A HIERCHICAL DESIGN METHODOLOGY

FOR DATA BASE SYSTEMS

by

Jerry Baker and Raymond T. Yeh

April 1977 TR-70

Acknowledgements are due to National Science Foundation for partial support of this work under grant DCR 75-09842.

DEPARTMENT OF COMPUTER SCIENCES

THE UNIVERSITY OF TEXAS AT AUSTIN

ABSTRACT

A formal design methodology for the development of data base system software is presented. This methodology provides a systematic method for the design, specification and implementation of a reliable data base system such that integrity and security constraints can be automatically included, and that correctness proofs can be established for the resulting system. The design process is top-down and the resulting data base system will consist of a hierarchy of formally specified data abstractions.

The methodology has several advantages over more ad hoc approaches including reliability and flexibility. Moreover, a performance modeling technique is introduced which will enable the designer to evaluate the performance characteristics of the system at each stage of the development process.

Some examples illustrating the application of the methodology to the design of a small relational data base system are also presented.

I. INTRODUCTION

The growth of the data processing industry has been spectacular. Almost every sizable business, government agency, and educational institution depends in a critical way on very large data processing systems. Moreover, as the cost of computing goes down it is expected that there will be a further explosive growth in the number and variety of other data bases.

While 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. Current approaches to the design and evaluation of data base systems, unfortunately, remain ad hoc and based primarily on experience. If, however, the technology of data base system design is to satisfy the requirements of future critical areas of application, then a general design methodology for data base systems is a necessity. Such a methodology ideally should provide guidelines for the systematic design and construction of any DBMS and its associated data bases such that the total system is

i) applications and machine independent

ii) intelligent

iii) efficient

iv) reliable

v) secure

We believe that knowledge in many disciplines of computer science should and could be utilized for the development of such a methodology. In this paper we make a first attempt in presenting a methodology in which we borrow heavily from software engineering. Our methodology is aimed to achieve all of the ideal goals except intelligence mentioned previously. Our methodology provides for the systematic design, specification, and implementation of a reliable data base system such that integrity and security constraints and performance characteristics can be automatically included, and that correctness proofs can be established for the resulting system. Using this methodology, a data base system 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. The approach not only enhances the reliability of the design but it also enables the designer at each step of the development process to evaluate the performance characteristics of the system.

Our approach to data base system (DBS) design is quite a departure from the traditional view that a DBMS consists of two levels - logical and physical, such that these two levels can be designed separately and connected by complex mappings. In our approach, the separation of logical and physical structures no longer exists.

It should be noted that the notions of top-down design and multi-level data base system architectures are not new. For example, Smith and Smith [1976], Weber [1976] and Aurda and Solvberg [1975] used the

concept of abstraction to design a hierarchical structured data bases. Our methodology, however, aims at the design of the whole system, i.e., both a DBMS and its associated data bases. Hierarchical data base architectures have also been proposed by many authors, e.g., Hammer and McLeod [1975], Madnick and Alsop [1967], Mealy [1967], Mylopoulous, Schuster, and Tsichritizis [1975], Schmid and Bernstein [1975], Kraegeloh and Lockemann [1975], and Senko [1976]. We do not, however, propose any specific architectures. Rather, the methodology presented here is a cohesive set of techniques which can be applied in a systematic manner to achieve a variety of data base system designs.

The methodology to be presented in the following sections has been applied to the systematic development of a very small relational data base system. A small subset of the example will be used to illustrate the methodology here.

II. Abstraction, Stepwise Refinement and Data Base System Design

One of the most powerful tools in software development is that of abstraction. The use of abstraction allows a designer to initially express his solution to a problem in a very general term and 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 programs 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 through a stepwise refinement process (function abstraction) such that each level consists of a number of modules (data abstractions). Note that such a hierarchically structured system is both horizontally and vertically modular, and hence is a useful model for a DBMS. The horizontal modularity provides machine independence, since change of machine architecture will only affect the bottom level of such a system. The vertical modularity provides application independence since the addition and deletion of applications can be accommodated through changes in columns (modules and their vertical refinements), but not the whole system.

Other benefits for a hierarchically structured DBS include:

i) Integrity and security constraints can be automatically implemented at each level (by means of exception conditions for example) since (Parnas) modules are designed to explicitly hide information from users.

ii) Proof of correctness of the system is now possible (this has not been a practice for DBS). Furthermore, due to the rigorous specification of modules, the proof is reduced to a proof of consistency between levels.

We will discuss these two items in more detail in later sections. We observe that the notion of abstraction comes naturally in DBS from

another aspect. Various users may view a DBMS quite differently, and their access rights to the system are also distinct. For example, only the top level of the system is visible to a casual user, whereas an applications programmer is allowed to create new functions and hence may need to access information available in the second level in order to create efficient function implementation. There are other levels, e.g., system programmer level, privileged programmer level, and storage structure development level.

Our mention of specific levels is neither intended to be exhaustive or even the best possible. We wish to emphasize that the notion of abstraction translates to a natural interpretation in the context of data base systems.

The methodology presented in this paper uses the concepts of abstraction, stepwise refinement and modularity to achieve a data base management system with a hierarchical structure. The structure consists of a set of n+1 abstract machines, M_n, M_{n-1}, ... M_0 connected by a set of n programs, P_n, P_{n-1}, ..., P_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. Thus, for example, the top level or "user machine" represents the end user's conception of the data base system both in terms of abstract data and operation. Each successive machine in the design sequence, moreover, represents a "refinement" of the previous machine in the sense that its data abstractions are used to "implement" those of the previous machine. This implementation

consists of a set of "virtual" programs each of which expresses an operation of the previous machine in terms of a (sequence of) operations of the current machine. A formal proof can then be constructed to verify that the implementation is consistent with the specification of each machine.

The stepwise process of machine specification, implementation and verification continues until the design reaches the level of some well-defined implementation language, M_0, which may be a programming language, low-level file management system, or the machine language of some appropriate hardware configuration. The result is a hierarchy of formally specified machines (Figure 1) and a corresponding hierarchically structured software system which expresses the concepts of the user level machine M_n within the framework of the implementation language M_0.

Before turning to a detailed discussion of the design process, it should be mentioned that there is a natural deficiency in the top-down approach proposed here. In this approach, it is necessary for the designer to choose among alternative designs at each level. The methodology, however, does not provide any guidance for the designer in making such choices. As a consequence, a system may be reliable but must be scratched because of reasons such as poor performance. In section VI, we shall point out how this deficiency can be eliminated by doing performance evaluation at the design level.

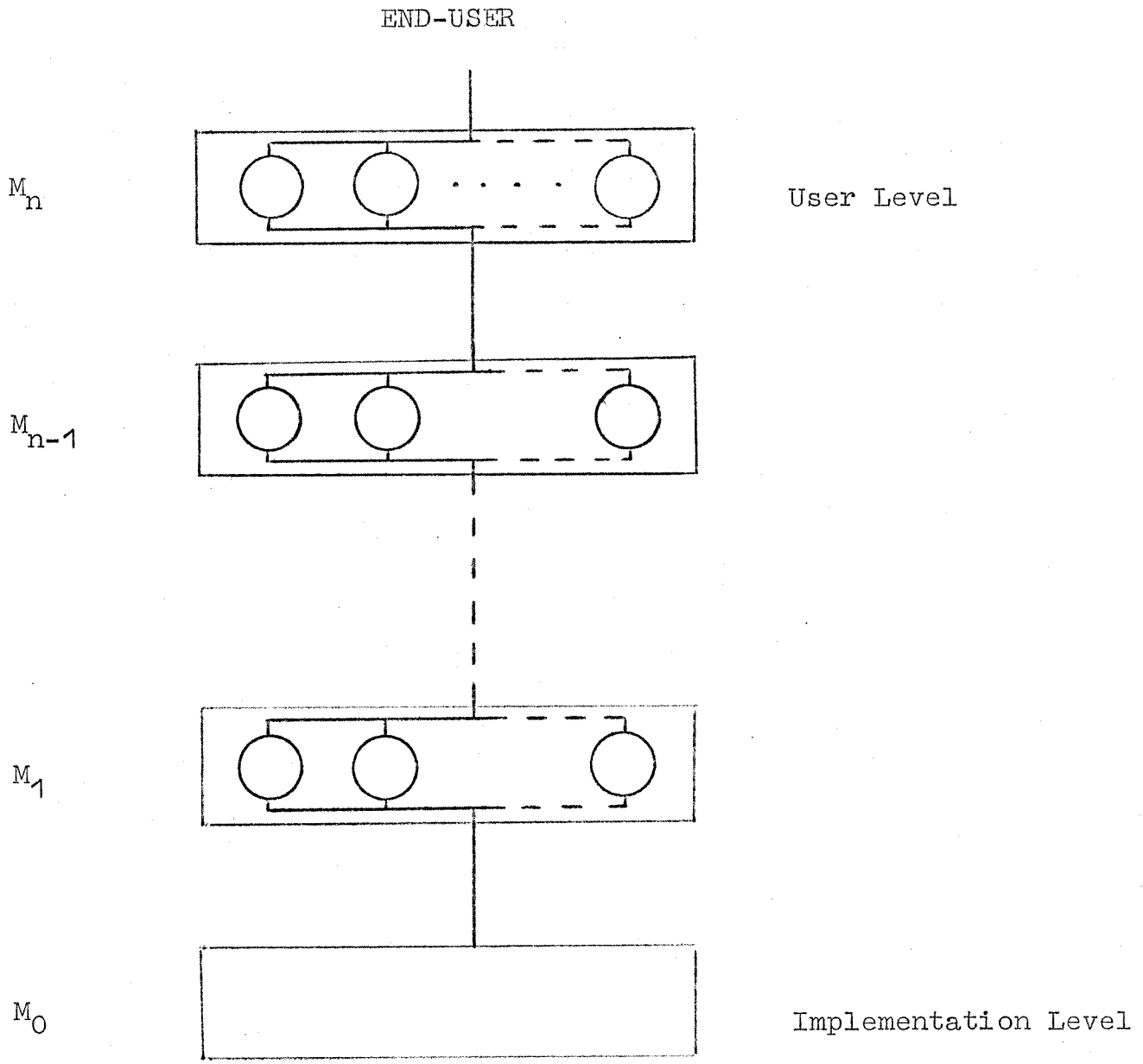

Figure 1 - A HIERARCHY OF FORMALLY SPECIFIED MACHINES SHOWING MODULARITY

III. Design and Specification

The design process of this methodology may be viewed as a sequence of decompositions of abstract concepts. The user level machine, for example, consists of those concepts visible to the end-user of the system. The design of the next level, then, is concerned with determining what concepts are necessary and desirable to represnet those of the previous level. This decomposition process continues until the concepts of the system are represented within the framework of the implementation machine M_0. The decomposition process is, of course, not unique and, therefore, there is no rigid rule for determining the number of levels used in a design. Thus, two designers may derive different hierarchies for the same system. In this paper, we decide that at the user machine level we are concerned with structures such as "relations", "networks", etc. At each succeeding level, less abstract concepts such as access paths, indexes, etc. are used to implement comcepts used at the higher level.

Each abstract concept within a machine is realized by a collection of one or more formally specified Parnas modules (Parnas [1972]). The role of a module in this methodology can be seen from two different viewpoints. First, a module may be viewed as defining an abstract type of data object. The module's specification, then, serves the purpose of defining the manner in which "instances" of the object type may be created and manipulated. In the second viewpoint a module itself is considered as an abstract machine which is characterized by its state and a set of machine operations. Each operation may serve one of two functions - that of causing a state change in the module or that or returning a value which partially defines

LEVEL	CONCEPTS VISIBLE	OPERATIONS	CONCEPTS HIDDEN	MODULES
4	relations, tuples authorization, semantic integrity	algebraic relational operations, creation and enforcement of authorization and integrity assertions	access path information, storage of data	RELATION, AUTHORIZATION, INTEGRITY, TUPLE
3	images, links, catalogs of relation and domain information	creation and maintenance of links, images, and catalogs	implementation of logical access paths and catalogs	RNT,DNT,LNK,ING,TT, etc.
2	linked lists of finite length abstract tables, B-trees	creation and maintenance of linked lists, B-trees and data records	bit encoding of lists, B-trees and data records	LT,BTR,RT, etc.
1	linear bit streams	bitstream extraction and encoding	distribution of bitstreams into virtual storage	BTS, etc.

TABLE 1. AN OVERVIEW OF THE DESIGN DECISIONS FOR A FOUR-LEVEL RELATIONAL SYSTEM.

the module's state. Both viewpoints are useful and will thus be referred to throughout this paper.

We present now an example which illustrates the design and specification of a relational system achieved by the application of the hierarchical design methodology. The purpose of the example is to illustrate the top-down design process and the resulting hierarchical decomposition of abstract concepts which occurs from the use of the methodology. Tables 1 and 2 show the general architecture of this system. Table 1 specifies for each level the visible concepts, the operations, and the concepts hidden. Table 2, furthermore, shows the decomposition of each level into some of its component modules. The "uses" hierarchy (Parnas [1974]) which exists between the modules at different modules is shown in Figure 2.

Level 4

RELATION - relations and algebraic relational operations

AUTHORIZATION - assertions defining allowable user accesses

INTEGRITY - assertions defining the semantic integrity of the system

TUPLE - n-tuples and operations on them

Level 3

RNT - catalog of relation names and related information

DNT - tables of domain names and related information for each relation

TT - tables of data values of relations

LNK - abstract "links" between sets of relation tuples

IMG - logical ordering of relation tuples

Level 2

LT - linked lists of finite length tables

RT - file structure of records

BTR - B-tree structures

Level 1

BTS - linear bit streams

Table 2 - Abstract Concepts Represented by System Modules

The design of Level 4 includes four modules, each of which represents an abstract concept visible to the system end-user. The RELATION module, for example, defines the operations by which users are able to create, modify, and view the primary data objects (relations) of this level. Likewise, the TUPLE module enables users to create and manipulate n-tuples as seperate entities. The AUTHORIZATION module relates to the concept of security in the data base system. Through the operations of this module it is possible to create authorizations which are assertions defining the allowable user accesses to the abstract data objects of Level 4. Certain operations also make it possible to scan all defined authorizations to determine if a user has access to a specified data object. The INTEGRITY module provides operations for defining and maintaining the semantic integrity of relations. For example, it is possible to create assertions which define allowable domain values in relations. Likewise, it is possible to scan these assertions to determine if an update or modification would violate any of these.

In the design of Level 3

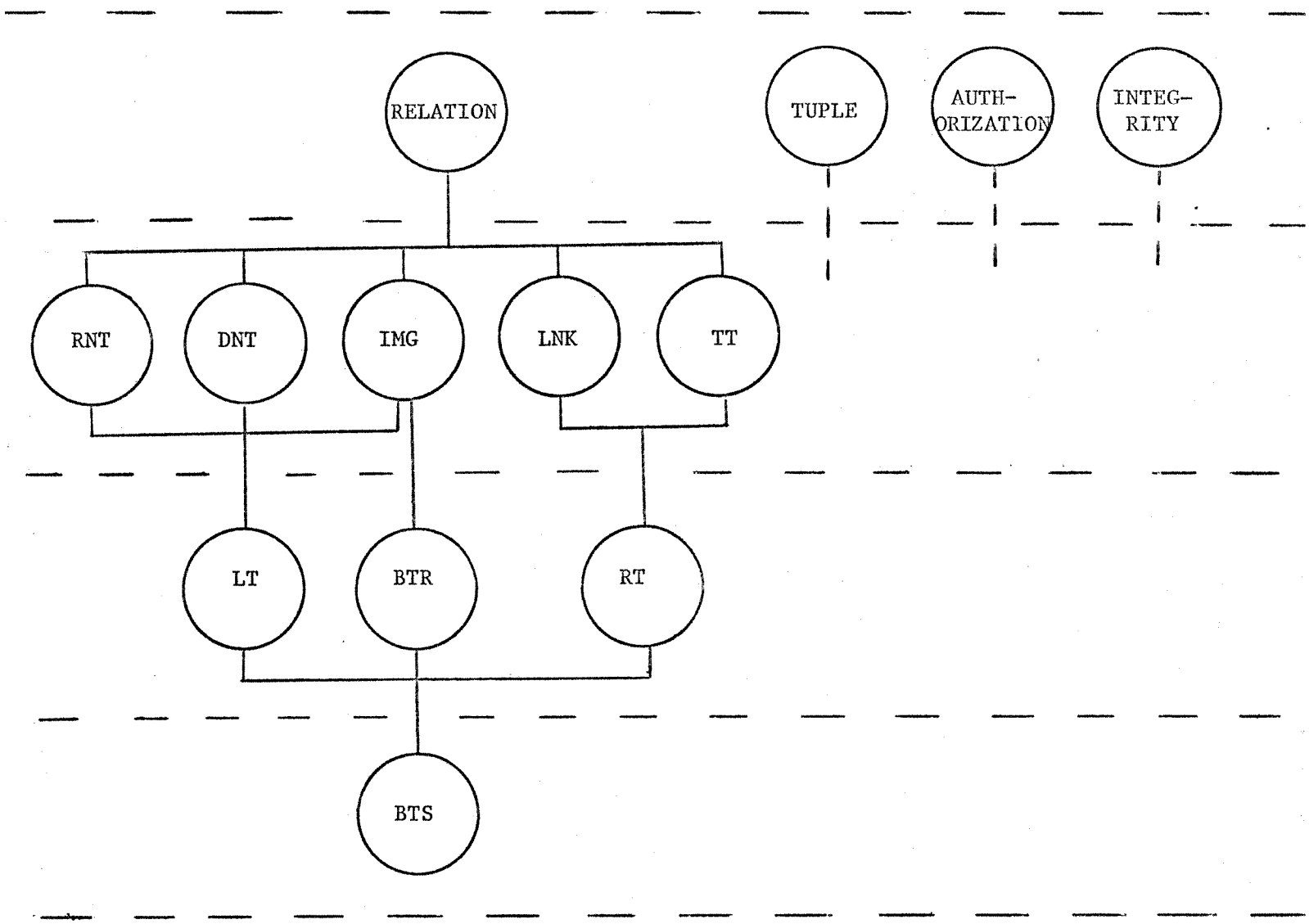

Figure 2. The "uses" hierarchy of modules in a relational data base system.

domain values in relations. Likewise, it is possible to scan these assertions to determine if an update or modification would violate any of these.

In the design of Level 3, of course, it was necessary to determine what concepts should be used to implement those of Level 4. Figure 2 shows some of the modules used to implement the RELATION module. The RNT module, for example, contains operations for creating and maintaining a catalog containing information about defined relations. Once created, this table contains one entry for each relation which is defined at Level 4. Each entry contains a relation name, the number of tuples contained in the relation, the "type" of the relation ('BASE' or 'TEMP'), the number of domains in the relation and a "pointer" to another abstract table containing domain information.

In a similar manner the operations of the DNT module allow the creation and maintenance of catalogs each of which contains information about the domains of a defined relation. Each entry in such a table contains a domain name, the data type of domain values (REAL, INTEGER, etc.) and information as to existing access paths for the values of the domain.

The IMG and LNK modules represent logical access paths corresponding to the concepts of links and images (Astrahan et al [1976]), respectively. Through the use of the LNK module it is possible to create logical links between tuples of the same or different relations. Such a concept is useful in efficiently implementing the relational join operation of Level 4. The IMG module makes it possible to create

logical reorderings of relation tuples based upon domain values. Such a concept is useful in efficiently implementing the 'select' operation of Level 4.

A further refinement of these concepts is shown at Level 2. The LT module, for example, implements the catalogs and images of Level 3 by linked lists of finite length tables. Likewise, the BTR module also implements the concept of an image as a B-tree structure (Bayer and McCreight [1972]). Also, the links and TT tables of Level 2 are implemented as "files" of "record" values by the RT module.

Finally, at Level 1 a single module representing linear bit streams is shown. This module can be used to extract and insert information into specified locations of user bit streams and is used to implement the LT, BTR and RT modules of Level 2.

The design shown in Figure 2 should by no means be construed as representing a prototype architecture for future data base systems. Rather, its purpose is to illustrate the process of viewing the operation of a data base system at various levels of abstraction. Consider, for example, the creation of a relation at the user level. This is viewed at Level 3, among other things, as being the insertion of an entry into the RNT table and the creation of a DNT table for the domains of the created relation. These processes, in turn, are viewed at Level 1 as being the manipulation of certain linear bit streams.

The primary importance of viewing a data base system in this manner is that it reduces the complexity of the design process. At each system level the designer is able to concentrate on the issues of

design relevant to the concepts at that level without being distracted by implementation details. The expected result is increased system reliability. Another advantage is that the various levels may naturally correspond to different classes of users (casual user, application programmer, system programmer, etc.) each desiring to view the system at a different level of abstraction. Thus, the operations defined by a machine's formal specifications not only may be used to implement a higher level machine, but also may serve as an interface to external system users.

Appendix A contains the formal specifications for some of the modules shown in Figure 2. The primary purpose of these specifications is to define the functions by which a module may be accessed. Basically, there are three types of access functions: SV, SE, and ST. SV and SE functions return values which partially define the abstract state of the module. Each SV function has a value associated with its name while SE functions return values indirectly through parameter lists. ST functions, on the other hand, are operations which produce a state change in the machine containing the module. The result of correctly invoking an ST function is specified as a sequence of effects each of which is a formal statement defining a resulting change in the values returned by SV or SE functions. Each effect may be classified as local or non-local. Local effects are those produced in the state of the module itself while non-local effects are state changes produced in other modules of the machine.

While non-local effects may seem to violate module independence, the concept can be very useful in data base system design. Consider, for example, the design of Level 4 of the system shown in Figure 2. It was decided that in this system any user creating a data object should automatically have complete access rights to it. Thus, a user invoking the operation

$$R := \text{join}(S,d_1,\theta,T,d_2)$$

should automatically have access rights to relation R. However, such an effect is impossible to specify within the traditional framework of the Parnas module. Therefore, the join operation has a non-local effect specified as:

$$\forall \text{op } (\text{op} \in \text{opset}) \quad [\text{check\_auth}(\text{uid},R,\text{op}) = \underline{\text{true}}]$$

where uid is the user requesting the 'join' operation and opset is the set of all relation operation types. The 'check_auth' function is a function of the AUTHORIZATION module which returns the value true if user uid has the authorization to perform the operation specified by op on relation R. It should be noted that non-local effects correspond to the concept of triggers as presented by Eswaran [1976].

IV. 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 $F_i = \{f_1^i, f_2^i, \ldots, f_k^k\}$ is the access function set for M_i then the implementation of M_i by M_{i-1} is defined by

$$I_{i-1}^i = \{\emptyset_{i-1}^i, p_1^i, p_2^i, \ldots, p_k^i\}$$

where ϕ^i_{i-1} is a mapping from the states of M_{i-1} to the states of M_i and p^i_j is a "virtual" program which implements the function f^i_j on machine M_{i-1}. The mapping function ϕ^i_{i-1} 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 ϕ^i_{i-1} is defined such that for every state $s_i \in S_i$ we have $s_i = \phi^i_{i-1}(s_{i-1})$ for some state s_{i-1} of S_{i-1}. The mapping function is actually constructed by expressing each SV and SE function of M_i as an expression containing the SV and SE 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 ϕ^i_{i-1}.

Appendix B illustrates the process of mapping function construction. Shown there is the set of partial mapping functions between Level 4 and Level 3 which corresponds to the SV and SE functions of the RELATION module.

The purpose of virtual 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.

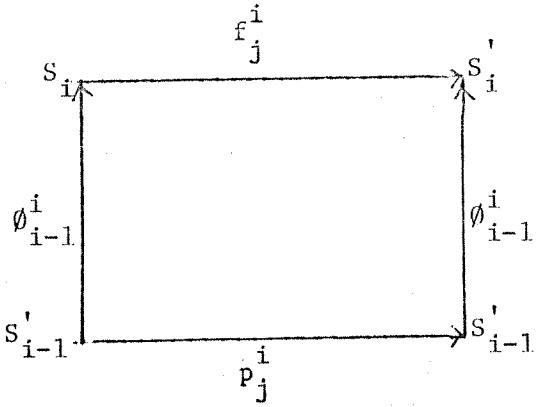

Figure 3

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}. Appendix D contains a program which implements the "select" function of the RELATION module.

The verification of the implementation I^i_{i-1} requires a formal proof that the commutative diagram of Figure 3 is satisfied for every virtual program p^i_j of the implementation I^i_{i-1}. This verification process is basically a standard inductive assertion proof 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 virtual program p^i_j is true because the program contains its own mechanisms for exception handling. The output assertions for p^i_j are derived from the assertions in the EFFECTS section of the specification for function f^i_j and from the mapping function $\emptyset^i_{i-1}$. Each output assertion is obtained by taking an EFFECTS assertion and replacing each reference by the instantiation of the appropriate partial mapping function of $\emptyset^i_{i-1}$. This process is demonstrated in Appendix C which shows the derivation of output assertions for the virtual program implementing the 'select' function of the RELATION module.

Intermediate assertions for p^i_j can be taken directly from the EFFECTS sections of the ST operations used to construct the program. Loop invariants and verification conditions can then be derived and proved to establish the validity of the program's output assertions.

V. Exception Handling

Exception conditions have been defined as "those conditions detected while attempting to perform some operation" which are "brought to the attention of the operation's invoker" (Goodenough, [1975]). In the hierarchical design methodology the mechanisms used to deal with exception conditions play an important role in the development of (reliable) integrity and security subsystems. This section, then, contains a description of the methodology's approach to the definition, detection, and handling of exception conditions. Also included are several examples which illustrate the utility of this approach.

In the methodology exception conditions are defined during the design of each module and are a part of each module's formal specifications. Each exception condition is specified as a name with a formal parameter list and is defined as an assertion containing the SV and SE functions of the module.

Each condition thus corresponds to a machine state or set of machine states about which the user must be informed. Consider, for example, the design of the RELATION module of Level 4 (Appendix A). One exception condition defined for this module is

NO_RELATION(R): relation_exists(R) = false

where the formal parameter R represents a relation name. This definition indicates that the NO_RELATION exception condition for actual parameter R' corresponds to any state of the machine in which the SV operation relation_exists(R') returns the value false.

In the formal specification of each module function a list of exception conditions may be included in which the formal parameters of

the function call are used as actual parameters for the exception conditions. The existence of such a list for a function indicates to the implementor of the module what conditions detected during the execution of the operation require notification of the invoker. The existence of an exception condition for an ST function also indicates that the function will have no effect upon the state of the machine if the condition is detected when the function is invoked. In a similar manner, if a specified exception condition exists when an SV (or SE) function is called then the value(s) returned by that function is (are) undefined.

Consider, again, the design of the RELATION module of Level 4. One function defined for this module is the 'select' operation which returns a subset of some specified relation based upon a domain value. The function has the prototype call

R:= select(S,d,θ,v)

where R and S correspond to relation names, d is a domain name, θ is a relational operator, and v is a domain value. One exception condition specified for this function is

NO_RELATION(S)

Thus, invoking the 'select' operation would have no observable effect on the system if the machine's state was such that the 'relation_exists' function returned the value false when supplied with relation name specified in the user's call to 'select'. Also, the user would be informed that this condition existed.

Exception conditions may correspond to local or non-local conditions.

For example, the NO_RELATION exception condition of the RELATION module corresponds to a local condition because its definition uses only an SV function of the RELATION module. Consider, however, the RELATION module exception condition definition

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

where the AUTHORIZATION module function 'check_auth' returns false if user id does not have authorization to perform the operation type op on relation R. This non-local exception condition definition can be used to prevent unauthorized access of relations. For example, the operation

R : = select(S,d,θ,v)

has the exception condition

NO_AUTH(uid,S,'VIEW').

Hence the operation cannot be performed unless the user has VIEW access to relation S. Non-local exception conditions also correspond to the notion of triggers (Eswaran [1976]) and can be a useful concept in the design of a data base system.

One important advantage of this approach to exception handling is that the definition and detection of exception conditions occurs hierarchically within the design and implementation process. This allows the designer at each level of abstraction to concentrate on defining the exception conditions which are relevant to the concepts represented by that level. Also, because the specification of exception conditions is separated from their detection the designer is free to

concentrate on what conditions are to be detected without worrying about how they are to be detected.

The reliability of exception handling is enhanced by the hierarchical nature of exception condition specification and detection. At the initial level of definition an exception condition is expressed in terms of the SV and SE operations of that level. Each of these operations is then, of course, implemented and verified in the hierarchical manner prescribed by the methodology.

Figure 4, for example, shows the decomposition of the BAD_DOMAIN exception condition into its component functions at each system level. Figure 5, on the other hand, shows the hierarchical expansion of the exception condition definition at each level. This expansion is obtained from the exception condition definition by the formal mapping specifications of each level. Thus, at level 4 the BAD_DOMAIN exception condition is defined in terms of the 'domain_exists' function by

BAD_DOMAIN(d,R): ¬domain_exists(d,R).

The 'domain_exists' function, in turn, is implemented at level 3 by the functions 'get_rnval' and 'get_dnval" and the resulting expansion of the exception condition definition is

BAD_DOMAIN(d,R) : ¬ if get_rnval(R) = false
then undefined
else get_dnval(get_rnval$_6$(R)$_1$d).

For each successive level, the two figures show the decomposition of

the functions of the previous level and the resulting expansion of the exception conditon definition in terms of the functions of the present level.

Exception handling also plays an important role in the development of reliable security and integrity mechanisms. This is explained in more detail in the following section.

BAD_DOMAIN(d,R): domain_exists = false

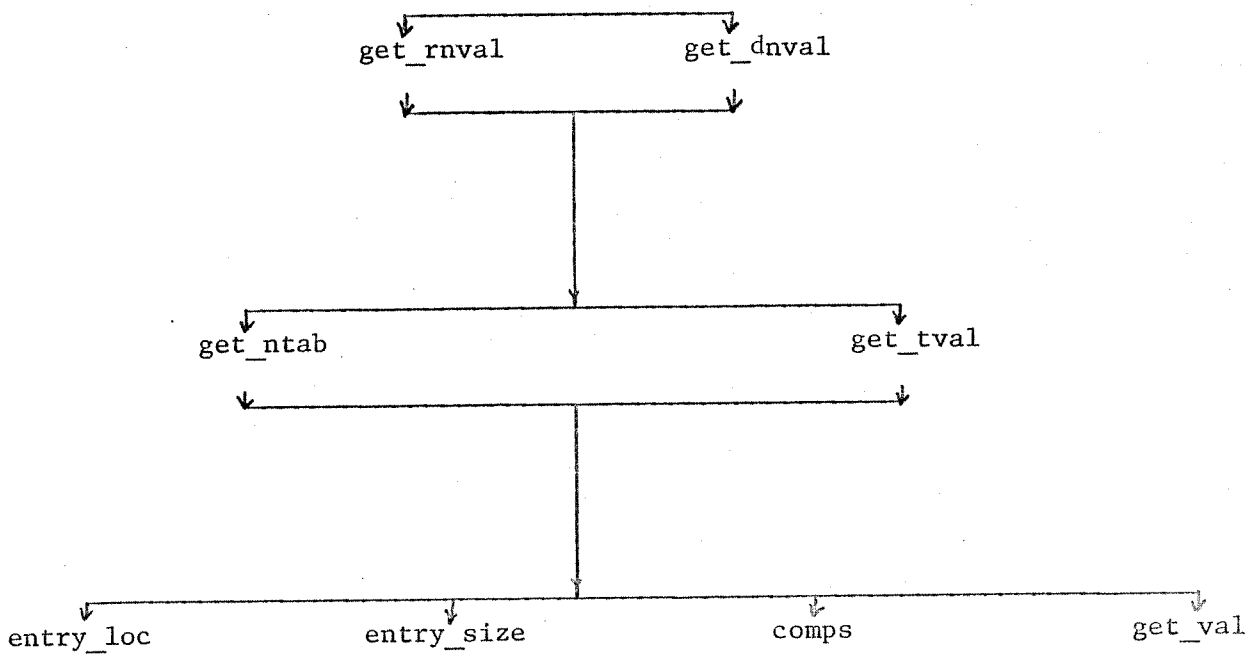

Figure 4 The hierarchical decomposition of the BAD_DOMAIN exception condition of level 4.

Level 4

BAD_DOMAIN(d,R): ¬domain_exists(d,R)

Level 3

BAD_DOMAIN(d,R): ¬if get_rnval(R) = false

then undefined

else get_dnval(get_rnval$_6$(R),d)

Level 2

BAD_DOMAIN(d,R): ¬if ∃j(1 ≤ j ≤ get_ntab(rptr)) [get_tval(1,j,rptr)=R]

then undefined

else ∃i(1 ≤ i ≤ get_ntab(get_tval(6,j,rptr)))

[get_tval(1,i,get_tval(6,j,rptr)) = d]

Level 1

Miscellaneous Definition

A(i,j,tptr): get_val(entry_loc(j,tptr),comp_loc$_3$(i,tptr),

comp_loc$_4$(i,tptr),tptr)

BAD_DOMAIN(R): ¬if ∃j(1 ≤ j ≤ get_val(nloc,0,nsz,rptr) A(1,j,rptr)=R

then undefined

else ∃i(1 ≤ i ≤ get_val(A(6,j,rptr))

[A(1,i,A(6,j,rptr)) = d]

Figure 5 Hierarchical expansion of the bad_domain exception condition of level 4

VI. Implementation of Security and Integrity Mechanisms

The hierarchical design methodology provides an excellent framework within which to develop reliable integrity and security mechanisms.

Using the methodology an AUTHORIZATION module and an INTEGRITY module have been designed to operate with the RELATION module described earlier. The specifications for these modules are contained in appendix A.

The AUTHORIZATION module is designed to provide facilities for defining the authorized use of data objects (relations) and for detecting any unauthorized uses which may be attempted. For example, the function

define_auth(id,rname,op)

can be used to create an authorization for a user to perform a certain type of operation on a specified data object. In this function call id represents a user identification number for which the authorization is being created, rname is a relation name, and op is an operation type from the set {'VIEW', 'MODIFY','INSERT', 'COMPUTE', 'GRANT', 'AUTH', 'REVOKE', 'INT', 'DROP', 'SAVE', 'CREATE'} . Thus, execution of

define_auth(id,'EMP','INSERT')

creates an authorization which permits the user specified by id to insert tuples into relation EMP.

To determine if a user has the authorization to perform a particular operation, the boolena function

check_auth(id,rname,op)

can be invoked. This function returns the value true if an authorization exists for the user to perform the operation type on the specified relation and false otherwise.

Examination of the specifications of the RELATION module indicates how the appropriate use of exception conditions can be used to ensure that no unauthorized use of the data base can occur. The function

insert_tuple(r,R), for example, contains the exception condition

NO_AUTH(uid,R,'INSERT')

which indicates that the function cannot be successfully executed unless the user has been given the required authorization. Each function available to external users of the system contains such an exception condition and thus at the level of specification for the RELATION-INTEGRITY-SECURITY machine the security of the system is assured.

The INTEGRITY module is designed to provide facilities for defining the data types and allowable values for relation domains and for insuring that the defined integrity of the data base is not violated by users of the system. Data types of relation domains can be specified using the function

set_dtype(d,R,dt)

where d is a domain of relation R and dt is either a system defined data type from the set {real, integer, boolean, char} or a user defined data type. Using the function 'define_range' it is possible to specify that all values of a domain lie within a particular range of values. For instance,

define_range('AGE','EMP','≥',15,'∧','<',65)

creates an integrity assertion which specifies that all values of domain AGE of relation EMP must be greater than or equal to 15 and less than 65. In a similar manner it is possible to create an integrity assertion requiring that all values of a domain be contained in a specified set. Execution of

define_set('SEX','EMP', {'M','F'})

for example, creates an integrity assertion specifying that all values

for domain SEX of EMP must be in the set {'M','F'} .

Several functions of the module can be used to detect attempted user violations of the integrity of the data base. The boolean function check_dtype(d,R,v) returns the value true if the data type of variable v is the same as domain d of relation R and false otherwise. In a similar manner the boolean function check_value(d,R,v) returns true if v is an allowable value as defined by the integrity assertions for domain d of relation R and false otherwise.

Again, the definition of the RELATION module indicates how the specification of exception conditions can be used to ensure the integrity of the data base. The function insert_tuple(r,R), for example, contains the exception conditions

BAD_TUPLE(r,R)

and

BAD_TVAL(r,R)

That is, if the data type of any component of tuple r is not the same as the corresponding domain of relation R then the function cannot be successfully executed. Also, if any component value of r is a value not allowed by the integrity assertions of the corresponding domain of R, then the function cannot be invoked. Any function which somehow changes values in defined relations contains similar exception conditions to insure that inserted values are of the correct data type and are allowed by the corresponding integrity assertions.

The specifications of the INTEGRITY module are also designed to insure that non-meaningful integrity assertions cannot be created.

For example, the BAD_RANGE exception condition of the function 'define_range' prevents the creation of a range integrity assertion for a domain which already has values which violate the assertion. Likewise, a RANGE_CONFLICT exception condition prevents the creation of a range integrity assertion which cannot be satisfied (e.g. $i < 5 \wedge i > 12$).

The reliability of the security and integrity mechanisms is enhanced by the fact that they are also hierarchically implemented and verified. At the highest level the designer is concerned only with identifying and specifying the types of authorization and integrity control allowed in the system. The implementation and verification of these concepts then occurs in the same step by step manner as described previously.

VII. Hierarchical Performance Modeling

Performance evaluation is generally carried out after a system has been designed at a fine level of detail using simulation techniques, or more often, after the system has been constructed using measurement. This approach has the disadvantage that performance characteristics of alternative designs cannot be compared at the design phase, and may result in discarding a design at a detailed level due to performance reasons. In current modeling approaches a data base system is considered to be structured into two well-defined representations: logical and physical. The function of performance modeling, then, is to allow the system designer to determine the most appropriate physical structure to which to map his logical design. Based upon the results of current research efforts, however, there seem to be some problems with this approach.

For one, the complexity of the physical structure of a data base system makes it difficult to derive meaningful performance models. Simplifying assumptions which are often necessary to make the performance analysis tractable also significantly affect the utility of the model. Also, the results of such performance models are generally stated within the framework of low level query structures. The designer generally has no basis for which to judge the performance characteristics of high level user operations.

In the hierarchical design methodology, however, the distinction between the logical and physical description of a data base system is not so well defined. That is, through a stepwise process of design and specification the designer gradually refines an abstract representation of the system into a sequence of less abstract representations. Thus, at each step in the developmental process the designer is required to make decisions which determine the eventual performance characteristics of the system. However, in the hierarchical design methodology discussed so far, there is no provision for the designer to evaluate the effects of his design decisions at each level. As a result, the system developed may not be meaningful due to performance reasons.

In this section, we make an attempt to incorporate the performance evaluation into the design process. The technique to be presented will enable the system designer to construct a performance model in parallel with the design and implementation of this system. This performance model can be used at each stage to evaluate the performance characteristics of the current design. Based upon this evaluation the designer may derive alternative designs or backtrack to a pervious design level to achieve the desired system performance.

We propose a technique of hierarchical performance evaluation by describing the process at the i^{th} level of system design. That is, for adjacent machines M_i and M_{i-1} and implementation I^i_{i-1} the evaluation process consists of two phases:

1. a formal verification of the performance properties of each operation $f^i_j \in F_i$ based upon the implementation I^i_{i-1}, and

2. an analysis process in which the overall performance properties of implementation I^i_{i-1} on machine M_{i-1} based on the expected use of machine M_i is predicted.

The formal verification of the performance properties of operation f^i_j is accomplished through an inductive assertion proof on virtual program p^i_j (Wegbreit [1976]). This process results in a performance assertion which expresses the expected performance of f^i_j as a function of the expected performance of the operations of M_{i-1}. Applying this process at every level of design and implementation, then, results in a hierarchy of performance assertions which can be used to express the expected performance of each operation of the user level machine M_n as a function of the expected performances of the operations of the current machine design M_{i-1}.

The analysis process is achieved through the use of certain design and useage parameters. Each level of a hierarchically designed data base system represents a collection of abstract data objects which in some way can be used to store and access data. Such data objects may correspond to indexes, links, directories or simple storage structures for data. The design parameters for each level then, represent information about

the expected number of, size of, and access time for each abstract object type for that level. Usage paramenters on the other hand relate information about the expected pattern of user operations. Consider, for example, a relational system in which a particular type of access path may exist for any relation domain. One usage parameter, then, would be the probability that a user operation specified a domain for which this access path existed. A design parameter, however, would be the average probability that any domain would have this access path.

The first application of these parameters is in the derivation of the expected storage requirements for the system. That is, certain design and usage parameters may be used to form an expression which relates the expected storage requirements of each object type of M_i in terms of storage requirements of each object type of level M_{i-1}.

The second part of the analysis uses the hierarchical performance assertions derived in the formal verification process to achieve a case by case analysis of the properties of the design of M_{i-1}. Each case corresponds to a different set of assumptions regarding the relative values of design and usage parameters.

Results from this analysis can be used by the designer in two ways. First, they can be compared with the analysis results for an alternative design to determine the most appropriate implementation path. Next, the results can be used to aid in any data base design process for the system.

There are several advantages to this approach to performance modeling. For one, the nature of the hierarchical model constructed using this approach enables the derivation of the performance properties

of the user level structures and operations at each step in the design process. This has not been achieved in current performance models. This method should allow the construction of more complex models of data base system performance. The hierarchical nature of the model enables performance issues to be understood at different levels of abstraction. Moreover, simplifying assumptions can be made at any level to reduce the complexity of the model as necessary. This approach also provides the designer with a much greater degree of flexibility. At each level the designer is free to evaluate the performance characteristics of alternative designs and choose the one which most satisfies his requirements. Finally, the performance analysis should make the actual data base design relatively straightforward. That is, the structure of the performance model is such that values of design parameters may be altered at each level to obtain performance characteristics of a wide range of data base designs.

An example illustrating the hierarchical performance evaluation technique is given in Appendix E.

It should be pointed out that the technique presented in this section is a very crude first try toward the development of hierarchical performance models. In general, design alternatives may be characterized by means of a decision tree. At the top levels of the tree, the designer has to make basic decisions regarding the structure of the system while detailed decisions about smaller subsystems are made at lower decision nodes. At top levels, performance models are only used to decide whether certain branches of the decision tree should be extended or whether searches along these branches should be stopped at this time

since these avenues are not likely to yield satisfactory designs.

It is not possible to have accurate models at the conceptual design level since accurate models require the specification of detailed system parameters which the designer cannot provide at top levels of the decision tree. Gross models focus attention on a small number of key parameters which can be specified by the designer. These models should test the sensitivity of system performance to assumptions made in the model.

Another important feature of the use of hierarchical performance modeling is that it enhances the development of "families" of data base systems which share certain operating characteristics and yet which are implemented in different ways. This concept is perhaps best illustrated by the tree structure shown in Figure 6. Each node in this tree represents a particular machine design at some level while the arcs between nodes are implementation "pathways". (Thus, for example, the root node represents a user level machine while each terminal node corresponds to a different implementation machine or operating environment.) Any node above the first level may have one or more sons, each of which is a possible implementation alternative for the node. A complete data base system design, therefore, corresponds to a single pathway through the tree.

It should be clear from this diagram that the methodology provides the designer with great flexibility in developing alternative designs for different operating environments. The cost of developing these

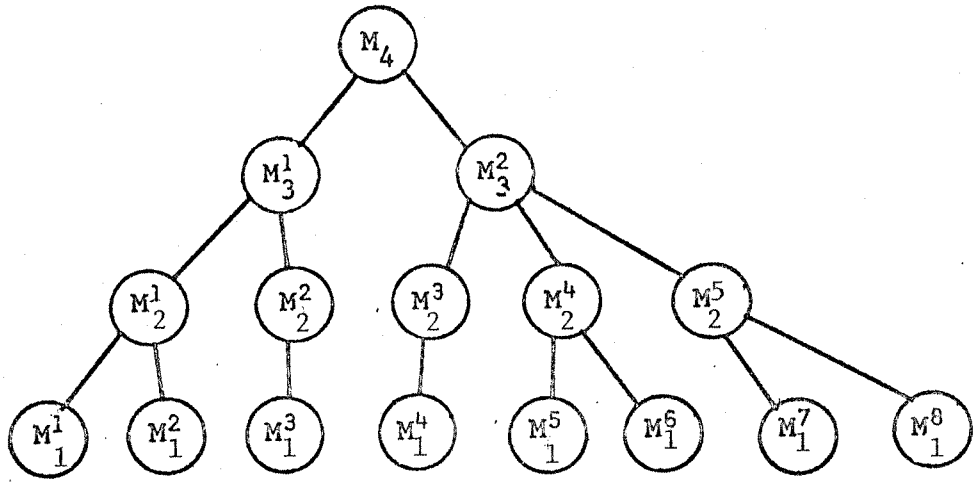

FIGURE 6 Tree representation of a "family" of data base systems. Each path through the tree (e.g., $\{M_4^1, M_3^1, M_2^1, M_0^2\}$) is a complete system design.

alternative designs is minimized because they may share design characteristics as much as possible. This, of course, enhances the portability of a particular data base system design because small perturbations in the operating environment require only small changes in the system design.

Perhaps the best example to illustrate the "family" concept is the development of a data base system to operate in a network environment, each node of which represents a different operating environment. The methodology would allow the construction of system software for each node in the configuration at minimal cost because design decisions would be shared among the systems as much as possible. Moreover, as new nodes were added to the configuration appropriate systems could be achieved by slight modifications to existing designs.

VIII. An Assessment of the Methodology: Promises and Problems

The methodology presented in this paper is but a small step in the development of a design theory for data base systems. This approach to data base system design and implementation 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. 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.

3. Localized Effects of Modification

A data base 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 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. For a hierarchically structured system discussed in this paper, at each level of the system design an abstract concept is realized by a formally specified (Parnas) module. Because the module structures hide all aspects of the implementation, modifying a machine design or implementation requires only localized changes in the system.

4. 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. The formal specification of each level also serves as a record of the design decisions made during the development process. As the design process unfolds, it is important to document the design cycle itself, document which alternatives were rejected and for what reasons, and document model inputs and predications. Such documentation is helpful in avoiding earlier mistakes, provides continuous evaluation of the models and enhances the understandability of the final product.

5. Formal Specification of Exception Conditions

The formal specification of each module provides for the designation of exception conditions for machine operations. As a result, integrity and security checking can be automatically specified. Also, the hierarchical nature of the design process enables the specification of exception conditions at the most appropriate level of abstraction.

6. Machine and Application Independence

The operational view of data semantics inherent in Parnas modules enhances the achievement of machine and application independence in the data base system.

7. Design Flexibility

The methodology enables the development of data base system "families" in which systems are implemented differently and yet share design concepts as much as possible.

8. Hierarchical Performance Modeling

The methodology includes a performance modeling technique which enables the designer to estimate the performance characteristics

of his design at each stage of the development process.

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. Systems which are intended to support multiple users are typically designed and modeled as single user systems. Necessary modifications to support concurrent access are accomplished once the design is completed. This, however, can have a substantial impact upon system performance. Guidelines, therefore, need to be developed for incorporating the concept of multiple user support into the initial phases of design and modeling.

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 to formalize the technique presented informally in this paper. The performance modeling sybsystem 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. It seems that some of the existing work in queueing network theory and Baysian decision theory could be useful in this regard.

4. Automatic aids are needed for the documentation of design process.

5. Incorporating intelligence in the specification and/or design. Although this is a problem that researchers in artificial intelligence area have been working for many years, it seems that this problem is

finally tractable, at least to a certain degree, due to emergence of many similar results in artificial intelligence, data base engineering, and software engineering. For example, the notion of a "demon" as a basic unit of intelligence used in many AI systems is discovered independently in data base systems as a "trigger" system, and in programming the ⟨ON⟩⟨condition⟩ statement in PL/1.

Our overall assessment is that the time has finally arrived for the development of a comprehensive design methodology for data base systems.

IX. Acknowledgment

It is a pleasure to acknowledge the stimulating discussions and suggestions of Daniel Chester, Sakti Ghosh and T. L. Kunii during the evolution of this paper.

Acknowledgments also are due to National Science Foundation for partial support of this work under grant DCR 75-09842
and to IBM for the support given to the first author under an IBM fellowship.

X. References

1. Astrahan, M.M. et al [1976], "System R: Relational Approach to Database Management", ACMTODS, Vol. 1, No. 2, 97-137.

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

3. Bayer, R., and McCreight, E. [1972], "Organization and Maintenance of Ordered Indexes", Acta Informatica, Vol. 1, No. 3, 173-189.

4. 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.

5. Codd, E.G. [1970], "A Relationsl Model of Data for Large Shared Data Banks", CACM, Vol. 13, No. 6, 377-387.

6. Engles, R.W., [1972], "A Tutorial on Data Base Organization", Annual Review in Automatic Programming, Vol. 7, Part 1, 1-64.

7. Eswaran, K.P., [1976], "Aspects of a Trigger Subsystem in an Integrated Data Base System", Proc. 2nd Int. Conf. on Software Engineering, 243-250.

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

9. Hammer, M. and D.J. McLeod, [1975], "Semantic Integrity in a Relational Data Base System", Proc. Int. Conf. on Very Large Data Bases.

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 from Interfering with Programs", IEEE Trans. on Software Engineering, Vol. 1. No. 1, 19-25.

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

13. Mealy, G.H., [1967], "Another Look at Data", Proc. AFIPS, Vol. 31, 525-534.

14. Mylopoulos, J., S. Schuster, and D. Tsichritzis, [1975], "A Multi-level Relational System", Proc. NCC 1975, AFIPS Press, 403-408.

15. Parnas, D.L., [1972], "A Technique for Software Module Specification with Examples", CACM, Vol. 15, No. 5, 330-336.

16. Parnas, D.L., [1976], "On the Criteria to be Used in Decomposing Systems into Modules", CACM, Vol. 15, No. 12, 1053-1058.

17. Parnas, D.L., [1976], "On a Buzzword: Hierarchical Structures:, IFIP Proc.

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

19. Schnid, H.A. and Swenson, J.R., [1975], "On the Semantics of the Relational Model", Proc. 1975 ACM-SIGMOD Conference, 211-223.

20. Schmid, H.S. and P.A. Bernstein, [1975], "A Multi-level Architecture for Relational Data Base Systems", Proc. Int. Conf. on Very Large Data Bases, 202-226.

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

22. Smith, J.M. and D.C.P. Smith, "Data Base Abstraction:, CACM, (to appear).

23. 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).

24. Wegbreit, B., [1976], "Verifying Program Performance", JACM, Vol. 23, No. 4, 691-699.