On automatic design of data organization

by WILLIAM A. McCUSKEY

Burroughs Corporation
Paoli, Pennsylvania

INTRODUCTION

A number of research efforts have contributed to the beginning of a methodology for the automatic design of large-scale information processing systems (IPS). See for instance Nunamaker.1 One facet of study in these efforts is the design of data organization.

Such a study was undertaken in the context of Project ISDOS,* now at the University of Michigan. The purpose of the study was to develop a model of the data organization design process and to create from this model a method for generating specifications of alternative data organizations. The first step of the study was to obtain a view of data organization uncomplicated by data usage. To this end the design of data organization (DOD) was divorced from the total IPS design process. A method for decision-making, which relates data organization to data usage and a measure of effectiveness, was to be a second phase of the study.

The purpose of this paper is to outline some initial results and implications of a set-theoretic approach to DOD which was developed for ISDOS. The assumed framework of the DOD process is described briefly. Within this framework concepts of data are defined in terms of sets. The DOD process can then be described in terms of set-theoretic operations. Finally some implications of the approach are given.

ORGANIZATION OF DATA—A FRAMEWORK

The term data is used here to mean the IPS representation of objects which are used as a basis for decision or calculation. The term data organization is used here to mean the set of relationships among data established by the problem definer or created by the system designer, as well as the representations of these relationships in the IPS. A design of data organization is a specification of these relationships, of their representations in the IPS, of the representation of the data in the IPS storage, and of the logical access and storage assignment processes which will operate on the data organization. The term process is used here to mean an operation or set of operations on data, whether that process is described by the problem definer or defined by the system designer. The system design procedure is itself a process and will be referred to as such.

The procedure for organizing data for an IPS may be thought of ideally in terms of four operations. First, a problem definer interprets a problem in his environment and defines a set of requirements which are as complete and concise as possible and which any solution of the problem, manual or automatic, must satisfy. A problem definition is complete if, in order to solve the problem, a system designer needs no further information from the problem definer. The problem definer defines the information processing problem in terms of sets of data, membership relationships among these sets of data, processes operating with the data, time and volume requirements on the processing, other constraints, and a measure of effectiveness for the solution. In order that the best possible design be produced, relative to the given measure of effectiveness, the problem definer should place as few restrictions as possible on the number of alternatives the system designer may consider.

Second, the system designer develops a specification of logically ordered structure for the data and the logical access processes which may be used to find any element in the structure. This structure will be called the logical organization of the data. An example is a binary tree, array, or any directed graph.

Third, the system designer specifies for these logically structured data the corresponding representations in the storage and the strategies for storage assignment. The resulting structure will be called physical organization of the data.

And fourth, the implementor of the system converts
the actual data from its present form to a form which meets the new specifications. Within this framework the approach was to view all concepts of data in terms of sets and then to define the design process, steps one through three above, in terms of set-theoretic operations on these sets. The set-theoretic details may be found in McCuskey. The following attempts a more narrative description.

CONCEPTS

The concepts of data organization described below must be viewed in the context of an ideal automated design system such as ISDOS. The problem statement, written in a formal problem statement language, is input to a system design program. This program specifies how processes should be organized into programs, how data should be structured logically and physically, and how the programs and data should be managed as a complete system. The system design is then automatically implemented.

The goal of this description of data concepts is to provide a framework within which to formulate a precise, simple algorithm. The algorithm must operate on a problem definition of data to produce a specification of IPS storage organization for the actual data.

Because of this goal the sets of data which the problem definer describes are viewed here as set-theoretic sets related by unordered cross-product relations. The algorithm must then establish what redundancies to keep, specify how the data should be ordered and then specify how this logical structure should be represented in storage.

The goal requires that the distinction between logical and physical data organization be defined precisely. The logical structure discussed below is the structure which is directly represented in storage. It incorporates some features, like redundancy specification, which are generally considered in the realm of "storage structure".

Problem description

From the problem definer's point-of-view an IPS operates on symbolic representations of conceptual or physical characteristics such as name, age, address, etc. The elementary object used to build such IPS representations will be called a symbol. The problem definer must specify an alphabet, the set of all symbols which are valid for the problem he is defining. One such alphabet is the EBCDIC character set.

Each occurrence of a characteristic, such as sex, amount, or title, may be thought of as an ordered pair of symbol sequences. The first component of the pair is the data name; the second component is the data value. The ordered pair will be called, generically, a data item. A data item will be denoted by its associated data name. An instance of a data item is a specific data name/data value pair. Thus (NAME, JONES) is an instance of the data item NAME. Common usage abbreviates this statement to "JONES is an instance of NAME". A data item has sometimes been referred to as an attribute, data element, or datum. In common high-level programming language usage the data value is the "data" stored while the data name is "data about data" which appears in the source program and enters a symbol table during compilation.

From the problem definer's point-of-view the IPS at any point in time will contain representations of many different occurrences of a given characteristic, say warehouse number. Disregarding how warehouse numbers are associated with other data in the IPS, one can describe a set of all distinguishable instances of a data item, named WHNO, existing in the IPS at the given time and having the same data name. Instances are distinguished by data value. The set WHNO contains no repeated occurrences of warehouse number. Such a collection will be called a data set at level 0 (henceforth, data set/0). The data set is referenced, like a member data item, by the data name common to all its elements. Context determines whether a data name refers to a data item or a data set.

Associated with a data set/0 is a number, called the cardinality of the set, which specifies the anticipated number of elements (unique data item instances) in the data set. Among data sets/0 exist cardinality relationships such as:

"at any given time approximately three unique instances of ITNO and exactly one unique instance of CITY will be associated with a unique instance of WHNO".

The anticipated cardinality and cardinality relationships among data sets, as defined here, are characteristics of the information processing problem and must be specified by the problem definer. The elements of a data set represent unique occurrences of an object, such as warehouse number, used in the problem as a basis for decision or calculation. What objects are used and how many unique occurrences of each must be represented in the IPS at any one time depend on how the problem definer interprets the problem.

These cardinality specifications eventually will help the system designer determine how much storage space

---

*A pair of right-angle brackets, ( ), will be used to indicate an ordered n-tuple (here a 2-tuple).*
may be required for any data organization design which he considers.

The concept of data set may be extended to higher levels. Data sets/O may be related by a named set membership association. The problem definer then describes processes in terms of operations on these associations as well as data items. For example, an updating process might be defined for INV (inventory) where INV is the data name associating the data items WHNO (warehouse number), ITNO (item number), and QTY (quantity). Nothing is said about ordering or logical structure on INV except the specification of set membership. In set-theoretic terms INV is a subset of the unordered cross-product of the three data sets/O.

INV names the data set/1 (data set at level one), the next level above its highest level component.

Such set membership relationships may be visualized in a non-directed graph as a tree in which data set names are associated with vertices and dotted arcs represent membership relationships. A graphic representation of the data set/1 INV is given in Figure 1.

A data set/n (data set at level n) (n≥1) may be thought of as a set of (distinguishable) ordered pairs. Each ordered pair is unique within the data set/n. The first component of the pair is the data name of this data set/n. The second component of the pair is an unordered m-tuple. Each component of the unordered m-tuple is an element (itself an ordered pair) of a data set/j (0≤j≤n−1). At least one component of the unordered m-tuple is from a data set/(n−1). The term data set component refers to a component of this unordered m-tuple. A data set component is referenced by its data name. Data set element refers to a unique member element of the data set/n. Component instance refers to the instance of a data set component in a given data set element. Figure 2 gives an instance of the data set

\[ \{\langle WHNO, 3 \rangle, \langle ITNO, 2 \rangle, \langle QTY, 2 \rangle\} \]

\[ \{\langle WHNO, 1 \rangle, \langle ITNO, 3 \rangle, \langle QTY, 7 \rangle\} \]

\[ \{\langle ITNO, 1 \rangle, \langle WHNO, 3 \rangle, \langle QTY, 1 \rangle\} \]

\[ \{\langle QTY, 2 \rangle, \langle WHNO, 2 \rangle, \langle ITNO, 2 \rangle\} \]

\[ \{\langle WHNO, 3 \rangle, \langle ITNO, 3 \rangle, \langle QTY, 7 \rangle\} \]

Figure 2—Instance of data set INV

INV.* The data set contains five data set elements. The data set components are WHNO, ITNO, and QTY. \(< WHNO, 3 >\) is a component instance of WHNO in three data set elements.

The concepts of cardinality and cardinality relationships, described above for data sets/O, are extended to data set/n. As with data sets/O cardinality specifications for data sets/n must be given by the problem definer.

According to the above definitions a data set/n element is unique within its data set. However, multiple instances of the same data set element may appear as component instances in a data set at a higher level. In Figure 2 (WHNO,3) is a unique data set element of WHNO but is a component instance in three data set elements of the data set INV. This multiplicity of occurrence of the same data set element is referred to here as redundancy. The amount of redundancy—the multiplicity of occurrence of the same data set element—in a data set/n is determined by cardinality relationships among the component data sets, by the cardinality of each component data set, and by the association of data sets defined by the problem definer.

The design of logical data organization may be viewed as a specification of the amount of redundancy and ordering of data set elements and component instances. For the design process to consider as many alternative logical structures as possible, as little structure—redundancy reduction and ordering—should be implied by the problem definition. The above view of data sets admits as much redundancy and as little ordering as the problem definition can allow and still be complete and concise.

**Logical data organization**

The first problem for the system design process is to take a specification of these data sets and, by performing

* A pair of braces \{\}, will denote an unordered m-tuple.
a sequence of operations, obtain a specification of logical data organization for the data set. Logical structure is provided for two reasons. First, the logical structure maintains in some form the membership associations established and referred to by the problem definer in his problem statement. Second, the logical structure provides a path or sets of paths to any element of the structure. Logical access processes, for example binary search, depend on such paths.

The logical structure of data may be visualized as a directed graph and will be called a data structure. Each vertex of the graph represents either a data item or a data structure. A data item or data structure represented by a vertex will be called a data structure component. An arc of the graph then represents a logical ordering relationship between two data structure components. Such a directed arc is an ordered pair of data structure components and will be called a connection. The logical connection described here is the connection which will be represented directly in storage by a fixed or variable distance in the address space of the storage. A data structure can then be viewed as a set of connections—that is, a set of ordering relations among its data structure components. A series of contiguous connections, called a logical access path, may be formed between two data structure components. Logical access processes use these paths to access components in the structure. A specification of data structure is a pattern which when applied to an instance of a data set yields an instance of the given data structure.

Consider the data set INV, revised and described by the non-directed graph given in Figure 3. INV has been redefined to be a data set/2. An instance of data set INV is given in Figure 4. To avoid the confusion of multiple brackets, the depiction of the data set instance in Figure 4 omits the bracket symbols of Figure 2 and factors the data names to the heading of the figure. Each circled single data value represents a data item instance. Data set membership relationships are represented by bounding lines in the heading. Each entire row represents a data set element of INV. Each column represents instances of the specified data item. While a horizontal ordering of data items has been introduced in the figure for ease of reading, it must be remembered that this ordering is only artificial: the data set components WHNO, CITY and STOCK actually form an unordered 3-tuple and ITNO and QTY form an unordered 2-tuple.

In the development of a data structure from the data set INV the system designer might specify the connections (WHNO,CITY), (CITY,STOCK) and (WHNO,STOCK). Similarly the connections (ITNO, QTY) and (QTY,ITNO) might be specified within the data structure developed from STOCK. The data structure components of the data structure developed from INV are WHNO and CITY, which are data items, and STOCK which is itself a data structure. The structure indicated so far is depicted in Figure 5a. For convenience, INV and STOCK will temporarily be the names given to the data structures developed from the data sets INV and STOCK.

Consider now the connection from WHNO to STOCK. This connection creates an ambiguous reference because there are two data structure components in STOCK. If a logical access path is to be constructed from, say, WHNO to the data structure

```
Figure 3—Graph representation of revised data set INV

Figure 4—Instance of revised data set INV
```
Figure 5—Development of a data structure for INV
STOCK, then through STOCK to QTY, the questions can be raised: At what point or points, ITNO and/or QTY, can the path enter the data structure STOCK and at what point or points can the path exit STOCK? What is the precise path from WHNO to QTY and out to another component?

It is important that this ambiguity be resolved. When the data structure is represented in storage, the programs which represent the logical access processes will operate on the storage representations of the logical access paths in order to access a given component representation. The ambiguity in the path from WHNO to QTY must be resolved if the program representing the logical access process is to reference the representation of QTY from the representation of WHNO.

The ambiguity is resolved here by designating one or more of the data structure components as entry components and one or more components as exit components of the given structure. A data structure component may be an entry component, an exit component, both, or neither. The set of entry and exit components will be called the boundary of the data structure. Since a data item may be considered an elementary data structure, the boundary of the data item is the data item itself. A data item is its own entry and exit component.

Thus, the connection to a data structure means a logical reference to its boundary; that is, to each of its entry components. A connection from a data structure means a logical reference from each of its exit components. This interpretation of connections makes no assumptions about the storage representation of the connection or of the boundary. When the boundary consists of multiple entry and exit components it will be called the boundary of the data structure. Since a data item may be considered an elementary data structure, the boundary of the data item is the data item itself. A data item is its own entry and exit component.

In the graph depiction of a data structure the boundary may be represented by broken arcs from the vertex representing the data structure to the vertices representing entry components; and by broken arcs from the vertices representing exit components to the vertex representing the data structure. The graph representation of the data structure then has a vertex denoted by the name of the data structure and a sub-graph representing the logical relationships among the data structure components. The arcs representing the boundary specify which subgraph is associated with which data structure vertex.

In the data structure INV, Figure 5b, WHNO has been designated as the entry component of the data structure INV and STOCK as the exit component. ITNO has been designated as both an entry and an exit component of STOCK. QTY occurs also as an exit component of STOCK. The boundary of INV is the component set consisting of WHNO and STOCK.

One piece is still missing from the picture of a data structure. An instance of a data set may contain multiple instances of its components. For example, for each WHNO there may be one CITY but many STOCKs. In the data set INV the same WHNO instance and CITY instance, for example (WHNO,3) and (CITY,A) in Figure 4, were associated redundantly with each of three different STOCK instances. The logical design of the data may specify that for each STOCK instance the corresponding WHNO and CITY instances will be repeated and maintained in the logical structure. In other words full redundancy of data will be maintained. If this design is implemented in storage, the same values of WHNO and CITY will be stored for each related instance of STOCK. On the other hand the logical design may specify that only one occurrence of the redundant WHNO and CITY will be maintained and with that occurrence of WHNO and CITY will be associated a new data structure each of whose components is one of the related instances of the data structure STOCK. The redundancy of WHNO and CITY has been reduced. This structure is depicted in Figure 5c. A structure of multiple instances of the same data structure is sometimes called a repeating group.

Within this newly created structure the instances of STOCK for a given WHNO/CITY combination are given some ordering, e.g., ascending numerical order by ITNO value. In addition, a boundary is specified for this new data structure; for instance, the entry component is the numerically first STOCK (f(ITNO)) and the exit component is the numerically last STOCK (l(ITNO)). In the graph these ordering and boundary specifications can be attached to the arcs to and from the STOCK vertex. The system designer may give a name to this new structure, as DS(1) in Figure 5c.

Assuming the given redundancy reduction, one can apply similar reasoning at the level of INV. According to the cardinality relationships given earlier, several instances of the data structure for INV will occur in an instance of the data structure stocking, one for each instance of WHNO. Each instance of INV will have the logical structure described in Figure 5c. This new structure has three components: WHNO, CITY, and DS(1). In each instance of INV, WHNO and CITY appear once and are connected to a data structure whose components are instances of STOCK. The data structure, DS(0), combining all instances of INV structure will be an ordering of instances and will have a specified boundary.

The complete specification of the data structure is given in Figure 5d. The design gives a specification of both ordering and redundancy and establishes the
network by which a data structure component may logically be accessed. Note that the membership relationships given by the problem definer have been maintained.

Associated with a data structure is one or more logical access processes which function to find or access the relative logical position of a component instance in an instance of the data structure. A logical access process uses contiguous connection instances to construct a path to the relative position of the desired component instance. For example, to find the data value of CITY for a given data value of WHNO, an access process for the above structure might create a path which starts at the entry component instance of the data structure DS(0) and courses through each instance of INV until it finds the given WHNO value which connects to the required CITY value. In each instance of INV the path leads from WHNO to DS(1) and exits via the DS(1) vertex. The access path does not course logically through the instances of STOCK.

From the point of view of the system designer a logical access process is like a problem-defined process. The system design process must incorporate it with the problem-defined processes into the program structure. Any data which the logical access processes need in order to function properly are designer-defined data sets which themselves must be organized logically and physically. At this point the system designer becomes a problem definer.

**Physical organization**

Physical organization of data means here the IPS storage representation of the given data structure. Two degrees of physical organization should be recognized: relative organization and absolute organization. Relative organization involves the manner in which the data structure components, connections, and boundary of a data structure will be represented in IPS storage. Such a specification involves questions of numbers of cells of memory required, sequential or non-sequential storage of components, header cells, etc., but not the actual locations of the data in storage. Absolute organization involves the actual locations of the components, connections, and boundary representations in storage. Absolute organization is specified by a set of storage assignment processes and must maintain the specified relative storage organization. In the following discussion major consideration is given only to the relative organization.

For the design of relative physical organization a relative storage area is defined here. This conceptual storage area is an ordered set of cells. Each cell is uniquely identified by an order-relative position and a length. Cells are non-overlapping. The length of a cell is measured in common units, such as bits.

Looked upon as elements of a set, and regardless of what they will represent, the cells in relative storage may be grouped together conceptually in many different ways. A storage node, or simply node, is defined to be a collection of elements each of which is a cell or a storage node. The cells in a storage node need not be contiguous relative to the ordering. In Figure 6 node A consists of three elements: two nodes and a cell. A single cell may be regarded as an elementary node.

For convenience in referencing a node relative to another node a measure of separation between the two nodes is defined. A separation between two nodes will be defined as the number of units of length, bits or words for instance, which, according to the given order-relative positions of the cells in relative storage, separates a reference cell in one node from a reference cell in the other node. The cell from which reference is made in the given node to another node will be called an exit cell of the given node. The cell to which reference is made in the given node from another node will be called an entry cell. The reference cells of a node will be called the node boundary. An elementary node or single cell is its own entry and exit cell.

Specification of entry and exit cells for a node is required for much the same reason that entry and exit components are specified for a data structure. If particular boundary cells were not specified, then reference to or from a multi-cell node would be ambiguous. In Figure 6 cell 0 has been designated the entry cell of node A (denoted by \(A_0\)). Cell 7 has been designated the exit cell (denoted by \(A_e\)).

It should be noted that the choice of the boundary
of a node in the conceptual relative storage is arbitrary. Multiple entry and exit cells may be designated in a node. Several different separations can occur between two nodes if one or both have multiple entry and exit cells. In Figures 6 and 7 only a single separation has been defined between nodes A and B. This separation is one cell—A→B. Figure 6 and following assume that all cells have the same length and that separation is measured in number of cells.

The system designer must specify first how to represent a data structure by a node in the conceptual relative storage and then specify a storage assignment process for mapping the node into absolute storage. The relative storage representation of the components, connections, and boundary of a data structure will be called here a storage structure.

A data structure component is represented in the relative storage by a node. If the data structure component is a data item, this node may be a set of cells which are contiguous relative to the ordering of relative storage. In Figure 8a the designer has decided to represent the data item WHNO by a two-cell node with the first cell being both the entry cell, WHNO₁, and the exit cell, WHNO₀. The system designer has decided that the first cell of the node will be the reference cell in any future separation specifications. The specific order-relative position of this node in relative storage is unimportant. Only the separation between it and other element nodes of the given storage structure is important. Figure 8a also represents data items CITY, ITNO, and QTY. The number of cells required to represent the data item is determined from the problem-defined size of the data value. This representation assumes that only the data value is to be represented in the node.

If the data structure component is a data structure itself then the storage structure may be defined recursively by representing the components, connections, and boundary of this component.

A connection in a data structure may be represented in one of two ways:

1. by a fixed separation between an exit cell of the node representing the first component and an entry cell of the node representing the second component;
2. by a variable separation which will not be given actual value until storage assignment takes place.

In either case the IPS will maintain a record of the separation. In common practice the fixed separation is recorded as a fixed address offset in program instructions. To handle variable separation the system designer may define another data item, a pointer, to be associated with the data structure component from which the connection is defined. The system designer also defines maintenance processes to update the pointer and perhaps other associated data sets, such as headers and trailers, to aid in maintenance. In Figure 8b the connection (WHNO, CITY) has been represented by a variable separation in the form of a pointer. A fixed

![Figure 8—Development of storage structure](fromcollectionoftheComputerHistoryMuseum)
separation of two cells has been specified to represent the connections (ITNO,QTY) and (QTY,ITNO).

The data structure boundary is represented by a node boundary developed in the following way. The designer may specify that the boundary of the node representing the whole data structure consists of the entry cells of nodes representing the data structure entry components and the exit cells of nodes representing the data structure exit components. Alternatively, the designer may incorporate additional cells in the node and define them to be the entry and exit cells of the node. He then defines a fixed or variable separation between these cells and the respective boundary cells of nodes representing the data structure entry and exit components. The additional cells and the boundary cells of nodes representing the data structure entry and exit components together represent the data structure boundary.

In terms of the graph representation of a data structure, for instance Figure 5d, the use of additional cells corresponds to treatment of the broken arc, say from DS(1) to STOCK, as a true connection; DS(1) is represented by the additional cells and the connection is represented by fixed or variable separations between these additional cells and the ENTRY and exit cells for the first and last instances of STOCK, respectively. If no additional cells are used, the broken arc is not viewed as a true connection and is therefore not represented explicitly in relative storage.

In Figure 8c the data structure boundary of STOCK has been represented by the entry cell of the entry component ITNO and the exit cells of the exit components ITNO and QTY.

Associated with a storage structure is one or more storage assignment processes. A storage assignment process will apply the relative storage structure design to an instance of the data structure and assign actual physical hardware memory locations. The storage assignment process is responsible for maintaining all “data about data” which is necessary for assignment of positions and all positional information which is necessary for use by the implemented logical access processes. The anticipated number of length units, e.g., cells, required by a node to represent a data structure instance may be developed from the size of data item nodes, the cardinality relationships given by the problem definer, the amount of redundancy defined by the system designer, and the requirements of pointers and related additions. See McCuskey for details.

A storage assignment process, like logical access processes, must be incorporated with the problem-defined processes to create program structure, whether at the operating system level or elsewhere. Any “data about stored data” which the storage assignment process requires is, from the point of view of the system designer, just like problem-defined data—data sets which must be given logical and physical structure.

**DESIGN PROCESS**

The goal of the concept descriptions above is to provide a framework within which to formulate an algorithm which, given a specification of problem-defined data, would specify how the actual data will be stored and accessed in the IPS. Figure 9 gives a very general flow chart of a design process for data organization.

In the design process the data structure designer accepts a specification of data sets and generates a specification of data structure (components, connections, and boundaries) and of logical access processes. While generating a specification of data structure, the designer acts itself as a problem definer; the problem is logical access to components of the data structure. The
definition of logical access processes must be input to the process structure design in order to be incorporated in the program specification. The structural information must be specified in terms of data sets and then input to the design algorithm.

The storage structure designer accepts the specifications generated by the data structure designer and produces a specification of storage structure (relative storage representation of data structure components, connections, and boundaries) as well as the storage assignment processes which will map the storage structure into absolute storage. Like the data structure designer, the storage structure designer is a problem definer; the problem is storage assignment. The storage assignment processes and information required by those processes must be defined and run through the design algorithm.

The process structure designer organizes the original problem-defined processes, the logical access processes, and the storage assignment processes and generates program specifications. How the logical access processes are represented in programs depends on how the storage structure and storage assignment processes have been designed. How the storage assignment processes are represented in programs depends on the characteristics of the absolute storage of the IPS.

In the context of this general picture of the design process only the specification of data structure and storage structure is considered below. An initial attempt at a method of generating alternative designs is described. The purpose of this attempt was to gain an understanding of what decision points are involved. No decision-maker has been developed. How a decision should be made at each point depends on the relation between the designed structure, the processes operating on it, and the performance criterion. As yet this relation is not understood.

Consider the specification of data structure for set INV (Figure 3). Suppose first that the given redundancy is to be retained. Then a general, recursive data structure design procedure might be:

**Process D**

1. For each component of the given set, if the component is not a data item then apply Process D recursively.
2. Define connections among the components of the given set.
3. Define a boundary from among the given components.

The process assumes all instances of a component are structured alike. A component may be a data set component or, in a repeating group, a data structure instance. The result of an application of Process D to INV, yielding a structure similar to that in Figure 5d, is given in Figure 10. Note that redundancy of WHNO and CITY will be maintained here while in Figure 5d it is not retained.

Suppose now that Process D has not been applied to INV. Suppose one wishes only to reduce redundancy. Reduction of redundancy may be accomplished in the following way:

**Process R**

1. Partition the original set according to the data values of instances of one or more components. A partition element is a subset of the original set. In a partition element each criterion component maintains multiple instances of the same data value.
2. Replace the partition element by a new element in the following way:
   a. one instance of each criterion component replaces the multiple instances;
   b. the remainder of the original subset is grouped by itself as a separate element.

The replacement operation will be called here truncation. The remainder will be called the truncated set. Figure 11 develops from Figure 4 a partition and truncation of INV according to the values of WHNO. The deleted component instances are blacked out. As in Figure 4 rows represent (unordered) data set elements and columns represent (unordered) data set components.
In Process R step one establishes which redundancies may be reduced in step two. The partition in Figure 10 establishes the redundancy of WHNO by definition; redundancy of CITY is established because the problem definer specified only one CITY instance per WHNO instance. The truncation operation performs the actual redundancy reduction. Neither, one, or both of WHNO and CITY may have redundancy reduced. In Figure 11 both were chosen.

These operations may be extended. A sequence of partitions leads to several levels of redundancy reduction. The sequence of partitions establishes a set of candidates for redundancy reduction at each level. The candidates are the criterion components established at that level or above and other components which are in one-to-one correspondence to criterion components. Starting at the deepest level and working upward, the design process can decide which candidates to choose for redundancy reduction. For a given candidate to have its redundancy reduced to a minimum its redundancy must be reduced at each level from the deepest up to the level at which it originally entered the set of candidates. If its redundancy is not reduced at the deepest level then its full redundancy is maintained. Partial redundancy for a component is established by not selecting the component for reduction at some level in between. Once the component fails to be chosen it drops out of the set of candidates; its redundancy is fixed.

This expanded redundancy reduction scheme at each level factors out selected components to the next higher level and leads to a hierarchical form of organization. The scheme may be combined with Process D above to form Process DS:

**Process DS**

1. Define \( n \)-level partition.
2. For level \( n, n-1, \ldots, 0 \):
   a. Define a truncation at this level.
   b. In the truncated set:
      i. apply Process D with data set components and, possibly, truncated sets as components.
      ii. apply Process D with truncated set elements as components.

Operation 2.b.i specifies the structure of an element of a repeating group or data set. Operation 2.b.ii amounts to specifying the structure of that repeating group. Once a component or truncated set has been structured it is a data structure component.

Figure 5d shows the pattern resulting from one application of Process DS to INV. Figure 12 shows the results of applying this pattern to the instance of INV given in Figure 4.

Consider next the specification of storage structure. A data structure INV has been specified in Figure 5d. The cardinality relationships among WHNO and STOCK are known. Following the definitions given earlier, one can specify a storage structure design process as follows:

**Process SS**

1. Represent each data structure component.
   a. If the component is a data item then specify the storage representation of its value and represent the data item by a set of contiguous cells large enough to contain the storage representation of the value.
   b. If the component is a data structure, apply Process SS recursively.
2. Represent each connection by either a fixed separation or variable separation.
3. Represent data structure boundary by a node boundary.

Figure 13 shows the result of an application of this process to the instance of the data structure given in Figure 12. The decisions shown in Figure 8 have been repeated here. Note that in the storage representation of data structure DS(1) the entry and exit cells of the node are additional cells; the broken arcs between DS(1) to STOCK in Figure 5d have been treated as true connections. These additional cells may be viewed as a header and a trailer for the data structure DS(1). Variable separation represents connections among instances of STOCK in DS(1) and among instances of INV in DS(0). One could question the value of this
of the data design problem. The model of data organization and the initial design procedures, as described above, are based on an ideal. The ideal is that a problem definer can provide a complete, concise and consistent problem statement which is completely sufficient for a system design process to generate specifications of an IPS to solve the information processing problem defined in the problem statement. One goal of Project ISDOS is to develop a problem statement language to facilitate such problem definition. Another goal is to define a design procedure to generate specifications. The following comments and implications must be viewed relative to this ideal.

First consider data sets. The set-theoretic approach was stimulated by Information Algebra described in CODASYL. Many of the ideas developed there find similar concepts in the present formulation. For instance, property and value correspond to data name and data value. Property space corresponds somewhat

**CONCLUSION**

**Comments and implications**

The intent of the research on data organization described above was to build a model of the data organization process and to develop a method for generating specifications of alternative data organizations. The set-theoretic approach was helpful in abstracting and precisely defining the component pieces

**Figure 12—Instance of data structure**

**Figure 13—Instance of storage structure**
to data set, except that property space is a set of ordered n-tuples. However, the goal of Information Algebra was a non-procedural language. The concepts of data in themselves do not lead to a solution of the problem of automatic design. The abstract formulation was also influenced by Young and Kent\(^6\) who specify information sets and relationships which are the counterparts of data sets and cardinality relationships.

Throughout the current paper, data sets were assumed to be problem-defined, or in special cases system-defined. In order to obtain a good system design, the designer may wish to redefine and coalesce some problem-defined data sets before proceeding to logical and physical organization. Such redefinition does not alter the logical and physical structure design processes just described. A pre-processor—data set redesigner—is introduced between the problem statement and the data structure designer.

Next consider data structures. Several levels of meaning are commonly attached to the term “data structure”. At the source language level, e.g., in ALGOL, the programmer considers an ARRAY to be a data structure and thinks in terms of, say, dimensions. However, the structure represented in storage may have array elements in row major order. The programmer may never see evidence of this actual logical structure (row major order) but the IPS operates on the directly represented structure and must be designed to handle it. The term data structure, used in this paper, refers to the latter structure, the logical structure which is actually and directly represented in the IPS storage. In order to consider all possible alternatives, including structures which perhaps have no common names, the design process must operate in terms of the structure actually represented in storage. The design procedure for data structures involved partitioning, truncation and ordering. The given procedure developed a generally hierarchical logical organization. The implications of applying multiple partition sequences and truncation to the same data set must be studied. Multiple partitioning would be required to develop more complicated, IDS-type structures involving multiple paths through the same data structure components.

The idea of generating logical access processes for a given data structure has interesting implications. In current practice data sets are structured to take advantage of the known, good access processes. For instance, data is often organized to take advantage of binary search or indexed sequential access. Often the data and its usage may not be suited to the known access processes. Perhaps what is needed is a procedure to generate logical access processes to fit the given data structure. The appropriate access process would be chosen dynamically at run-time according to the current status of the data.

Consider next storage structures. The distinction between storage structure and “data structure” has been made before by others. See for instance D'Imperio\(^6\) and Knuth.\(^7\) However, their distinctions are apparently not specific enough to suggest the automatic design generator desired here.

The idea of generating storage assignment processes to fit a given storage structure has implications similar to those of logical access process generation. Instead of designing a storage structure to fit the results of a given storage assignment process, perhaps we should have a generator to provide assignment processes tailored to the given storage structure. Within the context of automatic design and implementation of IPS these alternatives are worth considering.

Finally, the utility of design procedures which generate alternatives is minimal until a decision-making process is developed to select among the alternative designs. The decision-making process relates the data organization to data usage, i.e., to the process and program structure design, and will require much more research.

REFERENCES

1 J F NUNAMAKER JR
   On the design and optimization of information processing systems
   PhD Dissertation Case Western Reserve University 1969
2 W A McLUSKEY
   Toward the automatic design of data organization for large-scale information processing systems
   PhD Dissertation Case Western Reserve University 1969
3 Ibid
4 OODASYL DEVELOPMENT COMMITTEE (LANGUAGE STRUCTURE GROUP)
   An information algebra
   Comm ACM Vol 5 No 4 pp 190-204 April 1962
5 J W YOUNG JR H K KENT
   Abstract formulation of data processing problems
   J Ind Engr Vol 9 No 6 pp 471-479 November-December 1958
6 M D'IMPERIO
   Data structures and their representation in storage
7 D E KNUTH
   The art of computer programming
   Vol I Addison-Wesley Reading Mass 1968