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

September 1977 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

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 example, 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 such 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

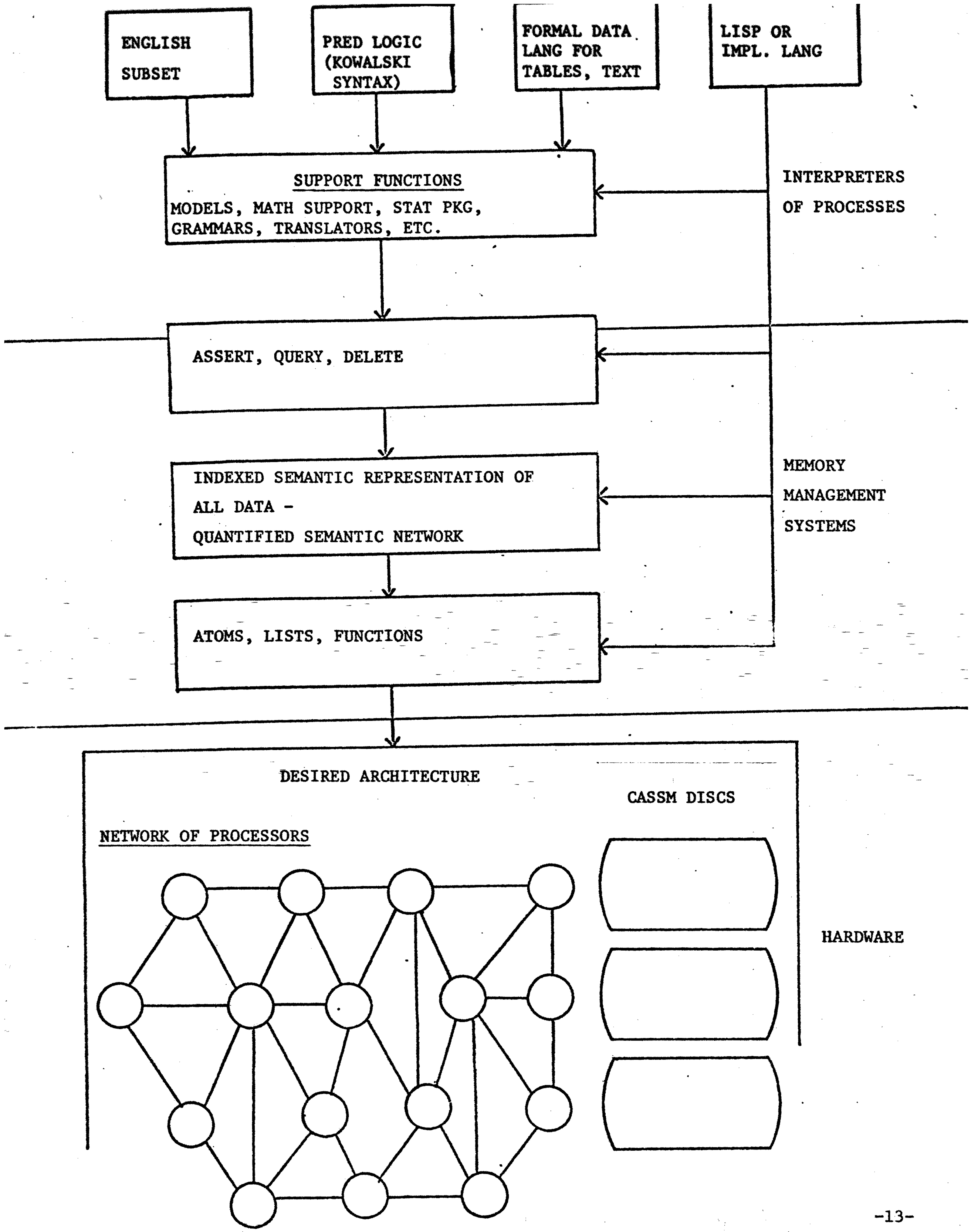

ENGLISH SUBSET
PRED LOGIC (KOWALSKI SYNTAX)
FORMAL DATA LANG FOR TABLES, TEXT
LISP OR IMPL. LANG
SUPPORT FUNCTIONS
MODELS, MATH SUPPORT, STAT PKG, GRAMMARS, TRANSLATORS, ETC.
INTERPRETERS OF PROCESSES
ASSERT, QUERY, DELETE
INDEXED SEMANTIC REPRESENTATION OF ALL DATA - QUANTIFIED SEMANTIC NETWORK
MEMORY 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) <u>Subsystem Performance: Competition and Interference in Distributed Systems</u>

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) <u>Reliability Models</u>

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 obtaining 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) <u>Design of Evolving Systems</u>

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

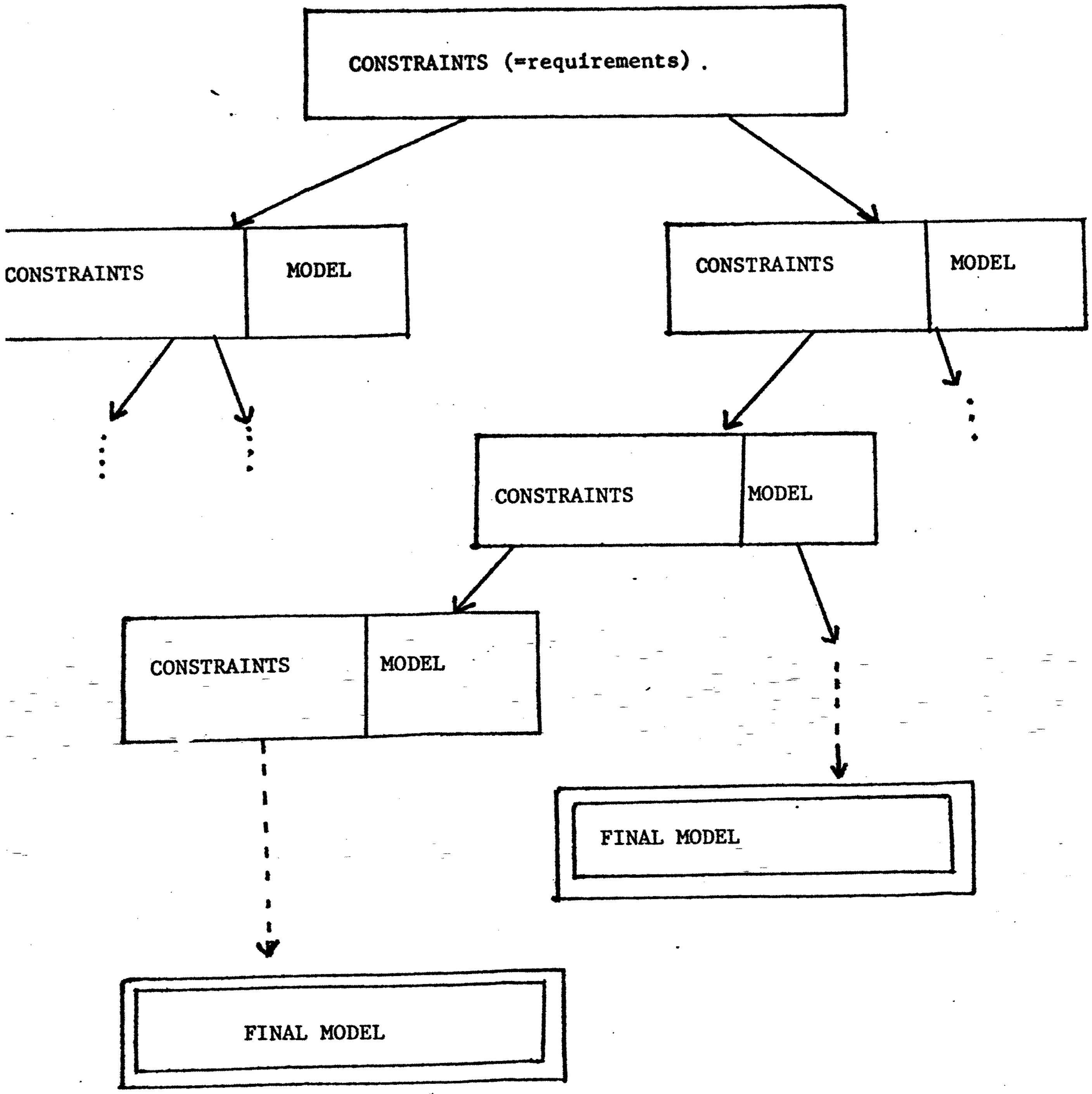


FIGURE 2. - Possible paths in the design process.

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 $I_i(1 \leq i \leq n)$ 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.

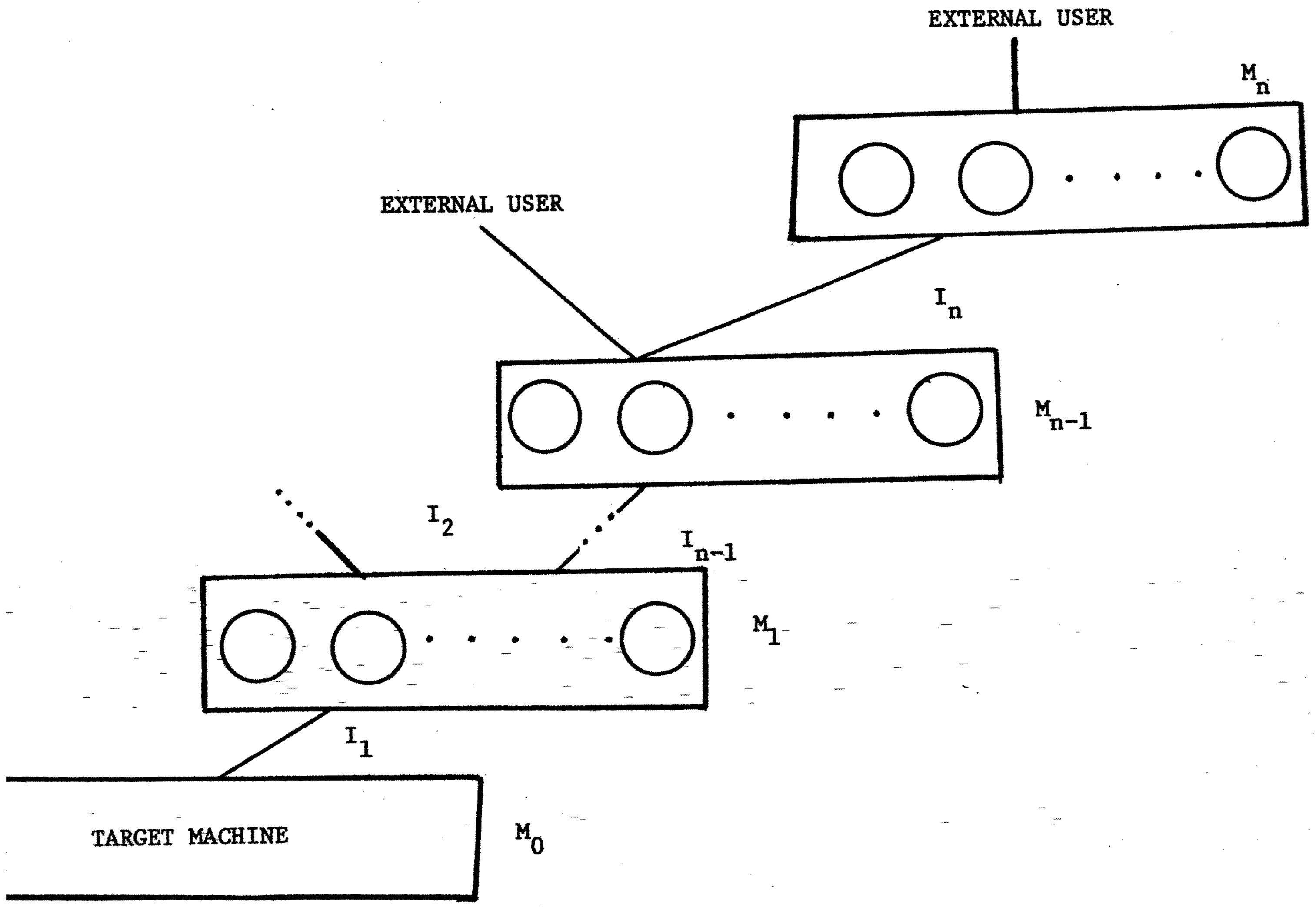


Fig. 3. A hierarchy of formally specified machines showing modularity. Levels may be constructed to accommodate the different views required by various users.

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

We have introduced a <u>System Design Langauge</u> (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 <u>hierarchical performance evaluation</u> 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 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 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.

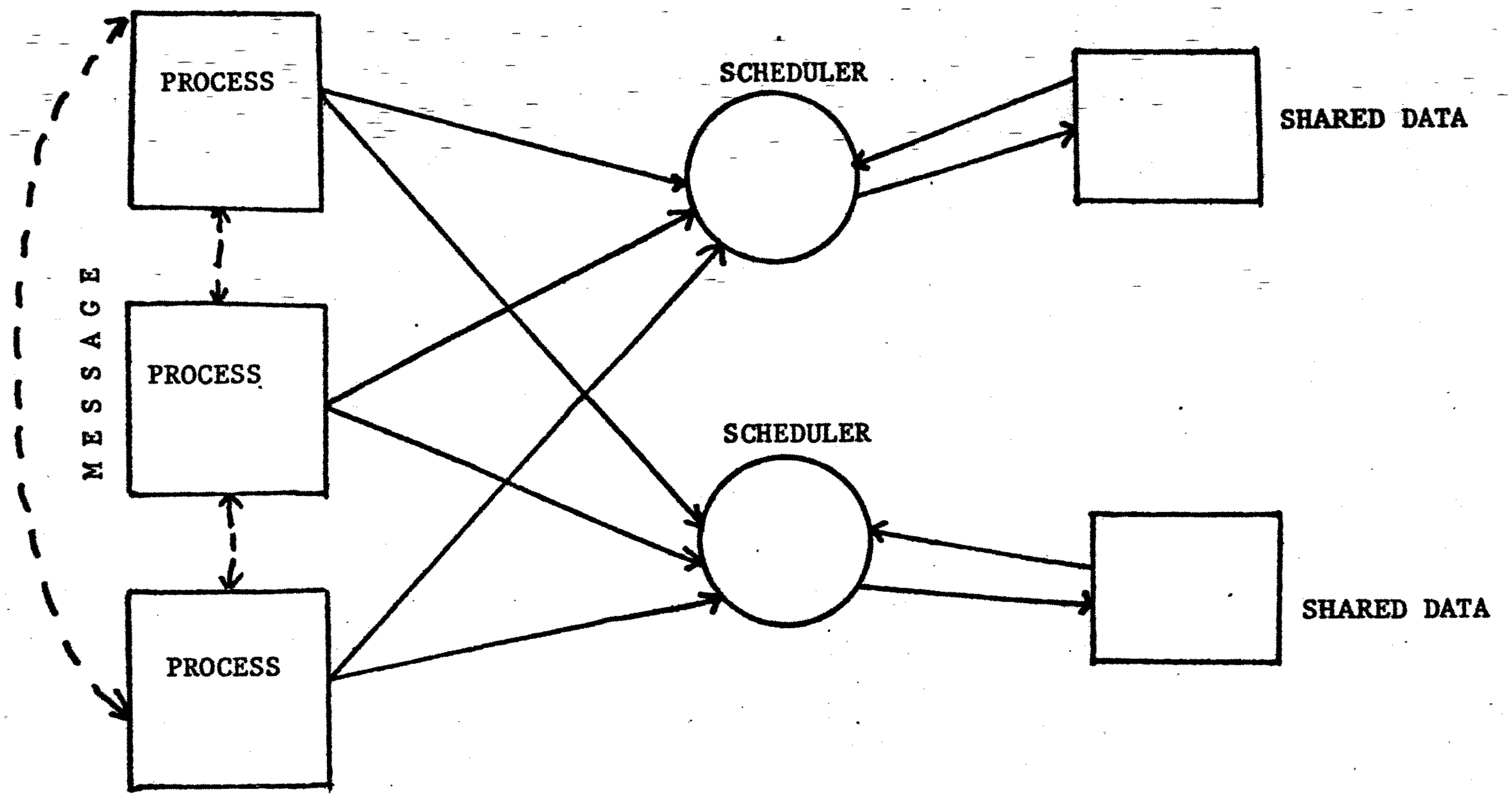
(iii) Enforcement of Coordination: Coordination of the interactions among processes has been studied at great depth, since the pioneering work by Dijkstra. On cooperating Sequential Processes [Dijkstra, 1968], solutions using P,V semaphores dealt with machine level concepts. Ultimately synchronizing 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 coördination.

"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 and 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 programs 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 model the required data and can support automatic linkage between different decision models. A strategic model (e.g., a cooperation 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 <u>most of the time</u>; 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) <u>Tools</u>

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: <u>languages</u> for communication, <u>modeling system</u> for testing out our ideas, <u>reasoning systems</u> for exploring the consequences of our ideas at a general level, and <u>knowledge systems</u> for gaining from our past experiences. <u>It is envisioned that such an integrated set of tools is itself a decision support system</u>. 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) <u>Languages</u>

   i) A requirements language. This should 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 more 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 <u>cell</u>. 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, <u>so that retrieval time is relatively independent of the size of the data base</u>. (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 <u>must</u> 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, <u>independent of any specific solution</u>. 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, <u>to within one day</u>.

(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 <u>integrity</u>. It is a constraint either dictated by the application or enforced by the data base administrator. <u>An integrity constraint is an assertion about the data base which holds following every transaction</u>. 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 <u>structure</u> of the data rather than the <u>value</u> 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 <u>incrementally</u>. 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 <u>locks</u> all pieces of data used by it prior to any <u>unlock</u>. 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."

$$\begin{tikzcd}
\text{enroll} & \text{student} & \\
\text{enroll1} \arrow[u, "\text{SUP}"'] \arrow[r, "\text{AGT}"] \arrow[dr, "\text{TH}"'] & \text{STUDENT1} \arrow[u, "\text{SUP}"'] \arrow[r, "\text{NBR}"] & 5 \\
 & \text{CS381*1} \arrow[r, "\text{SUP}"] & \text{CS381}
\end{tikzcd}$$

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 $\xrightarrow{\text{MOD}}$ (STATELY GRACEFUL)). This proves most economical for representing tables and texts.