NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

DATABASE FACILITIES FOR ENGINEERING DESIGN

by

Charles Eastman

DRC-15-08-81

September 1981

Database Facilities for Engineering Design

By Charles Eastman[1]
Institute of Building Sciences and
Design Research Center
Carnegie-Mellon University

1 Introduction

1.1 Evolving Needs for Databases in Engineering Design

Handling large amounts of data is an integral part of modern engineering practice. The traditional work area of engineers is filled with drawings, books of specifications, handbooks of standard tables and product catalogs. It is not surprising, then, that data files have been part of most large scale computer use in engineering.

An early need was to store constants such as material properties or other standards for analysis programs. This data was sequentially read into main memory at the beginning of an application, but not modified during use. It was stored separately because of its use by multiple applications of a program, but still occasionally extended or modified. When changes were made to the data, it usually applied to all projects using the program [Roos, 1967]. A later development was the use of files for pre- and post-processing. Examples are textual or graphic programs that expand simple input to the proper format for a single application, or that takes output from an application and reformats it into a chart or graphic review. Files provide communication between processes. This use is being applied frequently today [Butlin, 1974; Kamel and Shanta, 1974; Fredriksson et al, 1981].

An extension of the concept of the pre-processor has also been proposed by several groups. A common project description is expanded into input required for multiple application programs. This method of data generation guarantees the data to be consistent over all input descriptions. A change made to design requires changing the input description and re-expanding the data as needed. Such an approach requires a separate expansion module for each application. It does resolve the issue of having consistent input.

[1] This work was sponsored by a grant from the National Science Foundation, grant number MCS-78-22328.

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

1.2 Integrated Design Databases

The above file operations provide interfaces with particular engineering programs. But as the number of programs has grown, the generation and management of input data they use have also become a problem. Each file of input data has a separate structure; data preparation and the writing of interfaces between file formats has become a major endeavor of many engineering groups.

The concept of data capture suggests that various input processes could all feed a common data repository, from which is extracted the particular data needed for an application. Data common to a number of applications need only be entered once, and used as input for all applications, with reformatting as needed on input and output. Such a common repository has come to be called an **integrated design database** or just **design database.** Such an integrated approach can greatly reduce the cost of writing a pre-processor for yet another application or for implementing yet another set of file transfer and mapping routines. In the longer context, a design database's benefits include:

1. supporting new forms of integration, such as the automatic generation of production data, such as drawings and numerical control machine tapes, or for generating summary reports to a division - or company wide management information system.

2. by keeping all design data in a common machine readable form, the possibility exists to check consistency of data, reducing or eliminating conflicts.

3. by keeping data already prepared for use, integrated design databases support further automation.

4. eventually, a design database becomes an environment in which designers work directly. It supports generation as well as analysis, with the possibility of greatly improved productivity.

Efforts to develop design databases are underway in many engineering fields, including: electrical engineering [Matelan and Smith, 1975; Melanson and Spurlin, 1975; Kim and Siewiorek, 1980], process plant design [Tsubaki, 1975; Leesley, Buchmann and Mulraney, 1978], shipbuilding [Bandurski and Jefferson, 1975], manufacturing [Fisher, 1979; Foisseau et al, 1977], aerospace manufacture [Miller et al, 1979] and building [Eastman, 1976; Hoskins, 1973; Bijl and Shawcross, 1975].

This paper reviews standard practices of database development for CAD and presents a step by step outline of graduated target capabilities for a design database. In describing the target capabilities needed for design databases, it uncovers some unique needs that distinguish them from business applications, resulting from:

1. the unique nature of design decisionmaking

2. the need to support multiple design professions

3. and the special problems of concurrent design users.

The current database capabilities and research issues associated with these needs are reviewed. This article is meant to serve as a tutorial for those considering development of a design database and as a guide to the literature. It also attempts to define certain hazardous areas that production oriented efforts should avoid.

2 The Traditional Approach to Database Support[2]

The term *database* refers to an organization of data, too large to be held in main memory of a computer at once, upon which effective operation requires some form of structuring. The structuring of data distinguishes a database from a file system; file systems are sequentially ordered. A database provides the means to organize data, means to access it according to pre-defined access structures, and brings parts into main memory for use by a variety of applications. It also provides common support facilities for such tasks, including recovery from errors or crashes, report generators for making reports from database contents, interactive query languages and statistical analysis packages. The size and structuring capabilities of a database system distinguishes it also from virtual memory.

2.1 Database Organization

The basic organization of a database consists of three components: a data definition language (DDL); a data manipulation language (DML), and a set of utilities.

The *DDL* is the means to define the data organization. The unit of data is a record, composed of a set of fields. The fields are the primitive information elements, defined according to the data type they hold. Most often, records have a fixed composition, so that the location of particular fields within the record can be assumed by the system. Some implementations allpw variable numbers of a particular field. The format of a record is defined in the record *type,* in a manner similar to record types in programming languages such as PL1 and Pascal (roughly). A variable and usually very large number of record *instances* are expected to exist over the database lifetime.

---

[2] Good references on database design and implementation are [Date, 1975], [Martin, 1975] and [Wiederhoid, 1977]. The major journals for databases are the *ACM Transactions on Database Systems* and the *IEEE Transactions on Software Engineering.* Important surveys are sometimes presented in the *ACM Computing Surveys.*

For organizing data, it is useful to have one record refer to several others, such as would be needed for an entity to refer to each other entity connected to it. Similarly, it is useful for several records to refer to a common one, as when several entities are made up of common material, with a single set of properties. Thus one-to-many relationships are desired between records as well as many-to-one relationships. The structuring mechanism of different database systems allow either a *hierarchical* structure, i.e. a tree without circuits, or *network* structures which allow circuits. The main difference is that hierarchical structures require copying of shared records, while network structures do not. Typically, the concept of a *set* is used to define the many in a one-to-many or many-to-one relation. A set in database terms is an unordered collection of references to records [Taylor and Frank, 1976].

In addition to the relations between records, most database systems provide means for direct accessing of records with certain data. For example, it might be desirable to access all elements provided by a particular contractor or all the elements using a particular material.

Alternative accessing mechanisms consist of path traversals within a data structure and/or special directories. All require pre-organization of the data and are justified when the number of accesses of items is high over the data's lifetime. Some special accessing schemes include:

1. ordered directories, known as index sequential
2. pre-sorted binary trees
3. directories built of all records having common attributes, called *inverted lists*
4. *hash coding* schemes that map a unique attribute identifier into a address that holds records with this value.

All special accessing schemes provide access according to desired contents. The desired value for a specified variable is called the *access key.* For a detailed review of accessing mechanisms, see [Wiederhold, 1977, Chap. 3-4; Martin, 1975, Chap. 19-28].

Records and the structure between them define a complex data structure scattered between main memory and disc files. This record structure is called a database *schema.* The schema is sometimes represented explicitly and stored on a separate file so as to be accessible by different database support facilities. This file is called a *data dictionary.*

The DDL is managed by a *database administrator* (DA). The DA controls record definitions and their relating structure and is responsible for compiling them into the system.

The data manipulation language or DML, on the other hand, provides the procedural accesses for retrieving records from the structure defined in the DDL. The access of data for an application involves defining what data is needed, allocating space to hold it, fetching it into main memory and running the application against it, then possibly storing the results, additions or changes back on to the data structure. The subset of the schema used for an application is called a *subschema, view* or *externalschema.*

Other *supportfacilities* include:

1. *a query language* which provides means for defining accesses interactively by an end user, esp. for information retrieval systems. Query languages are not always provided; when they are, they are often imbedded in the DML [Reisner, Boyce and Chamberlin, 1975; Zloof, 1975].

2. *back-up and recovery* are required because of the great investment in the contents of a database. Records of database actions, called a transaction file or a file of updated records, called a checkpoint file, and other techniques are commonly used [Bjork, 1975; Wiederhold, 1977, Chap. 11].

3. *access control and security* allow management by the DA of who can Read or, more importantly, Write into particular data objects. Access control is usually defined for records and occasionally for subschemas. They are almost never defined at the field level within records [Patrick, 1974; Wiederhold, 1977, Chapter 12]. Access control is often an influence on the record structure within the database schema.

4. report generators, statistical analysis packages, and plotting report generators are sometimes provided in particular database systems, depending upon their intended use.

The sophistication of these facilities vary greatly, as do their cost. The quality of support and maintenance for such a facility is a critical issue if an organization is going to build its own commitments and investment "on top of" a database system.

2.2 Alternative Data Models

The complexity of database organization has led to alternative conceptual models for describing it to different classes of user. These models are largely independent of the database's physical implementation and define only functional capabilities. The principal data models are:

1. the *CODASYL* Report description is a proposed standard for databases developed by a committee that grew out of the Cobol business computing language design effort. This standard has been revised and extended several times. Most database vendors relate their capabilties to those specified in the CODASYL Report [CODASYL, 1971].

2. *Relational structure* is another major data model. It is composed of tables, called

relations, in which columns are fields, called *domains,* and rows correspond to records and are called *tuples.* Indirect references are resolved by formal operators on relations. The Relational data model has gained strong advocates because it allows formal analysis of database structure and operations [Codd, 1970,1971].

3. ANSI/X3/SPARC is a second committee that is deriving standards. A prominent result of their efforts has been to elaborate the notion of schema, with three different ones. *External schema* is the definition of part of the database needed for an application, defined in a manner independent of its physical implementation. *Internal schema* defines the physical implementation of a schema, esp. accessing mechanisms. *Conceptual Schemas* are the logical structures that may be used to describe data organization to the user [ANSI, 1973].

The terms used in the three models is shown in Figure 1.

CONCEPTS:		DATA MODELS:	
	CODASYL	ANSI-SPARC	Relational
field	data item	data item	domain
entity type	record type	record type	relation
instance	record instance	record instance	tuple
relationship	set type	set type	comparable domains
DB admin view	schema	internal schema	data model
user view	subschema	external schema	data submodel

Figure 1: Comparative terminology of different data models.

2.3 Example Development of Simple Design Database

The above facilities can be used for design databases. An example is developed in this section that will be used later to elucidate a variety of functional and implementation issues. While databases are large and complex in their variety, we will keep the example small, to allow focussing on esssential issues.

The initial example is of a piping system, a common subsytem in many engineering projects. The set of application programs are those supporting piping design for such projects as building and process plant design. See Figure 2. They include:

1. piping system topology design; it supports interactive definition of end node and source node requirements and their locations.

2. a pipe sizing design program that defines pipe parameters(diameter, schedule.material) from flow requirements, given the material properties of what is being transported.

3. a detail fitting and shop drawing program that identifies all fittings that will be required and also computes the finished length of pipe elements, for shop drawings.

4. a cost analysis program that estimates the cost of piping and all fittings.

5. a program for the production of numerical control tapes, for automatic pipe cutting, bending, welding, threading, etc.

Input of all but the first program is the output of previous programs. Each program also generates an evaluation report; most also create a new, more detailed definition of the piping system. The example includes two cost evaluation programs, at two different stages of design development. It is a common need to have multiple similar analyses of the same system at different stages of its definition.

Each application program has a mapping procedure that reads from the stored data the needed input into a common workarea, all at once (most common) or incrementally, as needed. On a single user system either strategy can be used and accessing the database can be based on optimal resonse.

The diagram shows four different databases; these could be separate, or they could be subschemas of the same database, with an extended set of fields for each pipe and fitting so that they cover the range of information needed by each application. If an application can access multiple databases during its execution, then the distinction between having one or several databases is insignificant.

The common databases, denoted $DB^{P}_{1,3}$, hold design project* information characterizing a particular design effort. They are called the *project dependent database.* Notice, in contrast, the cost, material properties and standard fittings files: those are used in their same form in many projects and are thus called *project independent databases.* Most applications need both project dependent and project independent databases. The project independent databases tend to be file oriented, similar to the earlier uses described at the beginning of the paper. They are usually accessed sequentially and have fixed size requirements. Thus a complex structure is not needed.

The database shown in Figure 2 supports a fixed development sequence, because of the sequential ordering of applications; it allows iterative design cycles, as the result of any application can be

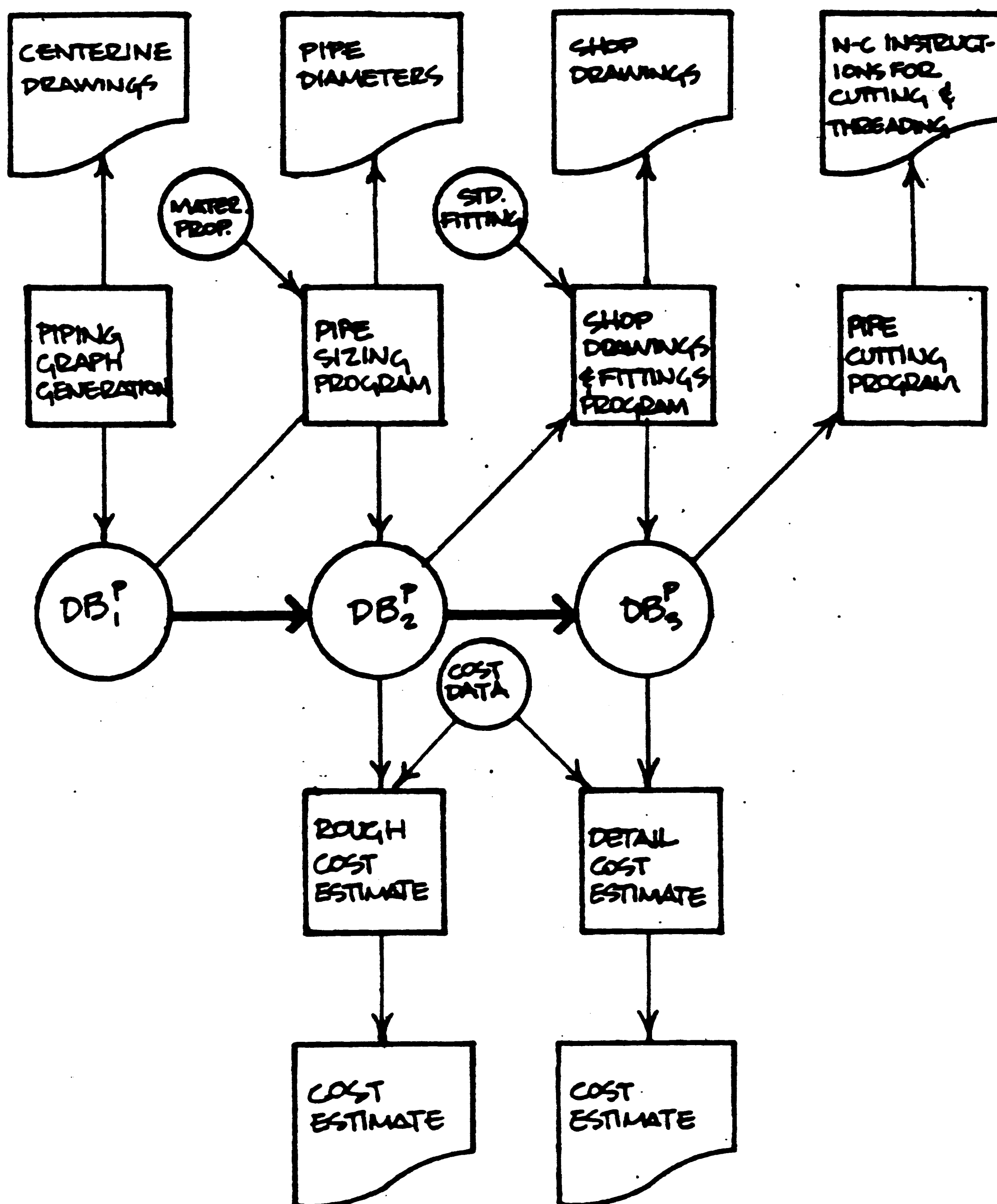

Figure 2: Example sequence of application programs in one functional area and the input and output files they require.

thrown away and regenerated until one is found that is satisfactory. A user also can always discard several previous stages and iterate.

The linking of data between several applications, as described here, certainly supports multiple applications and thus is properly a design database. The level of complexity is managable within

current software development practices. Thus this example and ones similar to it can be realized in many design areas. See for example [Beiland and Friedenson, 1980]. The structure described, however, is responsive to only one subsystem of an engineering design project and to a single user at a time; it also allows only a single, fixed development sequence. Relaxing these limitiations will be taken up in the following sections.

It should be noticed that the applications define a structure of *dependencies* in which existing data is used to compute new data. However, if the new data is not realistic, such that no new data can be defined that is consistent with physical laws or design objectives, then the existing data is not acceptable and its computation must be iterated, so as to generate different results. The dependencies among applications will become significant as capabilities are added.

3 Development of Multi-Disciplinary Design Databases

The above example ignores the many other functions that any engineering product must finally respond to. Some engineering projects largely respond to only one function, but even they always include a wider range of considerations. Maintainability and constructability almost always impose distinct issues beyond the primary function. Other fields inherently deal with the integration of many functions; building design is the most obvious example.

To elaborate the example, structural design capabilities will be added to the piping example. The choice of structural design is not important. Rather it was selected as an example of any other related function. It shall serve here to allow focussing on the generic problems of databases supporting multi-disciplinary design.

A typical structural design development sequence might be as shown in Figure 3. The development sequence is: generation of the structural network, determination of section properties, member selection and cost estimation, and joint design, followed by production of shop drawings. Again, the $DB^{s}_{1\text{-}4}$ could be merged into a single database that is their union. The loads that must be supplied to the member sizing application come from the rest of the project design, including piping. The input in the figure called Design Properties includes the material properties and joint detailing assumptions. Steel sections available from industry are provided in a catalog along with their properties. We can see that this information is used in 3 different stages of structural design and is thus an obvious candidate for storing in a common, centralized project independent form.

The set of structural design programs could be developed in the same manner as the piping

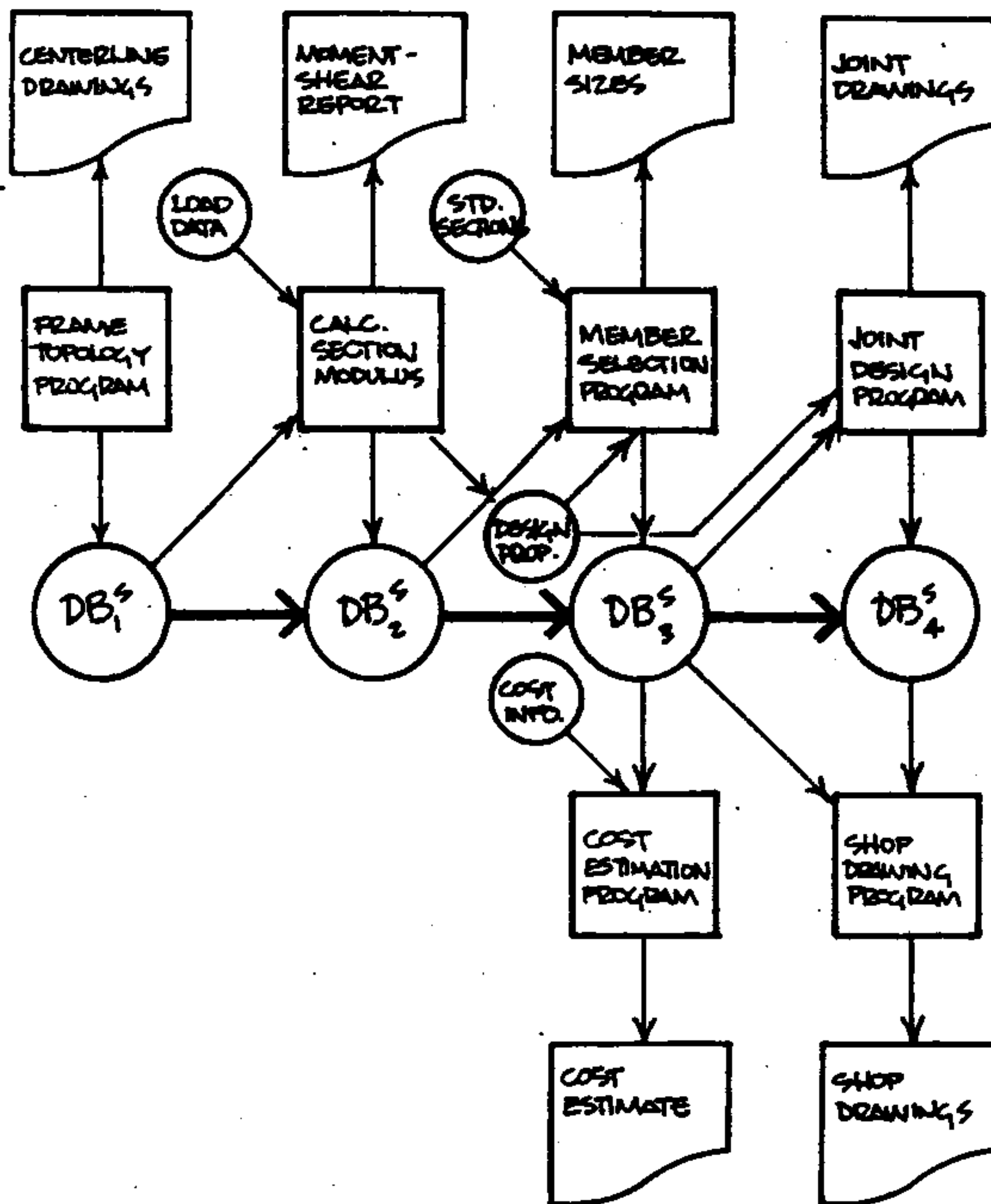

Figure 3: The sequence of applications supporting structural design and the corresponding databases involved.

system, using either separate databases or a combined one. The structural and piping databases also could be separate from each other. However, their definition in a common database would allow management of several forms of important interaction, including:

1. loads generated by the piping that must be picked up by the structure

2. spatial conflicts between piping and structural elements

The value of a design database is expressly for dealing with these sorts of subsystem interactions. By having the piping and structural data in a jointly accessible form, both interdependencies can be evaluated by applications that access the data describing both subsystems.

Spatial conflicts are most easily managed centrally and evaluated when entities are placed. The geometry of a shape can be represented" explicitly, as a set of bounded surfaces [Baer, Eastman and Henrion, 1979; Boyse, 1979] or in some approximated form. Means for handling spatial conflicts are described in [Eastman, Lividini and Stoker, 1975] and are based on methods for accessing entitites by their location within a project.

The load transfer of the piping to the structural system occurs after the pipes have been sized. For a given piping wall thickness, there will be a maximum span (under static conditions). By displaying both systems together, an engineer can easily pick locations for supporting the piping and transferring the loads to the structure. By evaluating these for their cost then iterating, the designer could hill-climb toward an efficient piping support design.

3.1 The Ordering **of Multi-functional Applications**

Up to this point, we have relied on a static and comprehensive means for diagramming a database system. Yet as new capabilities are added, the static representational schemes shown in Figures 2 and 3 quickly become too complex and don't incorporate needed information.

One useful alternative structure for representing operations on a design database is a *transaction graph.* Consider any one of the application programs. In order for it to execute correctly, a set of pre-conditions must hold in the database. These include the existence of valid input data and possibly adequate contextual data to allow identification of side conditions. In our example, the contextual data might be the identification of special structural loadaor the reservation of places where piping cannot be routed. The set of these pre-conditions can be denoted as a *database state,* which is realized by the successful completion of one or more previous applications. A database state is denoted in a transaction graph as a node and operations on the database as directed edges. The state required for an application to properly execute is denoted by the beginning node of the operation. The operations that realize the pre-condition state are the preceding edges to the beginning node. The state resulting from the successful execution of the operation is its end node. All the preceding edges incident to a node must, have their corresponding operations successfully executed for the state corresponding to the node to be realized. That is, the preceding edges

incident to a node are logically combined with an AND. When an application is iterated, the applications succeeding it are invalidated. Thus the succeeding edges must be traversed again [Eastman and Lafue, 1981; Fenves, 1975].

A transaction in a database is an operation within which some logical relations within a database are not valid, due to their modification. Before and after the transaction, the logical relations are assumed to hold [Eswaren et al, 1974]. In CAD, it has been shown that applications add to the pool of logical relations that are satisfied as well as maintain existing ones [Eastman and Lafue, 1981]. A transaction may be a large application program that completes a design task. Alternatively, it may be a single update of a variable and maintain only a trivial logical relation (such as an identity or summation). The scale of transaction is largely a matter of the frequency in which data from a workspace is entered as updates into the database system. All other things equal, the more frequent the update, the smaller will be the transaction. Initially, we will consider transactions to be the complete application programs. Thus there will be relatively few, large ones.

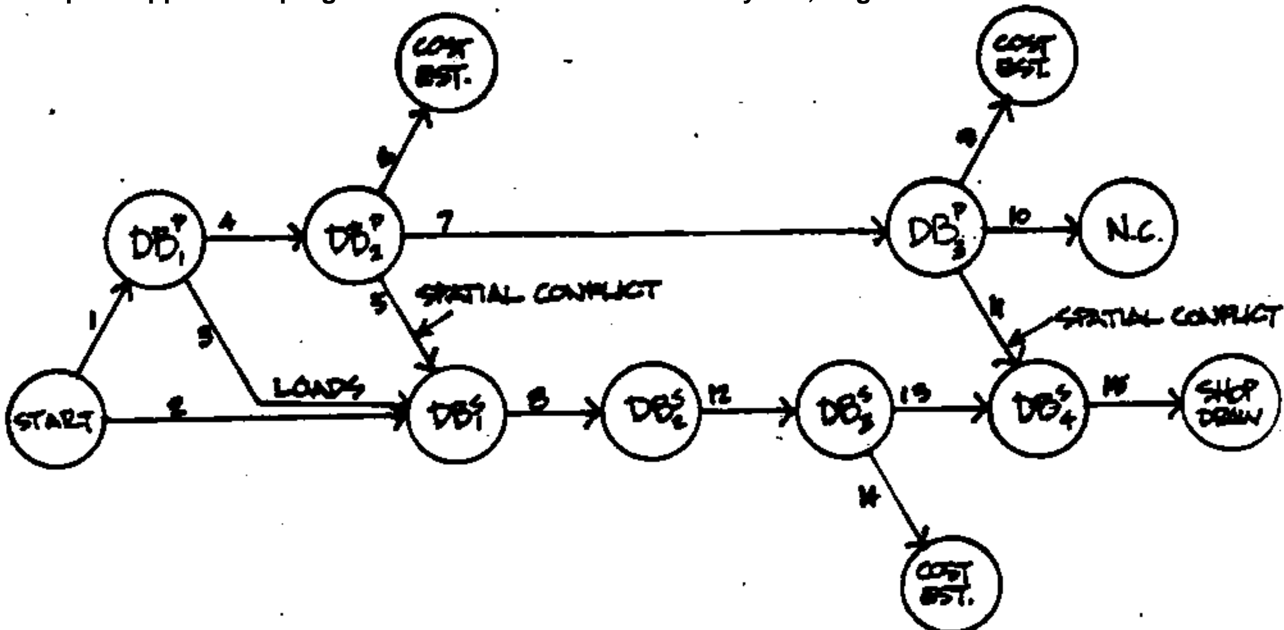

Figure 4: The directed graph shewing the merger or a piping and structural database.

In Figure 4, the two application programs are structured in their corresponding transaction graph. The application of the load definition program logically goes from DB^P_2 to DB^S_2, but the application of the spatial conflict program can go between a variety of database conditions. Here, we have limited to interaction to two places in the development sequence: between DB^S_1 and DB^P_1 and DB^S_4 and DB^P. While there is a strict precedent dependency for the load generation, spatial conflict has no precedent dependency, but a joint one.

Transaction graphs can be used to organize the integration *61* a wide range of applications and serves as a useful tool for organizing large design databases. They are useful in both defining how applications should be related within a CAD system, when they are being organized by the software developers, and in identifying what application options are available at each point to the end user. The organization and uses of transaction graphs are more fully developed in [Eastman and Lafue, 1981].

Notice that up to now, the transaction graph is long and thin, with only a few traversal paths within it The system defined thus far provides only a few sequences of design development. Notice too that it is assumed that an application resolves the conditions it is responsible for <u>everywhere they exist</u> within the database.

3.2 Alternative Design **Development Applications**

Alternative technologies exist, however, for realizing an engineering function. The design programs supporting the entry and definition of these technologies vary, though the resulting systems serve the same overall purpose. It is desirable to support the incorporation of multiple technologies in design databases. In the organization of a design database, these take the form of multiple parallel development sequences.

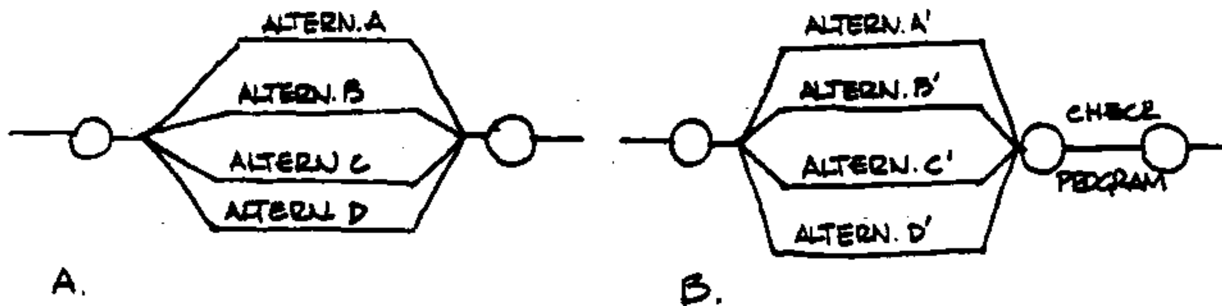

Figure 5: The transaction graph with several OR-applications.

For example, there can be multiple graph definition programs for structural system definition, as might be desired for different types of structural systems, such as rigid frames, bearing walls and space frames. These become parallel edges in the transaction graph, with an OR condition

associated across them. Complete execution of any one of these programs is satisfactory for the succeeding state. A set of OR applications can be denoted as shown in Figure 5a.

It is sometimes desirable to use more than one design development program, for different parts of the design, within a single design stage. For example, it may desirable to use first a program for defining and laying out bearing walls and then another for defining space frames, as might be appropriate for a building design project-

While this kind of mixing can be realized within the organization of a database, mixing application programs to satisfy a state condition makes defining the completion of the state difficult. The state conditions are not guaranteed by completion of any one program. Instead, the pre-conditions must be evaluated according to the data values themselves.

In many application areas, a separate checking program for evaluating a state condition can be readily defined. For example, it is straightforward to check if all nodes in a piping system are connected or all loads in a structural system are transferred to the frame. By allowing multiple programs to satisfy a pre-condition, the state checking program becomes a necessary separate application. An example transaction graph that includes such a checking program is shown in Figure 5b.

The checking program can be thought of as a pre-processor to the succeeding application that determines the existence of the.required database state. There is some advantage in maintaining the checking programs separately from their application, especially when there is also a choice of several succeeding applications. State checking programs are especially important for applications that incrementally extract database information. Applications that extract all the data they need at their initiation can usually check the adequacy of the data during the extraction process.

3.3 Automatic Management of Precedence Ordering

Up to this point, the sequential management of the execution of applications has been managed manually. But design databases easily can have such a range of applications that serious errors can arise due to running applications out of their expected order. The most serious problem is only partially complete data, usually added late and not fully integrated. Examples might be additional structural loads that are never incorporated into the structural analysis or additional end nodes to the piping system that are never connected. These may not be incorporated into analyses and may either never be caught or else they are dealt with much later and re-analysis and re-design of some portions of the project will be required.

Managing the traversal of the transaction graph can be automated. A direct approach is *management by transactions*. Each application, including the state checking ones, has an associated global flag. For the purposes described here, the flag may be boolean. It is initially set to FALSE and changed to TRUE when the associated application is executed. Each application checks that its preceding tasks have been completed, eg. the flags of its data are set to TRUE, before it is initiated.

When an application is iterated, its flag is found to be TRUE rather than FALSE. In this case, it switches to FALSE all transactions that succeed it. In order to set these succeeding flags, it is necessary for an application to know the succeeding applications that use its integrity conditions. This is difficult to embed into the applications themselves. Bookkeeping is most easily managed by a procedure that centralizes knowledge of the complete application sequence [Eastman and Fenves, 1978].

One example of the data that could be used in such a central control program is shown in Figure 6. It shows an edge-by-edge matrix of the transaction graph shown in Figure 4 and a set of boolean flags, one for each transaction. When a transaction is completed, including a state checking transaction, its boolean flag is checked. FALSE means that it has not been executed previously and should be set to TRUE. If it is already TRUE, then it is left that way, but the matrix is used to identify all the successor edges to the one updated and they are set to FALSE. The edge by edge matrix allows traversals of the transaction graph in either direction, as is needed for checking if needed precedences are resolved and in propagating the effects of an update.

The example design database at this stage, with centralized transaction checking, allows a wide variety of development sequences, with effective automated control of the sequence of application execution.

4 Design Databases Supporting Concurrent Users

Most design projects are of such magnitude and under such time constraints that parallel development by multiple designers is an imperative. In order to support multiple users, the system must manage the multiple requests for data, resolving conflicts that occur between simultaneous Reads and Writes and when one user updates information that is being used in decisions by several others.

The standard method for controlling concurrent use within operating systems is a system of locks, so that only one user may have Read-Write access to adata item at a time. The data items used for

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	FLAG
1			+1	+1												
2								+1								
3	-1							+1								
4	-1				+1	+1	+1									
5				-1				+1								
6				-1												
7				-1					+1	+1	+1					
8		-1	-1		-1							+1				
9							-1									
10							-1									
11							-1								+1	
12								-1					+1	+1		
13												-1			+1	
14												-1				
15											-1		-1			

Figure 6: One set of data that can be used to manage the transaction graph of a design database.

locking are usually files or records [Eswaren et al, 1974]. This mechanism gives adequate protection if the information element for which locks is allowed is small, such as a record, but not a file (it is not helpful if the unit for locking is the whole database). Read-Write locks respond to access to records but do not deal with the management of dependencies. One user may be using certain data created by an earlier transaction. If this earlier transaction is iterated during the first user's processing or any of the transaction's predecessors are revised, then the user's work may be invalidated.

4.1 Transaction Management of Concurrent Users

The precedent ordering of applications, as controlled in transaction management, allows concurrent use in a manner that deals with dependencies. Any application that has its precedent transactions complete may run, in parallel with others that have the same set of completed predecessor states. No transaction may be executed whose beginning node state is an end node state of another concurrent transaction or is a successor of a concurrent transaction. For example, in Figure 4, the application to size structural member cannot be intiated until the program that rough sizes the piping is completed. Notice that, with these rules, shared data is only Read. If data has alread been Read for an application, however, then that application may be invalidated by any application that accesses the same data in Write mode. Similarly, if the first access is in Write mode, then no other Read-write or Read accesses can be allowed. These rules apply when data is first Read, normally at the beginning of a transaction.

Alternatively, applications can make any access, but checks are made at update time. If any Writes are done to previously Read data, then all the Read transactions are invalidated. This however, is more difficult than checking at access time, as a record must be kept of all intervening Reads. Because it is not obvious which Read may eventually Write back to a data item, the records of all Reads of the extant transactions must be maintained. Iteration of an application makes results of all succeeding applications invalid.

Concurrent use requires structuring of data on records so that each stage of application creates new records, rather than writing back onto empty fields on existing records. This guarantees that there will not be conflicts of accessing records in Read/Write mode with data defined earlier. The separation of design data into its records corresponding to each design stage suggests an ordering of data describing a system in a hierarchy of detail. This organization will be developed more fully in Section 7.

Spatial conflicts or other interaction checking is applied after each merger, with iterations on the transaction until they are eliminated.

Logically separating concurrent tasks at a finer grain, such as at the record-by-record level of interaction, involves both gross checks at the transaction precedent ordering level and, within dependent transactions, record-by-record dependency checking. Separation of logical dependencies at the record-by-record level requires keeping the dependencies resulting from individual computations, usually by storing the parameter lists of the checking procedures. This level of management has only begun to be studied for possible efficient implementations and as to its effectiveness [Lafue, 1980]. See Section 5.

Another form of separation in concurrent processing is by spatial isolation. The assumption is that by defining the spatial boundaries of updates, dependencies can be localized. Such spatial partitioning is common in manual design and should be available as a tactic in CAD. Required is a means to define the subschema for a transaction by its spatial locality, eg. the structural or piping graphs will be modified in only certain locations in the project. Currently, the only way to do this is by having the schema organization incorporate this type of partitioning. This has the effect of having all such partitions being organized integrally into the database. An example is shown in the building design schema shown in Figure 7, where each story and each space of a building are distinct and organized so that the components within them can be accessed separately. Ideally, one would like to identify such partitions dynamically, without pre-organization. Concurrent use can be based on many forms of logical partitioning.

4.2 Communication Among Multiple Users

Concurrent designers have need for a range of communication. They need to point out special conditions they have created, the status of parts of the design they have been working on but have stopped, and the assumptions they made in certain decisions. These communication requirements are similar to those needed among software development teams.

These should include "talk mode", sending direct messages to another current user. It should include "mail", for leaving messages for other users that can be read later that tell of special conditions or questions needing resolution. A design database should support "attached messages" that are associated with particular data items, so that other users of data may learn of special assumptions or considerations. Special controls are needed for reading these. It is important to recognize that a design database is a communication system as well as a database system.

5 Localized Integrity Management

Most databases are passive stores of information. All processing capabilities are embedded in either their utilities or in applications. When data is updated, it is the application that must know what logical structures must be modified to keep the database logically correct. This includes embedding new data in accessing structures as well as making the required corollary modifications when one piece or a small set of data is changed.

5.1 Data Abstraction

If we consider the structures of a database as complex data types, then the operators on these structures also carry much meaning regarding the semantics of the types. The operators have embedded in them knowledge of the structure and how it is to be managed to have meaningful results. Software engineering emphasizes the need to define data structures and their operations jointly [Parnas, 1972; Dahl and Hoare, 1972]. This merging of data and operations is commonly referred to as *data abstraction.*

In most databases, of necessity, the operators on the data are in applications. This means that every application programmer must know all the bookkeeping required to update parts of the database. However, if the database has an active, procedural capability, then the schema definition can include the procedures for modifying the database and these can be used by all application programmers. There have been several calls for such capabilities within database systems [Weber, 1978; Lockemann et al, 1979; Lafue, 1979] and a few research database languages have been developed that incorporate them [Eastman and Thornton, 1978; Rowe and Shoens, 1979].

Data abstraction in database systems can be achieved with current production databases also, with some effort. It is possible to define a set of applications for the various parts of a passive database so that they provide all access capabilities for the types in that part of the database. These applications then become the sole interface with all other applications.

Localization requires that procedures be available for the following operations:

1. creation: to correctly initialize its value(s) and install it in bookkeeping and access structures.

2. reading: to check status of data being used for computation.

3. writing: to determine if a new value assigned is acceptable, for example, within pre-defined limits.

4. deletion: requires correctly altering any references to deleted item, to eliminate dangling references.

The last operation, deletion, is by far the most difficult and may require massive checking [McLeod, 1976].

Localization of transactions brings with it the problem of *propagation:* one change may invalidate other data, resulting in concatenating changes. The result is that one change may lead to extensive updating [Armstrong, 1974; Maier, Mendelzon and Sagiv,1979]. The propagation may be done all-at-

once, with the result that there may be high overhead for any database update. There has been as a result exploration of incremental propagation of integrity checking. These efforts are described in the next section. One important use of local integrity checking is to maintain data status continuously, without developing a separate application.

Having the database do its own integrity checking means that there is a logical partitioning between the checking done in the database and that done in applications. There has not been enough experience to determine what such a policy would consist of. It is clear that database integrity checks are especially useful for managing inter-subsystem relations, such as the spatial conflicts and load definition tasks in our example. For an example of database operators that manage integrity and are embedded in a schema, see [Eastman and Yasky, 1981]. Localized integrity checking, as allowed by data abstraction, greatly eases the task of application programming.

5.2 Extending the Concept of Type in Databases

Transactions consisting of large applications are inefficient in a concurrent environment. A change to a small part of the project requires a full design application run and also iteration of the corresponding checking programs. These will lock all the data accessed by the application. Basically, the granularity of the transactions is too coarse. Data abstraction can be used to greatly reduce the granularity of transactions, and allow users to update a design database interactively.

Full realization of data abstraction hs led to constructs that associate checks with individual data items. Access operations are defined integrally in the data type. This allows a concept, such as volume, to be defined once, along with one of more methods for computing it or checking it within the various contexts, i.e. records, in which it may be defined. Thereafter, the definition maintains its logical relation with other existing data. This allows a concept to be named once and used many times, with most of its semantics being automatically transferred to those uses. This transfer is accomplished by embedding of parameterized run-time checks into the code of all applications by the compiler.

The effect is to allow transactions to become more tightly inter-meshed, with parts of several applications running concurrently that would otherwise require sequential execution. The result is a more efficient and less restrictive style of concurrent design [McLeod, 1976; Lafue, 1980].

This goal is a long-term research goal only being realizable now in small programs applying artificial intelligence efforts to CAD. More powerful programming techniques are needed if this is to

become a common form of design database development. Such checking is only allowed now in AI languages such as LISP and PROLOG.

5.3 Delayed Integrity Checking

It may not be effective to check the integrity on a data item basis immediately. It is not only expensive in CPU cycles, but also do much needless testing. Many changes involve sets of modifications that only should be checked after ail are completed. For example, moving a column, if full structural integrity was applied, would require the structure on top to be dismantled or temporary supports brought in underneath. However, several changes in a row may bring the structural relations back into harmomy. In interactive design, when to check integrity is best controlled by user.

The effect is to allow user defined transactions on top of the itegrity defined within the database definition. Lafue has proposed putting integrity checks on a list to be evaluated later. This allows holding of tests, then flushing them according to alternative strategies, eg. by function or area. Currently, such capabilities are only research ideas and have not been extensively tested in implementations. Local integrity checking, with user control, has also been shown to be useful in low level design decisionmaking, as a design aid [Sussman and Steele, 1980; Preiss, 1978].

This sequence of design database features has progressed from a single function, single user system to a multi-function, multi-user system with automatic checking of state dependencies. In this transition, we have also moved from practically realizable systems to those that reflect major research endeavors. The practical boundary seems to be at the simplest levels of concurrent operation, with coarse scale transactions and loose interaction between designers.

6 Varying Design Development Using Normative Data

In Figure 4, DB^P_2 must be developed before^executing the structural analysis program that results in DB^S_2, in order to determine the rough sizes of mechanical equipment that impose loads on the rough sizing of the structure. In some design projects, this is not a realistic or practical ordering. Many types of projects typically determine the structural system before definition of the mechanical equipment loads.

Many such examples of conflicts in precedence orderings arise in large design projects. Cases arise where there are circularities in the precedent ordering. In manual design these precedent ordering conflicts are dealt with intuitively as an everyday occurance. Designers estimate a likely

value that has not yet been analytically determined for some critical data item, so that decisionmaking can proceed. Later, the estimate will be checked by computing the actual values in a full analysis. Multiple computations and iterative convergence is sometimes required. As long as actual values are on the conservative side of the estimate, design development resulting from the estimate may not have to be re-calculated. (If the estimate is too conservative, computations may be re-done to gain better efficiency.) The use of normative estimates is common in all areas of engineering design.

Checking programs or local integrity management that evaluate if the pre-conditions for an application are met can be extended to ask the user for missing data or can use pre-defined estimates. If such data is used, the results must be distinguished from results gained from data derived from full analyses. The database should guarentee that the estimated data will be re-evaluated and checked after the necessary inputs are available. This can be accomplished by extension of the flag capabilities used in automatic transaction management. The needed flags are:

1. NIL value, corresponding to no value yet assigned.
2. INVALID ESTIMATE, resulting from an estimate that has been determined to be invalid but not yet corrected.
3. ESTIMATE, corresponding to an estimate of the required value.
4. VALID VALUE, a computed value based on other computed values.

The flags are associated with records and denote data *status*. Data status is checked as input data for applications is accessed. The worst case of input is used to determine kind of output for an application. For example, if data for some computation includes "invalid estimate", then the data result has a status of "invalid estimate" [Fenves, 1975; Eastman and Fenves, 1978].

7 Database Schema Organization

Design projects describe inherently complex systems, with a tremendous variety of information. This is in contrast to management data, which usually has very large numbers of instances of a few record types (order less than 100). It is likely that most engineering databases will require several thousand different record types. Because of this complexity, there is a need to conceptually organize data: for the database development team, the application programmer and for the end user. The use of traditional data models helps only minimally; they are at too low a conceptual level.

A concept proposed in database research and developed in several CAD database systems is the *abstraction hierarchy*. An abstraction hierarchy consists of an engineering system being described

redundantly, at several levels of detail. The most schematic and abstracted description is defined by records at the top and corresponds to the root of a directed graph. At the bottom are the most detailed record types. In between are records that detail the top record types and are abstractions of the detailed bottom records. Several classes of abstraction have been identified, suggesting the outlines of a theory of representation [Smith and Smith, 1977].

In terms of an engineering artifiact, the development of an abstraction hierarchy consists of decomposing the system definition into records that are defined during single stages of design. Each project subsystem is described in a sequence of records defined it in increasing detail. The separation provides isolation of the previous stage design information for later concurrent accesses. An example abstraction hierarchy for building design is shown in Figure 7. In the context of applications, earlier records are the basis for developing current record definitions. Levels of detail can serve as reference marks in progress development, for project management.

The result of an abstraction hierarchy is a redundant database in which any one change implies changes to all its redundant cohorts. But the style of use is somewhat more subtle than requiring simple consistency. The assumption is made that decisions are made "top-down", from the root to details (though possibly developing the branches corresponding to different subsystems in varied orders [Eastman, 1978]). The data at any one level of detail is assumed to be consistent across subsystems when the user is finished with it. When a user adds information to the model, he is elaborating a subsystem previously defined more grossly. The gross description defines the assumed performances and resources that the subsystem is to achieve. If they are accomplished, then evaluating them in light of other subsystems is not necessary, as these have been resolved at the previous (higher) levels. Thus the previous descriptions provide local goals for the current level of detail and puts bounds on the values that result in no interaction with other parts of the design. However, if a detail, when checked against its previous more aggregate description is not consistent with it, then the previous description must be re-considered and evaluated at its level, possibly against other subsystems.

The abstraction hierarchy allows a design element to be part of more than one subsystem, yet maintains well-structured relations that can be used for communication and planning. It has been developed on slightly different forms in several areas of design [Bandurski and Jefferson, 1976; Foisseau et al, 1977; Eastman, 1978; Sussman and Steele, 1980] and [Eastman and Yasky, 1981].

Clearly, design databases are complex. It does not seem desirable to therefore accept the added complexity and overhead imposed by database systems systems that allow only tree structures. They

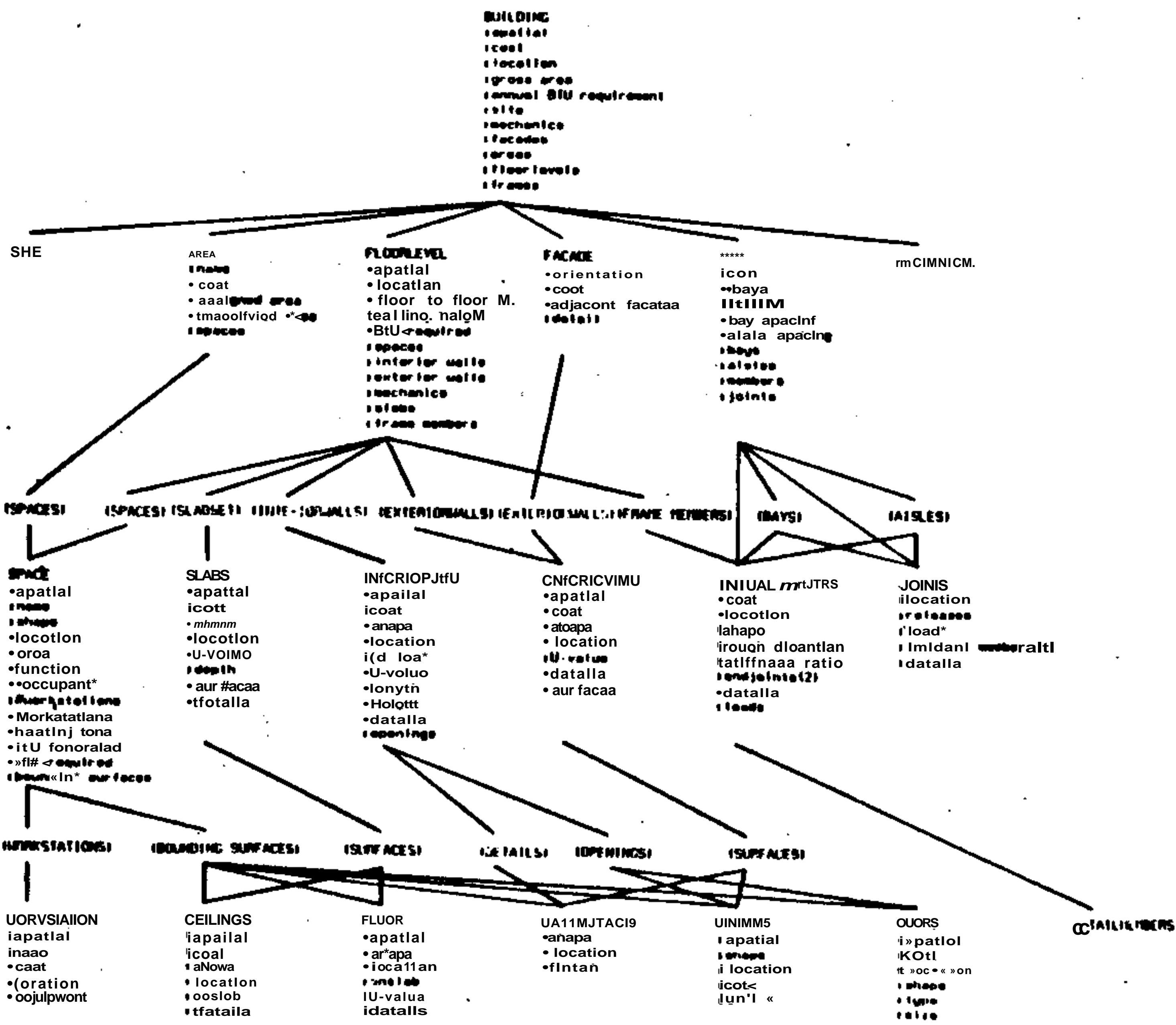

Figure 7: Example abstraction hierarchy for architectural design.

require that any data item shared at a low level by several references must be copied. Updates must be applied to all copies, adding greater management of data on the application programmers, something that constantly should be avoided, when possible.

8 Special System Requirements of Engineering Design Databases

In addition to the functional capabilities defined earlier, design databases involve a number of other issues of a more general nature, that have not yet been touched upon. Some of these are of a system facilities nature. We have already seen how localized integrity checking is limited by current language and database technologies. Some other limitiations are identified in this section.

8.1 Dynamic Schema Extension

In section 3.2 were described alternative design development sequences. One reason for needing them is the different technologies that can be used in an engineering system. Different technologies require their own analysis programs as well as development applications. Because of the varying technologies that may become relevant during the development of a design, it is difficult to specify the appropriate schema at the outset of the project. Two possible strategies may be followed. One is to incorporate within a "master schema" all combinations of records needed for different technologies. The difficulty is that the user is required to anticipate all possible combinations of technologies. In addition, many combinations of technologies are not meaningful and there should be some logical method to block these from being developed. Also, the result is a cumbersome database schema, with extra pointers and accessing schemes that must be maintained (and thus impose an overhead), even though they are not used. There are no techniques that help define schemas with these combinatorial properties.

Far preferred is the capability to dynamically load new subschemas into the database as design proceeds, using some form of incremental compilation or interpretation. In this way, the subschema for some technology can be selected during design development and the schema extended, by compiling in the needed record types and operations, while the existing data is still loaded. At least one example of such a database capability exists, in GLIDE [Eastman and Thornton, 1978].

8.2 Unique Record Access Requirements

Taking full advantage of localized integrity management leads to small transactions, in order to update the database and determine integrity effects. The corresponding speed of direct database accesses becomes an important issue, because of their frequency. The current lines of research anticipate that integrity checking facilities and status checking will become system level facilities, as extensions to the concept of data type. Beyond the justifications associated with data abstraction, there is the benefit of optimizing the implementation of these checking operations.

The problems of accessing speed are made more serious because geometry information is inherently defined by a varying number of records, of several types. If modification capability for shapes results in randomly accessed records, then the access time of a shape becomes intolerably slow. A shape can easily consist of hundreds of record instances and a drawing of several hundred shapes.. Their access should be in the thousandths or hundredths of seconds for design use then.

certain facilities, if incorporated into design databases, could alleviate this problem:

1. *repeating groups are* specified in the CODASYL Report, but seldom implemented. They consist of a variable number of sub-records or fields that can be dynamically instantiated during execution [CODASYL, 1971].

2. *priority sets* that affect the physical organization of disc. Systems with this feature group the instances in the priority set, when possible, in contiguous locations on disc, so that they may be accessed in a continuous Read.

3. *physical restructuring of Relations.* Some Relational databases provide operations that physically restructure data. These operations could be used to group data to optimize accesses.

Each of these capabilities respond to the special problem of accessing collections of diverse data quickly.

8.3 Unique Support of Multiple Disjoint Alternatives

It is common practice in design to develop multiple alternatives and compare them, finally selecting one. To accomplish this, a subschema can be copied and an alternative corresponds to any portion up to a complete transaction. Only one transaction can be merged back into the database. With small transactions and/or localized integrity checking, meaningful alternatives do not correspond to transactions. The complete database can be copied, but this is both expensive and eliminates integrity checking; when an update is made, to which copy of the database should it be checked? A set of transactions should be composed into a set of changes and evaluated globally prior to final

commitment. One means to achieve this is by provision of a general check-point facility. It saves separately any records that have been modified and stores these separately instead of overwriting the previous version of the records [Eastman and Thornton, 1978; Eastman, 1980]. Multiple set of changes can be generated, each considered as a design alternative. Only one check-point file may be merged back into the database however.

8.4 Conclusion

Some features described above are part of the CODASYL specifications, but are not implemented in many supposedly CODASYL compatible systems. Other features do not yet exist in any production database system. A valuable contribution to both the CAO fields and to the database industry would be the development of a standard specification of common features needed for design databases. Some steps have been made in this direction, but a thorough study has not been completed [Eastman, 1980; Sidle, 1980].

9 Unique Architecture Considerations

Design databases rely heavily on interactive graphics for user communication. Data structures for the management of the graphic display data are presented in [Newman and Sproull, 1979]. This structure is needed by application programs to allow identification of operations and operands and to execute the resulting procedures. Graphic, then, is an additional system structure that should be considered in an overall system implementation. The result is a three-part system design, as shown in Figure 8, consisting of display structures, application subschema and database schema.

The three parts may be allocated to computer hardware in a variety of ways, but there seem to be three principal options, currently. These are shown in the figure as b,c and d.

The first option, and until recently the clearly dominant one, is shown as Figure 8b. All operations take place in a single processor with strict sequential control. While the display may have hardware graphic operators, for vector and character generation, or even more powerful ones for display, control for graphics resides in the single processor; it is involved in every action. The implementation of this alternative is straightforward and usually treats graphics as an I-O device. Its shortcoming is in imposing graphics control on the host processor and demanding a high degree of communication with it. The host is usually time-shared, especially in a concurrent environment. Both graphics speed and host cpu cycles are compromised, with a relatively expensive processor being relied on for predominantly 1-0 operations.

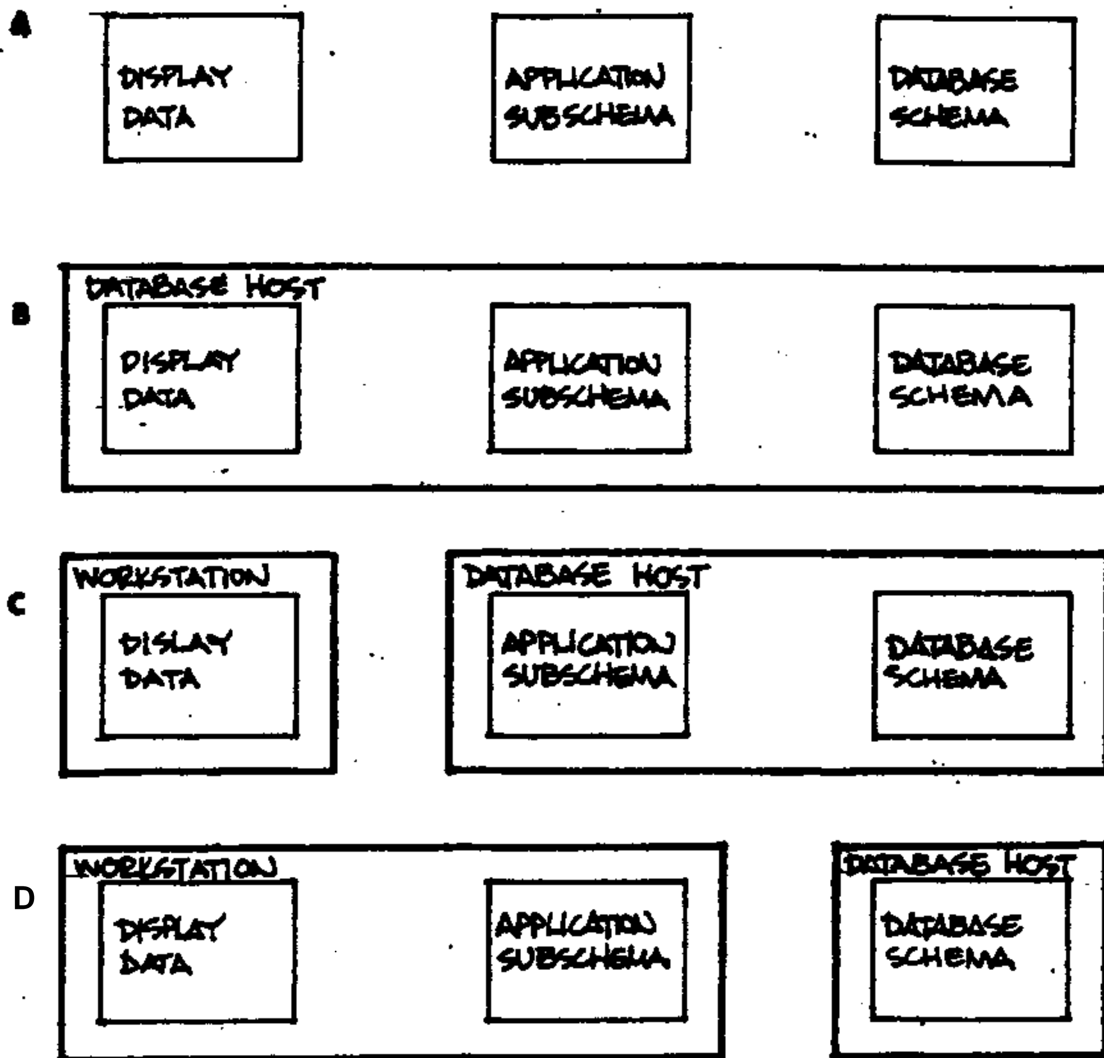

Figure 8: The three principal options for structuring graphics, applications and database operations to processors.

A second option is to pass graphics control to a micro-processor that manages all communication with the graphics device. See Figure 8c. The micro-processor takes high level operations from the host and expands them into multiple graphic operations. An example might be 'return a sequence of points[1]. Since the graphics processor is dedicated, it can respond as its capability provides. Correspondingly, the host processor is not interrupted on every operation. Both the speed of graphics interaction is enhanced and other users of the host are not interrupted for frequent 1-0 operations.

In this organization, checking for legality of an action requires interaction with the application program (and possibly the database), as the graphic display structures support only graphic operations. Thus a local change may be undertaken within the graphic display, but its legality only checked when the application or database is updated. This means that updates to the host must still be called frequently, as it executes the application. The degree of control transferred to the graphics processor is largely a function of its speed, relative to the host, and the commonality that can be defined for high-level graphic operations.

A third option is to execute the applications on the Iccal workstation processor (that also acts to control graphics). See Figure 8d. In this option, a database subschema is sent to the workstation processor, but all interaction with it is managed locally. The graphics display structures can be embedded with the interaction of the application. The mapping of data to the workstation can allow great flexibility as to hardware, programming languages and operating systems. Intermeshing of concurrent operations suggests that small transactions and frequent updating of the host is still desired, but this configuration provides a wide degree of flexibility in this regard.

Each of these options has its own influences on the user communication with both the application and the database. While more power in the local workstation allows faster interaction with larger amounts of program and data, reducing the frequency of communication results in aggrevating the management of integrity. It can probably be recognized that this author believes that advantage exists in small transactions and high degrees of concurrency in design databases. Little attention has been given to the management of dependencies within totally distributed databases. For that reason, it seems premature to consider them for design use. But see [Rothnie et al, 1980; Johnson et al, 1980].

10 Conclusion

Design databases are large systems and implementing them has all the incumbent pitfalls, such as turnover of personnel, resource requirement under- estimation, complex interfaces, and naive technical ambitions that exist in this type of enterprise. Though they hold promise for great quality and productivity benefits, the development of design databases is best approached cautiously, in a fashion commensurate with the development of a new database management system or language compiler.

Today, design databases are custom products, without many shared modules. As we better understand their design, standard systems are expected to emerge, distributed in a fashion similar to today's database management systems.

Any implementation effort must recognize the commitment that grows around a design database system. Many additional applications will be proposed and added, with the need for additional fields and record types. The system design must anticipate such growth. It should allow upgrading as new equipment becomes available.

Design databases are important systems for technological innovation. It should be recognized that

they will seriously influence how organizations develop and manage engineering projects. They provide communication within an engineering group. They will begin to reflect the organizational structure (and possibly modify it). It will also become a central repository for methods of analysis, optimization and other forms of technological advantage for the organization. Advances in design databases are likely to be reflected in advantages in engineering performance and in the marketplace.

1 REFERENCES:

ANSI, "Interim report from the study group on database management systems", Bull. SIGMOD, 7:2, (1973).

Armstrong, W.E. "Dependency structures for database relationships", Information Processing 74, IFIPS, (1974), North-Holland Press, pp.998-1006.

Bandurski, A.E. and D. Jefferson, "Enhancements to the DBTG model for computer-aided ship design", Proceedings Of The Workshop On Databases For Interactive Design, University Waterloo, Ontario, Sept.15-16, 1975.

Bandurski, A.E. and D. Jefferson, "Data description for computer aided design", ACM-SIGMOD Conf. on Management of Data, (1975b), San Jose, CA.

Baer, A., C. Eastman and M. Henrion, "Geometric modeling: a survey", Computer Aided Design, 11:5, (Sept. 1979), pp. 253-272.

Bijl, A. and G. Shawcross, "Housing site layout system", Computer Aided Design, 7:1 (January 1975).

Bjork, L.A. "Generalized audit trail requirements and concepts for database applications", IBM Systems J., 14:3, (1975), pp. 229-245.

Boyse, J.W. "Interference detection among solids and surfaces", Communic. ACM, 22:1, (January, 1979), pp.3-9.

Brieland, J.R. and R.A. Friedenson, "Designer's Workbench: The User Environment", Bell Telephone Technical J., 59:9, (November, 1980), pp.1767-1792.

Butlin, G.A. "Interactive graphics for finite elements", Finite Element Symposium, Atlas Computer Laboratory, Chiltern, UK, March, 1974.

CODASYL, Database Task Group April 1971 Report, ACM, New York, 1971.

Codd,E.F. "A relational model of data for large shared data banks", Communications Of The ACM, 13:6, 1970 pp.377-387.

Codd,E.F. "Further normalization of the data base relational model", Courant Computer Science Symposia & Database Systems, New York, 1971.

Dahl, O.J. and C.A.R. Hoare, "Hierarchical program structures", Structured Programming, by O.J. Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, 1972.

Date, C.J. Introduction To Database Systems, Addison-Wesley, Reading Mass, 1975.

Eastman, C. "System facilities for CAD Databases", Proc. 17th DA Conf. ACM, (June, 1980), pp.50-56.

Eastman, C. and G. Lafue, "Semantic integrity transactions in design databases", Institute of Building Sciences Res. Rep. No. 12, Carnegie-Mellon University, March, 1981.

Eastman, C. and Y. Yasky, "The integrated building model and database schema for the second phase of integrated CAEADS", CAD-Graphics Laboratory* Institute of Building Sciences Research Report, Carnegie-Mellon University, (January, 1981).

Eastman, C.and R.Thornton, "A report on the GLIDE2 language definition", preliminary draft, March 22,1979, Institute of Physical Planning, Carnegie-Mellon .University, Pittsburgh, PA.

Eastman, C. "Representation of design problems and maintenance of their structure", IFIP Working Conference fin Applications Of Artificial Intelligence To CAD, Grenoble, France, 1978.

Eastman, C M. and S. Fenves, "Design representation and consistency maintenance needs in engineering databases" Conference gn Engineering and Scientific Data Management, Hampton VA, May 1978.

Eswaren, K.P., J Gray, R. Lorie ad I. Traiger "The notions of consistency and predicate locks in a data base system", IBM Research Report RJ1487, San Jose CA, (December, 1974).

Fenves, S. "Integration of database management and project control for engineering design", Proceedings On The Workshop On Databases For Interactive Design, ACM, New York.

Fischer, W.E. "PHIDAS - a database management system for CAD/CAM application software" Computer Aided Design, 11:3, (May, 1979), pp.146-150.

Fredriksson, B., J. Mackerle and B.G.A. Persson. "Finite-element programs in integrated software for structural mechanics and CAD", Computer Aided Design, 13:1,(January, 1981), pp. 27-39.

Hoskins, E.M.,"Computer aids in building", Computer Aided Design, J.J. Vlietstra and R.F. Weilinga (eds.) American Elsevier, N.Y. 1973.

Johnson, H.R., D. Comfort and D. Shull, "An engineering data management system for IPAD", Proc. IPAD Nat. Symposium, NASAConf. Pub. 2143, Sept. 1980, pp.145-178.

Kamel, H.A., and Shanta, A solid mesh generation and result display package" J. Pressure Vessel Tech., AMSE, 96:3, (August, 1974), pp. 207-312.

Kim, J.H. and D. Siewiorek, "Issues in IC implementation of high level, abstract designs" Proc. 17th DAC, (June 1980), pp: 85-91.

Lafue, G. Integrating language and database for CAD applications", Computer Aided Design, 11:3, (May, 1979), pp. 127-130.

Leesley, M.E., A. Buchmann. D. Mulraney, "An approach to a largely integrated system for computer aided design of chemical process plants", int Congress gn Computers to th£ Development of Chemical Engineering and Industrial Chemistry, Paris, (March, 1978).

Lockemann, P.C., H.C. Mayr, W.H. Weil and W.H. Wohlieber, "Data abstractions for database systems", ACM TODS 4:1, March, 1979, pp. 60-75.

Maier, D., A. Mendelzon and Y. Sagiv, "Testing the implications of data dependencies", ACM TODS, 4:4, (December, 1979), pp. 455-469.

Malenson, G.S. and S.A. Spurlin, "A fundamental approach to data base implementation for design automation ",Proc. Workshop on Databases for Interactive Design ACM, (Sept. 1975), pp. 120-129.

Martin, J. Computer Database Organization, McGraw-Hill, N.Y. 1975.

Matelan. M.M. and R.J. Smith, "A database design for digital design automation", Proc. Workshop on Databases for Interactive Design, ACM, Sept, 1975, pp. 85-92.

McLeod, D.J. "High level expression of semantic integrity specifications in a relational database system" MIT Laboratory for Computer Science, TR-165,1976.

3 flMfl2 OOtiT 1M33

Miller, R. E. et al. "Feasibility Study of an integrated program for aerospace vehicle design (IPAD)" Boeing Commercial Airplane Company, Seattle 1973.

Miller, R.E., J. Southall and S. Wahlstrom, "Requirements for management of aerospace engineering data" Computers and Structures. 10, (1979), pp.45-52.

Newman, W. and R. Sproull. Principles Of Interactive Computer Graphics. McGraw-Hill, N. Y. 1973.

O'Neill, L.A. et all The designer's workbench' Bell System Technical Journal 59:9. November, 1980.

Parnas, D.L. "Information distribution aspects of design methodology", Prog. IFIP Congress 1971, North-Holland Press, pp. 339-344.

Patrick, R.L., Security System Review Manual. AFIPS Press, Montvale, N.J., October 1974.

Preiss, Kenneth, "Engineering design viewed an an activity in artificial intelligence", SRI International Technical Note No. 167, 24 July, 1978.

Reisner, P., R. Boyce and D. Chamberlin, "Human factors evaluation of two database query languages - Square and Sequel", Proc. 1975 NCC. AFIPS, (1975), pp. 447-452.

Roos, D.T. Ices Systems Design. MIT Press, 1967.

Rothnie, J.B., P. Bernstain, S. Fox, et al, "Introduction to a system for distributed databases (SDD-11". ACM TODS. 5:1. (March. 1980), pp. 1-17.

Rowe, L. and K. Shoens, "Data abstraction, views and updates in RIGEL", Proc. 1979 SIGMOD Conf. on Manao. of Data, ACM, (May, 1979).

Sidle, T.W. "Weaknesses of commercial database management systems in engineering applications", Proc. 17th DA Conf. ACM. June, 1980), pp. 57-61.

Smith, J.M. and D.C. Smith, "Data abstractions: aggregation and generalization", ACM TODS. 2:2, (June, 1977), pp.1-5-133.

Sussman, G.J. and G.L. Steele, "CONSTRAINTS - a language for expressing almost hierarchical descriptions", Artificial Intelligence. 14:1, (August, 1980), pp. 1-40.