AD-A042 482 MINNESOTA UNIV MINNEAPOLIS MANAGEMENT INFORMATION SY--ETC F/G 9/4
DEVELOPING A DATA PERSPECTIVE ON THE SPECIFICATION OF INFORMATI--ETC(U)
MAY 77 G C EVEREST, O BRAY, I VALTERS N00167-76-M-8476

UNCLASSIFIED MISRC-TR-77-05 NL

1 OF 2
AD
A042482

(12)

Report Number: MISRC-TR-77-05

DEVELOPING A DATA PERSPECTIVE ON THE SPECIFICATION OF INFORMATION SYSTEM REQUIREMENTS

Management Information Systems Research Center
Graduate School of Business Administration
University of Minnesota
Minneapolis, MN 55455

1977 May

FINAL REPORT under contract N00167-76-M-8476

UNCLASSIFIED: DISTRIBUTION UNLIMITED

Prepared for and delivered to:

David W. Taylor Naval Ship Research and Development Center
ATTN: Code 1821, Dr. David K. Jefferson
U.S. Department of the Navy
Bethesda, MD 20084

Report Number: MISRC-TR-77-05

DEVELOPING A DATA PERSPECTIVE ON THE SPECIFICATION OF INFORMATION SYSTEM REQUIREMENTS

Gordon C. Everest, Principal Investigator
Olin Bray
Indulis Valters

Management Information Systems Research Center
Graduate School of Business Administration
University of Minnesota
Minneapolis, MN 55455

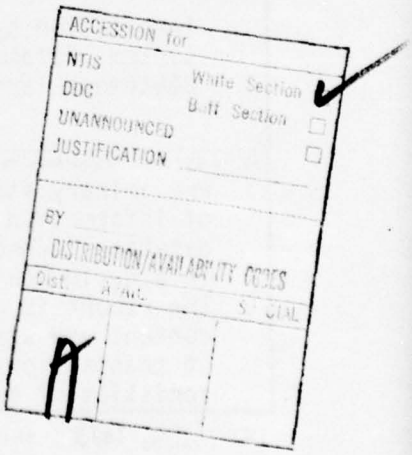
1977 May

FINAL REPORT under contract N00167-76-M-8476

UNCLASSIFIED: DISTRIBUTION UNLIMITED

Prepared for and delivered to:

David W. Taylor Naval Ship Research and Development Center
ATTN: Code 1821, Dr. David K. Jefferson
U.S. Department of the Navy
Bethesda, MD 20084



UNCLASSIFIED, DISTRIBUTION UNLIMITED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) MISRC-TR-77-05	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) Developing a Data Perspective on the Specification of Information System Requirements		5. TYPE OF REPORT & PERIOD COVERED FINAL REPORT 1976/09/20 - 1976/12/20
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Gordon C. Everest, Principal Investigator; Olin Bray; Indulis Valters		8. CONTRACT OR GRANT NUMBER(s) (15) N00167-76-M-8476
9. PERFORMING ORGANIZATION NAME AND ADDRESS Management Information Systems Research Center, Grad. School of Bus. Admin., 269 19th Ave. So. Univ. of Minnesota, Minneapolis, MN 55455		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (11)
11. CONTROLLING OFFICE NAME AND ADDRESS David W. Taylor Naval Ship Research & Development Center; ATTN: Code 1821, Dr. David K. Jefferson U.S. Department of the Navy; Bethesda, MD 20084		12. REPORT DATE 1977 May 77; 13. NUMBER OF PAGES 125
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) (9) Final rept. 20 Sep – 20 Dec 76,		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) DISTRIBUTION UNLIMITED. (12) 121p.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Information system requirements, database system, database management system, database design, database definition, data dynamics, problem statement language.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Preliminary statement of the information needed to prepare a specification of information system requirements; developed from the perspective of database technology. The information would be collected from users, stored in a design database, and used to support the process of database design. The report is divided into four sections pertaining to information about: content and structure of the database, system data including the definition of transactions and reports, application processes, and behavioral characteristics of processes and data.		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED, DISTRIBUTION UNLIMITED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409153

AB

PREFACE

This report provides a preliminary statement of information to be collected in preparing a specification of information system requirements. This preliminary statement has been developed from the perspective of data and database management systems. Ultimately, this information would be collected from users and stored in a design database. The information should provide an adequate description of the user problem environment and should be sufficient for the process of file design.

This report is divided into four major sections:

1. information about the content and structure of the database.
2. information about the system data maintained in support of the database system (definitions of reports and transactions).
3. information about the application processes which act upon the database.
4. information about the behavioral characteristics of the processes and the data.

The organization of the material is motivated from the perspective of the user who supplies the information and for convenience in writing this report. It is not organized for the designer who will use the information once stored in the design database, nor does it necessarily indicate the importance of each piece of information to the design process. There is no correlation between the amount of space and attention given to a topic in the report and the level of importance to the file designer or the proportion of use in subsequent design procedures.

This is a distinctly preliminary statement and as such will be rough, vague in spots, and incomplete. An attempt was made to cover all the relevant areas of

required information even at the expense of including some sections that are sketchy at best. At least the spot is made explicit where further work needs to be done.

Subsequent research will compare this preliminary statement to actual user experiences in designing and developing database systems, to well-developed database design procedures, to manuals of existing DBMS's, and to any relevant literature. Through this process the preliminary statement can be refined, clarified, and made more complete. The current report serves as a springboard for this subsequent research activity.

This final report is partly based upon the material in the items of a bibliography prepared under a previous contract with the Navy, entitled An Automated, Annotated Bibliography on the Specification of Information System Requirements (Minneapolis, MN: University of Minnesota, Management Information Systems Research Center, MISRC-TR-77-01, 1976 October 11, 156 pages; Final Report under contract N00167-76-M-8441; available from NTIS, AD A038 398). The Bibliography constitutes the set of references which could logically be appended to this final report.

A description of the context for the information in this final report is contained in the research proposals for this contract (8476) and the prior bibliography contract (8441). Portions excerpted in Appendix A.

TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

This report provides a preliminary statement of the information necessary to describe an application/database system environment which is the object of a design process. The information is gathered in a design database. In this report the information is divided into four areas: information about the database, system data, application processes, and behavioral characteristics.

First, it is necessary to make some observations on several interesting aspects of the relationship between the database and the system data, between the defined data/system data and the application processes, and between exogenous and derived information.

The Database versus System Data

When speaking of a database system, the term "database" is used in its broadest generic sense, namely, the collection of data relevant to the operation and management of an enterprise. Database is not used in the narrow sense of referring to a physical entity stored under a DBMS although that would likely be the ultimate representational form of a database.

From a data perspective, the database is the most obvious system component which must be defined, but other components exist which must also be defined -- input data transactions, interrogation requests, output reports, users, mappings between two data structures, validation criteria, Boolean selection expressions -- in fact, the list gets longer as we consider formally capturing more information about the object system of application processes and files which is the object of the design activity. All of the information which defines these nondatabase components is called "system data."

A picture of the application/database system which is the object of the design activity is shown in Figure 1. The components are divided horizontally

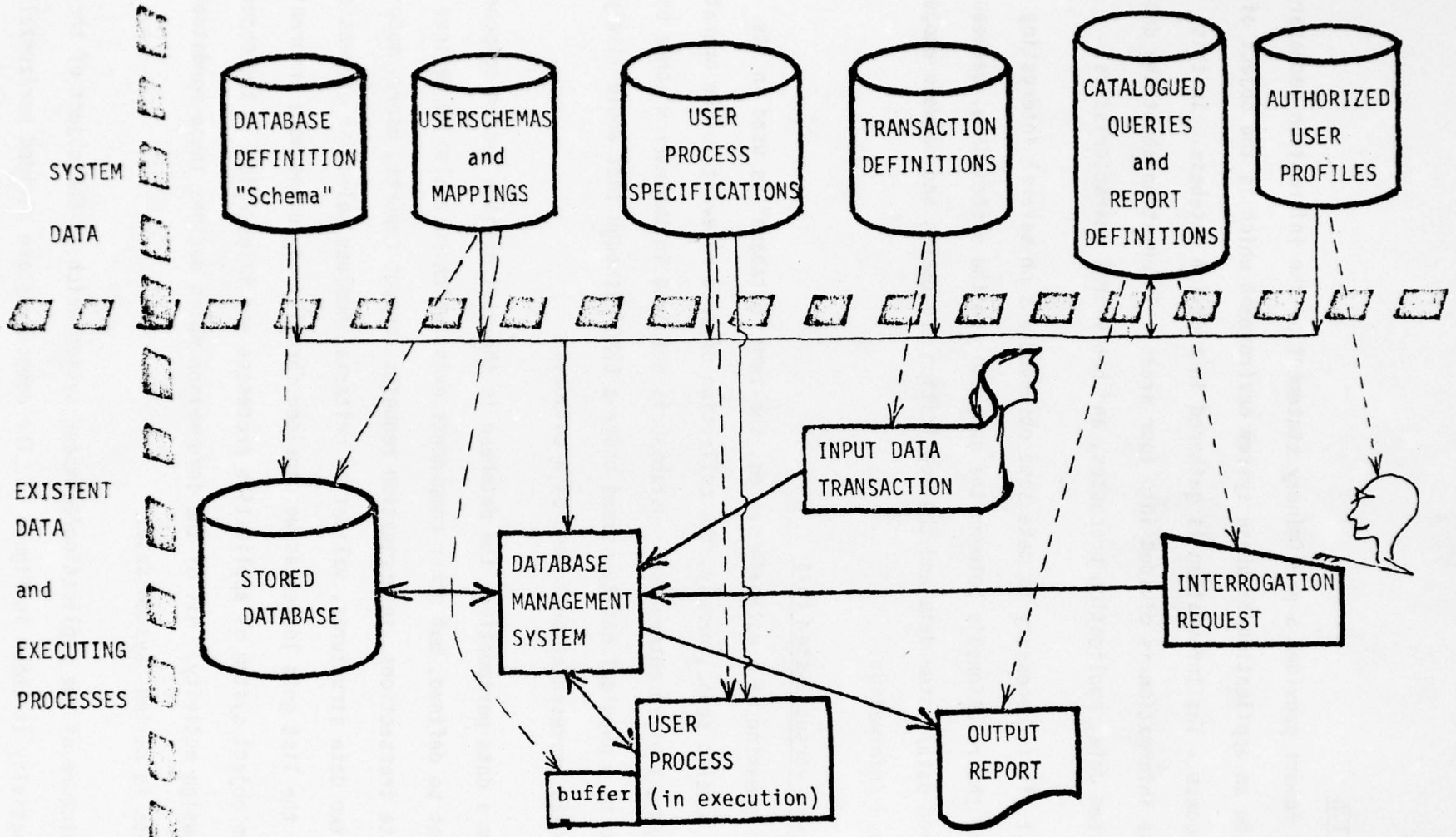


Figure 1. Components of the Object Application/Database System.

into system data, and the real data (whether in the database, input transactions, or output reports) and the processes which act upon the real data. The system data above the horizontal line describes the various components below the line, which is the meaning of the thin broken lines. The database management system has access to all the system data during execution, which is not shown in the figure.

The Duality of System Data and Processes

In contrast to data which is a passive component, a process is an active component of a system. A process is something that happens over time; something that acts upon data; something that moves and transforms data. There are processes which create and maintain a database schema, which create and maintain data according to the definition, which create and maintain system data such as transaction or report definitions, which process transactions, which process interrogation requests, and which generate output reports. Processes which operate on system data are sometimes called utilities. System data so established then drives the processes which act upon the real data -- in the database, transactions, queries, or output reports. This second set of processes is embodied in the database management system which in turn may interact with user-defined application processes.

Could an object application/database system operate with no explicitly defined system data? Certainly, and the information would be embodied or buried in the programs; all the "intelligence" about the database and related application processes would not be formally stated. The only necessary prespecified process would be the compiler for the programming language. In effect, the system data becomes the set of all written programs.

Every process needs a process specification (commonly called a program). A process specification can be written in a programming language or a special purpose language such as that used to write a report definition or a Boolean selection expression. The process specification can be stored as is, or it can be "compiled" or translated into another form of process specification written in another language. Eventually, the process specification is "interpreted" or executed by a processor which understands the language. The execution generates a process, which might also be called an "instance" of the process specification.

The lower the level (more micro) of process specification, the longer the time it takes to develop the process specification, the more prone it is to error, the more difficult it is to comprehend, and the more difficult it is to modify later. (These are all simply the arguments for using high-level languages.) Using a low-level language, the programs are highly tailored. The trend in the evaluation of programming languages, however, has been to factor out the process specification to higher levels -- the Data Division and Report Writer of COBOL are good examples. Then the underlying processes become more generalized in interpreting a broader range of high-level input statements. The database management system becomes a highly generalized interpreter of the system data. The intelligence of the object application/database system is embodied in the system data, not the underlying processes.

A central objective in the development of the design database is the explicit, formal expression of all the information or intelligence that is to drive the designed object system. The formal expression of system data should be at the highest possible or macro level and in a form that most directly relates to the real world environment of the object system. This calls for the existence (or development) of very high level languages with which to formally express and capture all the system data, that is, all the intelligence information needed to

drive and guide the ongoing operation of the database system. The information to be stored in the design database is actually all the system data. Hence, the central thrust of this research is the development of high-level languages for defining all the system data.

Exogenous versus Derived Information

The objective in developing a specification of information system requirements is to capture all and only the exogenous, user-oriented information about the data structure, the application processes against that structure, and the behavioral characteristics of processes and data. The remaining information needed to ultimately establish a database system is then derived from the user-supplied exogenous information using appropriate database design procedures -- some which may not now exist but which we know must exist, or some which are not yet precisely defined.

Most information relating to how data is to be physically stored is actually derived information, even though many DBMS's today require explicit declaration of such information. Since physical organization and access paths are a major determinant of DMS performance, they should be designed in response to user-stated performance requirements. Currently, the DBA must provide much of this information, based partly on algorithms and partly on judgments. Actually, given a more complete description of the data, its behavior, and the processes that operate on it, much of these physical design decisions can be derived by the system. Initial design decisions must be based on user estimates, but revision and reorganization can be made after monitoring actual performance.

The following are examples of information currently supplied directly by the user in many systems but which could be derived by appropriate database design procedures.

- physical record placement, that is, proximity of records to each other.
- ordering of record instances as stored.
- ordering of dependent records within each parent.
- item value encoding.
- representation of a relationship.
- inter record connections.
- access methods (indexing or hashing on what data items).
- padding of storage space to allow for future growth.

More and more of the information assumed exogenous today will be derivable tomorrow as more database design procedures are developed.

In many cases, it may be desirable to have users provide the nonexogenous information as they are now accustomed to doing. The same information can be derived by the design procedures based upon the other exogenous information such as performance requirements, response requirements, volumes, and other behavioral information. Then the information as derived can be compared to the same information as supplied by the user. Significant deviations can be reported to the user. This can serve to educate the user, to validate the user's expression of requirements, and to verify the accuracy of the builtin design procedures.

I. DATABASE DEFINITION

Although considerable information may exist about a database structure, we are here only concerned with the information that must be expressed directly by the user. This section attempts to specify all the information needed to accurately model the reality of the user environment in a database structure. We assume an entity-attribute-relationship construct for this modelling process. We need a user-oriented formalism for defining a database model of the real world; a formalism which is unconstrained by any particular representational model and unencumbered by considerations of physical data storage. The formalism must be embodied in a graphical notation and that in turn must have a linear form which is convenient for machine processing and storage in a design database.

The definitional process will attempt to explicitly capture all relevant information, leaving nothing hidden or assumed. For example, all relationships in the data will be based upon criteria expressed in terms of data content. The execution of processes may depend upon time or priority attributes that exist in reality. The design process may choose not to represent such attributes but, from a user perspective, they are real and should be defined.

The Entity-Attribute-Relationship Model

The entity-attribute-relationship model of data is preferred because it seems to relate most directly to the real world of a user. The pivotal construct is the entity. The user first identifies classes of entities, describes them in terms of attributes of interest, and then describes all first order (that is, direct) relationships between entities. This report specifies the characteristics of attributes and relationships to be defined by the user. Then the user supplies additional information in the form of validation rules and semantic constraints on the defined data structure, and behavioral information such as volumes and rates of change.

An entity is a real world thing or event of interest and on which data is to be collected and stored. An entity type is a named class of entities. An entity instance is a collection of attribute-value pairs describing an entity. An attribute is a named characteristic of a class of entities. A data item is a stored representation of an attribute. Each attribute/data item has a domain of values from which values are selected to describe an entity instance. A relationship is a correspondence or association between entities.

Database Schema and Userschemas

The complete logical and physical definition of the stored database is called the database schema. A userschema is the view a particular user or application process has of that database. It can be limited to relevant portions of the stored database and also define additional data which is derivable from the stored database. If the userschema is the data the user sees and needs, who defines the schema? What is it that relates to the specified information requirements of the using environment?

In some cases, the schema could be considered derived information. Each user's requirements would be expressed in one or more userschemas and the Database Administrator would then develop a composite schema which satisfies the collective information needs of the using community. In the case of an already established database, the schema would be taken as given and a mapable userschema defined to meet the information needs of a new user application. Whichever is the case, two principles emerge: (1) the user never directly specifies the schema of a stored database, and (2) both schemas and userschemas are definitions of data and therefore require a data definition language facility. The central requirement of such a data definition language is to model the reality by capturing all the essential information about the entities, attributes, and relationships in the user's reality. No further distinction between database schema and userschema will be made here. The only further requirement is the definition of a mapping between a userschema and a database schema but this would be provided by a database administrator, not a user.

ENTITIES

An entity is any object, thing, or event in the real world of the user; anything about which data is to be defined, collected, and stored.

Entity Type

An entity type is a named class of entities. Its characteristics include:

1. Name of the entity type.
2. Description - A narrative explaining the named class of entities.
3. Criteria for Inclusion - What special characteristics, if any, an entity instance must exhibit to be included in this entity class? Frequently all entities of a particular type will be included in the same class. However, if there are no special membership rules then the general rule should be stated explicitly. For example, a personnel file contains all personnel currently employed by the firm.
4. Criteria for Retirement - What characteristics must the entity instance exhibit to be removed from the entity class? Logically, this criteria is the inverse of the above criteria for inclusion. There may be a set of criteria for retirement, with retirement to occur either when any of the criteria is met or when all of the criteria are met. For example, a person would be retired from the personnel file when resigned, fired, or deceased.
5. Derived - In some cases different entity types may be related to each other. A derived entity type can be obtained from another type based on some construction or derivation rule.

Behavioral characteristics relating to entity type include the number of types, the number of instances of each type, and the growth or decay in these numbers.

Entity Instance

An entity instance is an unnamed collection of attribute-value pairs describing a real world thing or event of interest and on which data is collected and stored. Some entity instance characteristics are:

1. Date and Time of creation.
2. Date and Time of last change.
3. User/process which created the instance.
4. User/process which made the last change to the instance.

Characteristics one through four are derived based on performance monitoring of the database. One and two are needed for backup and recovery procedures. Three and four are needed for audit trails and (along with other information such as patterns of retrieval) for later analysis to determine reorganization and restructuring points.

5. Size - This is the aggregate of all value sizes plus any additional system information created and appended to the entity instance. The user should not have to consider or be concerned with this characteristic.

Behavioral characteristics of entity instances include the rate at which instances are changed both with respect to time and to the number of instances (e.g. 200 changes/day and ten percent of the instances are changed per month) and the size both of individual instances and of all instances of a given type of entity. The system monitoring should provide both average behavior and extremes (maximum and minimum), or even develop a frequency distribution.

Entry

A file is a collection of entries each conforming to a common definition. Each entry describes a particular entity instance in the real world. The file contains descriptions of all the entities of a class of entities.

An entry is similar to a record (which has a more physical connotation), so a file could be defined as a collection of records of the same type. In the simplest case, an entry would consist of a set of single-valued data items. Such a file is called a flat file. Adding multivalued items and repeating groups to an entry definition results in a more complex definition for an entry. Each nested repeating group corresponds to a subentity which has a special relationship with the entry, namely, a hierarchical relationship. These characteristics of an entry are discussed in following sections.

ATTRIBUTES

An attribute is a named characteristic of a real world entity. The name should be unique for a given entity type, but can be the same for different entity types (e.g. age can apply to both personnel and equipment). Synonyms may be used. A data item is the formal representation of an attribute within the system. A data item takes on values from a single domain of values or value set. Except for multivalued items a data item takes on a single value for each entity.

An attribute or data item has the following characteristics:

1. Name of the attribute.

2. Description - A narrative explanation of the attribute.

3. Role - An attribute can be used in any of three roles. First, it may be a unique identifier used to specify the entity of interest. Second, it may be a key used for the selection of a set of entities of interest. While an identifier is a key, this more general definition of a key does not require uniqueness. The entities of interest are those which meet a certain Boolean selection expression which contains the names of various key attributes, values, and operators (arithmetic, relational, and logical). The third role of an attribute is to describe the entity. The distinction between a key and a descriptive attribute may be in the use not the attribute itself. While some attributes may in practice never be used as keys, in principle any attribute can be used to select an entity and therefore be a key. When it is not used as a key, an attribute is simply a descriptor.

4. Value Domain/Type - The value domain is the central characteristic of a data item. It can be explicitly specified by enumerating all of the acceptable values or by specifying the range from which the values can be taken. This specification defines the value set of the data item or attribute. (The way the values are physically stored is the value representation and should not concern the user.) The value set may be defined implicitly be specifying the type of the attribute.

Any value encoding should be specified if it is meaningful to users, including a description and meaning of each value. Behavioral characteristics include the size and growth rate of the value set.

5. Existence - For each entity type each attribute may be mandatory, optional, or irrelevant. If mandatory, the attribute is an essential characteristic of the entity. Optional attributes may or may not apply to a specific instance of the entity type. The existence of an optional attribute is conditional on some characteristic of the entity or its relationship to other entities. If the particular characteristic or relationship, specified by a Boolean expression, is met then for that entity instance the attribute is mandatory. If the expression is not met, then the attribute is irrelevant and does not exist for that entity instance. Any of this information that can be specified to the system can then be used in more complete validity testing.

Identifier attributes or characteristics are usually mandatory. Many, but not all, keys are also mandatory. Descriptive attributes can be optional or mandatory.

6. Number of Value Occurrences - For a given entity an attribute may have one or many values. Whether the attribute is single or multivalued is a property of the attribute, e.g. a person has only one job code or spouse, but can have multiple skills or children. For multivalued attributes the actual number of occurrences may be constant or dependent on the specific entity. If the number of values is in some way dependent on other characteristics of the entity, i.e. the number of occurrences can be predicted, limited, or bounded, this dependence should be specified so the system can maintain it. In general, the number of attribute values can be specified in the following ways: (1) constant or fixed; (2) predictable based on certain entity instance characteristics; (3) variable but bracketed within a certain range, e.g. more than one but less than ten; (4) variable but where either a maximum or minimum number of occurrences can be specified; or (5) nondeterministic but where some form of frequency distribution can be specified (or approximated) for the number of value occurrences per record (or parent group).

Although the values of a multivalued attribute may have an ordering, e.g. skill codes could be ordered by difficulty, the occurrence of the multiple values have no inherent meaning. In other words, the skills for an individual could be considered ordered by when acquired or level of proficiency. These additional factors are attributes of each skill and should be defined and provided explicitly, rather than implicitly by the ordering.

Behavioral characteristics for number of occurrences include the average, and range of occurrence for each multivalued attribute.

7. Length - A user may specify the length of an attribute either explicitly or implicitly by giving the value set of the attribute. A length may be fixed or variable. However, this is basically an efficiency question with which the user should not be concerned. Factors which affect the decision include the frequency of existence of the attribute, the number of occurrences and how predictable they are, and the distribution of the value occurrences for the attribute and their length. Therefore, the actual length of an attribute is a derived characteristic, just as record length.

Behavioral characteristics of length would include a frequency distribution of length and changes in the frequency distribution. This distribution can be derived from a frequency distribution of the values and their representation.

8. Units of measure - provide a way to interpret the actual stored value. This is necessary if attributes of the same value class (therefore comparable), but different domains are to be operated on, for example, money class, but one in dollars and the other in pounds. Units of measure information would allow the system to make many automatic conversions for the user. Two pieces of information are important to the unit of measure:

radix - the base against which the value is to be interpreted.

subfields - in this case an attribute may need to be interpreted as an organized set of subfields. For example, time may be specified as

year, month, day, hour, minute, second. The significance of these subfields becomes important when computing the period between two time points or doubling a time period. However, the conversions and shifts between subfields should be done by the system without user intervention once the appropriate units of measure have been specified. Another example would be with geographic coordinate systems such as X, Y, Z or latitude and longitude.

Value Class and Operators

Attributes can be of three general types - arithmetic, string, or special. The arithmetic types have some value interpretation. They may be simple scalars or compounds of these. Arithmetic items can have a variety of representations such as integer, decimal, exponential, fraction, probability, or percentage.

The value class of an item and the operators that can operate on it are closely related. It may not make sense to allow operators of one class to operate on the items of another class (e.g. arithmetic operators on string items). However, within a value class there may be different ways to represent the same value (e.g. integer and real for arithmetic data). In this case the user should be able to freely mix the ways in which the items are represented and the system automatically convert them to a common representation before performing the operation. However, for consistent and predictable results there must be a procedure ordering or a rule which defines how the conversions will be made. For example, if the result is real then all integers in the expression would be converted to real and real operations performed on them. Depending upon how the value classes are defined, there may not be any comparable automatic conversion between certain value classes. Different value classes tend to be used to represent different types of attributes.

If two attributes have the same value class, then they have both the same value set and the same set of operators. However, the values may have different representations. This should not concern the user, in fact he should not even need to be aware of it. If the attributes are of the same value class, the system should make the conversions between the various representations automatically. This also implies that when the user defined the value set he also defined the units of measure for the value.

The arithmetic type implies some quantitative relationship among the values in the value set. The most frequent relationship is ratio, i.e. the attribute has a zero point and the intervals on its value scale are equal. All of the common arithmetic operations can be applied when this relationship holds. The next less rigorous relationship is interval. It is similar to the ratio scale except that there is no zero point on the scale. Temperature and date are examples of this relationship. The difference between 0 and 10 degrees is the same as the difference between 90 and 100, but it makes no sense to describe one temperature as twice as hot or cold as another one. All the relational operators (EQ, NE, LT, GT, LE, GE) apply to interval values. The least rigorous relationship is ordinal. In this case, matching and ordering relationships apply; the values can be ranked -- one value is greater than or less than another value, but the difference between two neighboring values does not have to be the same. Moving from ratio to interval to ordinal fewer of the traditional arithmetic operations are legal. In the extreme case with nominal values no arithmetic operations or relations except equality are applicable.

There is also a class of mathematical value sets which are composed of scalar numbers but subject to additional composition rules and operators. These include complex numbers, vectors, matrices, tensors, etc. Groups of the simple arithmetic items are collected into these larger entities and new operators are defined on the group (e.g. dot product, cross product, pre-multiple, post-multiple, inverse, transpose, etc.). These operators are collections of the simpler operators. With these entities the relational operators normally do not apply. The entities tend to be equal, not equal, or not comparable (e.g. different dimensions). Automatic conversion of the components of these entities is possible (e.g. integer complex to real/floating complex), but such conversion between the larger entities cannot be done (e.g. convert a vector to a matrix).

Strings - are formed from an alphabet, such as bits, numbers, or alphabetic characters. A string value consists of a string of instances from the alphabet. (Either a bit or a character.) A string may be either fixed or variable length. String operators may change a string's length, content, or both. The main operators are concatenate, deconcatenate, replace, and substring match (search by content), or position find (which is preparatory for another operator and does not itself change the string). Searching a string, which is implicit in both the replace and the find, may be in terms of position or content. Textual data is a special type of string, where one might or might not keep any superfluous blanks. The alphabet is all alphanumeric and special characters.

Geographic Coordinates - The type of value locates points in an N dimensional space. There may be several different units of measurement (e.g. in planar system, X & Y, or P, θ or latitude, longitude from known origin). However, there is still the units of measure question, for example, is θ in degrees or radians. However, by knowing these units the system should be able to freely convert between them. To describe a point, one must provide its coordinates. For this value class there are several special operators: given two points (of the same dimensionality), compute the distance between them; given a set of points (of the same dimensionality) and an order to traverse them, define an enclosed area; given an enclosed area, determine if a given point is inside or outside the area. When time is related to geographic coordinates, a new set of operators are developed which define and operate on velocity and acceleration.

GROUPS

The previous discussion focused on data items (or elements), the smallest, unstructured components. (Exceptions such as time and geographic coordinates were represented by a collection of subfields. They were included under the discussion of value class of attributes because logically they are a single value.) Several individually meaningful attributes can be related or grouped together to represent a more complex attribute of an entity.

A group is a collection of named attributes which may be considered as a unit. A simple group is a collection of data items. A compound group is a collection of data items and/or other groups. This allows the nesting of groups, theoretically to any level. A group may have at most one value in each parent instance (nonrepeating) or it may be a repeating group (rgroup).

A group can actually be considered another entity type. It is like a subentity with a subordinate relationship to the entity within which it is defined. The subentity is presumed not to have an existence independent of its parent entity (which may be a higher level subentity). That is, the parent instance must exist before a related instance of the dependent can exist; if the parent entity instance is deleted, so will all subentities. It is also presumed that the attributes of the subentity must be interpreted in the context of the parent entity. (This corresponds to the hierarchical relationship defined later.)

The value of a simple group is considered to consist of the values of each constituent data item. For example:

the employee attribute	the value of
ADDRESS	ADDRESS
could be a group	for one entity
with the items:	could be:
APARTMENT NUMBER	B2
STREET NUMBER	1678
STREET NAME	LAPPAND AVENUE
CITY	MILLERSVILLE
STATE	VIRGINIA
ZIP CODE	23771

The second column is called the instance (or value) of the simple group defined in the first column above.

A nonrepeating group is actually an alternative naming mechanism to enable a user to reference the whole address or to reference parts of an address. It may also be desirable to define different groups over the same set of data items. For example, LOCATION may be just CITY and STATE. Furthermore, a compound group could be formed by alternatively defining ADDRESS as consisting of: APARTMENT NUMBER, STREET NUMBER, STREET NAME, LOCATION, and ZIP CODE, where LOCATION has replaced CITY and STATE. Similarly, the compound, nonrepeating group BIRTHDATE may consist of the three data items: YEAR, MONTH, and DAY.

A repeating group can have multiple instances for a particular parent entity instance, (analogous to a multivalued item). This represents a significantly different structural construct than the nonrepeating group. A repeating group can be either simple or compound. In effect, a repeating group is a little flat (simple) or hierarchical (compound) file within each parent entity instance. Thus,

all of the definitional information discussed for files can be repeated here (recursively). The following characteristics of repeating groups are particularly important.

Number of occurrences may be specified as a single fixed integer, a range of integers, an enumeration of integers, and the simultaneous exclusion of enumerations and/or ranges of integers. At a minimum, it is necessary to specify the maximum number of occurrences of instances of a repeating group for any given parent entity instance. More information is provided by a frequency distribution on the number of instances of the repeating group for an instance of the parent entity. For initial establishment of the database, the user would have to estimate the information regarding number of occurrences. After the database is implemented and used, the system can monitor this characteristic and perhaps even automatically initiate some form of reorganization if changes become significant.

Ordering may be defined on the occurrences of the group. Ordering may be implicit, i.e. in terms of either position or date/time of creation of the occurrence. However, usually the ordering, if it applies, will be explicit in terms of specific item(s) within the group. (See section on Ordering.)

Identifier item -- a data item (or items) in the repeating group may serve as an identifier of repeating group instances in the parent entity instance.

RELATIONSHIPS

A relationship is a correspondence or association between entities. Entities are related in the real world and the defined database structure should reflect these relationships. A relationship is defined between two (and perhaps more) entity classes or types, that is, between two entries or records. A relationship defined on the same entity class is called a reflexive relationship, as with parent-child on a person entity class.

In general, a relationship has a name. Although a relationship has no directionality from a static perspective, directionality is implied when a user addresses the data structure in a query language. The relationship is generally interpreted in the context of one entity or the other. Hence, the need for two names, depending upon the context of the query. For example, with EMPLOYEE and SKILL entities, the user may be interested in the skills of a particular employee (or set of employees) in which case EMPLOYEE is the context. Alternatively, the user may be interested in those employees possessing a particular skill. These two segments in a query could refer to the relationship as SKILLS-OF-EMPLOYEE or EMPLOYEES-WITH-SKILL, respectively. If, during the parsing of a query, the context is known unambiguously, then a single name for the relationship will suffice.

A relationship is described in terms of three characteristics: exclusivity, exhaustibility (or dependency), and degree. In addition, a relationship must be based upon some criteria which is expressed in terms of the value contents of the participating entries. A graphical schema diagram is a most helpful representational form of a relationship.

Different methods of formally representing relationships in a defined data structure give rise to various data models -- most notable are the hierarchical, network, and relational models. The representation of relationships and the associated data model is of little importance to the user. It is actually a

secondary design choice which is not part of the basic exogenous information needed from the user in the specification of information system requirements. The user should provide a generic definition of entities, attributes, and relationships unencumbered by any particular view or representational form. (Of course, the mere writing of the next few pages will develop another representational form, but hopefully it will be closer to the way the user thinks and more directly relatable to the real world being modelled in the database structure.)

Graphical Schema Representation of a Relationship

Each entity class (or record type) participating in a relationship is represented by a box labeled with the name of the entity record. A relationship between two entity classes is represented by an arc between the two boxes. The form of the arc depends upon the characteristics of the relationship between the entities (as discussed in succeeding sections). The arc is labeled with the name of the relationship (but can be omitted if no ambiguity results). Figure 2 illustrates the basic graphical representation of a relationship.

Figure 2.

Basic Graphical Representation of a Relationship

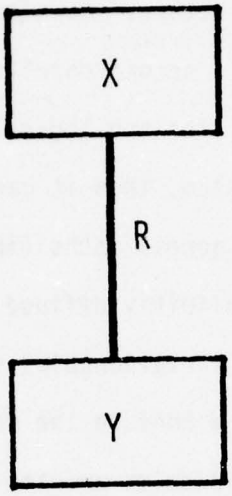


Figure 2. Basic Graphical Representation of a Relationship.

Reversibility, Directionality, and Accessibility of Relationships

All relationships are reversible. Given two entities, X and Y, if X is related to Y then Y is related to X, even if it is not explicitly represented in the database structure. As a corollary, no directionality is ever implied by a relationship. As a second corollary, there is no necessary dependence between physical access paths and logical relationships. If a relationship exists, and is known to the system, then it can be a basis for accessing entities in the database whether or not access paths (indexes, chains, pointers, hashing algorithms, etc.) have been explicitly defined. Since some widely used systems cannot access records unless a "relationship" is explicitly defined, we have become conditioned to comingle and confuse the notions of access path and relationship. The nondirectionality of relationships and the separation of logical relationships from physical access paths have both been further confused by the common practice of using an arrow to represent a "relationship."*

---

*See Charles W. Bachman, "Data Structure Diagrams," Data Base (1:2), 1969 Summer, pages 4-10; and John K. Lyon, An Introduction to Data Base Design, New York: Wiley, 1971, (Section 3).

In the absence of defined access paths, if a relationship is formally defined and known to the system, then entries or records can be accessed even if it requires an exhaustive search of the entire database. There is only an affect on performance, not on feasibility of access.

Exclusivity of a Relationship

The exclusivity characteristic of a relationship indicates whether an instance of one entity class (X) is related to at most one or more than one instance of another entity class (Y). If an X is related to at most one Y, then the X-Y relationship is said to be exclusive on Y. In set theoretic terms, such a relationship is a function from X onto Y, that is, given an X a unique Y is determined (or perhaps no Y, but that depends upon the exhaustibility characteristic). A function is a special case of a relationship. We say that the function maps or transforms X onto Y.

Not only must a relationship be defined in terms of its exclusivity on Y but is must be defined in terms of its exclusivity on X. If a relationship is exclusive on both entity classes, it is a one-to-one (1:1) relationship. In general, there are four possibilities for the exclusivity characteristic of a relationship: one-to-one, one-to-many, many-to-one, and many-to-many. Three are shown in Figure 3 along with their graphical representation and a sample instance picture.

Figure 3.
Exclusivity Characteristics of a Relationship

The fanning out of the arc visually communicates the inherent exclusivity characteristic of the relationship. As such it is preferred to the use of alternative graphical notations such as the arrow. The arrow can be misleading because it inherently communicates directionality and implicitly depicts an access path.

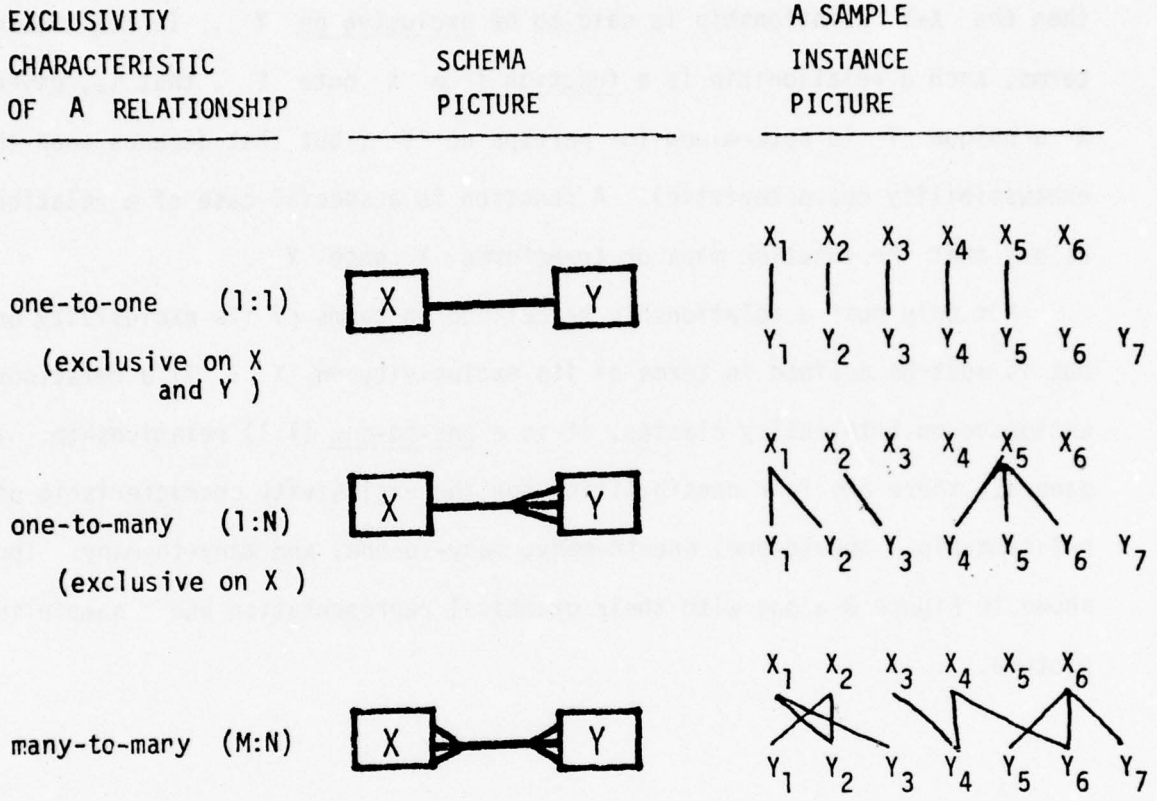


Figure 3. Exclusivity Characteristics of a Relationship.

If a relationship is nonexclusive on either entity class, it is desirable for the user to provide some information concerning the size or cardinality on the "many" side(s) of the relationship. For example, the relationship may be better characterized as "one-to-a-few" or "one-to-at-most-four" or "one-to-between-two-and-four." To the extent possible, the user should provide information concerning the cardinality of the relationship which is known or can be estimated. If the "many" is variable, the file design process is even better served if the user can specify a frequency distribution on the cardinality of the relationship in each direction. For a one-to-many relationship, such a cardinality frequency distribution might be as follows:

RELATIONSHIP: 1:N between X and Y

NUMBER OF Y's FOR AN X.	NUMBER OF X's WITH n Y's
n	f(n)
0	300
1	750
2	1000
3	500
4	250
5	200
TOTAL NUMBER OF X's =	3000

As a second-order behavioral characteristic it would be useful to know the direction and rate of shift in this frequency distribution. For example, of the number of Y's is growing, where are the new ones being added? If the number of X's remains constant, then the mean of the distribution is most surely increasing.

Exhaustibility or Dependency in a Relationship

The exhaustibility characteristic of a relationship indicates whether all or just some of the instances of one entity class (X) are related to instances of the other entity class (Y), in other words, whether or not the relationship exhausts the members of an entity class. From the opposite perspective, if the relationship exhausts the set of X's then every X must be related to at least one Y , and X is said to be dependent upon Y .

If a relationship between X and Y is a function of X onto Y , and the relationship exhausts X , that is called a total function; otherwise, it is a partial function. Notice that whether a function is total or partial with respect to X , implies nothing about whether or not the relationship exhausts Y . The exhaustibility characteristic of a relationship independently applies to each of X and Y . Therefore, there are four possibilities: some of X are related to some of Y , all X to some Y , some X to all Y , and all X to all Y .

The all or exhaustive characteristic represent more restrictive semantics on the data structure and its behavior. With a one-to-one relationship between X and Y which exhausts the X's (that is, with X dependent upon Y), there must be at least as many Y's as there are X's. An X cannot exist or be added without a Y , and a Y cannot be deleted without first deleting the dependent X . If the relationship is mutually exhaustive on X and Y (that is, X and Y are mutually dependent upon each other), there must be exactly the same number of X's and Y's. With mutual dependence, an X and a Y must be added or deleted simultaneously.

With a one-to-many relationship between X and Y which exhausts Y (that is, Y is dependent upon X), an X cannot be deleted without first (or simultaneously) deleting all of its dependent Y's. If a Y is added, it must

simultaneously be related to an X or else a new X must first (or simultaneously) be added. This relationship is very common and is given the special name of hierarchic relationship. The hierarchical data structure is built up from a set of hierarchic relationships.

If a one-to-many relationship exhausts both X and Y, the number of Y's must remain greater than or equal to the number of X's. With such mutual dependence, the addition of an X requires the simultaneous addition of (at least) one Y, and the deletion of an X requires the simultaneous deletion of all dependent Y's. Addition (deletion) of a Y requires the addition (deletion) of the X if the Y is the first (last) of the dependents of X.

There are several possible ways to graphically represent the exhaustibility of dependency characteristic of a relationship. The one recommended here is to add the letter "D" on the arc of the relationship close to the dependent entity. If the relationship is mutually dependent, add the letter "D" at both ends of the arc. Some selected relationships are graphically shown in Figure 4.

Figure 4.

Graphical Representation of Exhaustibility or Dependency in a Relationship

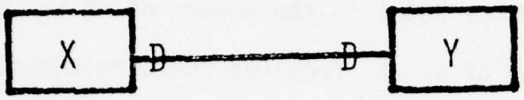
---

Since exhaustibility or dependency is the more restrictive condition on a relationship, it is selected as the one for explicit notation in the graphical schema diagram. The less restrictive nonexhaustibility characteristic is the default if no notation is added to the representation of a relationship.

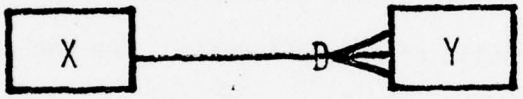
In capturing information about a relationship that is nonexhaustive, it is helpful for the user to indicate what proportion of the entities in the nonexhausted class do not participate in the relationship.

one-to-one (1:1) relationship between X and Y, with X exhausted or dependent:

one-to-one (1:1) relationship between X and Y, mutually exhausted or dependent:



one-to-many (1:N) relationship between X and Y, with Y exhausted or dependent:
(called a HIERARCHIC Relationship)



one-to-many (1:N) relationship between X and Y, with X exhausted or dependent:

one-to-many (1:N) relationship between X and Y, with mutual dependence:

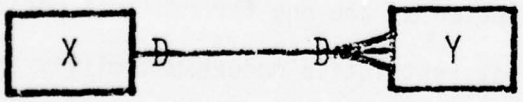


Figure 4. Graphical Representation of Exhaustibility or Dependency in a Relationship.

Degree of a Relationship

The degree of a relationship consists of the number of items of information needed to represent the relationship. The minimum is two pieces of information corresponding to the two entities participating in the relationship. It is helpful to consider the two minimum pieces of information to be the identifiers of the entity records for each arc in an instance picture.

From the sample personnel-organizational database shown in Figure 5, one relationship would be represented by the two data items: EMPNO and UNITNO. It is a one-to-many relationship which exhausts the set of employees, that is, every employee must be assigned to exactly one organizational unit.

Another relationship exists between organizational units within the hierarchy of an organization. Such a relationship is reflexive since it relates entities from the same entity class. The two items which represent the relationship are UNITNO and the organizational unit number of the parent, referred to as REPORTS TO in the sample database. The reflexive one-to-many relationship with dependency on the "many" side, effectively constructs a tree of organizational units. The "D" on the arc indicates that each organizational unit must have a parent. For a reflexive relationship, this dependency implies that the instances are on one big cycle (or ring) or else the rule must be relaxed slightly to say that each organizational unit except one must have a parent unit; that one being the top level organizational unit or the root of the tree.

A third relationship exists between employee and skill which can be represented by the pair: EMPNO and SKILLCODE. In fact, there are three such relationships between employee and skill which is why they must be named. All employees must have exactly one skill which is their primary skill, and they must have one skill which relates to the job they fill. The third relationship, secondary skills, is a many-to-many relationship and is nonexhaustive on either the employee or the skill entities.

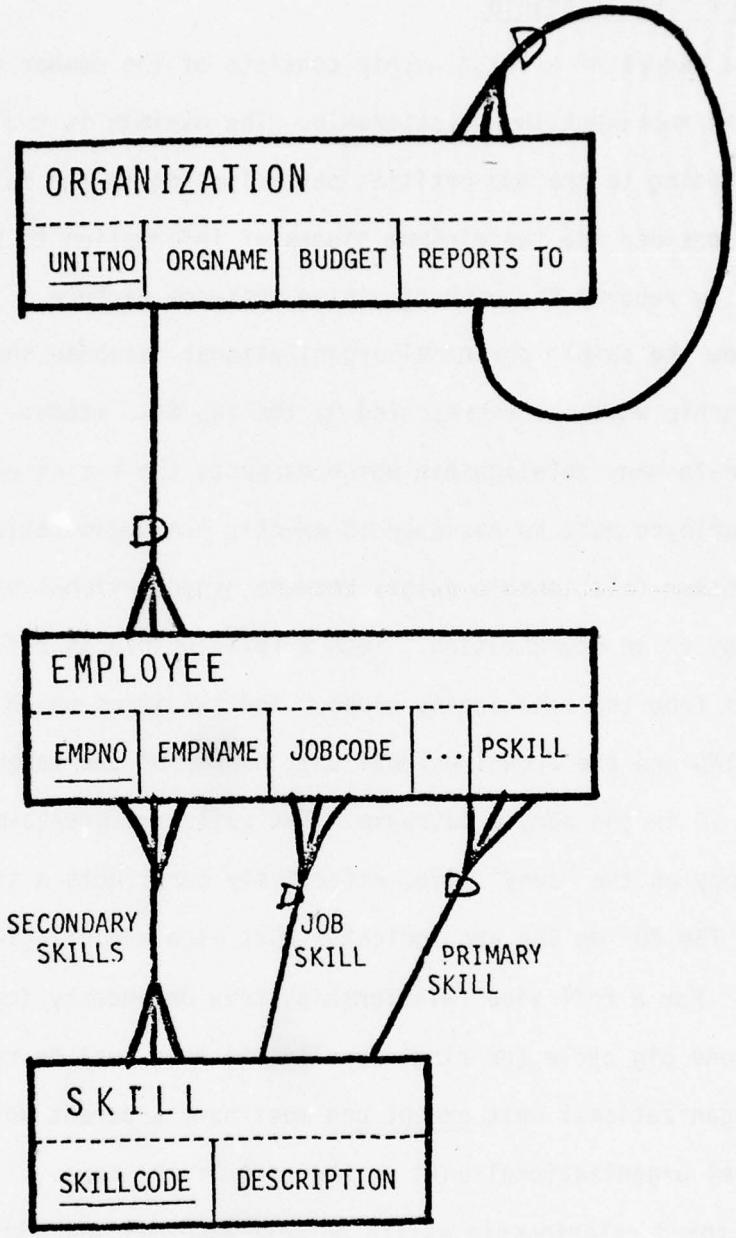


Figure 5. Relationships in Sample Personnel-Organization Database.

Notice that the first two, job skill and primary skill, are total functions *onto* skill. This means that, given an employee record, a unique skill is determined for each relationship. As a consequence, it is rather simple to represent each relationship by storing the SKILLCODE (the identifier of the skill entity) in an appropriately named data item within the employee record as shown. (The organization UNITNO could similarly have been stored within the employee record to represent the organization-employee relationship.) The third relationship, secondary skills, is more difficult to represent as will be shown later.

All of the relationships discussed so far have been binary relationships. Now consider extending the definition of the database to include additional attributes associated with a relationship, for example, a proficiency rating for each skill possessed by each employee. Such a relationship is ternary and would consist of three pieces of information: employee identification, skill, and the proficiency rating. The degree of a relationship refers to the number of individual pieces of information required to represent it.

Criteria of a Relationship

Every relationship must have some criteria or rule which provides the basis for an association between two entity instances. The criteria is generally stated in the form of a Boolean expression on the values of attribute from each of the entities participating in the relationship. The following paragraphs describe the various types of criteria.

Two entities which possess the same common identifier are automatically related. For example, an organization may consider one entity type to be active employees currently on the payroll and another entity to be all personnel ever associated with the organization including applicants never hired and those on leave or retired. In both files the entities may be identified by some personal identifier (such as the U.S. Social Security Number or the Canadian Social Insurance Number). A simple matching of the value of the identifier for each entity establishes an association between two entities. This criteria is only possible for one-to-one relationships.

With mutual identifiers, entities in two files are related because there is a one-to-one relationship between the identifiers. For example, one file may have organizational unit number as an identifier and another may have organizational unit name as an identifier. As a second example, one file of motor vehicles might be identified by a Vehicle Identification Number (VIN) while another file is identified by a license number. Since names and numbers, or VIN's and license numbers, are assigned uniquely to the entities, there will be a one-to-one correspondence. In most cases, such files essentially describe the same real world entity class. Defining a relationship based upon mutual identifiers requires some representation which relates each value in one set of identifiers with each value in the other set of identifiers. This criteria is only possible with one-to-one relationships.

Two entities are related if they have a common partial identifier, in other words, part of the identifier (which is a ranked set of data items) of one entity corresponds to the identifier (all or part) of another entity. For example, if the identifier of a position is organizational unit number and job code which is from the skill domain, then the position entity is related to both the organizational unit entity and the skill entity. The relationship is based upon a match of the appropriate data item values in the related entities. This criteria would be possible on one-to-many relationships.

Two entities may be related through a foreign identifier, that is, one entity possesses a nonidentifier attribute which is actually the identifier for another entity. For example, the organizational unit within which an employee works is an attribute of the employee and is also the identifier of the organizational unit entity. A one-to-many relationship could be based upon a foreign identifier.

Two entities can be related through a common nonidentifier domain. If two entities have an attribute which is based upon the same domain they are related, although such a relationship may not always be meaningful. For example, position has an AUTHORIZED SALARY attribute and employee has an ACTUAL SALARY attribute, therefore, these two entities are related through the salary values. For a second example, entities in a customer file could be related to entities in a salesperson file through the common attribute of LOCATION indicating the city where they are located. This relationship through a common nonidentifier domain could be quite meaningful and useful. Such a relationship is automatically established as soon as the system knows that the two data items in the two files are from the same domain, even if they are named differently. Two entity instances would be related if the values are equal in the common domain. In general, such a relationship would be many-to-many.

Except for mutual identifiers, all of the above criteria establish the relationship between two entities based upon a data item value in one entity instance

being equal to a data item value in the other entity instance where the two data items are from a common domain. In general, the criteria for a relationship can be any Boolean expression in which the operands are data items from both entity classes. The mutual identifiers is a simple example of such an expression which is a one-to-one transformation from the identifiers in one set to identifiers in the other set.

Sometimes a relationship may exist between three or more entity classes. In this case, one entity class must be designated as the parent (various other terms used include master, header, or owner) and the others are dependent entity classes (detail, trailer, or member). Such a relationship requires that there exist a common domain across all the dependent entity classes. If there is no common domain, there can be no single criteria upon which the relationship is based. As an alternative, a separate relationship can be defined between the parent entity and each dependent entity class.

Representing Relationships

As was seen in the previous section, a relationship is always based upon an equality (or other more general functional relationship) between values of data items in each of the entity classes participating in the relationship. Although such data items are real attributes of the entities represented, they need not always be explicitly defined within the formal database structure.

A relationship may be represented in the data structure using one of three forms: association through the values of data items in each of the entity classes, physical contiguity, or chaining. More than one form can be used simultaneously resulting in greater redundancy and greater reliability. It is primarily the choice of representational form for relationships that gives rise to the various data models in the current literature.

A one-to-one relationship can be represented by a data item value in one entity being equal to (or otherwise functionally related to) a data item value in the other entity. There can be no relationship between two entities unless there is some common domain on which to base the relationship (or a defined correspondence between two domains). As an alternative, one of the two entity types could be designated as "parent" with the dependent entity instance physically stored "next" to the parent entity instance, or a physical pointer added to the parent entity instance to point to the dependent entity instance. In these two representational forms it is no longer necessary to explicitly store the data item of the common domain in the "dependent" entity. The only problem is that the proper recording of the relationship is dependent upon physical contiguity or the correctness of the physical pointer. If the attribute value in the dependent entity instance changes, it must be reflected in a physical move of the entity record or the requestor must identify both the old and the new entity instances so that the system can appropriately update both physical pointers. Also, the physical contiguity and pointer representational

forms imply a directionality and access path which must be known and adhered to on the part of the user. It is more difficult (and sometimes impossible) to access dependent entity instances independent of parent entity instances. It is generally more desirable to store the data item explicitly, and then let the use of pointers or physical contiguity be dictated solely by the need for more rapid access or better utilization of storage. At the very least, the user should be insulated from these physical considerations. In defining the database structure, the user should only be concerned with the relationship and not its representational form -- that is a matter for the file designer or the file design routines.

Figure 6 shows three possible representations of a one-to-one relationship based upon a foreign identifier or a mutual identifier.

Figure 6.

Representation of a One-to-one Relationship

In these two cases it is necessary to add a data item to one record which contains the identifier of the related record from the other entity class. Either the identifier of X is stored in the records of type Y or vice versa. Doing one or the other has some access implications. If the identifier of Y is stored in the records of type X , then finding the record of X that corresponds to a given record of Y is not obvious. Such retrieval however, is feasible since it is always possible to scan the records of X until one is found with the matching identifier of Y . Alternatively, the records of X may be indexed on the data item containing the identifier of Y . (There is actually a fourth representational form in which the matching identifiers of X and Y are stored together in matching pairs in a separate table).

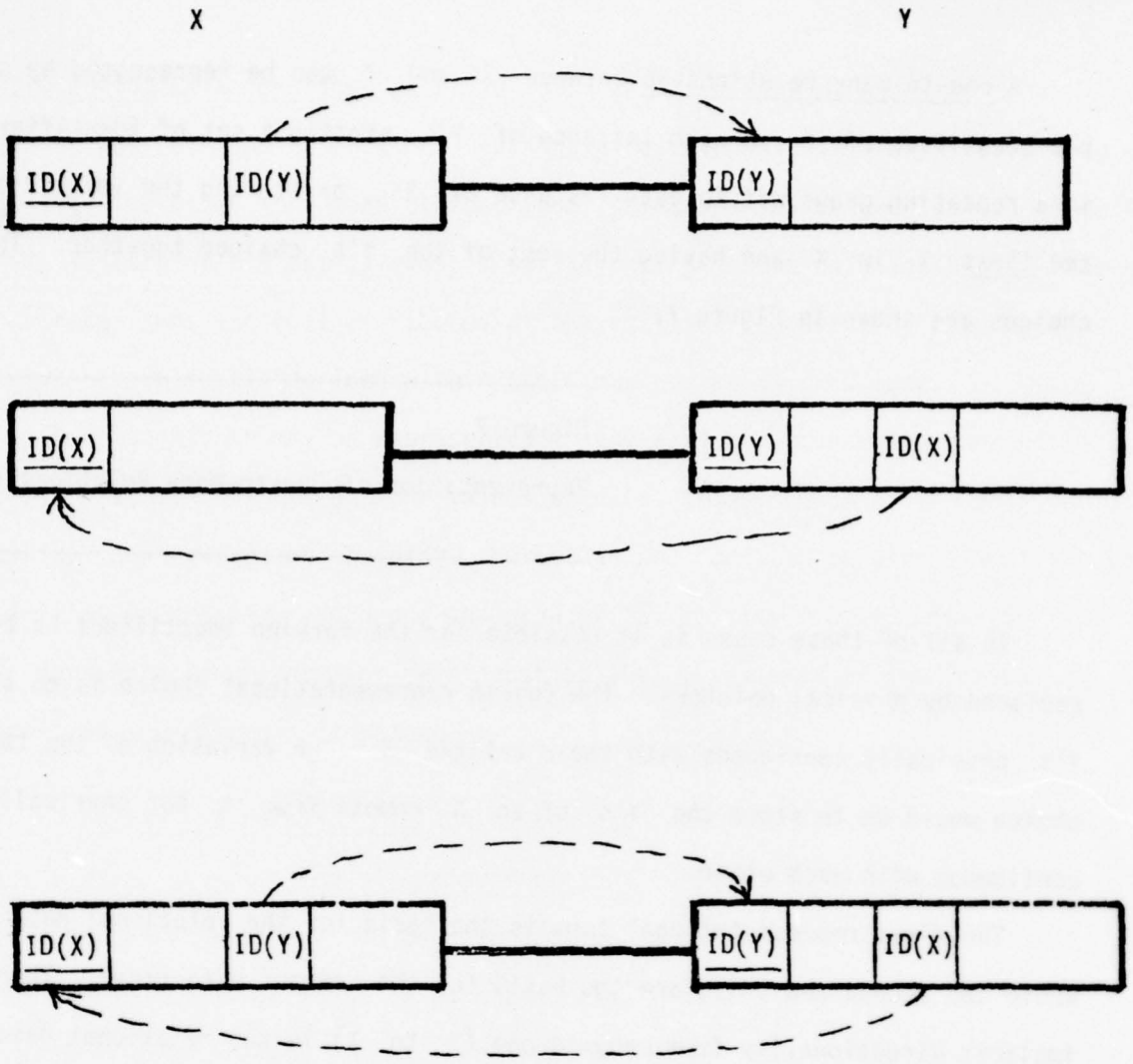


Figure 6. Representation of a One-to-one Relationship.

For relationships based upon a foreign identifier or a mutual identifier, record X can contain the identifier for record Y, or record Y can contain the identifier for record X. For greater reliability and redundancy, each can contain the identifier for the other. For relationships based upon common data items, whether identifiers, nonidentifiers, or partial identifiers, no additional data items are required -- the relationship is fully represented in the existing data items.

A one-to-many relationship between X and Y can be represented by storing the identifier of X in each instance of Y , storing a set of identifiers of Y in a repeating group within each instance of X , or storing the identifier for the first Y in X and having the rest of the Y's chained together. These choices are shown in Figure 7.

Figure 7.

Representation of One-to-Many Relationships

In all of these cases is it possible for the foreign identifiers to be replaced by physical pointers. The fourth representational choice is to store the Y's physically contiguous with their related X . A variation of the third choice would be to store the Y's of an X remote from X but physically contiguous with each other.

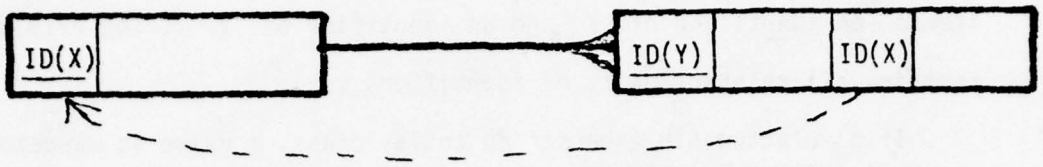
The first representational form is the basis for the relational data model while the second and third are the basis for the network data model. There is an implicit directionality from many-to-one (Y to X) in the relational data model and a directionality from one-to-many (X to Y) in the network model. Neither one of these directions is inherent in the relationship to be defined by the user and captured in the stored database. The particular representational form chosen depends upon the behavioral characteristics of the database -- how it is used and the more frequent avenues of access. It is obviously possible to store sufficient information to permit efficient access in either direction. The X records can be indexed on the identifier of the Y's and the Y records can be indexed on the identifier of X . Each form shown in Figure 7 is a minimal set of information for representing the relationship.

In representing many-to-many relationships the repeating group cannot be avoided without setting up a special linking record type. Either each record of

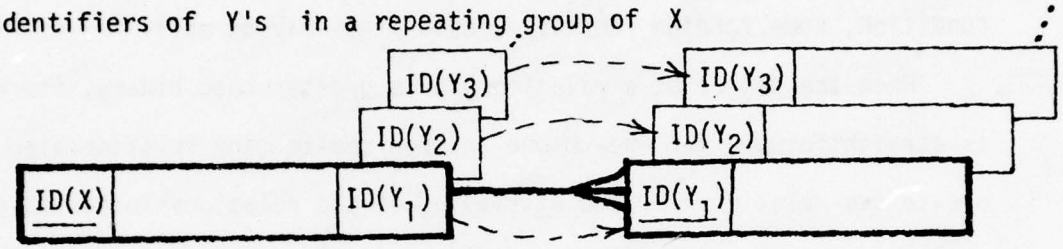
X type records

Y type records

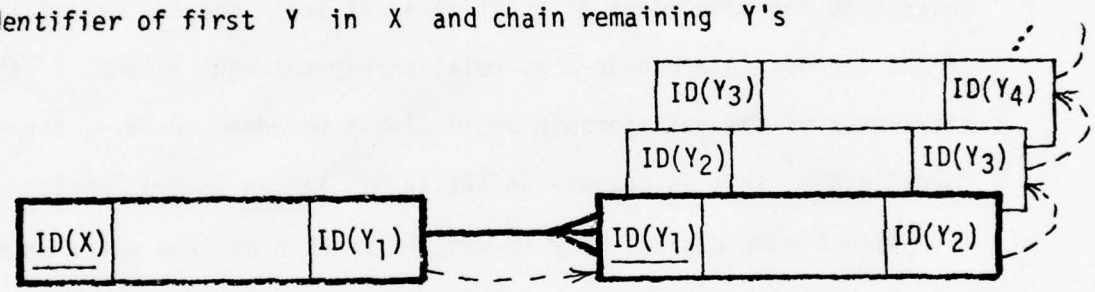
(1) identifier of X in Y .



(2) identifiers of Y's in a repeating group of X



(3) identifier of first Y in X and chain remaining Y's



(4) Y's physically contiguous with X

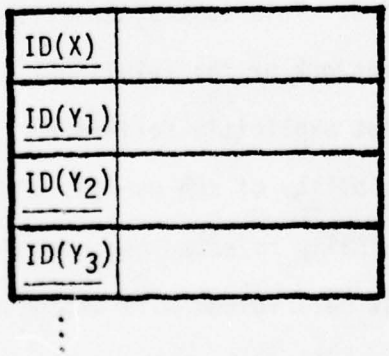


Figure 7. Representation of One-to-Many Relationships.

X contains a repeating group of identifiers of Y , or each record of Y contains a repeating group of identifiers of X . The options of physical contiguity and chaining are not viable for representing many-to-many relationships. The third possibility is to set up a special linking record to contain two data items: an identifier of X and an identifier of Y . The file of such records contains all relevant pairs of identifiers.

If a relationship exhausts an entity class, a value is mandatory for the foreign identifier in that exhausted entity record. Without the exhaustibility condition, some foreign identifier data items may be null.

When the degree of a relationship is greater than binary, its representation is straightforward for one-to-one and for one-to-many relationships. In a one-to-one relationship, the attributes of the relationship can be added to the records of either entity class. This is because the records are both essentially describing the same class of entities or at least one set of entities is a subset of the other. In a one-to-many relationship between X and Y the extra attributes of the relationship would always be added to Y , the entity on the "many" side. This is because an entity of Y can have at most one parent.

With higher degree, many-to-many relationships, the extra attributes of the relationship (and/or physical pointers) can be added to the repeating group of foreign identifiers in either the X or the Y record. Alternatively, the extra attributes can be added to the special linking record. This last alternative is the only reasonable one permitted in either the network or the relational data models (unless the full nature of the relationship is not explicitly defined to the system, and therefore its maintenance is the responsibility of the users). Physical contiguity is another representation possibility, usually rejected because of its inherent redundancy. Ideally, the user should not be involved with definitions at this level of representation. It should be sufficient to declare two entity classes, a relationship between them, the characteristics of the relationship, any attributes of the relationship, and the behavioral characteristics of the relationship.

SEMANTIC CONSTRAINTS

Semantic Constraints are a generalization of the concept of validation criteria which is an extension of the definitional information. A semantic constraint defines the acceptable value domain for an attribute or a consistency relationship between several attribute values (e.g. the total payable on a bill equals the total for all of the items purchased, minus any discount, plus tax). There are tradeoffs to be made in deciding whether to store an attribute or to derive it when necessary. This is the reason for generalizing validation criteria to semantic constraints. If the attribute is actually stored, then the constraint becomes a validation criterion. However, if the attribute is derived, then the expressed semantic constraint is the derivation rule. In the former case a true-false value is returned depending on whether the value is valid, but in the latter case the derived value is returned. Whichever is implemented, the user needs to specify the semantic information as part of the data definition process. With semantic constraints as validation criteria, there are two basic approaches to maintaining validity within the database -- constructive and functional.

With constructive validity, a set of legal operators are defined such that, given a set of valid operands, these operators will always yield valid results. Many of the possible operators are described in the previous section of Value Class. Two benefits of this approach are that repeated tests are not required each time any attribute is changed and the database can move through temporarily invalid states (e.g. in the above example with the bill if the total were actual instead of derived two operations are necessary to add an item to the bill and maintain consistency -- include the item, and increment the total). For this approach to be used the user must be able to define his own value classes and the operators on them, and the system must check to insure that the operators are indeed legal and will always yield valid values. With a functional approach

the system must be provided three pieces of information: (1) the validation criteria, (2) the action to be taken when the data is not valid, and (3) when the validation criteria are to be invoked and tested. The general format would be:

```
WHEN (condition)
     IF (premise) THEN continue normal processing
                  ELSE exception/error processing.
```

Both the condition and the premise are Boolean expressions, with the premise being the actual validation criteria. The premise can be as complex as necessary. For example, it may define the domain of the attribute and consistency relationships with several other attributes or groups.

These semantic constraints may apply to either the database or to a query response a user obtains from the database. Edward Yourdon identifies three database problem conditions (in Design of On-line Computer Systems, Englewood Cliffs, NJ: Prentice Hall, 1972).

1. obsolete -- the data is out of date, but will be corrected by a later transaction. There is no way for the user or the database to recognize this condition, although an effective date would provide some assistance.
2. inconsistent, but automatically self-correcting -- certain consistency conditions are not met because the database was observed in the middle of an update data transaction. By specifying when to test the validity, this problem is ignored and allowed to correct itself within the database. However, a user's query may occur at this point and result in an inconsistent response that will not be self-correcting. This problem could be avoided by specifying that the query is to have exclusive use of the object data with respect to concurrent updates.
3. consistent, but incorrect -- this condition indicates improperly defined or inexpressible validation criteria.

Semantic Constraints may be applied to the database or to transactions against the database.

Database -- When applied directly to the database, semantic constraints provide validation criteria or derivation rules.

Transactions -- When applied against transactions, semantic constraints can occur at four levels.

1. Item -- Within the transaction the constraint can apply to a single item. In this case it specifies a condition on the value domain of the attribute or some other characteristics of the attribute. (See Domains and Value Class.)

2. Interitem -- Within a transaction the constraint may apply to the relationship between attributes. For example, the total price for an item may be defined as the unit price times the quantity. This constraint could then be checked and verified before the transaction would be accepted and processed.

3. Batch -- When transactions are entered and processed as a batch, certain control information (e.g. control totals) may be defined and used as semantic constraints. If they are not met, then the entire batch is rejected. (Note: it would be possible for individual transactions to be invalid and rejected, but for the control totals to be automatically updated so the entire batch would not have to be rejected).

4. Transaction-to-Database -- It is also possible to define semantic constraints relating a transaction to the current state of the database. For example, when a customer's purchase is being added, both the customer and the item being purchased must already be in the database. Similarly, when a new account is being added, a check must be made to insure that the new customer number is not already in use.

II. DEFINITION OF SYSTEM DATA

System data in an application/database system consists of all the information which defines the various static and dynamic components of the object system. In addition to the database definition or schema described in Part I, these components include:

1. Reports.
2. Transactions.
3. Mappings.
4. Authorized Users.

As more of the object system is formally defined independent of the programs, this list of components expands. Only the first two are discussed in this report.

The definition of system data raises the problem of the duality of processes and system data as discussed in the Introduction. The definition of a report, for example, implies the later generation of that report, the generation being the action of a process. Figure 8 attempts to show these relationships. The report generator is a generalized program (part of the DMS facilities) which is "driven" by a stored report definition -- part of the system data. Such a view implies that the report generator executes the definition interpretively.

At the other extreme, the report definition compiler could actually generate directly executable object code, in which case the embodiment of the stored report definition is, in fact, a program. Now there is no generalized report generator, only a collection of tailored report generators. The language used to write the report definition is actually a very high level programming language. The view taken here is that if the report definition language is descriptive and nonprocedural (as it should be), the user will not think of it as specifying a process but rather as specifying system data. Hence, the definition of system data is treated separately from processes (which are user application processes).

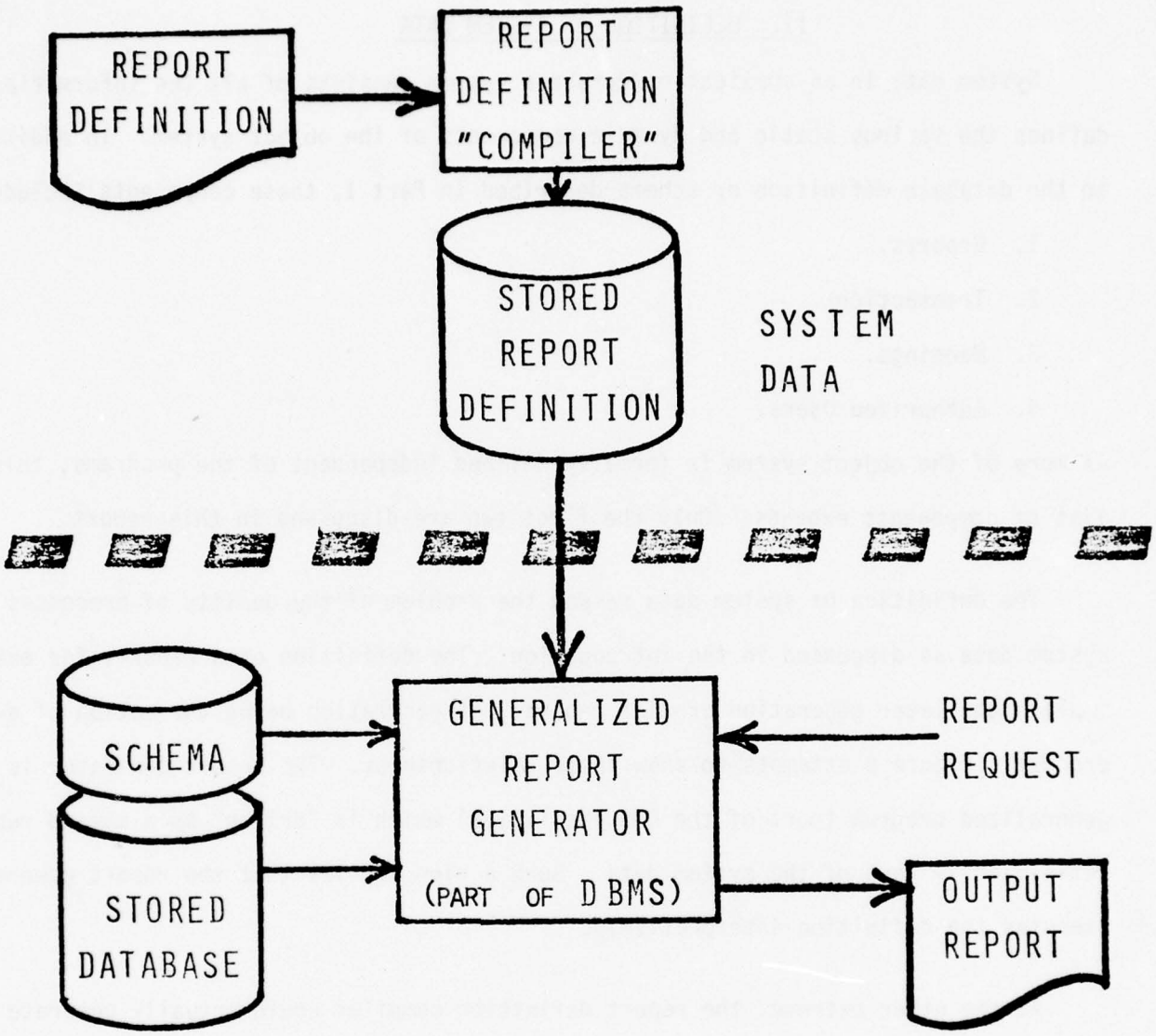


Figure 8. The Duality of System Data and Processes.

REPORT DEFINITION

A report is simply a very complex retrieval specification. There is a continuum of query requests -- the very simple which the user provides in its entirety, the more complex which is usually pre-defined and stored for him, and finally the very complex or report request which is usually pre-defined and specifies both the data to be retrieved, the form in which it is to be presented, its distribution, and a variety of other factors. The information to be output may be defined in terms of any of the data components -- data items, groups, relationships, entities, information elements, or files or subsets of files. The output or reports may be in either of two modes -- list or graphic. The output may be intended for man (either a programmer or a nonprogrammer) or for the database or another machine.

Reports are a generalization of derived information. The report definition is the derivation rule and the actual way the report is derived and/or stored should be determined automatically from the definition. We will now consider in detail the defining information required to completely specify a report.

Mode -- The information in a report can be presented in several different formats. It may be in the list or tabular mode or it may be presented graphically as charts or curves. This distinction is not frequently made currently because of the weakness of current graphic systems. However, there is much psychological literature to suggest that people get more information and get it quicker and more accurately from graphic than tabular form. Therefore, for reports destined for human users the graphic mode is often preferrable. However, when the report is primarily to modify the database or provide input to another machine, the tabular or list format would be better. Much of the report defining information is common regardless of mode, however, the specific differences will be pointed out when they occur.

The basic report definition must include the following items:

1. Titles -- ideally there should be default options which the user can override, for example, file name and Boolean selection expression could be a default title.
2. Items included -- should include the names of the items, their position on the report, and their value editing formats. Once again given the item name there should be a complete set of default options. (This is the main area where the different modes require different definitional information.)
3. Conditions under which the report is to be generated.
4. Access controls -- Who can cause the report to be generated (i.e. who can invoke it, if it can be generated on demand?) What is the report distribution, i.e. who gets copies?
5. Who defined the report?
6. When was the report defined, modified, and what is its current version number?
7. Where to send the report? Is it for internal system use or for external use by either another system or by humans?

The new field of word processing relates to the formation and presentation phases of report definition.

1. Titles -- When a report is generated there should be a complete set of defining information provided for the recipient. There should be a covering page including report title, date and time the report was generated, the date and time covered by the report (this may be either for a point in time or for a certain interval), why the report was generated (at the end of a standard reporting period, because certain exception conditions were found, or on demand and if on demand the requester should be identified). The report should also provide a distribution list and any information about restricted distribution. The report definition must provide the user with a way to specify all of this information which can be included as a default option or modified on an ad hoc basis for a specific report instance. All of the above information should be present regardless of the mode of the report.

2. Items/Attributes to be included -- The items/attributes to be included in the report must be specified. There must also be a selection expression specifying the entities on which to report. This information must be provided regardless of the mode of the report. However, there is additional information which is dependent on the mode.

LIST -- Given the item names and selection rules for the entities, there must be editing and placement information.

Editing -- Much of this information can be provided as defaults because it is included in the data definition of the item/attribute. The main editing information includes -- field size, data type, right or left justification, any encoding/decoding tables, any leading or trailing character fill, whether the value is signed or followed by CR, whether there is a floating dollar (or other currency) sign, what to do if the field overflows, or conditions under which a value should not be printed.

Labels -- Labels must be provided to identify the information in the report. The labels must identify both the item name (schema information) and the entity for which the data applies (instance or value information). Encoding/decoding tables may also be necessary and would have to be provided either in the data definition or the report definition. In most reports item names are the column labels and value labels are used on the rows, however, either type of label should be applicable for either row or column.

Placement -- This specifies where the item is to be printed on the page. Placement must be indicated both horizontally and vertically. It may be given in either absolute or relative terms absolutely (column 10, line 4 or relatively skip two lines before printing the next line). Relative placement has an advantage since it facilitates page mapping.

Page mapping -- Currently most reports are defined with absolute placement. This means that when the report is to be produced on a different device with a different page size, it has to be redefined. With an automatic page mapping capability reports with greater device independent can be defined. Certain default mappings (with possible user overrides) can be defined for converting from a given logical report page size to a larger or smaller physical page size required by the output device.

Paging -- Conditions must be specified for when to start a new page. These conditions may be value dependent. However, often they are simply dependent on the page position, not any data values. Often in this case certain additional special processing is necessary before the normal output can continue, e.g. reprinting the header labels at the top of the page.

Control Breaks -- These define special conditions on which certain exception processing, such as totaling occurs. Control break processing is a generalization of the exception processing described under paging. However, control breaks are always determined by data values, as opposed to page position.

TRANSACTION DEFINITION

Transactions are a specific type of system entity which is used to describe certain "real world" entities of interest. Each transaction type must be explicitly defined in terms of its input format, validation criteria, and processing. A transaction type definition includes many of the same characteristics as an entity type:

1. Name of the transaction type.
2. Description - A narrative explaining the transaction type.
3. Criteria for Inclusion - What are the characteristics which a transaction must exhibit to be classified as this type.

Transaction instance characteristics are similar to those for other entity instances, e.g. date and time of creation, user/process who created it, size. The previous discussion for attributes and data items for entities also applies to transactions. The input definition includes a definition of the attributes in the transaction, their value class, and validation criteria. In those cases where the transactions are batched additional definitional information such as batch identification, validation, and control totals must be defined.

III. DEFINITION OF PROCESSES

In the overall process of system design, both files and application processes must be defined. This research takes a data perspective with a focus on the database design. Therefore, for purposes of this research we only need to capture enough information about processes to enable file design to be accomplished. Processes here are defined in a generic way and only from the unique perspective of the interaction with the data. The user defines application processes in terms of the generic, data processes.

Classes of Processes

There seem to be some distinct classes of generic processes -- the classes being differentiable by level of interaction with the database, or time of functioning. These classes of data processes are shown in Figure 8. Data processes are divided into:

1. operators which act directly on data,
2. generalized functions which act directly on data (through the execution of a complex collection of actions built up from the data operators), and
3. sets of functions (generalized or tailored) which operate on system data.

Within each class of processes, we want to define those which are generic or basic -- those which represent the functional primitives within each class. At the same time we want to define processes that represent a complete unit of processing to the using environment. The user can then take one process (or some small number in combination) to define an application process.

In dividing data into actual data and system data, we find that some processes act directly on the system data. In fact, these could be the same processes, as, for example, when the data definition is stored in the database just like any other data. However, since the function is different we would consider them to be different processes (it is the mechanism which is the same). Moreover, the system data actually implies a process as, for example, a report definition implies the generation of a report when requested.

Defining User Application Processes

The user selects from the generic data-oriented processes and describes (or builds up a description of) user application processes that model events, activities, and operations in the real world. Just as logical database design and data definition attempt to model the statics in the real world, the user

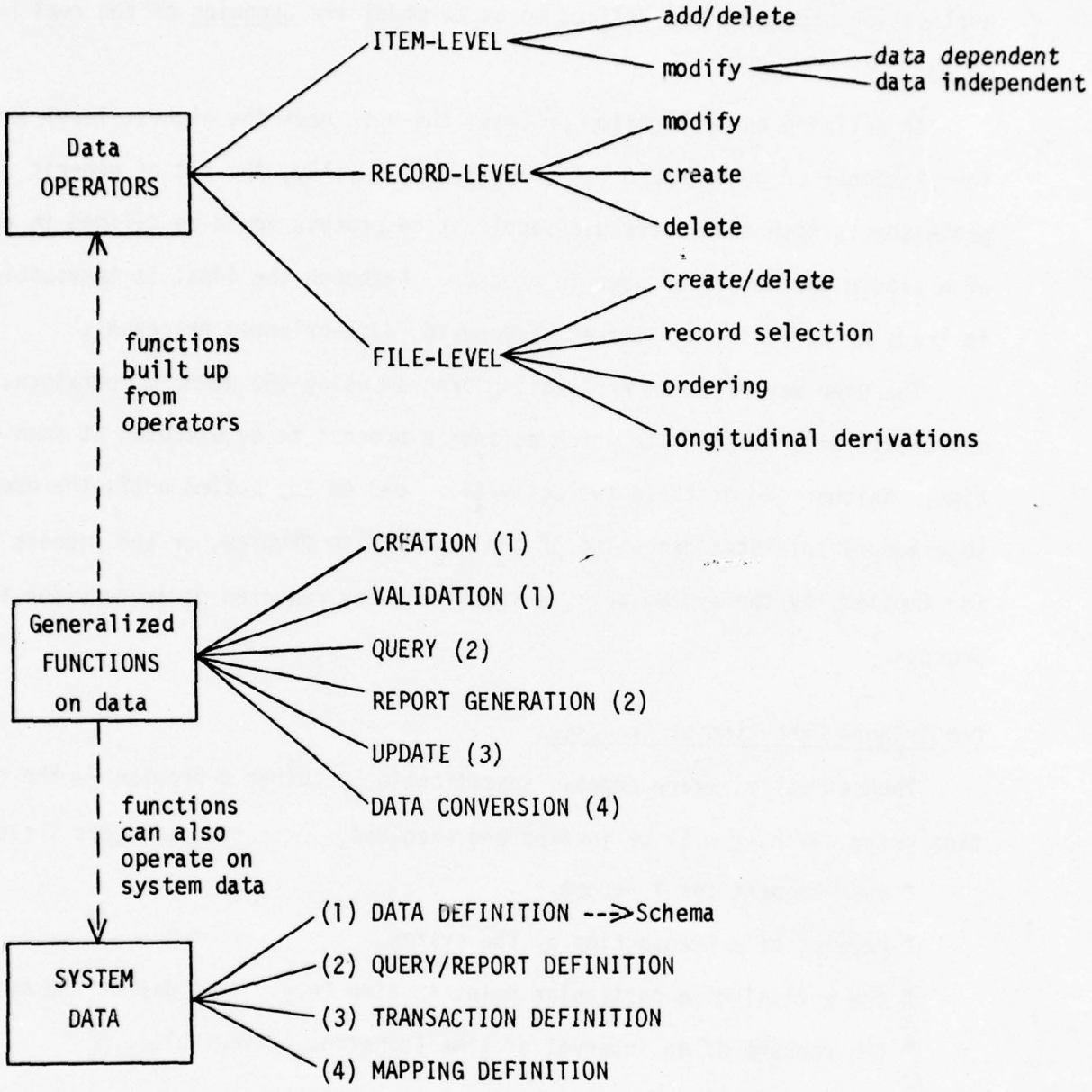


Figure 9. Classes of Generic Data Processes

application processes are defined so as to model the dynamics of the real world.

In defining an application process, the user uses the highest level and fewest number of generic process primitives. Ideally, the set of generic processes is such that every user application process could be defined in terms of a single parameterized generic process. Although the ideal is impossible, it leads us to define a rich set of generic, data-oriented processes.

The user may write an application program using the generic operators, or may create some system data which defines a process to be executed at some future time. Neither one of these two activities lead to any action until the user (or some event) initiates execution of the application program, or the process defined (or implied) by the system data, and delivers any required parameters for the process.

Conditional Execution of Processes

Theoretically, every process specification requires a premise -- the condition under which it will be invoked and executed. Execution triggers include:

- user request for a report.
- receipt of a transaction by the system.
- the arrival of a particular point in time (e.g., last day of the month).
- the passage of an interval of time (constant or variable).
- internal request from another executing process.
- the status of the database, that is, the truth evaluation of a Boolean expression.

Every process specification, from individual operators to system-wide functions, has a condition on its execution. Since a process is built up from more primitive processes and operators, there can be internal conditions expressed on parts of a process specification.

ITEM-LEVEL OPERATORS

These have been discussed in some detail in conjunction with data item types under database definition. Although a full definition of item-level operators is desirable here, it seems that defining generic data processes at higher levels (record, file, function) will have a greater impact on the macro aspects of file design.

Item-level operators create, retrieve, modify, and destroy individual data item values using specific verbs and derivation rules. They always take a record instance as given. They may or may not take an item value instance as given depending upon the interpretation of the processing action.

Item-level operations may be specified in a variety of ways. For retrieval, the simplest form is naming the item. At execution, the record instance is presumed to be specified by context and the item name is considered to be a variable which takes on the value of the data item from the object record instance.

For update, the simplest form is writing the item name to the left of a replacement statement. The specification of a value or expression on the right determines the value used to replace the data item in the object record instance. A value can be used directly. With an expression, all operands must be resolved (given a value) in context and the expression evaluated.

Examples of implicitly specified (that is, without using an explicit verb) item-level updates include:

item-name ⟵ value

item-name ⟵ expression

item-name ⟵ item-name + c

item-name ⟵ NULL

The first two are direct (value independent) item-value replacement operations, the third is a value dependent modification of the data item value, and the last operation effectively deletes any existing value for the data item. Increase and decrease are two simple, value-dependent operations.

The distinction between value dependent and value independent item-level update actions is extremely important to backup and recovery operations. In a value dependent update operation, the resulting data item value depends upon its previous value. Proper recovery requires recording one additional piece of information compared to value independent updating -- both before and after images, or one of them plus the change to the before image (say, + 10).

Update actions using explicit verb specifications provide added integrity. For example, one very would assign a value to an item only if it was previously null, while another verb would assign a value only if it was previously valued, and a third verb could operate either way. The implicit update specified using a replacement statement (as discussed earlier) is equivalent to the third case -- and is unable to distinguish the first two. To accomplish the first two would require a condition to be associated with the replacement statement. It could be of the form:

IF condition THEN item-name ← value.

where, for example

condition := item-name IS NULL

The proper execution of an item-level operator may depend upon the role of the object data item in the entity record. For example, changing the value of an item upon which the records are physically ordered implies that the object record be moved in storage to maintain the defined ordering. Similarly, changing the value of an identifier may change the entity (e.g. part number) and, therefore, such action should be disallowed, or at least restricted to a well-defined set of semantic constraints.

Other instances of changing the identifier may not indicate a change in the entity, for example, changing an employees maiden name to a married name, or reassigning part numbers. Note that changing an identifier may affect the representation or relationships with other entities.

Item-level operations are often implicit in a record-level operation.

Time Dependent Operators

Adding the time dimension to the definition of a database leads to the use of (at least potentially) specific operators which depend upon the time dimension for proper execution.

There are two basic types of time dependent data -- data pertaining to a specific point in time (such as level of inventory) and data relating to a time interval (such as sales, or flow of goods through a manufacturing process).

Examples of time-dependent operators include:

- EARLIER, LATER.
- BEGINNING OF, END OF, MIDDLE OF.

Time-dependent operators are most likely to appear in the Boolean selection expression.

(References: Randall, Sundgren, Taggart, Jones.)

RECORD LEVEL OPERATORS

A record level operator always acts upon or delivers a single record instance at a time:

create
retrieve
modify
destroy

} a record instance

The input to a record-level operation must identify a record instance and the data items to be operated upon (depending on the action). Retrieval involves identifying a record instance and then delivering all or named data item values from the record. In general, a record-level operation acts on a whole record instance. To modify a record is to identify an instance and execute item-level changes within that context.

Identification of a Record Instance

Before a record-level operation is executed, a single record instance must be identified. This can be done through a special selection expression or through positional verbs.

The special selection expression must be required to identify at most one record instance. A simple example would be

item-name = value

where the named data item is the record identifier.

For a single file with one record type, the positional verbs would be NEXT, PRIOR, FIRST, and LAST. For multiple, interrelated file structures, there may be additional positional verbs such as FIRST CHILD (of current parent record instance), NEXT WITHIN PARENT (or NEXT SIBLING), LAST CHILD, or PARENT (of current child record instance). These all refer to a particular relationship and assume a parent-child interpretation of the relationship.

Positional verbs generally require a prior designation of a "current" record instance and an assumed sequence of record instances. The positional verb is then executed relative to a current record instance and in a particular sequence from the current record instance.

Creating a Record Instance

Record creation requires the values for data items to be pulled together to form an instance of the object record which is placed in the file. Record creation must be done in a manner consistent with the definition of the file. All mandatory data items must have a value in the new record instance, otherwise the creation operation should not be executed. Any data items designated as identifiers must not have the same value existing in any other record instances of the same file, otherwise the creation operation should not be executed. If the file is ordered (presumably the ordering data items would be mandatory), then the record instance would be inserted in its proper place in the sequence of records.

Sometimes it is useful to separate the function of creating a record in a user buffer (followed by a series of item-level operations) from the function of placing the constructed record instance into the file.

Deleting a Record Instance

Deleting a record implies deleting all the data item values which exist within the object record instance. It may involve logical or physical removal from the database but that distinction is of no importance to the user. The choice of logical or physical removal is a function of space and time which can be derived from user performance criteria.

FILE-LEVEL OPERATORS

File-level operators act upon all (or some) records of a single file. In all cases it is necessary to first name the file which is the object of the file-level operation. This may be done with one global declaration or as part of each file-level command to the system. File-level operators include creation, destruction, selection, ordering, and file-level derivation functions.

File Creation and Destruction

The file creation operation begins by naming a previously defined file. File creation then consists of a series of record creation operations. File destruction is accomplished with a delete operator followed by the name of the file to be deleted.

Record Selection

The file-level selection operation essentially partitions the file into selected and nonselected record instances based upon a Boolean selection expression. The default with no expression is to select all record instances.

In general, a Boolean selection expression consists of operands, operators, and delimiters (parentheses). The operands are constant literals, named data items from the object record type, or other variables. Operators may be arithmetic (+ - * /), relational (EQ NE LT LE GT GE), or Boolean (AND OR). An expression consists of one or more conditions separated with Boolean operators and variously grouped with parentheses. A condition consists of a data item name, and either a binary relational operator with an object operand, or a unary operator. The object operand may be a constant literal, a named data item, or an expression of a type consistent with the type of the subject operand. A Boolean expression always evaluates to a true or false which indicates whether an object record instance is to be selected or not.

In a file-level command, the selection expression may be preceded by a keyword such as WHERE or WITH. If the selection expression is used on a record-level operation, a special designation is required (such as WHERE UNIQUE, SELECT UNIQUE, SELECT ONE, or UNIQUE WHERE). In fact, it may be the UNIQUE designation alone which identifies a record-level operation requiring the identification of at most one record instance.

Boolean selection expressions are the central means for identifying data by content. They are used in several different contexts within the database environment. When semantic constraints are used as validation criteria, they consist of Boolean expressions stored as part of the data definition. Boolean expressions are also used for access control. Attached to either the user (authorization) or the data (permission) can be a Boolean expression which must be true before the user can operate on the data. In these cases, the expression is prestored and automatically ANDed to the user's selection expression.

Selection may be based upon the values of data items, functions on data item values, nonvalue attributes of data items, or any combination of these.

Selection on an identifier data item implies that zero or one records selected is the only legal response. If one record is selected only an insert operation will raise an exception since it cannot be allowed. If zero records are selected a record-level retrieval, modification, or deletion raise an exception since they have no data to operate upon. If two or more records are selected the data is inconsistent with its definition.

Selection on a non-identifier data item means that the data item values need not be unique, and, therefore, zero or more records can qualify. In addition, the user may state exactly how many records must qualify, or the minimum or the maximum number of records that must be selected before the associated processing is to take place.

Selection on longitudinal operators specifies that an item is to be compared to a value derived across all (or selected) records of a file, based upon such functions as count, sum, minimum, maximum, average, or standard deviation.

Selection using a sampling function specifies the selection of records which meet a sampling function criteria. The user may specify the kind of distribution desired and on what item.

Non-value attribute selection is selection based upon attributes of a data item other than its value. These could be such things as value length, type, existence, substring match, masked match, or other defined or assumed attributes.

Ordering

File-level ordering is applied to all (or selected) records in a file based upon defined ordering criteria. An ordering is defined by specifying:

1. a ranked set of data items,
2. collating sequence on the value set for each data item (usually implicit in the character coding convention and the data item type declaration), and
3. an ascending or descending flag for each data item.

Normally, a file ordering is specified on selected records prior to printing or other form of output. Generally, users do not specify an ordering on the records as stored, unless it is a temporary prelude to subsequent processing or output.

File-Level Derivation Functions

File-level derivation functions operate on the values of a data item across a set of record instances. These include count, sum, minimum, maximum, average, standard deviation, and many higher-level statistics. They also include marginal and joint frequency distribution statistics from two data items. For example, AVERAGE SALARY would be calculated across all selected records and would produce

a single output value, whereas AVERAGE SALARY BY JOBCODE would produce an average salary value for each unique jobcode among the selected records. The latter is based upon a marginal frequency distribution. It requires either a prior sort on jobcode (which may or may not be explicitly stated by the user), or a complete scan of all instances of salary, tallying them with respect to each unique value of jobcode.

GENERALIZED FUNCTIONS

Generalized functions are defined to be generic, that is, to perform a single, well-defined action with respect to the database. They are built up from the operators discussed in the previous section. A user could specify the process for performing one of these functions using the data operators embedded in some programming language. On the other hand, the notion of a generalized or generic function implies a high-level system module and language designed specifically to perform that function. The user provides the necessary input parameters, data, or directives and the generalized function module can perform the processing (now, or later if the input is catalogued).

A generalized function has associated with it a generalized, self-contained language and a language processor. The language processor may be a two-stage processor. The first stage accepts input written in the language of the function, doing some syntactic and semantic validation, and storing the language statements in some internal or object form (which may be compiled code). The second stage takes the stored specifications and executes them in response to a specific request, within a specific context, and with a given set of input data or parameters. The second stage of the process is mandatory. The first stage always exists in some form, ranging from simply storing the source input statements without any validation, to performing extensive prior analysis on the source input statements and generating highly tailored machine language object code. It is also possible that the input source statement would be executed immediately upon receipt, not deferred. Conversely, a process specification may appear to be executed immediately when, in fact, it is compiled and run in quick succession.

There are several useful ways to classify the set of all possible generalized functions. One way divides them into <u>setup</u> functions, <u>run time</u> functions, and <u>resetup</u> functions. The first and the third are often called utilities

because they are not actually executed on a continuing basis (although that distinction quickly gets fuzzy when speaking of transaction-driven online systems with masterfiles being "open" for processing 24 hours a day). The resetup functions are executed in response to an exception or deviant condition; examples include creating and destroying indexes, physical database reorganization, logical database restructuring, and restart and recovery. Ongoing functions could further be divided into direct ongoing functions (such as interrogation and update) and underlying ongoing functions (such as validation, logging for backup, access control, and performance monitoring).

Generalized functions apply to the stored database directly or to the system data. Generalized functions which act directly on the stored database include:

- creation (set up)
- validation (ongoing, underlying)
- interrogation (ongoing)
- update (ongoing)
- backup logging (ongoing, underlying)
- recovery (resetup)
- dump, reload, reorganization, restructure (resetup)

All of these functions depend upon a prior stored database definition if they are to adequately manage the stored data. In addition, interrogation may use previously stored report definitions, update may use previously stored transaction definitions, and restructure/reorganization uses mapping instructions.

At a meta level, the functions of creation, interrogation (display), and update apply equally well to the stored definitions of data, reports, transactions, mappings, users, etc. Indeed, if this system data is stored in exactly the same fashion as the database, then these three functions can be applied directly to the stored definitions. Since the stored definitions would be much

less volatile than the stored data, there is less need for generalized validation, backup, logging, and recovery, and practically no need for generalized reorganization and restructure. In most systems, the processes for creating, maintaining, and displaying stored definitions (that is, system data) are special purpose and tailored with more restrictive access controls. Only the database administrator should be allowed to change stored definitions and then not too often. Users cannot tolerate much instability in system data.

The following subsections identify and briefly describe various generalized functions. There are many degrees of freedom possible in specifying any one of these functions. At this stage of the research it is impossible to go into much detail for any of these. Yet, the real payoff of this research stems from a comprehensive definition of each of these functions. A comprehensive definition would encompass all the major patterns of processing against a database within each function. This would enable users to specify their data processing requirements at a substantially higher level than is currently possible.

Database Definition

The generalized function of database definition takes as input the definition of a database written in a Data Definition Language, processes and validates those language statements, and produces a stored representation of the database definition. This function is performed by a DDL Compiler.

The definition of a database includes:

- the logical structure of items, groups, records, and relationships.
- the stored physical representation of the logical structure, including item value encoding, record formats, and links.
- access mechanisms on the data structure.
- validation criteria and semantic constraints.
- device-media control information.

Besides creating the stored database definition, it is necessary to be able to display, add to, modify, and delete parts of or all of the stored database definition. This begins to resemble all the operators discussed earlier and indeed it is, except that the function is now being performed on system data. Modifying the stored database definition is generally called redefinition. Redefinition is the first step in the overall function of database revision.

Database Creation

The generalized function of creation takes a given database definition and some input source data for which it knows the definition, and stores the input data according to the database definition. In effect, the creation function "populates" a database. This general process is shown in Figure 10.

The simplest form of creation is to store the data in its existing physical form. In this case, the definition of the database and the definition of the input source data must correspond exactly. Flexibility would be added to the creation process by defining a different encoding for data item values, different sizes for data items, different sequencing of data items within records, different inter-record relationships, and validation criteria on the input data or the resultant stored database. In fact, creation is one of a family of data conversion processes as shown in Figure 11. Therefore, a more flexible database creation function incorporates a generalized data conversion function (based upon a defined mapping) and may incorporate a generalized validation function. This example illustrates the extreme importance of defining generic generalized functions, and not redefining essentially the same function merely because it is used in multiple contexts.

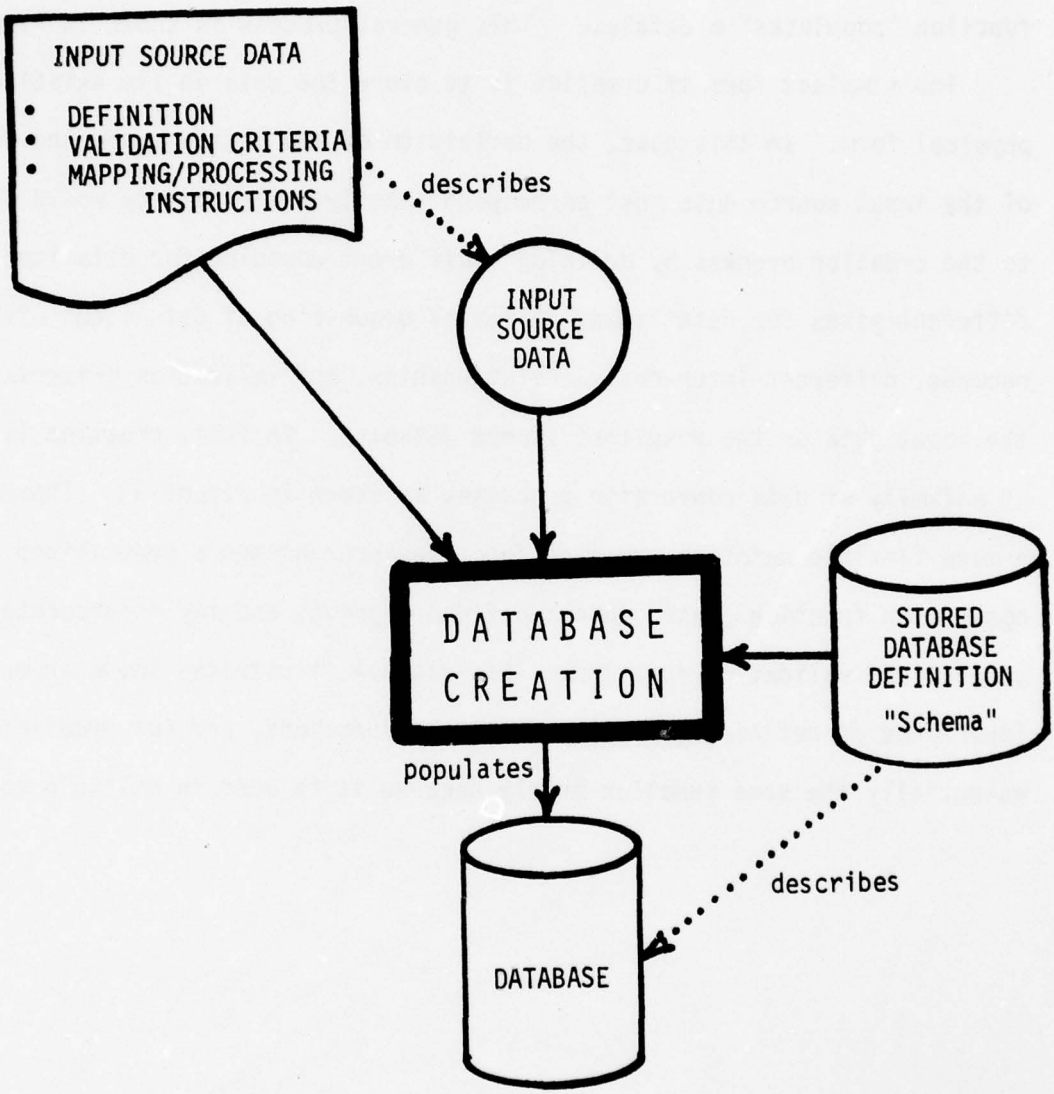


Figure 10. The Process of Database Creation.

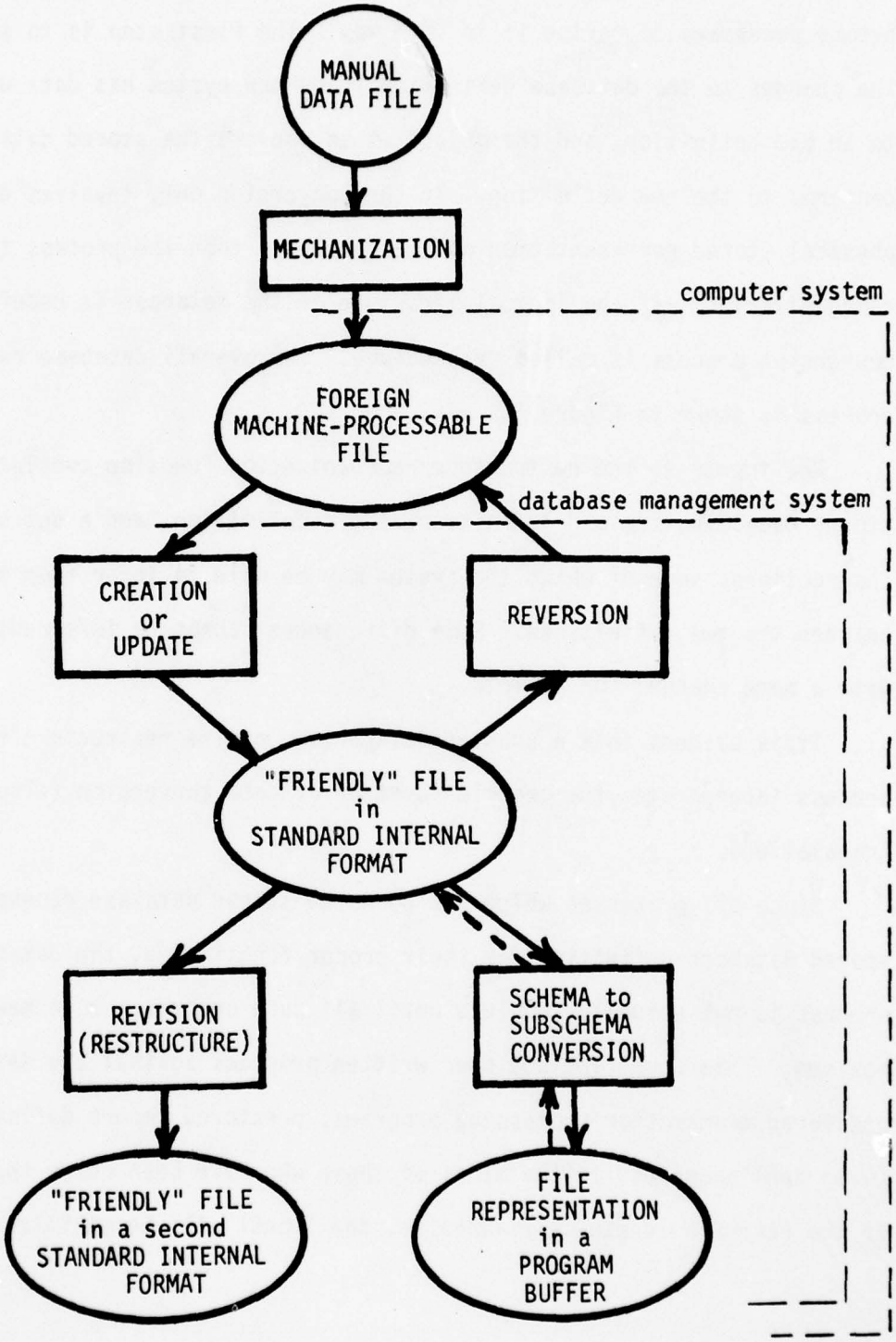


Figure 11. The Family of Conversion Processes.

Database Revision: Redefinition, Restructure, and Reorganization

After a database is established, that is, defined and created, it may become necessary to revise it in some way. The first step is to specify the changes to the database definition. Now the system has data which conforms to an old definition, and the object is to convert the stored data so that it conforms to the new definition. If the conversion only involves changing the physical stored representation of the database, then the process is called reorganization. If the logical structure of the database is redefined, the conversion process is called restructure. The overall database revision process is shown in Figure 12.

The inputs to the restructure/reorganization function consist of the stored database, the old definition, the redefinition, and a set of mapping instructions, some of which the system may be able to infer from the differences between the two definitions. Some differences cannot be inferred, however, as with a name change, for example.

It is evident that a substantial portion of the restructure/reorganization process incorporates the generic function of data conversion (also called data translation).

Since all processes which act upon the stored database depend upon the stored database definition for their proper functioning, the database revision process is not actually complete until all such processes have been appropriately revised. This would include user written programs against the database, prestored transaction processing programs, prestored report definitions, etc. (even the "programs" in the minds of those who have been using the old database). If the revision is strictly additive, the impact will be minimized.

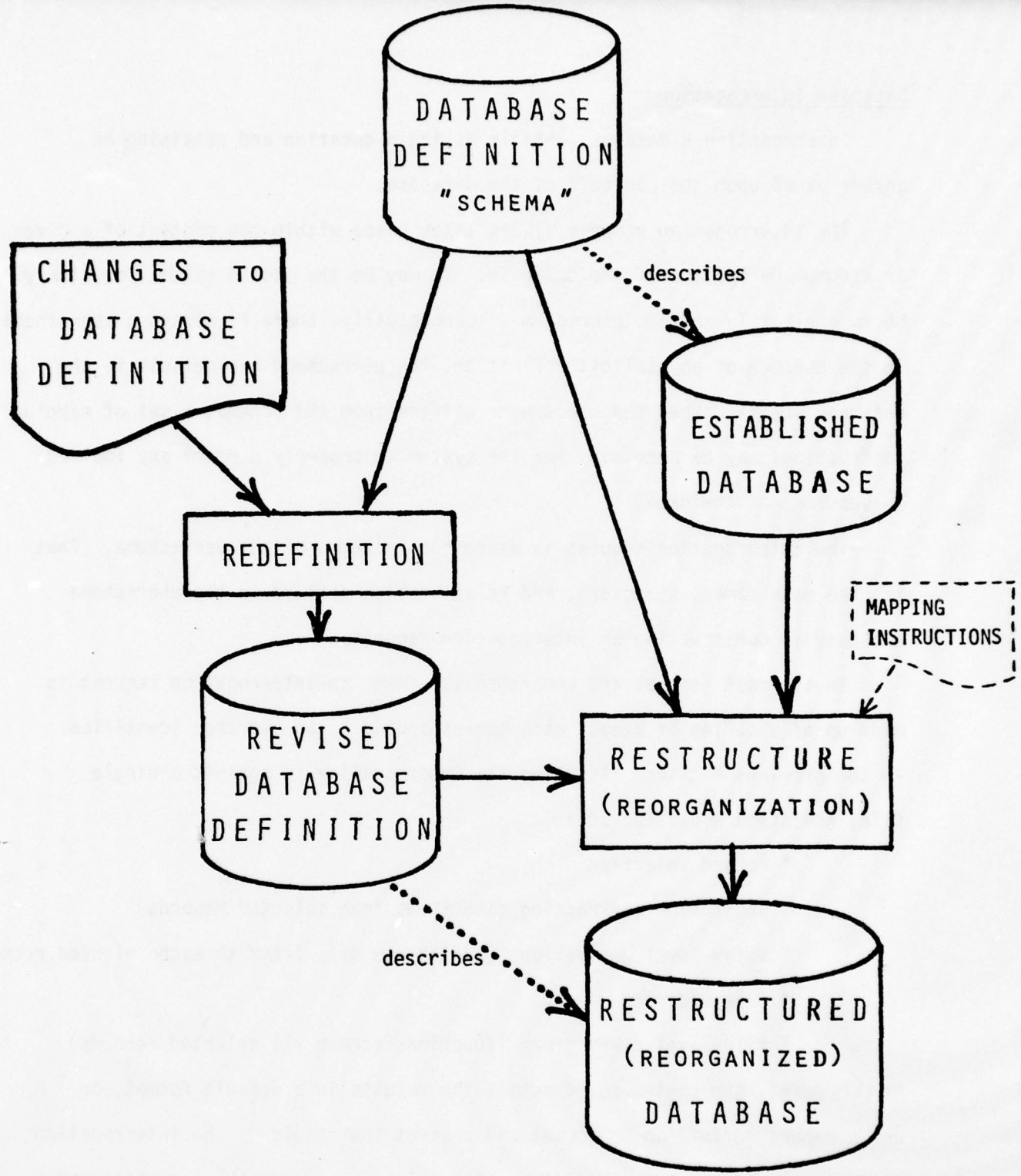


Figure 12. The Process of Database Revision.

Database Interrogation

Interrogating a database entails asking a question and obtaining an answer based upon the contents of the database.

The interrogation process always takes place within the context of a given or assumed definition of the database. It may be the stored schema, or it may be an explicitly defined userschema. Conceptually, there is always a userschema; in the absence of an explicit definition, the userschema can default to the database schema. When the userschema differs from the schema, a set of mapping instructions may be necessary for the system to properly perform any implied conversions of the data.

The interrogation request is expressed in terms of the userschema. That is, the data names, structure, and relationships defined in the userschema are used in constructing an interrogation request.

In its most general and comprehensive form, an interrogation request is made up of a series of steps, each corresponding to an operator identified in the previous section. Assuming the interrogation is against a single file, the steps would include:

- record selection
- projection (extracting data items from selected records)
- entry-level derivations (adding new data items to each selected record)
- file ordering
- file-level derivations (functions across all selected records)

At this point, the system could output the results in a default format, or use a report definition to format and present the results. The interrogation language itself may have facilities with which to specify the formation and presentation of results.

Database Update

The function of database update involves taking some input data or an update request and affecting a change in the stored database. Update changes the content but not the organization or structure of the stored database. An overview of the update process is shown in Figure 13.

Update may be accomplished with a generalized update language which is used to express all the necessary information for a complete update action. The primitive update actions correspond to the item and record level operators discussed earlier. Other update actions can be built up from these operators, for example:

- change all instances of a data item value to a new value.
- perform the same update action against all records which satisfy a given record selection expression.
- perform the update operation only if a specified condition is true.

Alternatively, an update action may be specified in terms of a transaction definition which contains validation criteria and rules for processing the transaction against the database. The transaction is defined in the same manner as a database -- in terms of the structure and format of data items and groups, batches, physical representation, validation criteria, responses to exception or invalid conditions, and the processing and mapping to the database. Batch header and trailer records are analogous to parent records of the transactions. With batches of transactions, additional processing directives are needed, such as ordering, and batch validation criteria.

With semantic conditions defined on a database, it is possible for such a condition to be temporarily violated after a single item-level or record-level update operation. Therefore, it becomes necessary to provide commands to suspend and restore the testing of semantic conditions on the database. These commands are needed only if an update process must be specified with multiple update actions using the functional primitives provided by the system.

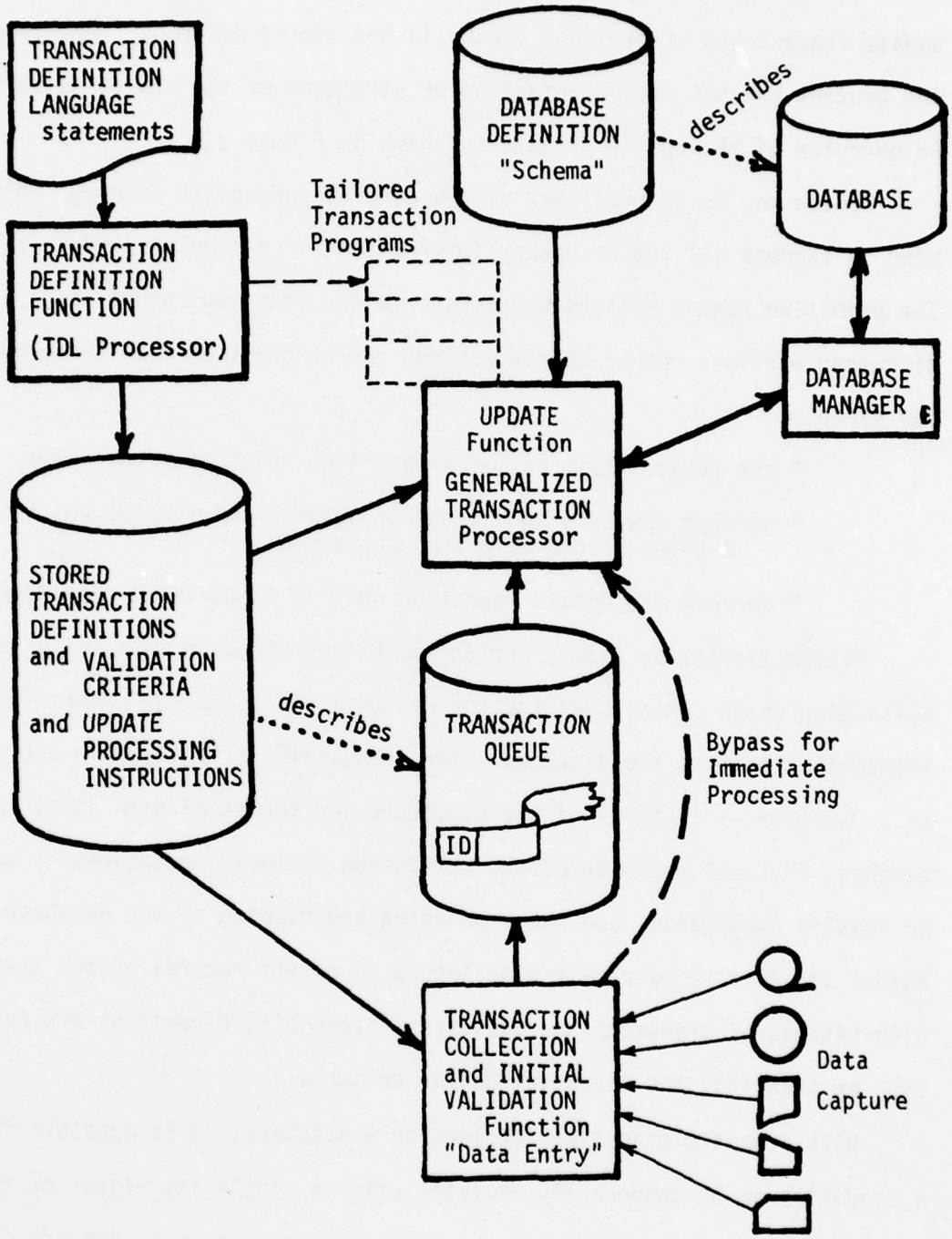


Figure 13. The Process of Database Update.

Userschema Definition

The userschema is system data and a facility is needed to define, process, and store userschemas. The userschema consists of a logical definition of data, its physical representation, perhaps validation criteria and semantic constraints, and the mapping between the data in the userschema and the database schema. In addition to defining and creating a stored userschema, processes are also needed to display, modify, and delete userschemas.

Transaction Definition

This function creates a stored transaction definition. A transaction definition includes the content and structure of a type of input data transaction, the validation to be performed on transactions of that type, and the mapping instructions for processing those transactions against the database.

Report Definition

The definition of reports desired on a recurring basis are stored as part of the system data. Functions are also needed to modify and delete a stored report definition. Some parameters may be omitted in the stored definition to be added at the time the report is invoked for generation. Invocation of report generation may be on demand, on a time-dependent trigger, or on some condition of the database.

User Profile Definition

Users must be defined to the system for the purpose of access control for privacy. For each bonafide user of the database system, the profile contains user identification and authorized actions (read, add to, modify, delete) against the database and against system data (for example, which userschemas and report definitions they are authorized to invoke).

Integrity and Performance Parameters

Some facilities are needed to define and modify parameters for the ongoing, underlying functions. These include:

- backup strategy (what to log, frequency of dumps).
- concurrency control.
- encryption-decryption keys.
- what to record on the audit trail.
- when and what performance statistics to record.

IV. BEHAVIORAL CHARACTERISTICS OF DATA AND PROCESSES

Behavioral characteristics of data and data processes constitute the third basic component of information requirements which the user and/or designer must specify. (Data definition and process specification are the other two.) The behavioral characteristics can be expressed in terms of either the actual real world entities and processes on them or their representation within the database as data and processes. In either case the user/designer specifies certain estimates of behavioral characteristics which the system can process in two ways -- (1) by comparing the various estimates and their derivatives for consistency and (2) by using the estimates as input to certain design algorithms. Once the database has been created and is being used the behavioral characteristics can be monitored both to determine the accuracy of the initial estimates and to determine when the use of the database has changed enough to justify reorganization.

The estimates of behavioral characteristics the user makes can be classified into two types -- mandatory and optional. Mandatory estimates are those which must be made for system design. Optional estimates are those which the user may supply to improve the design, but which are not essential. In either case, later performance monitoring would probably be done.

In discussing the behavioral characteristics, there is an underlying hypothesis. Behavioral statements can describe data or processes or both. To a certain extent these differences are simply differences in perspective. However, the assumption is that given a complete behavioral description of the processes, then the behavior of the data can be derived. Figure 14 attempts to depict the modes of arriving at the behavioral characteristics of data.

Behavioral characteristics of data can be obtained in three ways: direct estimates, derived from estimated behavioral characteristics of processes acting upon the data, or by monitoring actual usage and behavior over time. In practice, all three would be used. Direct estimates are needed to arrive at an initial file design.

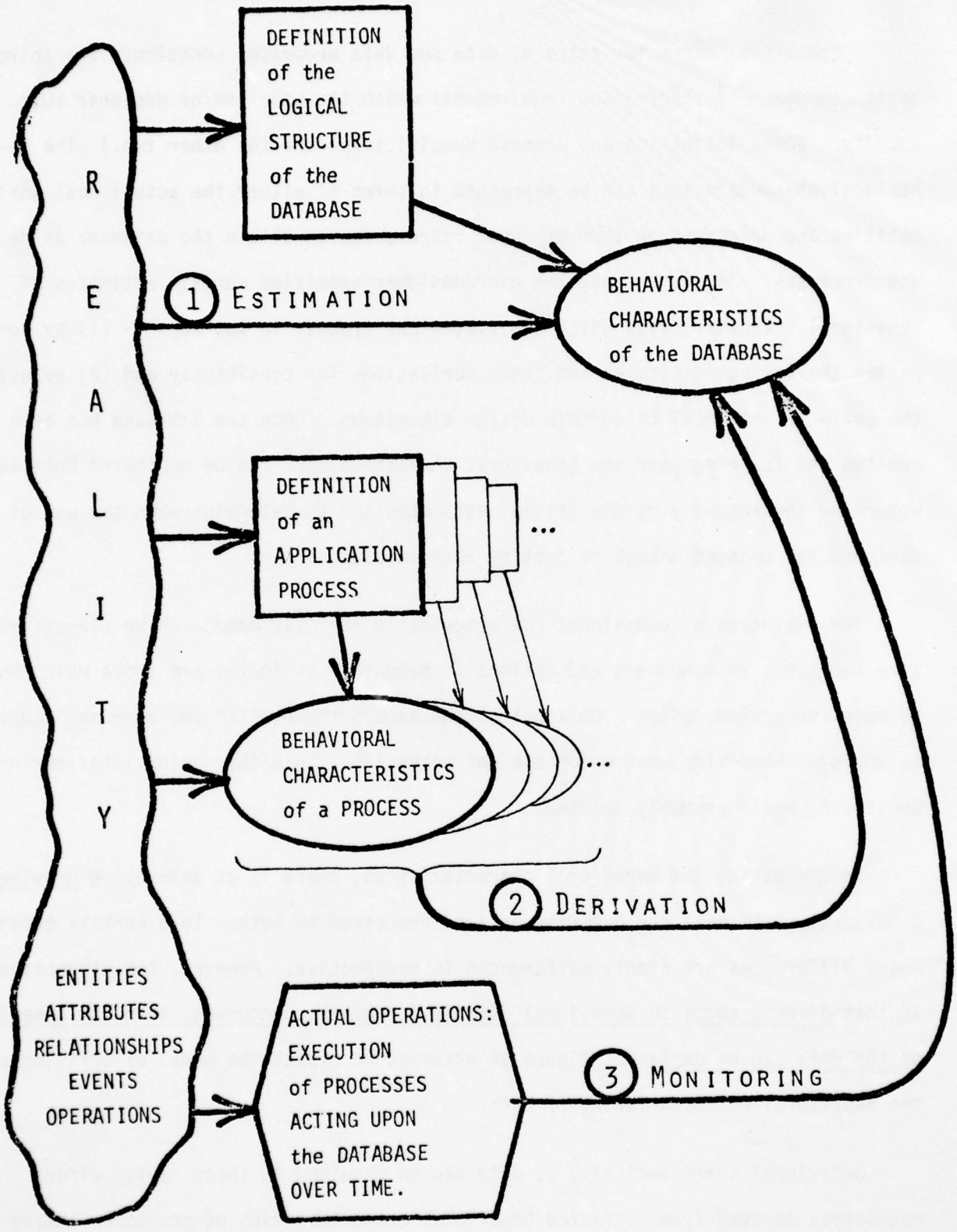


Figure 14. Obtaining Behavioral Characteristics of Data.

Characteristics derived from aggregate process characteristics are needed to validate the initial direct estimates. Once in operation with an established database, the actual usage and behavior of the database can be monitored. If usage patterns are or become significantly different from initial estimates, redesign may be required. The behavioral characteristics would be stored in the design database, both initial estimates and monitored statistics.

The derivation sequence for Behavioral Characteristics is as follows:

1. Behavioral Characteristics of external events (e.g. transactions, queries, or report requests), from which we can derive:
2. Behavioral Characteristics of internal application processes, from which we can derive:
3. The dynamics of the data:

   The database is a passive component of the system; it only behaves or exhibits dynamics in response to processes acting upon it.

Model statics relate to the data, while model Dynamics relate to the user application processes. Thus, we need a statement of initial conditions on the statics such as database size, and number of entities of each type. Then we can "run" the dynamics (upon which we have defined behavioral characteristics) to "derive" the behavior characteristics of the statics, that is, the database.

There are two types of behavior that can be described: static behavior describes the current state of the database or provides status information about the data, process, or users in the system; and dynamic behavior describes the change in the data or processes or the flow within the system.

There are three levels on which to describe Behavioral Characteristics:

1. First order characteristics:

   size or projected size,

   e.g. number of records in a file;

   e.g., distribution of the number of instances of a repeating group per entry.

2. Second order characteristics:

rate of change of size; growth.

e.g. number of insertions and deletions per unit time

e.g. number of requests for a report per unit time.

3. Third order characteristics:

changes in the growth rate.

First order characteristics describe statics and higher order characteristics describe dynamics. The first order or static characteristics provide the initial conditions on which we apply the higher order characteristics to yield projections of static characteristics in the future.

Behavior Relating to Structural Data Elements

There is behavioral information that can be monitored at each level of data (item, group, relationship, record, file and database). Two main uses of this data are to assist the design process in determining encoding schemes and access paths through the data. Some of the properties are cumulative, in other words, the effect of moving up the hierarchy of levels is simply a summation, e.g. lengths for items, groups, records, and files. However, other characteristics such as access paths are emergent and don't exist for lower levels (below the group).

1. Domains - The values within a domain have certain lengths. Behaviorally one can consider the probability distribution of the various values or lengths of values. These characteristics will help in designing encoding methods.

2. Items - Items are either present or absent. If they are present they assume certain values from the appropriate domains. Important behavioral characteristics are the percentage of time the item is absent and the distribution of the lengths for the values the item assumes. This information is useful in determining how to represent the item in the database. This representation has two components. First, how to signify the item's presence or absence: (1) always present by using a null value when necessary; (2) assume present, but use flag bits when not; (3) assume all items are absent and label all variables that are present by attaching a name to the value. Second, how to encode the information (fixed length binary, variable length, encoding table, Huffman code, etc.). Most of these behavioral characteristics are cummulative, since item lengths build into group, and record lengths.

Additional behavior which relates to an item are how many times it is retrieved, by which users or processes, for what purpose (on each of two dimensions -- (1) for use in selection or for presentation; and (2) for read only, or value modification). However, some of this behavior could also be considered to describe the

AD-A042 482 MINNESOTA UNIV MINNEAPOLIS MANAGEMENT INFORMATION SY--ETC F/G 9/4
DEVELOPING A DATA PERSPECTIVE ON THE SPECIFICATION OF INFORMATI--ETC(U)
MAY 77 G C EVEREST, O BRAY, I VALTERS N00167-76-M-8476

UNCLASSIFIED MISRC-TR-77-05 NL

2 OF 2
AD
A042482

END
DATE FILMED
8-77
DDC

process operating on the data rather than or in addition to describing the data behavior. Examples of how this behavioral information would affect database design are: (1) frequently retrieved data would probably be stored on fast, expensive devices (record segmentation) and not be encoded unless there was a great discrepance in value frequency and the values were, according to information theory, longer than they needed to be to distinguish among them; (2) use for selection rather than presentation would suggest the use of indexes and access paths; and (3) use for read only versus modification suggest the tradeoff between retrieval versus update efficiency.

At system design the user could probably only provide estimates of these characteristics. However, by performance monitoring, better estimates could be obtained and used for system redesign or optimization. Database reorganization points could be determined by comparing the differences in performance given the better estimates and the cost of actually reorganizing the database.

3. Groups - Groups are collections of items and may be linked to other groups. This suggest two types of behavioral information. For groups, as collections of items, most of what was said for items (presence-absence, length, and use) also applies.

4. Records - Record types describe the entities about which the data is being maintained. Behavioral characteristics include: (1) the number of record types; (2) the number of instances of each type; (3) the growth or contraction of the number of instances, by type; (4) the amount of access to each type (conceivably by instance); (5) by which process and operating for which user; and (6) the number of multivalued items or repeating group instances within a parent record. Record size (and their summation into file length) is simply a summation of the item and repeating group lengths of the record components.

5. There are relationships between various record or entity types and between groups which can be represented by a variety of linkage methods. These linkages

may be implemented with embedded links or pointer tables. The amount of usage for each relationship and the volatility of the relationship will determine the best way to implement the linkage. The method of establishing this linkage depends upon the patterns of group processing. If there is a high probability that two groups will be processed together then access of one group from the other should be relatively easy. This pattern of processing can be determined by monitoring the access paths through the data -- frequently used paths should be very efficient, while infrequent paths do not need to be. Once again the user may provide initial estimates of this behavior. Later this can be improved by performance monitoring.

Performance monitoring is not a substitute for the specification of behavioral characteristics of data and processes. During initial design the system does not yet exist. Once an initial design has been implemented, the performance of the system can be monitored. The performance statistics will indicate the need for redesign to the extent that the initial design parameters were not accurate or that patterns of processing change. Of course, if new or changed patterns of processing are known in advance, the system can be redesigned, and the database reorganized prior to the new or changed use.

Static Behavior

Behavior of either data or processes can be divided into either static or dynamic behavior. In their stored form, process specifications are simply other examples of data, therefore, some of their characteristics will be considered under data. For example, when the volatility of data is described this will also describe the volatility of the stored process definitions. The static information provides a snapshot of the state of the database at a point in time. It includes such information as: (1) the size of the database and the defined processes; (2) the number of users of the database; (3) the number of processes or user applications against the database (this is different than 2 which is the number of users each of which may use many different processes); (4) a confidence level or measure of the quality of the data in the database. Each of these behavioral characteristics can be described in more detail.

1. Size - The size of the database can be measured in a variety of logical or physical units (characters, items, groups, records). Different units are needed to measure the size of processes. However, they may still be physical (number of characters, instructions, words, storage units) or logical (number of processes independent of their size or how they are stored).

2. Users (or individuals) - Users can be defined in two ways -- users as individuals or departments which operate on the database making queries and updates or users as the processes which these individuals or departments use. Users in the latter sense (processes) are considered in the next section. Users in the former sense can be described in several ways. First, there is simply the number of users. How many different individual users are making requests of the database. This provides information about the sharability of the data, how responsive the system must be, and how many users it must be capable of handling. Obviously, all of these users will not be making their requests simultaneously, therefore two estimates are needed for the number of users -- total, and number at any one time.

A second factor is the dispersion of the users -- are they all at one site or dispersed throughout a plant, city, state, or country. This information will be important in designing distributed databases. Another characteristic of the user is his sophistication in dealing with the database. Is he a programming or non-programming user? If nonprogramming, is he a general, parametric, or casual user? How much training and experience has he had in working with the database? This determines the level with which the system can deal with the user. It may require extensive tutorial assistance providing the user with a list of options and a full explanation of each or at the other extreme provide virtually no assistance, simply accepting the user's requests as he enters them. To a certain extent some of this information would also be applicable at the other level, the process as the user. The sophisticated or a naive user will have a different interface with the database, but will often use the same underlying processes. Therefore, there would be additional information if the estimating and monitoring were done at the individual level.

3. Users (as processes) - In this case much of the same information described in the previous section would also apply, however, it would not be disaggregated to the level of the individual user. Also certain processes could not be completely classified. For example, if a single application process were capable of dealing with either a sophisticated or an unsophisticated user, it would have to be classified when it was invoked, depending on how it was used, rather than based on the characteristics of the process as it was defined.

Dynamic Behavior

Dynamic behavior refers to changes over time. These changes may be in either data or the processes. There are two basic ways to measure dynamic behavior -- (1) volatility = change/base in a unit of time or (2) the rate of an occurrence per unit of time. Percentages are a special case of volatility in which the change and the base are in the same units, however, a more general form of the measure might be queries/user/day.

In terms of data, two important volatility measures are the number of value changes to the database and the number of inserts and deletes (size modification). While these measures could be provided as a rate/unit of time, they are often dependent on the size of the database, therefore, it is appropriate to use a measure which will control for this factor. The selection of the proper control is critical, since it is usually the control factor which is varied to study various design alternatives. For example, consider the relative usefulness of queries/day versus queries/day/user. Other dynamic behavioral characteristics of data include:

1. the rate at which certain data (items, records, files) is used;
2. how it is used, e.g. is an item used for selection or for presentation;
3. how much data is retrieved by a request;
4. who it is retrieved for; and
5. how it is used functionally for read only, update, or insert/delete.

Similarly, we can consider the dynamic behavior of individual processes. Such measures include:

1. how many times a process is invoked;
2. when it is invoked;
3. by whom it is invoked;
4. which access paths it uses to obtain what data;
5. what is its response time to fill a request.

This latter measure should probably be controlled for type of request (read only, value modification, insert/delete, and more complex requests which may include file level derivations), size of the response set, and possible time of day.

Data Behavior Characteristics

The behavioral characteristics of data can be grouped into three areas: (1) the size and rate of change in the size of the database; (2) the volatility of the database, which would include changes in both the content and structure of the database; and (3) the access and use of the database.

1. The size of the database is a static characteristic. However, the change in the size (growth or decay) is a dynamic characteristic which describes the behavior over time. The size will be constant in either of two cases: (1) if there are no adds or deletes to the database (the only changes are for value modification -- regardless of whether they are data dependent or independent) or (2) there are balanced changes, e.g. approximately equal numbers of adds and deletes. To specify these characteristics, the user should estimate the initial size of the database and its growth factor per unit of time. This growth factor could be the net growth or the rate of adds and deletes separately. Obviously the latter approach would provide more information. If the initial hypothesis is correct, one could specify the frequency or amount of use of the add and delete processes and then derive the corresponding behavior for the data.

2. Volatility is a dynamic characteristic which refers to the amount of change over a specific time interval. It is basically the number of records or other units changed divided by the total or base number of those units. The volatility could apply to value modification changes to the database or to changes in the size of the database. Both of these approaches consider changes to the content of the database. Another more difficult type of volatility concerns changes to the structure of the database. This type of volatility would primarily involve adding or deleting relationships and access paths to the database. Estimates of this type of volatility provide a measure of the flexibility which should be designed into the database. A database designed for a relatively stable set of applications would have low

structural volatility, whereas one designed for a new, expanding set of applications would be expected to have high structural volatility. This structural volatility is important because it is usually obtained at the cost of efficiency of time, space, or both.

3. Access to the database includes access paths through the database and, as a subset, the question of who is allowed the use of those paths. A static characteristic is what access paths are currently in existence. Dynamic characteristics would be the frequency of use of specific paths over a period of time and the change in this usage. From one perspective the access path used to obtain a piece of data is a behavioral characteristic of the data. However, this access was for some individual user and through a specific process, therefore, the access path used to obtain the data could also be considered as part of the behavior of one of these entities. The reason for monitoring behavior with respect to access paths is to determine their relative usage so that tradeoffs can be made in determining which access paths to maintain to provide proper responsiveness to the users.

APPENDIX: RESEARCH CONTEXT
INFORMATION SYSTEM DEVELOPMENT AND THE DESIGN DATABASE

1. INTRODUCTION

The research reported herein is part of a larger research area which focuses on the portion of the information processing system (IPS) design database used as the basis for subsequent file design. First, all information needed for file design must be captured and stored in the design database. Three parts of this information relate to (1) the definition of data, (2) a high-level specification of patterns of processing against the data, and (3) the behavioral characteristics of data and processes. A full description of these three parts of the design database can provide the basis for evaluating the Problem Statement Language (PSL) [Hershey, 1974]. PSL would be evaluated as a basis for gathering and expressing the information needed for file design.

This appendix describes the context of the overall research area, namely, the ISDNLS prototype system for IPS development and discusses the specific ideas relating to the definition of data, its behavioral characteristics and patterns of processing, and testing the adequacy of PSL.

2. SYSTEM LIFE CYCLE

For a simplified view, the life cycle of a computer-based system goes through three phases: development, operation, and modification. The primary resources required in each of these phases are human resources during the system development phase, machine resources during the system operation phase, and again, human resources during system modification. This is depicted in Figure 1. Even before final installation of a system that has been developed, pressure builds to modify the system. As the months pass the pressure mounts until eventually management is forced to initiate substantial system revision. The perssures for change may stem from economic, technological, or human behavioral considerations.

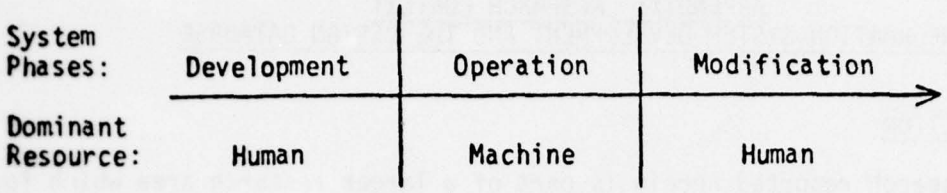


Figure 1. Dominant Resources in System Life Cycle

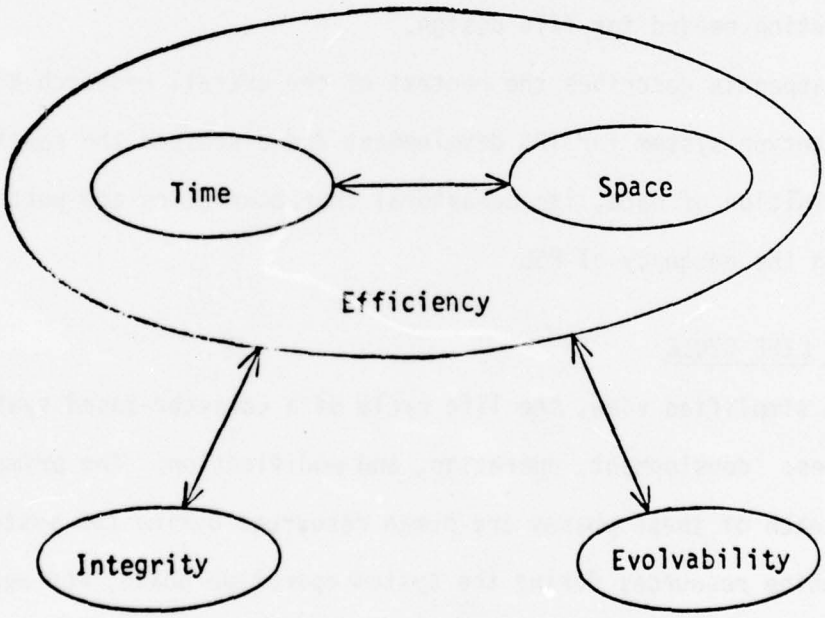


Figure 2. Multiple Objectives in Systems Development

PSL and database design to date have only considered the efficiency objectives of time and space utilization. This only reflects a desire to make best use of machine resources. It largely ignores the human resources necessary for system development and system modification.

The multiple objectives in systems development are machine efficiency, integrity, and evolvability (see Figure 2) [Everest, 1974 May]. Human resources are becoming increasingly costly relative to machine resources. Considerations of machine efficiency do not lead to effective use of human resources. The two objectives which primarily affect human resource utilization are integrity and evolvability. Greater system and data integrity leads to increased user and management confidence in the use of a system and its output. Greater evolvability leads to easier system modification in response to changing user demands, thus resulting in more responsive systems.

Integrity is becoming a significant concern in system and database development and operation. Organizations are increasingly concerned about the unauthorized disclosure of sensitive and classified information. Federal privacy laws demand that personal data be accurate and that its disclosure be controlled. The objectives of integrity and evolvability must be factored into the system development process in the future.

Unfortunately, the objectives of integrity and evolvability generally stand in opposition to the objectives of machine efficiency of time and space utilization. That is, a greater measure of integrity generally means less efficient machine operation. Furthermore, integrity and evolvability must be factored in at the beginning of the system design and development process. They cannot be effectively added on later.

3. SYSTEM DESIGN AND DEVELOPMENT PROCESS

Systems design and development is the first phase in the system life cycle. With an eye toward automating the systems development process, the design database takes on a pivotal position. The process of system development and design is assumed to be a highly iterative process. At the front end the process involves capturing requirements from the problem and application environment. At the back end the process involves using the information in the design database to design and optimize files and application processes. The design database is the repository for all information input from the environment. It contains all original and derived data captured and developed during a system design process.

As shown in Figure 3, there are two families of processes involved in the machine-based development of an application system. First is the PSL family of processes which capture and store problem application requirements. Then there is a second family of processes to verify the problem statement and to design and optimize the database and application processes.

From the reality of the using environment come information requirements and application process requirements. The reality of the using environment includes data stemming from various events reflecting status and changes in status of various entities in the real world. It also reflects an organizational context, activities of operating personnel, and decisions of managers. In the process of information systems design, the user needs guidance in how to look at the reality of the environment, and needs guidance on how to formally express these requirements given some requirement statement language such as PSL. The data resulting from a formal expression of these information and application process requirements is stored in a design database.

The design database then is the basis for the design of files, hardware configuration, and application processes. That is, the design database is input to the process of defining and generating files, specifying a hardware and

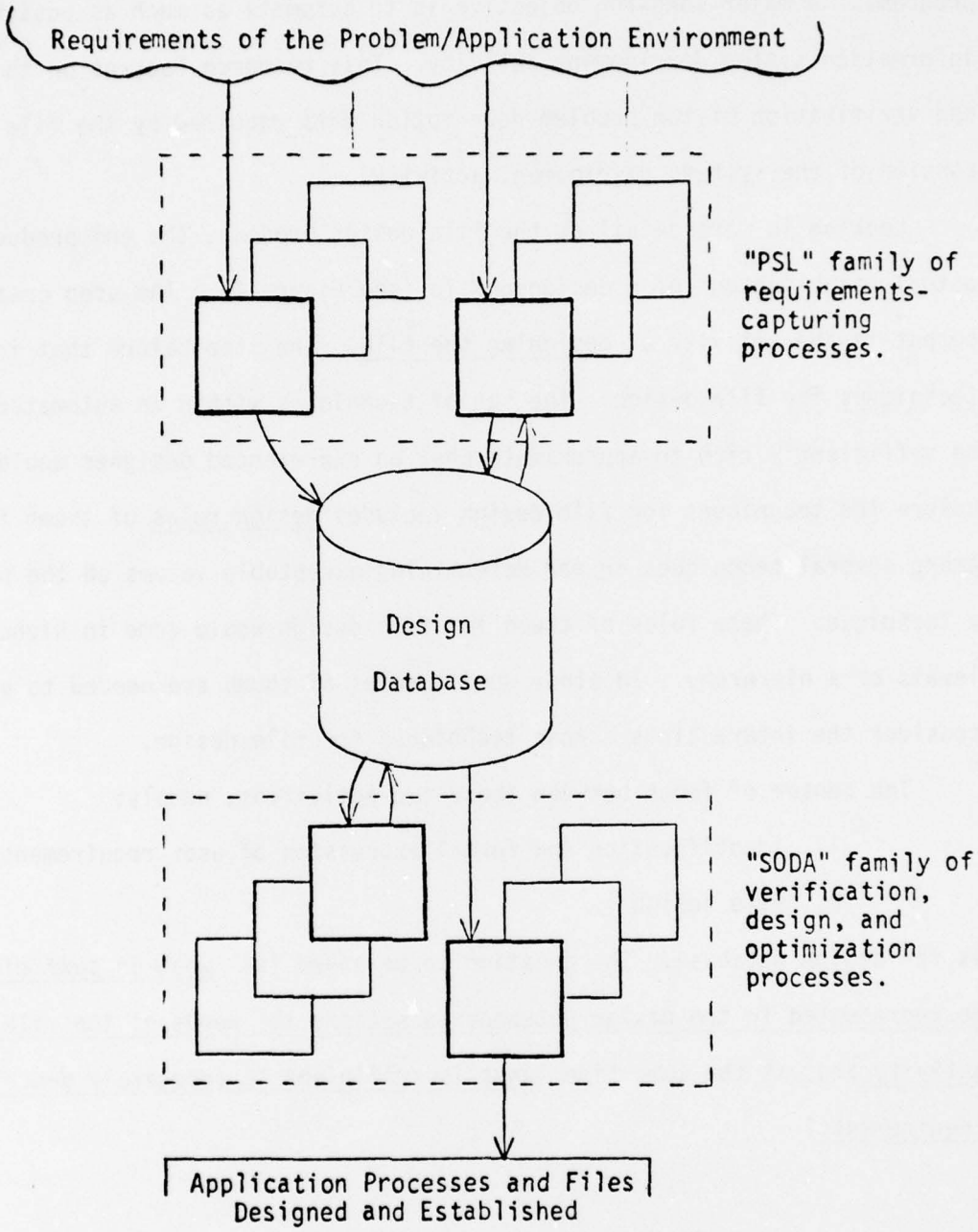


Figure 3. Design Database in the System Development Process

communication system configuration, and specifying and generating application programs. A major long-run objective is to automate as much as possible of this information system development activity. This research focuses on the collection and verification of the problem description data required by the file design portion of the systems development activity.

Looking in more detail at the file design process, the end product is the actual establishment of a designed file (see Figure 4). The step creating this output is the activity of designing the file. The step before that involves techniques for file design. The set of techniques within an automated system must be sufficiently rich to approximate what an expreienced designer would do. A step before the techniques for file design includes design rules of thumb for choosing among several techniques or for determining exceptable values on the parameters of a technique. These rules of thumb for file design would come in higher and higher levels of a hierarchy. In other words, rules of thumb are needed to explicitly consider the interactions across techniques for file design.

The center of focus between these two activities, namely:

1. identification and formal expression of user requirements and
2. file design

is the design database. The question to be asked is: what is sufficient data to be represented in the design database to satisfy the needs of the file design activity and, at the same time, what is sufficient to adequately describe the user requirements?

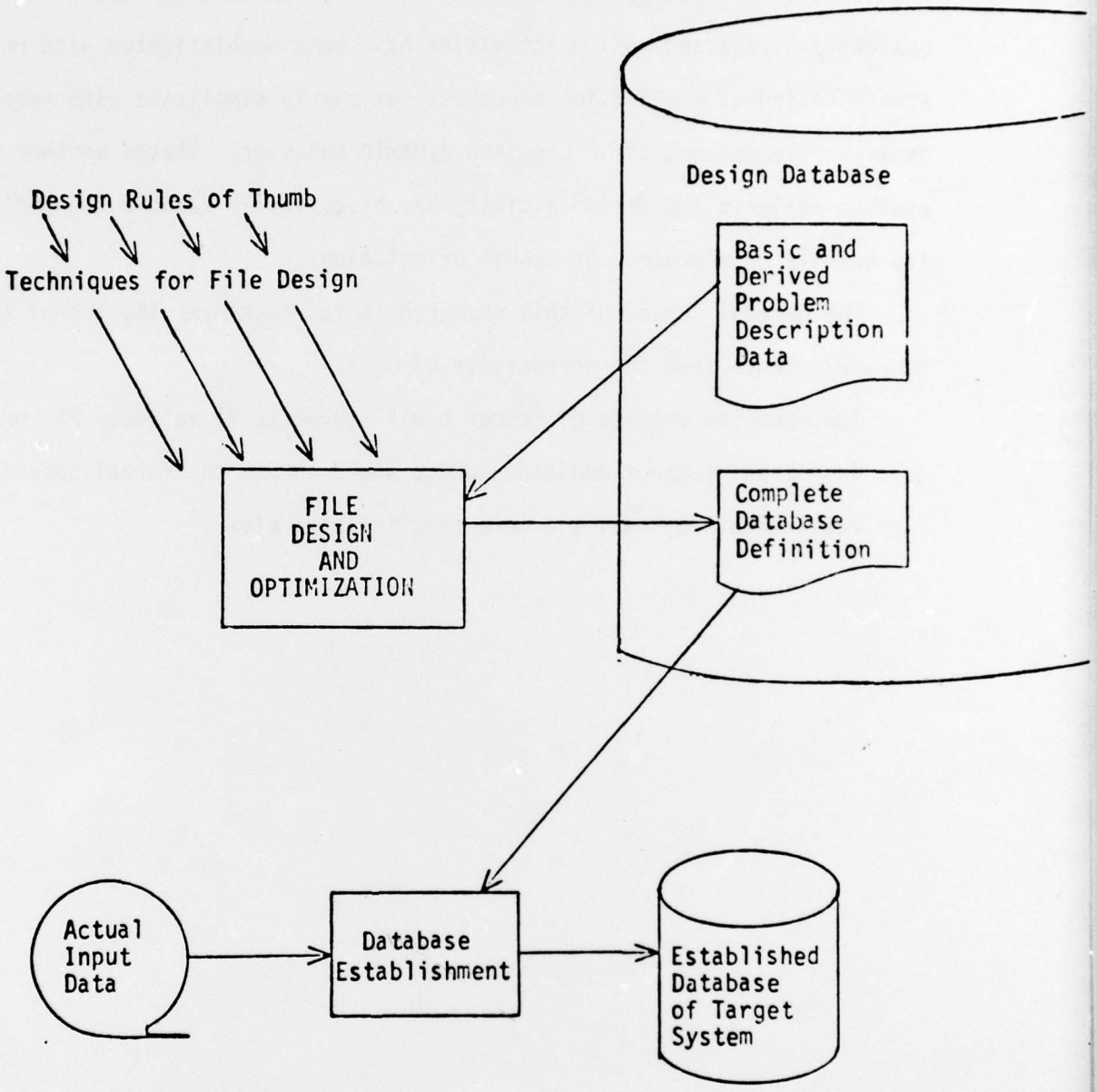


Figure 4. Steps Toward File Design

4. RATIONALE AND OVERVIEW OF THE RESEARCH

During the past decade the field of data processing has begun to recognize the significance of data *per se* in the system development activity. Traditionally, systems analysis and design activities have been sophisticated with respect to the specification of application processes but overly simplistic with respect to the data -- file design, structure, and dynamic behavior. Stated another way, the systems analysis and design activity has historically taken a view which leaned too heavily on a process or "runs" orientation.

The general thrust of this research is to re-examine the formal specification of requirements from the perspective of data.

The expected results of research will serve to 1) validate PSL or 2) identify gaps in PSL and suggest additions which would enrich the formal specification of user requirements by taking a more sophisticated view.

5. DATA-RELATED INPUT TO FILE DESIGN PROCESS

Out of the statement of user requirements and as input to the file design process, the design database must contain (but not be limited to):

1. a full definition of the static structure and semantics of the database.
2. a precise, high-level specification of expected patterns of processing against the database.
3. a full specification of the dynamic behavioral characteristics of the data and processes.

The database definition information must be adequate to represent the entities and relationships in the user environment. In other words, it must provide modeling fidelity. It must also be sufficient for the file design activity to build effective and efficient files and direct operations on the files. It must contain sufficient information to enable a system to automatically maintain the logical structure and semantic integrity of the stored data and to handle structural volatility.

A full specification of the dynamic behavioral characteristics of the data must be adequate to represent the user environment and sufficient for the file design activity. This stems from an expression of patterns of processing against the database from the perspective of the using environment. Patterns of processing would be expressed in terms of the highest (that is, largest or broadest) unit of processing which was independent of other units of processing against the data.

These three classes of data would be contained in the design database. The split between static structure and dynamic behavior may prove to be a difficult basis on which to classify certain design data.

5.1 Examples of Data Structure Definition Information

Information pertaining to a high logical level might include:

- The entities represented
- The attributes which describe those activities
- The domain of values for the attributes
- Intra-entity relationships among attributes
- Inter-entity relationships
- Definition and derivation rules for virtual and inferred data

Additional data needed to produce a physical realization of the database to be stored might include:

- Encoding of item values
- Sizes of item values
- Distribution of item value occurrences
- Physical access paths, including such things as indexes

Additional information may be included, such as:

- Validation criteria
- Semantic constraints
- Input/output formats
- Device/media control information

5.2 Examples of Patterns of Processing

At the lowest level, processing against the database can be expressed in terms of reading and writing individual records. This would be analogous to a COBOL programming language facility which provides the verbs OPEN, CLOSE, READ, and WRITE. However, when a user writes these verbs in a program there is usually some higher level pattern of processing implied in the particular sequence of issuance of these verbs. If an efficient and effective job of file design or process execution is to be accomplished, it is important that information relating to these patterns of processing be obtained at a high level and in advance of the performance of the processing. This is partially accomplished in a system such as COBOL with the prior declaration of whether or not a file is to be opened for input, output, or update and a prior declaration of whether the file is to be processed sequentially or randomly. However, much more information is generally available regarding patterns of processing.

To draw a couple of examples, consider one pattern of processing in a bank which is to calculate the accrued interest on every savings account and credit that amount to the account. A second level of processing would be represented when a check is presented for payment and the teller needs to know whether sufficient funds are available in the account. These two patterns of processing represent two extremes. Little research has been done on the development of a taxonomy of patterns of processing and of how to describe patterns of processing. Some work in this area could greatly aid the process of information systems design and development.

5.3 Examples of Behavioral Characteristics

The types of information included under behavioral characteristics are:

- Volumes of data
- Growth in volumes of data
- Volatility
- Timeliness
- Structural volatility
- Data conversion
- Geographical distribution
- Sensitivity to loss or damage

Behavioral characteristics would also relate to the queries directed to the database. This would include such information as:

- Source of the query
- Frequency of each type of query
- Volume of data involved in the response to the query
- The response time requirements

Behavioral characteristics also relate to transactions which update the database. These characteristics would include:

- Source
- Frequency
- Volume of data per transaction
- Response time requirements
- Validation criteria

6. PSL AND THE PROTOTYPE DEVELOPMENT

The Navy has begun with a prototype development utilizing the ISDOS Problem Statement Language (PSL) to formally express application system requirements and to serve as a basis for subsequent file design activities. The expected results of this research will serve to (1) validate PSL or (2) identify gaps in PSL and suggest additions which would enrich the formal specification of user requirements by taking a more sophisticated view.

The following reflect some initial perceptions of the adequacy of PSL.

While PSL allows fairly detailed definition of application processes in terms of inputs, outputs, triggering and triggered events, parent processes and subprocesses, it does not provide for the formal expression of the dynamic nature of the process. The only provision so far is for "comments" helpful to the designer. These are captured and merely echoed within PSL; they are not in a form that is usable by the machine. This would suggest the need for an exploration of patterns of processing. The development of a taxonomy within which to express patterns of processing could provide the basis for a more formal expression of processes within a formal requirement specification language such as PSL.

PSL is wholely inadequate in its expression of system and data integrity requirements. For example, it lacks the direct ability to express error checking and correction procedures, data validation on input data and stored data, access control authorization, process discipline required for controlling concurrent update, and data encryption needs due to geographical dispersion of data and users.

Another weakness evident in PSL is the inability to declare restrictions to the class of data structures to be used in any instance of an application system design. PSL implicitly assumes a network structure similar to that presented in Chapter 2 of the CODASYL Systems Committee Feature Analysis of 1971 [CODASYL, 1971 May] and the CODASYL Data Base Task Group Proposal for 1971 [CODASYL, 1971 April]. It may be highly desirable in the development of a system to restrict the class of

data structures to hierarchical structures or to relational structures, perhaps even binary relational structures.

Several questions can be posed relating to the adequacy of PSL for formally expressing system requirements as perceived by problem definers and users. Is it really possible for a problem definer in the application environment to express the system requirements using PSL? PSL is the basis for various file design activities but is the information known or attainable and expressable at the level and in the form required for PSL? Is PSL a sufficient vehicle for a user to describe his environment, the critical factors and relevant parameters of that environment?

7. SUBSEQUENT TASKS IN THIS RESEARCH AREA

This report (MISRC-TR-77-05) is a preliminary specification of the information needed to express information system requirements. This research naturally leads to a language for expressing that requirements information. Furthermore, to test the adequacy of the descriptive information requires that these ideas be communicated, in other words, that a language be developed which provides a formal and concise means of expressing the information needed to describe the information system requirements in a problem environment.

Given an initial specification of an information system requirements language, it is necessary to test and explore the validity, generality, completeness, expressibility, and sufficiency of the semantics of the language. Subsequent research tasks will seek to refine, modify, and extend this language to fit an ever widening spectrum of defined data structures and processing environments.

Ideally, the information content and language for expressing information system requirements should be generalizable and consistent across several complexity spectra:

a) Single flat file to. . . . . hierarchical and multi-file structures

b) Single user to . . . . . a multi-user or shared environment

c) Single application to. . . . . different application environments

d) Time and space efficiency to . . . . added data integrity requirements

e) Static requirements to . . . . . a dynamic environment wherein usage statistics are gathered and the possibility for redesign exists

f) A single data processing node to . . databases in a distributed processing environment

Most current research in file design assumes a priori a single entity connotation, which is to assume a single file. Much of it also assumes a single flat file, or makes no material assumptions regarding intra-record structures. The general approach to be followed in this research is to start by considering the single flat file and then to extend the descriptive information and language semantics to incorporate multi-file and hierarchic file structures. In making this generalization, it will be important to explore the degree of dependence of the language for expressing patterns of processing on the object data structure model. To what extent will expressions for processing against one data model be transferable to processing against another data model?

7.1 Test of Adequacy for Describing Application Environments

Several tasks can empirically test the adequacy of the information system requirements language for describing selected user application environments in the Twin City area of Minneapolis/St. Paul, in the Navy, and possible other user environments. The Management Information Systems Research Center of the College of Business Administration at the University of Minnesota has direct access to several large firms with substantial data processing activities. These firms would serve as the basis for the survey to gather indepth information and to test the initial statement of data definition, patterns of processing, and behavioral characteristics as expressed using the preliminary language specifications.

At the beginning of this task, the proposed methodology for visiting user application environments will be expressed in a working paper. Then the findings of these visits will be recorded in occasional working papers.

Users will be selected from organizations that utilize a relatively sophisticated approach to applications systems development and design. Such testing would first examine in detail selected application processes. Are such processes contained within the descriptive taxonomy of patterns of processing? Are the semantics of the processing expressable in the information system requirements language? At the same time, the established data structures will be examined to see if the proposed information system requirements language is adequate to describe the data structures. The semantics of the data structure can be gleened from formal data definitions, documentation, and application processes. Not only will such activity point out weaknesses in the proposed information system requirements language, but it will suggest new semantics which are not expressable and therefore require an extension of or change in the proposed language.

7.2 Test of Sufficiency for Database Design

This avenue of testing requires an examination of selected database design procedures. The research would assess the degree to which the required input parameters are provided by or derivable from the information system requirements language. To be sufficient, the information expressed in the language must all be directly available or derivable without further reference to the user environment for additional information. The Navy may suggest some database design procedures which can be used to test the sufficiency of the information system requirements language. Several design procedures developed by Severance, et al., under contract with the Navy would be obvious candidates. In addition, the design aids from various vendors can be examined.

7.3 Test Against Existing Database Management Systems

Differing database management systems require differing definitional information in order to establish a database environment. Some systems are considerably more sophisticated than others in the level of information required from the user of the

system. Several tasks under this test can review the preliminary language specification against the reference manuals for some of the more comprehensive and sophisticated database management systems. Alternatively, a task could look across several systems in terms of a particular function such as report definition or transaction definition.

7.4 Continued Review of the Literature

Throughout the course of this research, items in the literature will be reviewed, compared against the preliminary language specifications, and useful items will be added to the automated, annotated bibliography on the specification of information system requirements [Everest, Bray, and Valters, 1976].

7.5 Evaluation of Problem Statement Language (PSL).

With the rather comprehensive specification of information system requirements language to be developed under the above tasks, it is possible to evaluate the semantics of PSL. This evaluation would primarily take the form of identifying those semantics which are absent from PSL. The evaluation would further suggest ways in which PSL could be conveniently extended to make it more useful in the prototype development at the Navy. Is PSL complete in at least representing the information required to perform the file design task and to adequately represent the entities, relationships, and behavioral characteristics in the real environment of the users? An attempt will also be made to use PSL for expressing information system requirements in parallel with the visits to user application environments. This will provide an additional opportunity to evaluate PSL and propose modifications and enhancements.

BIBLIOGRAPHY

1. CODASYL Systems Committee, Selection and Acquisition of Data Base Management Systems, New York: Association for Computing Machinery, 1976 March, 252 pages.

2. CODASYL Programming Language Committee, Data Base Task Group Report, New York: Association for Computing Machinery, 1971 April, 273 pages.

3. CODASYL Systems Committee, Feature Analysis of Data Base Management Systems, New York: Association for Computing Machinery, 1971 May, 520 pages.

4. Everest, Gordon C., Olin Bray, and Indulis Valters, An Automated Annotated Bibliography on the Specification of Information System Requirements, Minneapolis, MN: University of Minnesota, Graduate School of Business Administration, Management Information Systems Research Center, MISRC-TR-77-01, (final report under contract with D. W. Taylor Naval Ship Research & Development Center). 1976 October 11, 156 pages, (NTIS: AD A038 398).

5. Everest, Gordon C., "Basic Data Structure Models Explained with a Common Example," Proceedings Fifth Texas Conference on Computing Systems, Austin, 1976 October 18-19, pages 39-46.

6. _____, "Database Management Systems Tutorial," Fifth Annual Midwest AIDS Conference Proceedings, Vol. 1, Minneapolis, Minnesota, 1974 May 10-11, edited by Norman L. Chervany, Minneapolis: University of Minnesota, College of Business Administration, 1974, pages A1-A12.

7. _____, "Concurrent Update Control and Database Integrity," Data Base Management, edited by J. W. Klimbie and K. L. Koffeman, Amsterdam: North-Holland Publishing Co., 1974, pages 241-270.

8. _____, "Managing Corporate Data Resources: Objectives and a Conceptual Model of Database Management Systems," unpublished doctoral dissertation, Wharton School, University of Pennsylvania, Philadelphia, 1974 May, 602 pages.

9. _____, "The Objectives of Database Management," in Information Systems: COINS-IV, Proceedings of Fourth International Symposium on Computer and Information Sciences (COINS-72), Miami Beach, Florida, 1972 December 14-16, edited by Julius T. Tou, New York: Plenum Publishing Corporation, 1974, pages 1-35.

10. _____, "Residual Dump Backup Strategy for Large Databases," submitted for publication to Transactions on Database Systems (TODS), a publication of the ACM.

11. Hershey, E. A. III, et all, "Problem Statement Language Version 3.0 Language Reference Manual," Ann Arbor: The University of Michigan, Department of Industrial and Operations Engineering, ISDOS Research Project, Working Paper No. 68, May 1975.

12. Severance, D. G., Some Generalized Modeling Structures for Use in Design of File Organizations, Ph.D. Dissertation, University of Michigan, 1972.

13. _____, "A Parametric Model of Alternative File Structures," Information Systems Journal, February 1975.