an "instruction" is executed on the entire file. A typical query consists of in the order of ten instructions which will be executed on the same file. Upon receipt of a query at an intelligent disc by the microprocessor that controls the disc, the file requested by the query is positioned under the heads, either by moving the heads or by moving the file in from tertiary memory. The heads remain positioned over the file as the disc revolves, to execute the query, for about ten revolutions. The heads are then positioned over the file needed by the next query. Each head will have a "microprocessor" similar in complexity to the popular microprocessors but having quite different organization and instruction set. It will be attractive to put each "microprocessor" in an LSI chip. A disc track and "microprocessor" are called here a cell. The logic looks like a chain of identical cells. See figure 1.

2.2 Storage of Data

The file consists of records of a variable number of words, and the words are fixed length and are divided into fields. Each word has a mark bit on the disc for content search operations. Records correspond to tuples in the relational data base system and to nodes in semantic networks. The first word of each record stores a bit stack for context search operations. Other words appear to store domain names and items in the tuple, or arcs incident from the node in the network. See figure 2 for storage of relational tables and figure 3 for storage of (semantic)networks. Figure 2a shows two relations while figure 2b shows their storage on the disc. Figure 3a shows a network like a semantic network, while figure 3b shows storage of the network on the disc.

Note, in figure 2b that a file can contain many relations (tables) and that each tuple (row) is stored as a record. In this figure, two relations (officer and parts) happen to be stored on the same file. Note that the tuples from different relations can be intermixed, but that a field in the first word in each tuple or record identifies the relation that the tuple is in by a code word. For each domain (column), a word containing a pair of code words in fields, domain name and domain value, are stored.

Note, in figure 3b that each node is stored as a record and records are numbered according to their position from top to bottom in the file. For instance, node "Tom" is stored in the 21st record from the top of the file. For each node, its corresponding record contains in its first word a field containing the code word of the node name and in succeeding words a pair of fields associated with each are in the network that is incident out of the node. The fields are best explained by example. For instance, in record 20 corresponding to the node "John", the arc (John, father, Tom) is represented by the word (father, 21) where "father" is a code word and 21 is a number, since 21 is the record number for the node "Tom".

Though not fully shown in the above examples, the key problem is efficient storage of data. That is why code words are used rather than character strings. A mechanism to convert between code words and character strings by means of hardware has been worked out. Moreover, the left field of each word can be generated by means of a counter in the "microprocessor" associated with the disc track rather than stored on it if the code words are consecutively numbered. These fields that are generated in the "microprocessor" are called imaginary fields. The user need not concern himself about whether data is stored in real or imaginary fields, for instructions will treat the files as shown in figures 2b or 3b, whether or not some left fields are actually generated by hardware.

2.3 Content Searching

Each word is provided with a bit (mark bit) which can be modified by a content search instruction. The bit is set if the content of the word matches the argument of the instruction, and is cleared otherwise. For instance, in figure 2b, if the operand of a content search were P#, 30, then the mark bit of the eighth word would be set and all other mark bits would be cleared. Content searching is normally used to single out individual words to be rewritten, output, or deleted.

2.4 Set Oriented Context Searching for Relational Data Bases

Each record is provided with a bit stack located in the first word of the record. If the argument of a context search "push", instruction is in a record, a 1 bit is pushed on the bit stack for that record, else a 0 bit is pushed on the bit stack. A context search instruction could "AND" the result of the search with the top bit on each bit stack, or "OR" or "AND the COMPLEMENT", etc.

Consider a query to locate Captain Smith in figure 2. The query is translated into the following program:

```
1)  PUSH            "IS-AN"        "OFFICER"
2)  AND             "RANK"         "CAPTAIN"
3)  AND             "NAME"         "SMITH"
4)  MARK(OUTPUT)    "LOCATION",     -------
```

Instruction 1 pushes a 1 onto the bit stacks for the first and third records, and a 0 onto that of the second record. Instruction 2 AND's a 1 bit onto the bit stack of the first record, but AND's a 0 onto the other bit stacks. Instruction 3 does the same (in this simple example). Instruction 4 marks, by content searching within records that have a 1 bit on top of their bit stacks, the words whose left fields are "LOCATION". The marked words are output as the response to this query. This query is effectively answered in four disc revolutions. Typical queries should be answered in ten disc revolutions, independent of the size or complexity of the file.

In hardware, the results of the search are stored temporarily in a one bit wide random access memory, which has 1 bit per record, and are processed by pipelining to appear to move the results to the bit stack so that one context search can be executed each disc revolution. Complex Boolean queries can be analyzed over all tuples in a file in a number of revolutions proportional to the number of terms in the query expression and independent of the size of the file. (No conventional data base management system can approach this ideal.) Moreover, there is no need for directories to locate relations within a file and tuples from a relation can be scattered throughout the file because the entire file is searched each disc revolution.

2.5 Other Set-Oriented Functions

It is possible to find the intersection of two sets in two disc revolutions. The one bit wide, RAM (mentioned in section 2.4) is initially cleared. In the first revolution, elements (code words) of the first set provide addresses to set bits of the RAM. In the second revolution, elements (code words) of the second set provide addresses to read bits from the RAM. If a 1 is read, the word of the second set is marked. Only if an element is in the intersection will that bit be both set and read, and the word marked. (Other researchers have also shown that duplicates can be deleted by a similar procedure.)

It is possible to execute an inner product "threshold search" as shown in figure 4. Each word on the disc has an associated weight, as word A in record 23 has weight 3. The argument of the instruction also has a weight. The argument, its weight, and a storage buffer are in registers in each head. If the word matches the argument, the two weights are multiplied and saved in the buffer. The bottom word of each record contains an accumulator, the number in the buffer is added to it. Thus, an inner product "threshold" search can be conducted simultaneously over all records in a file.

The buffer can also be used for simpler functions. The maximum, minimum, sum or count of marked words can be conducted in each record or the entire file. In particular, after an inner product threshold search, the set with the maximum accumulator value can be marked. Equally important, the number of marked words can be counted before they are output, to determine whether there are too many to be of interest.

2.6 Network Oriented Context Search Instructions

Pointers from one record to another are stored by putting the record number of the second record in the right field of a word in the first record. See figure 3 again, where the "father" pointer from record 20 (for "John") points to record 21 (for "Tom"). The RAM discussed earlier is used to transfer tokens. The RAM is initially cleared. If the argument of a token transfer search is found in the left field of a word in a record having a 1 bit on the top of its stack, the right field is used as an address to set a bit in the RAM. In the following revolution, a counter that counts records as they pass over the head is used to address the RAM to push the values stored there onto the bit stacks of the records. Pipelining allows the second revolution to be "hidden" so that tokens can be effectively transferred in one disc revolution.

Consider a query to find the grandsons of "John" in figure 3. The instructions are:

```
1)  PUSH           "IS-A",     "John"
2)  PUSH           "FATHER",    TOKEN
3)  PUSH           "FATHER",    TOKEN
4)  MARK(OUTPUT)   "IS-A",      ──────
```

After the first revolution, a 1 is effectively pushed onto the bit stack of record 20, and a 0 is pushed onto all other stacks as discussed in section 2.4. After the second revolution, a 1 bit is pushed onto the bit stacks of records 21 and 23 simultaneously, and after the third, a 1 bit is pushed onto that of record 25. After the next revolution, the work "IS-A", "BILL" is output. Such a query is effectively executed in four revolutions.

One of the most useful applications of pointers and token transfers across pointers is semantic paging for deep theorem proving programs. See figure 5. Context addressing, threshold searching and so on can be used to select one or more nodes of a network. Then tokens can be transferred without regard to pointer names from these nodes in n layers, one layer per revolution, to mark a subgraph containing the selected nodes and all nodes up to n arcs distant from the node. The records so marked can then be paged into a parallel computer for analysis by a deep theorem proving program. Semantic paging should effectively filter the data to a small size subgraph that is manageable in a parallel computer, so that it can thoroughly analyze the subgraph at high speed.

2.7 Other Hardware Functions

The disc "microprocessor" will also collect garbage words by a hardware mechanism that operates concurrently with instructions that are evaluating a query. Also, data can be input and marked words can be output while instructions are processing a query. (Interlocks will be provided so that inputs or outputs from one query are not mixed with those from another.) Character string to code word translation is carried out automatically upon input and code word to character string translation is automatically carried out on output. Finally, disc processor instructions are to be stored on and fetched from the disc itself to manage "demons".

The disc is capable of storing a large number of "demons" by storing data words and instruction words in records. Data words are searched by context or by token transfer to activate instruction words in the records satisfying a query. The activated instructions are executed on the disc one at a time as they are deactivated. These concurrent hardware functions increase the performance of the intelligent disc, and make possible some new and possibly revolutionary software techniques.

3. Parallel Computation in Automatic Theorem Proving

There are several ways in which parallel computation might speed up an automatic prover.

A. Evaluating an And-node.

Whenever the prover is asked to prove a subgoal like

$$Vx(P(x)_{\wedge}Q(x))$$

one processor can be asked to prove $P(x)$ (ie., to find a value or values, for x that will satisfy this formula), and another processor can be asked to prove $Q(x)$. The answers from these two would then be reconciled (if possible) to obtain a common value (or values) or x satisfying by $A(x)$ and $B(x)$.

B. Evaluating an Or-node.

Whenever the prover is proving an or-node of the form $A_{\vee} B$, or is trying a list of possible strategies to obtain the proof of a given subgoal, a separate processor can be assigned to work each of A and B , or each of the strategies. The subgoal would be satisfied when one of these processors succeeded.

C. Simplification and Reduction.

Much of modern theorem proving involves rewriting a formula into a canonical form. For example, the formula $(1 + y - 5 + x)$ might be rewritten as $(x + y - 4)$, or the formula $(x\varepsilon\ A_{\cap}B)$ might be rewritten as $((x\varepsilon A)_{\wedge}(x\varepsilon A))$. Parallel processors could greatly speed up this kind of process.

Many of these examples of parallel computation can be handled by an extension of LISP which would allow a parallel COND. That is for the command

```
(COND
    ( P  A )
    ( Q  B )
    ( R  C ) ),
```

it would simultaneously calculate P, Q, and R, determine which was true, and return accordingly A, B, or C (or some function of them if more than one of P, Q, and R was true).

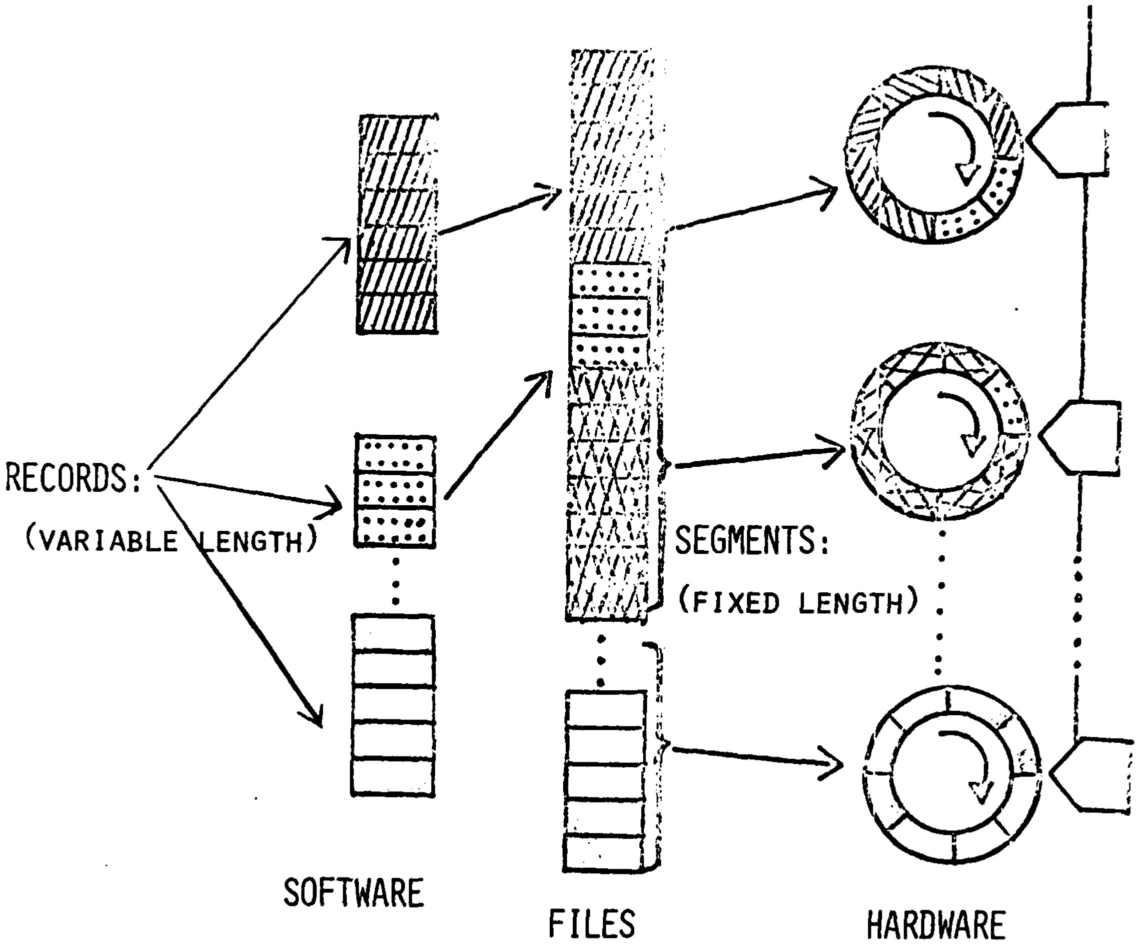
FIGURES

The figures are selected from the enclosed view graphs. Numbers are shown on the bottom left of each view graph.

The figures for this Appendix appear in the order in which they are listed.

Figure	Number of View Graph
1	7
2a	13b
2b	13a
3a	21a
3b	21b
4	17
5	22

INTELLIGENT DISC
(LOGICAL)


RECORDS:
(VARIABLE LENGTH)
SEGMENTS:
(FIXED LENGTH)
SOFTWARE
FILES
HARDWARE

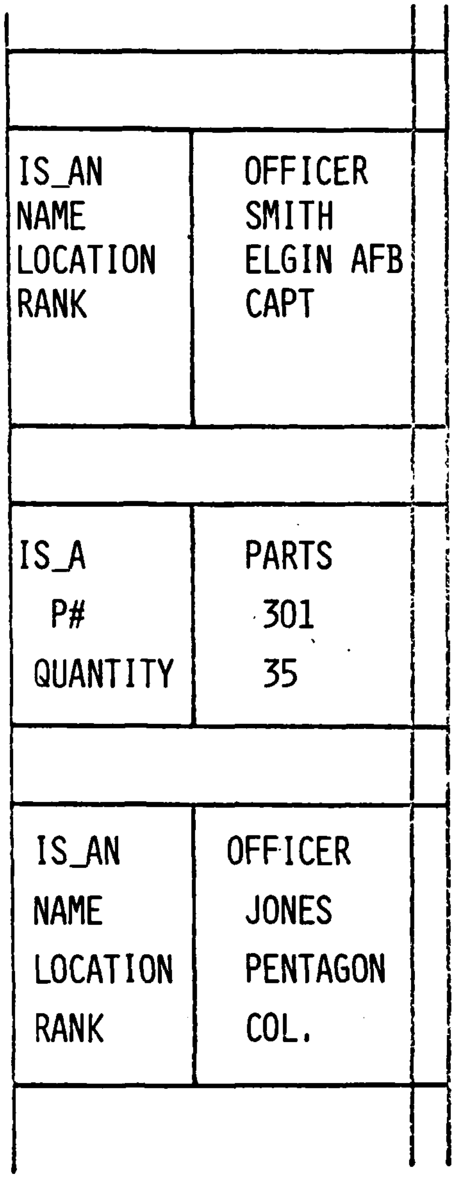
OFFICER

NAME	LOCATION	RANK
SMITH	ELGIN AFB	CAPT.
JONES	PENTAGON	COL.

PARTS

P#	QUANTITY
301	35

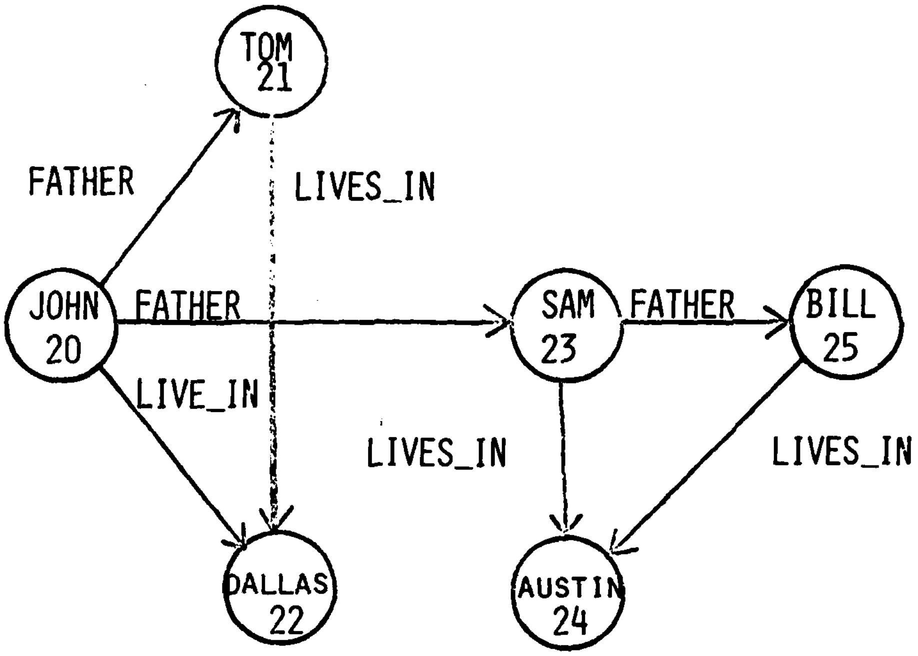
TUPLES ARE STORED AS RECORDS



IS_AN	OFFICER
NAME	SMITH
LOCATION	ELGIN AFB
RANK	CAPT

IS_A	PARTS
P#	301
QUANTITY	35

IS_AN	OFFICER
NAME	JONES
LOCATION	PENTAGON
RANK	COL.

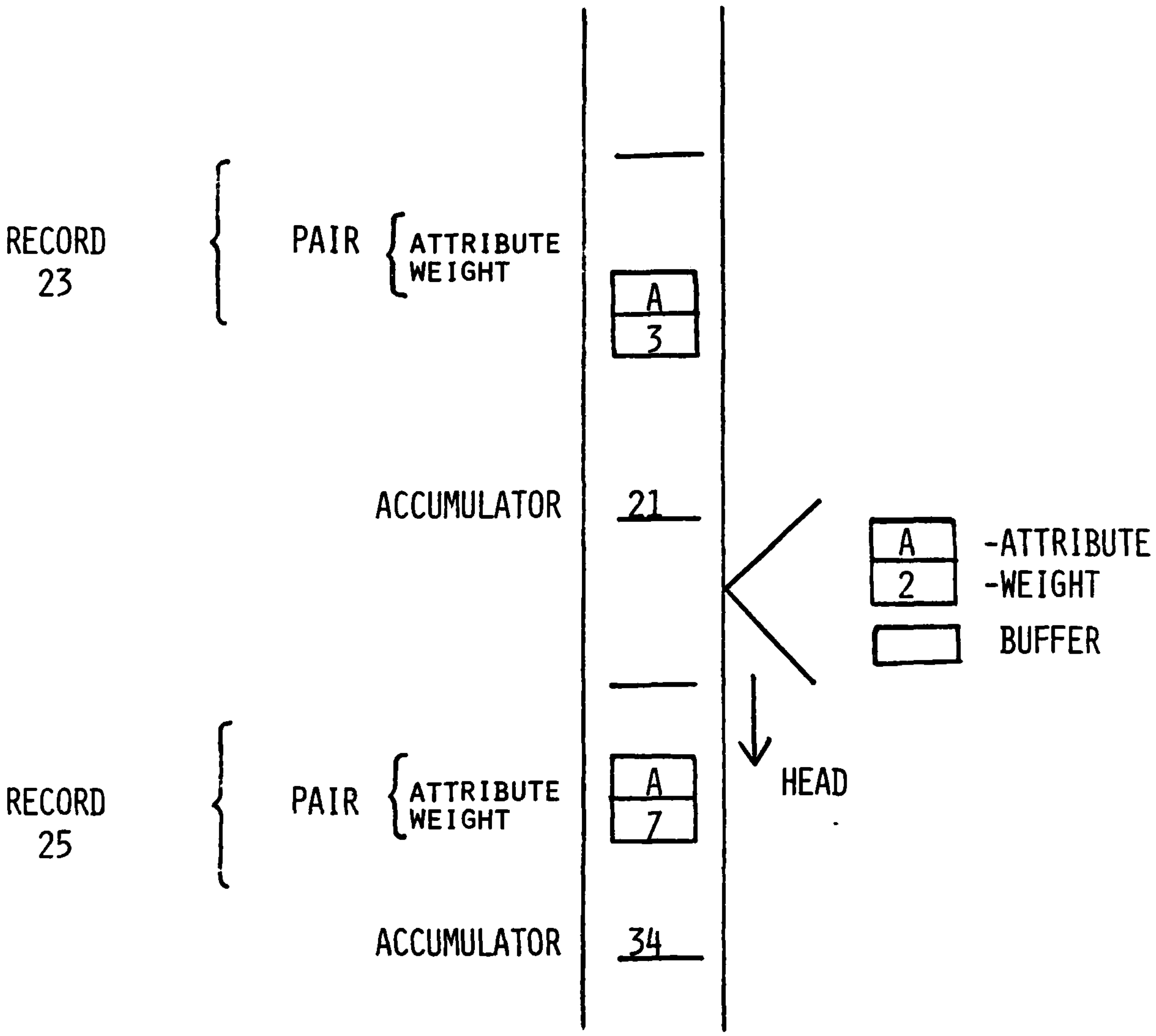

TOM
21
FATHER
LIVES_IN
JOHN
20
FATHER
SAM
23
FATHER
BILL
25
LIVE_IN
LIVES_IN
LIVES_IN
DALLAS
22
AUSTIN
24

20	IS_A JOHN FATHER 21 FATHER 23 LIVES_IN 22
21	IS_A TOM LIVES_IN 22
22	IS_A DALLAS
23	IS_A SAM FATHER 25 LIVES_IN 24
24	IS_A AUSTIN
25	IS_A BILL

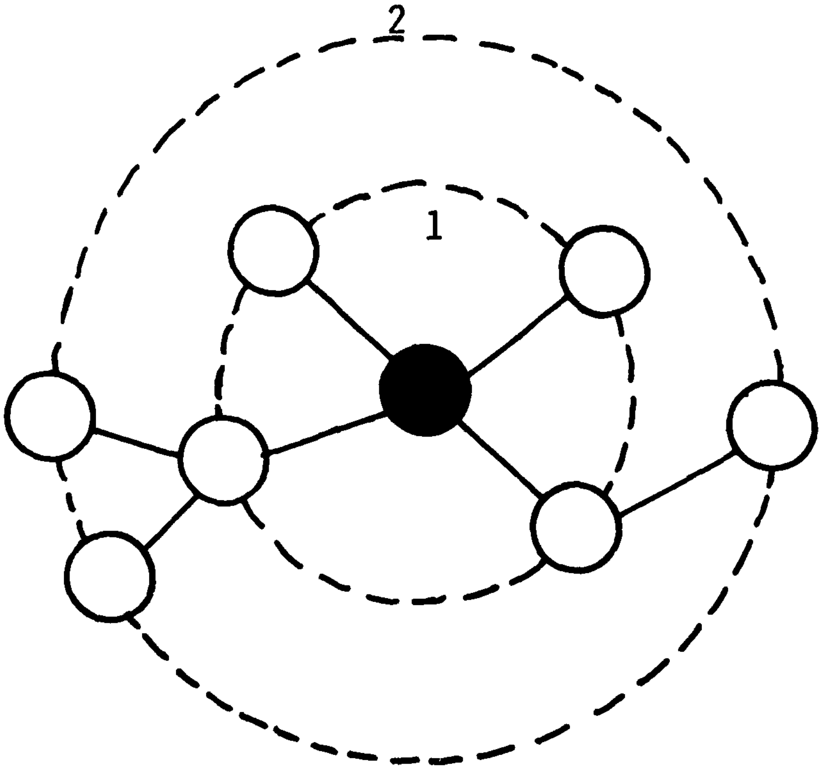
NODES ARE STORED AT RECORDS.

ARCS ARE STORED AS RECORD NUMBERS.

THRESHOLD FUNCTION SEARCH



SEMANTIC PAGING



1) SELECT NODE(S) BY CONTENT OR CONTEXT.

2) TRANSFER TOKENS OUT THROUGH ARCS N TIMES.

3) OUTPUT ALL NODES WHICH RECEIVED TOKENS.

FILMED
8