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))))
```

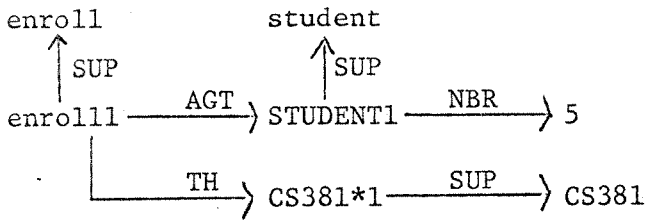
This will construct the new table, TEMP*TAB selecting NAMES, SALARY and DEPT from the old one when entries have a salary greater than 20000.

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

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 associated with a 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 $\underline{\text{TEXT1}}_{>}$(1.1)
2. FISH $\underline{\text{TEXT1}}_{>}$(1.2, 1.5 2.2 2.5)
3. EAT $\underline{\text{TEXT1}}_{>}$(1.3, 2.3)
4. LITTLE $\underline{\text{TEXT1}}_{>}$(1.4, 2.1)
5. LITTLER $\underline{\text{TEXT1}}_{>}$(2.4)

TEXT1 $\underline{\text{SEQ}}_{>}$((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 $I_i (1 \leq i \leq n)$ 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

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:

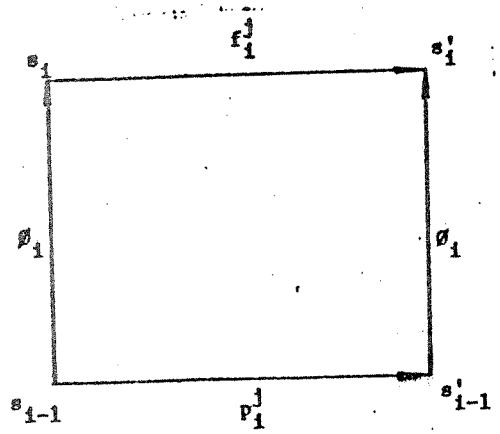


Fig. 1

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 <u>true</u> 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.

<u>Level 5</u>

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

<u>Level 4</u>

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.

<u>Level 3</u>

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.

Level 2

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.

Level 1

VP - Defines the concept of a virtual page space.

Level 0

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.

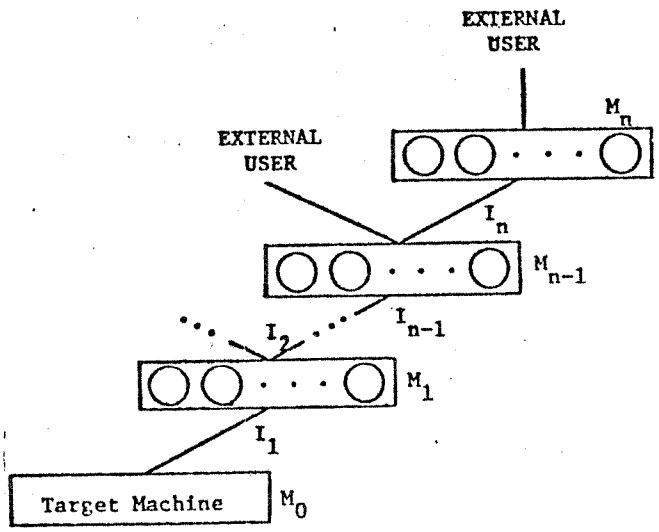


Fig. 2. A hierarchy of formally specified machines showing modularity. Levels may be constructed to accommodate the different views required by various users.

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 included 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 defined as follows:

$$A: \{a \mid a \in M_i \times \{C,G,I\} \times M_{i-1}\}$$

$$U: \{u \mid u \in M_i \times \{R,W,C\} \times M_{i-1}\}$$

$$P: \{p \mid p \in M_i \times M_{i-1} \times M_{i-1}\}$$

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

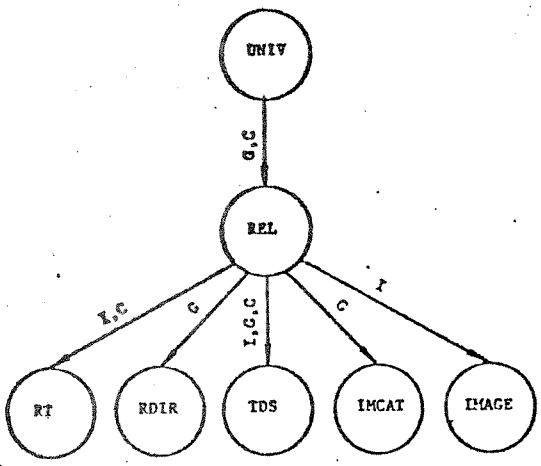


Fig. 3a. The has access relationship between several modules of Table 1. The types of access are Global (G), Indirect (I), and Creation (C).

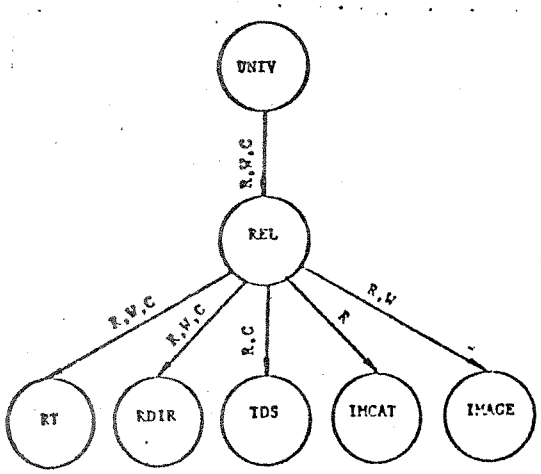


Fig. 3b. The uses relationship between several modules of Table 1. The types of usage are Read (R), Write (W), and Create (C).

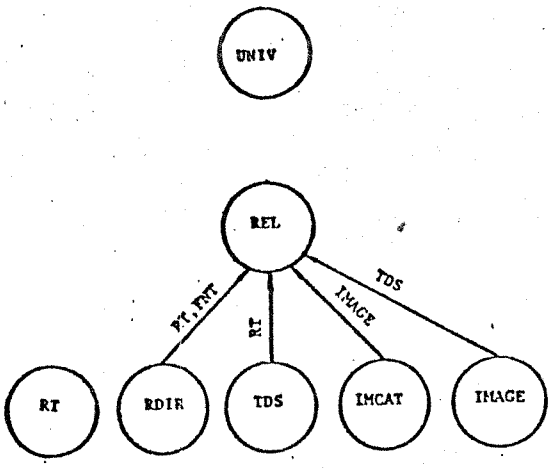


Fig. 3c. The provides relationship between some modules of Table 1.

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).

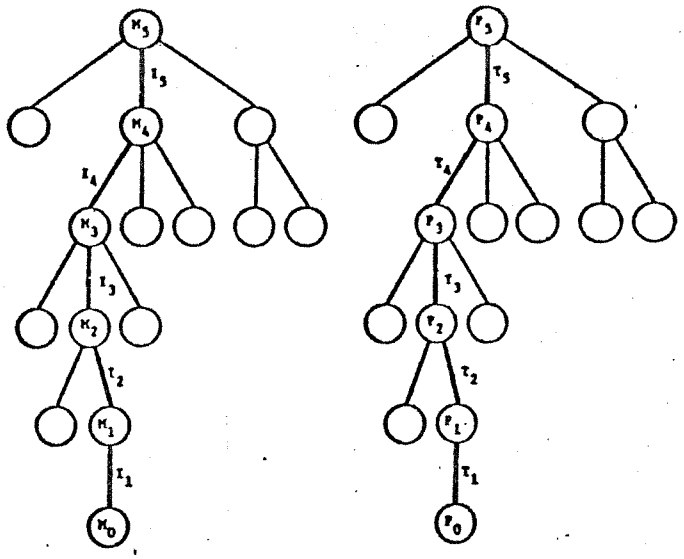


Fig. 4. A hierarchical DBMS design and the correspondingly structured performance evaluation model. Unlabeled nodes represent unused alternative designs.

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(R(X(1),...,X(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:

      (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",0R) 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
```

Fig. 6 (Cont'd)

```
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)
```

Figure 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))
```

```
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 performed 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 <u>shared</u> 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 <u>no</u>

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.

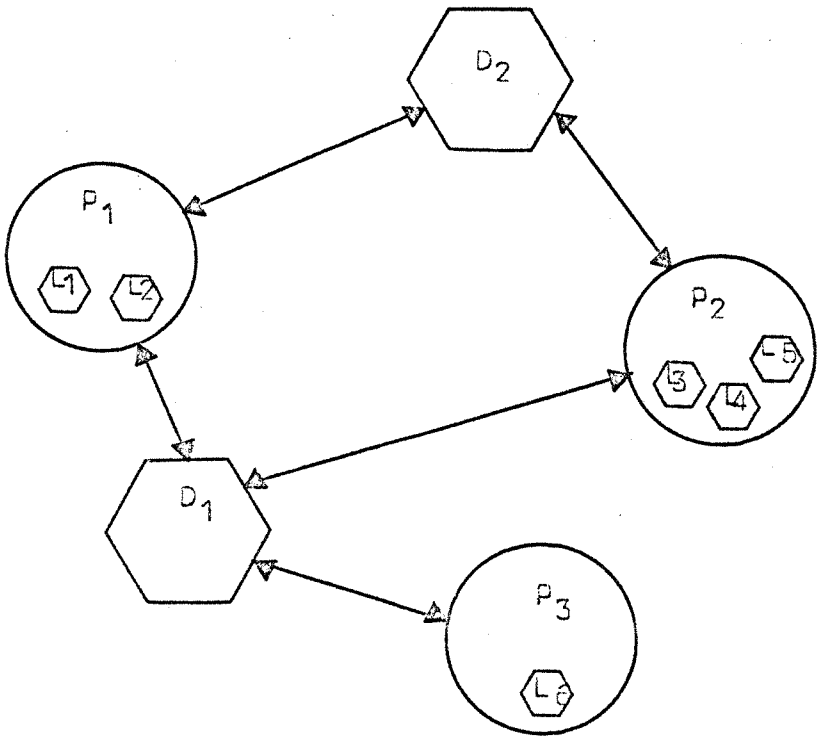
MOVE_RIGHT (TAPE): Moves the TAPE's tape head right.

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.

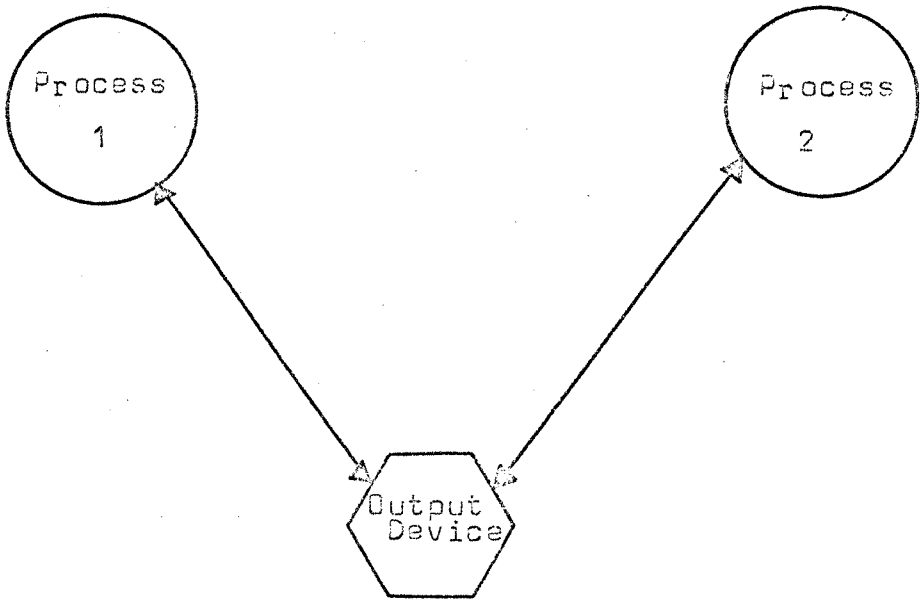
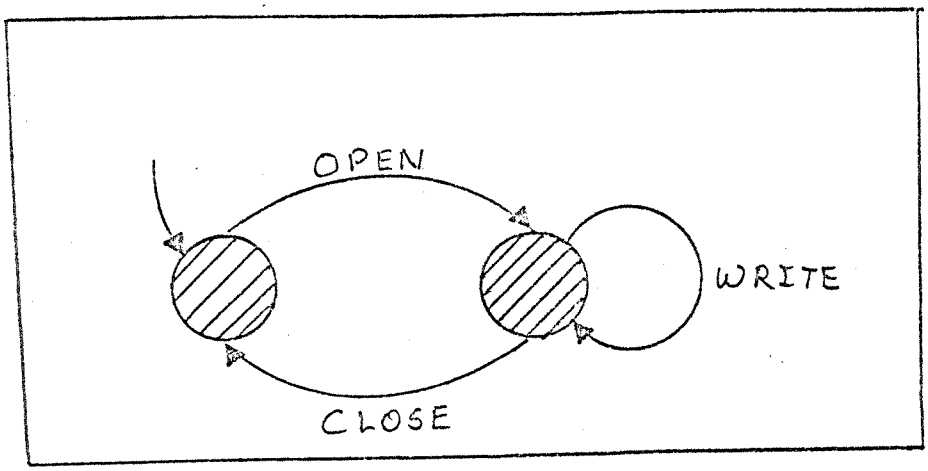


FIGURE 3. Structured environment model of a two process version of the output device example.

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

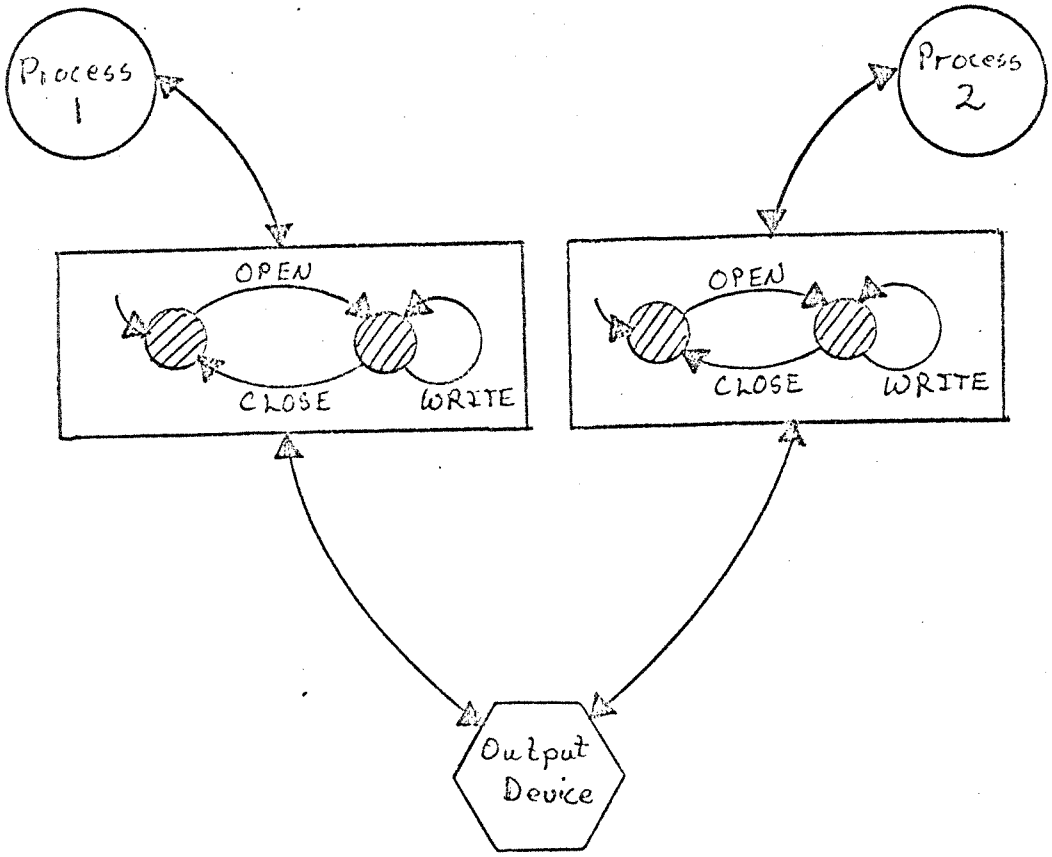


FIGURE 5. Two process version of output device example including rights contollers.

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 sequential 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.

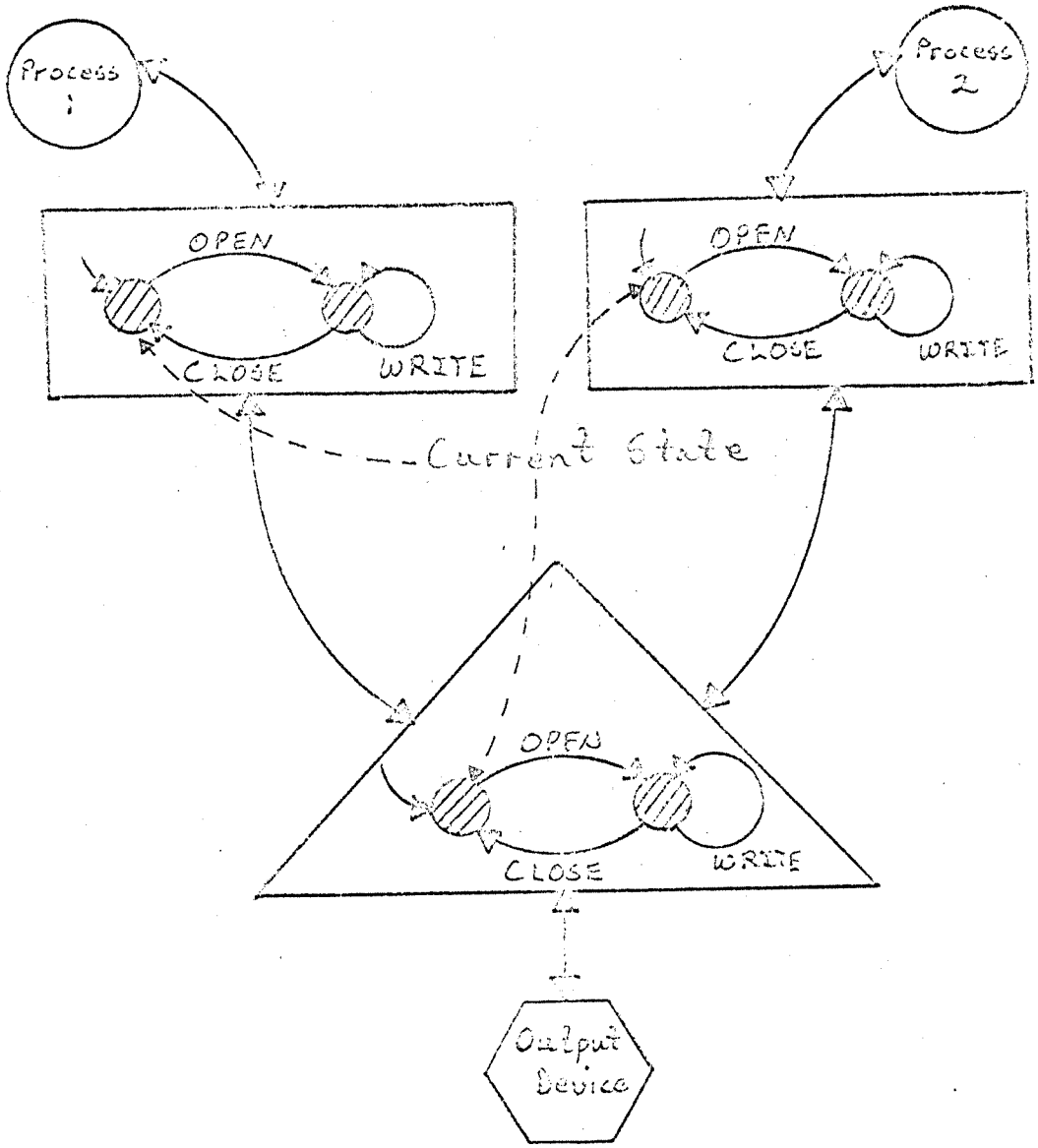

Process 1
Process 2
OPEN
CLOSE
WRITE
OPEN
CLOSE
WRITE
Current State
OPEN
CLOSE
WRITE
Output Device

FIGURE 6a

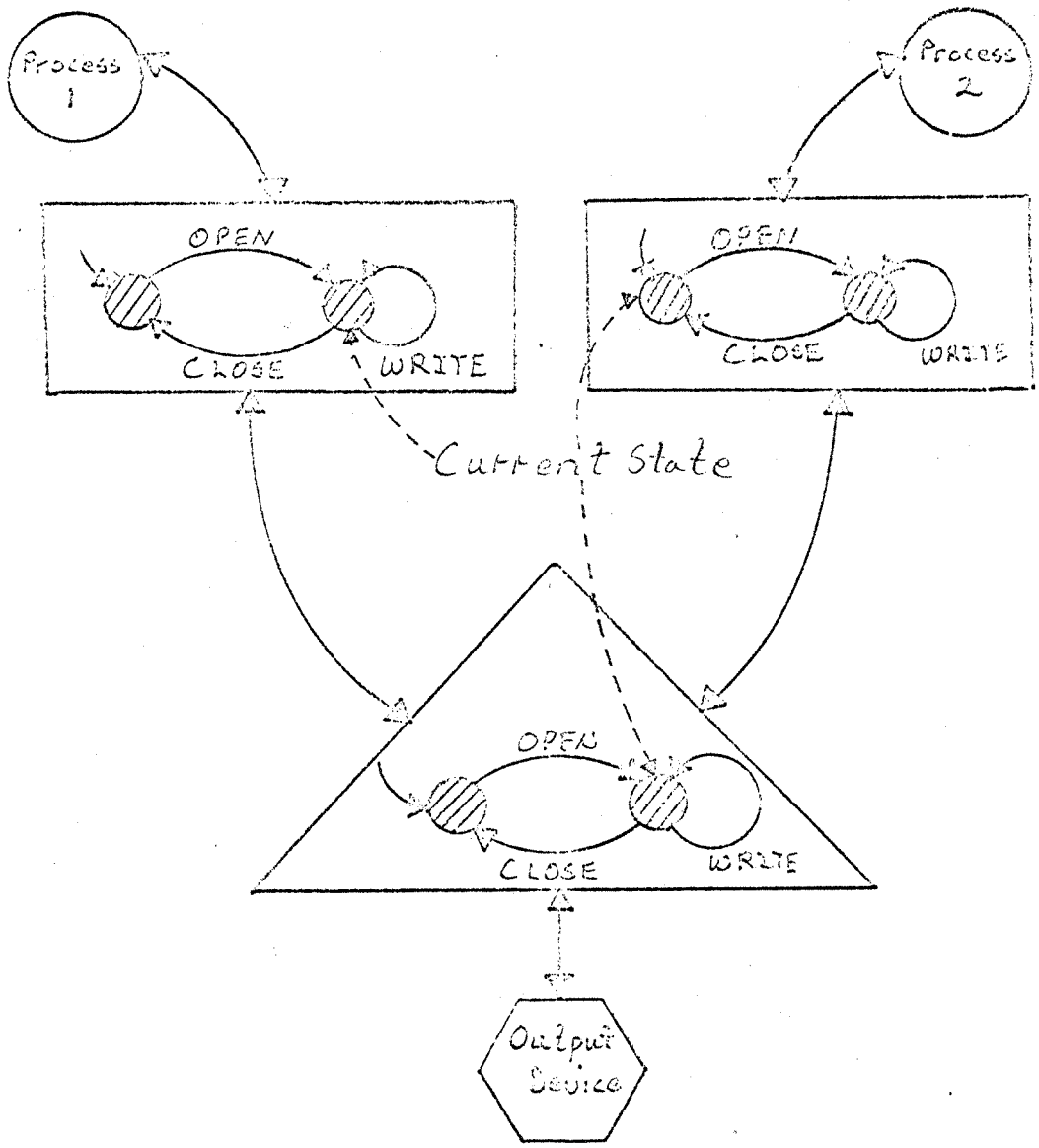

Process 1
Process 2
OPEN
CLOSE
WRITE
OPEN
CLOSE
WRITE
Current State
OPEN
CLOSE
WRITE
Output Device

FIGURE 6b

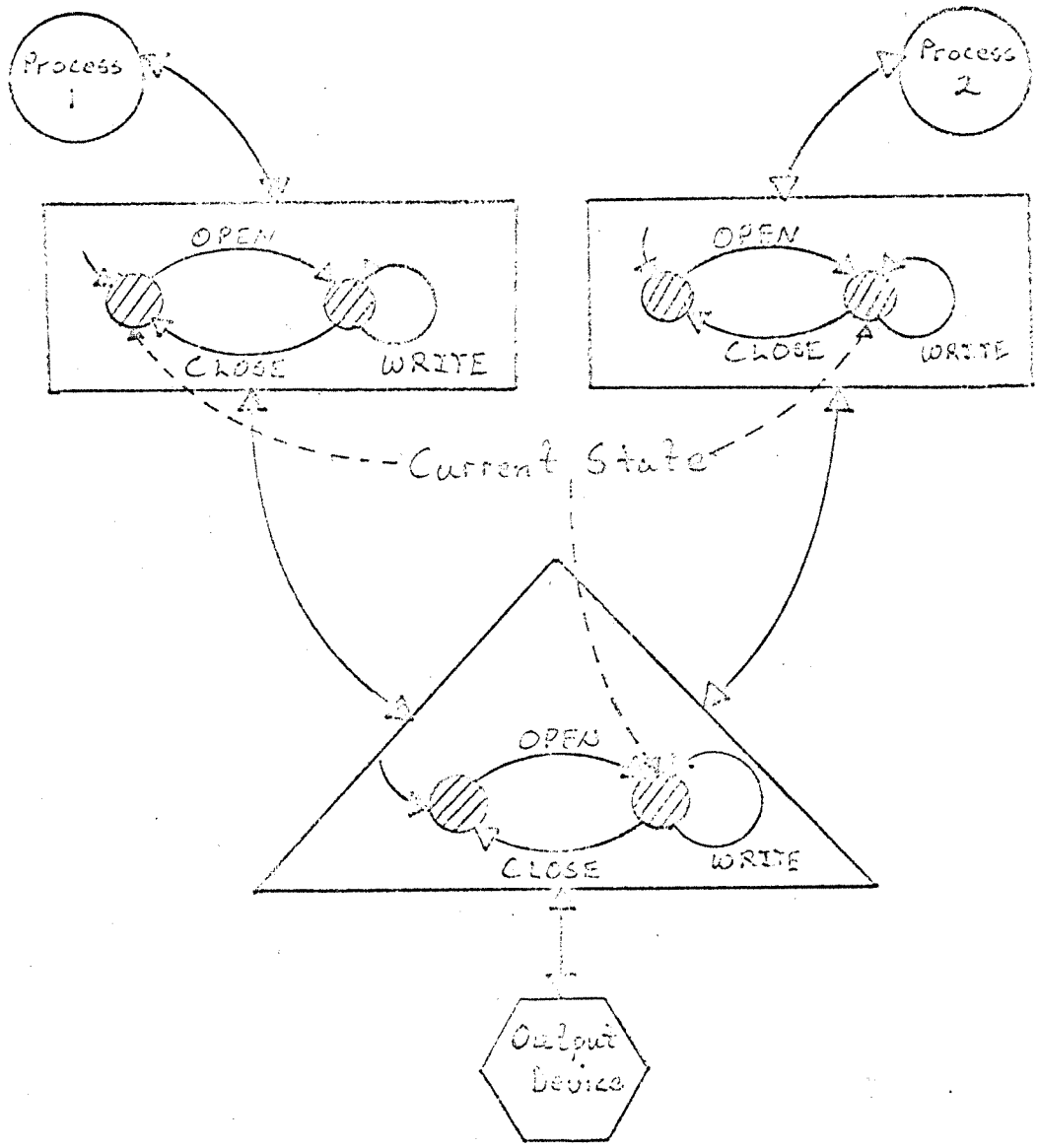

Process 1
Process 2
OPEN
CLOSE
WRITE
OPEN
CLOSE
WRITE
Current State
OPEN
CLOSE
WRITE
Output Device

FIGURE 6c

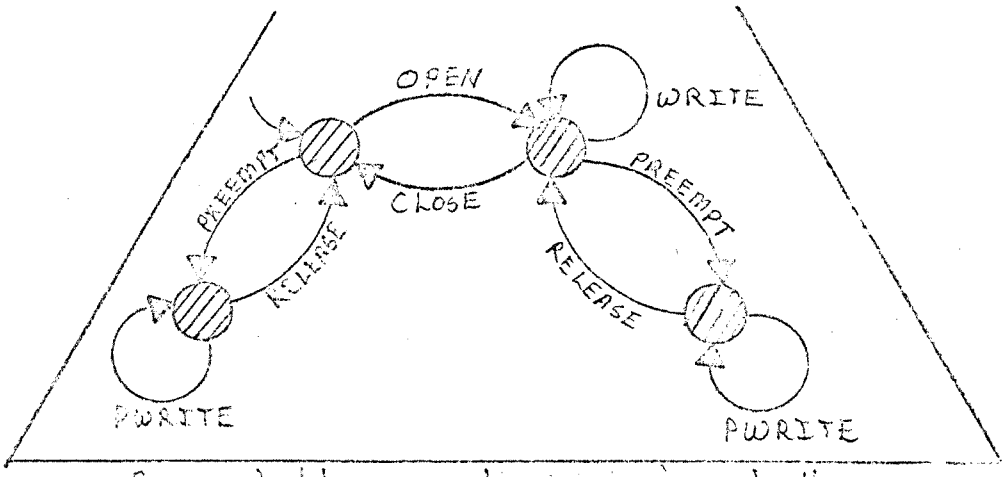
Thus, by the combination of rights controllers and synchronizing controllers, I am able to insure both of the properties concerning the sharing of the output device. Note that the synchronizing controller by itself would not have insured the proper synchronization property. For example, if the process 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.

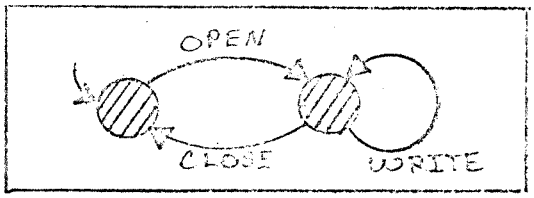
Let us suppose that our output device is used for messages to the machine operator as well as user output. Now, suppose some operator messages need to be output immediately, i.e., before the end of some user output task. Figure 7 shows how the synchronizing controller for the output device might be modified 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.

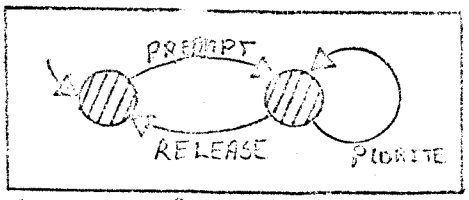
Let me continue to add complexity to this example by suggesting that



Preemptable synchronizing controller



Regular rights controller



Priority Rights Controller

FIGURE 7

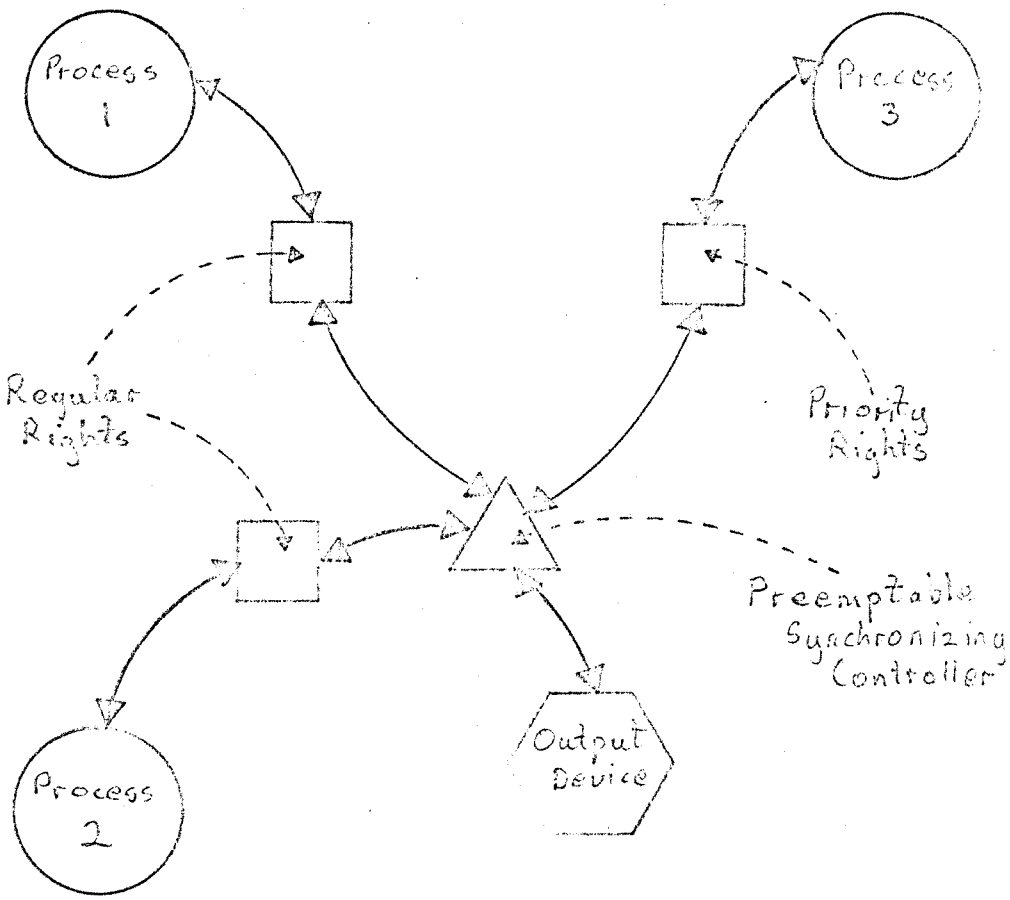

Process 1
Process 3
Regular Rights
Priority Rights
Preemptable Synchronizing Controller
Output Device
Process 2

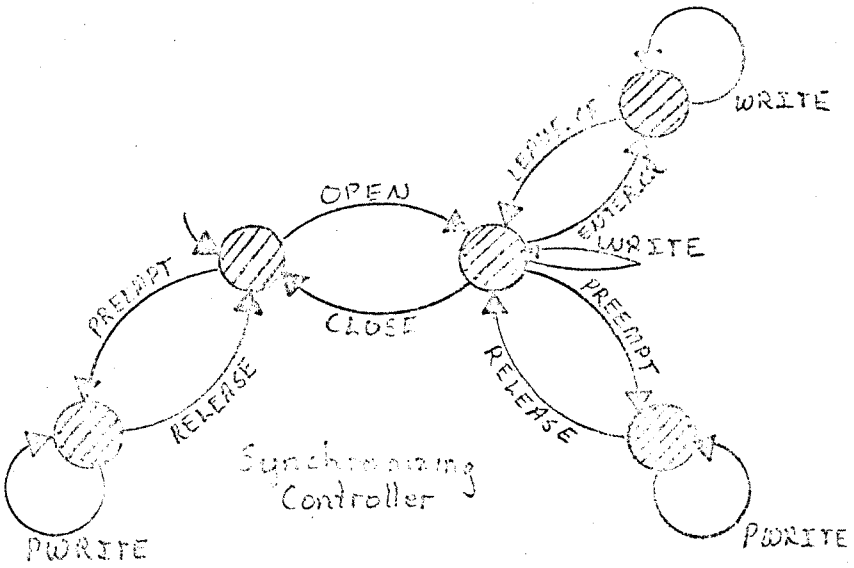
FIGURE 8

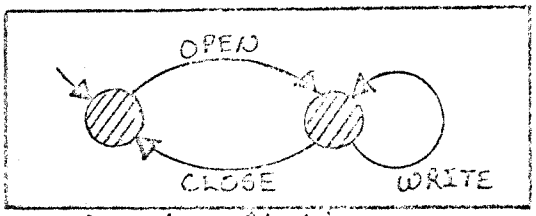
after our hypothetical system has been in use for some time, one of the system users might come in with the complaint that his output has operator messages in it. Now suppose that this user's output involves the use of expensive registered forms (e.g., payroll checks) and the system manager decides to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

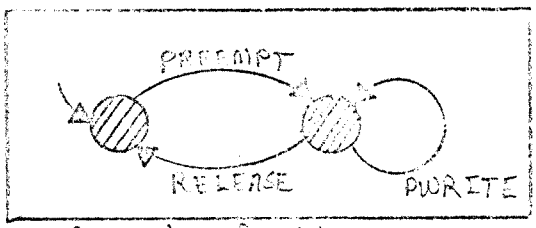
I think that this solution compares very favorably to a more traditional solution involving conventions over semaphores or such. I find especially impressive the way one is able to modify the constraints concerning the sharing of a data object without affecting those processes which do not wish to take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

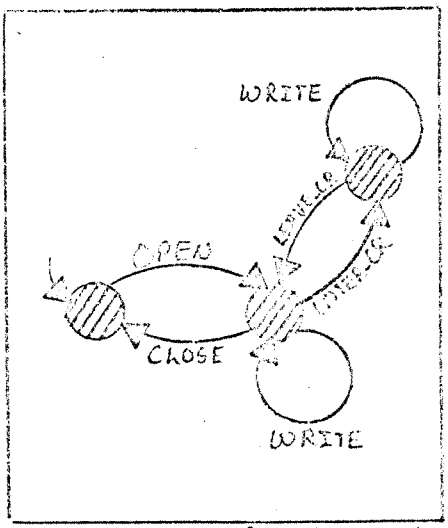




Regular Rights



Priority Rights



Nonpreemptable Rights

Figure 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.