Database Facilities for Engineering Design

CHARLES M. EASTMAN

Invited Paper

Abstract—As the complexity of engineering design projects increases and the objectives for their performance become more demanding, there is growing interest in integrated databases that support a broad range of applications and various forms of automated entry. This paper is a tutorial on the design and implementation of integrated design databases. It addresses a variety of functional issues, including subsystem integration, maintaining consistency among concurrent users and various system architectures.

I. INTRODUCTION

A. 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 and databases are becoming part of most large scale computer applications in engineering.

The need for handling extensive amounts of data arose concurrently with major applications. An early need was storing constants such as material properties for analysis programs. This data was stored separately and sequentially read at the beginning of an application, but not modified during use. By storing it separately, it could be used by multiple projects using the application, but still be occasionally extended or modified. When changes were made to the data, they applied to all projects using the program [53]. A later development was the use of files for pre- and post-processing. Examples are textual or graphic oriented programs that expand simple input to the proper format for an 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 [12], [30], [33].

B. Integrated Design Databases

The preceding file operations provide interfaces with particular applications programs. But as the number of programs has grown, the generation and management of input data they use have become problematical. 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. This type of 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 and improving control of the design product;

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 [34], [41], [43], process plant design [38], [60], shipbuilding [3], manufacturing [29], aerospace manufacture [46], and building [8], [21], [31].

There are many levels of sophistication in design databases. Design databases are complex systems that create an environment that easily impacts organizational decisionmaking, management and communication. 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, some unique needs that distinguish design database systems from business applications are identified, that require special consideration. This article is meant to serve as a tutorial for those considering development of a design database system and as a guide to the literature. It also identifies certain hazardous areas that production oriented efforts should avoid.

II. DATABASE MANAGEMENT SYSTEMS

The term database refers to an organization of data, too large to be held in main memory of a computer at once, upon which

Manuscript received March 23, 1981; revised June 10, 1981. This work was supported by the National Science Foundation under Grant MCS-78-22328.

The author is with the Institute of Building Sciences and Design Research Center, Carnegie Mellon University, Schenley Park, Pittsburgh, PA 15213.

1Good references on database design and implementation are [17], [42], and [62]. 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.
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 usually refers to a particular collection of data, whereas the system level program that supports the development of a database is called a database management system (DBMS). In this paper when reference is made to a database system this means a DBMS. A DBMS provides the means to organize data, to access it according to pre-defined access structures and to bring 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 tables and forms of database contents, interactive query languages and statistical analysis packages. The size, structuring capabilities and support utilities of a database system distinguish it also from virtual memory.

A. DBMS Organization

The basic organization of a database system 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. It corresponds to the declaration of variables in a programming language. The primitive unit of information is the data entity or field. Fields are defined according to the data type they hold, e.g., REAL, INTEGER, CHAR STRING, etc. Some DBMS’s allow users to define their own types, such as enumerated types and subranges, but strong user-defined type definition capabilities, as found in Pascal or ADA, are an exception. Fields are composed into larger collections called relations or records. Most often, records have a fixed composition, so that the location of particular fields within the record can be assumed at execution time. Some implementations allow variable numbers of a particular field. The format of a record is defined as a record type. A variable and usually very large number of record instances are expected to be created over the database lifetime.

The organization of fields into records can be approached intuitively and with good results in simple design databases. A record can correspond to a device or collection of common properties. But as the operations on a database become complex, its proper organization becomes a challenge and an area for research. This issue is taken up in Section VII.

For organizing data, it is useful to have one record refer to several others, such as would be needed for some device, defined in a record, to refer to each other device connected to it. Similarly, it is useful for several records to refer to a common one, as when several devices are of the same type, with a single set of properties. The records for the individual devices refer to a shared set of properties. Thus one-to-many relationships are desired between records as well as many-to-one relationships (and many-to-many). 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, i.e., they do not allow many-to-one relations, while network structures do allow them. 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 [59].

In addition to the relations between records, all database systems provide means for direct accessing of records with certain data, that is associative accessing. For example, it might be desirable to access all elements provided by a particular manufacturer or all the elements using a particular material. The specified field and volume is called the access key.

Direct accessing mechanisms consist of path traversals within a data structure and/or special directories. All require preorganization 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 the following.

1) Ordered directories, known as index sequential (or ISAM), allow accessing records by the value of any field that can be uniquely ordered. The facilities consist of directory entry, management and search processes. Presorted binary trees provide similar capabilities.

2) Directories built of all records having common attributes, called inverted lists are especially appropriate for accessing fields that hold nominal data. They also are fast because the relevant data is in the directory and search does not require reading values of data from records. Inverted lists therefore also are used for ordinal data. Inverted list structures greatly increase the size of a database.

3) Hash coding schemes that transform a unique key into an address that holds record entries with the desired key value are efficient of both time and space. However, they can only be used for one field, because they rely on a unique location of the record on mass storage.

For a detailed review of accessing mechanisms, see [42, ch. 19-28], [62, ch. 3-43].

The record types and the structures between them define a complex data structure scattered between main memory and mass storage. 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 data manipulation language or DML, on the other hand, provides the mechanisms 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 the necessary intermediate and final records into main memory and copying the application data into the allocated space, then running the application against it. Computed results, additions or changes may be stored back onto the data structure. The subset of the schema used for an application is called a subschema (or view or external schema).

Other support facilities 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 often imbedded in the DML [52], [64].

2) Backup 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 [9], [62, ch. 11].

3) Access control and security allow management 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 [50], [62, ch. 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.

Database models normally distinguish three classes of user. The database administrator (DA) is responsible for defining and managing the database schema, using the DDL. He is responsible for all recompilations of the database and for making sure that it supports the various applications. Application programmers define the data needed for an application and devise the accessing paths for retrieving it, using the access paths set up by the DA. They rely on the DML for defining the application interface. They also devise the necessary applications for updating the database. End Users call the application programs developed earlier and use them to operate on parts of the database. These applications may create new record instances, modify or delete existing record instances.

Most DBMS’s are host-based systems. That is, they provide accessing facilities that may be embedded in programs written in some other language that is the host for the application. COBOL, Fortran, and PL/I are common host languages. Some DBMS’s are self-contained systems that provide all their own computational facilities. Superficially, self-contained systems should not be attractive for interfacing with existing CAD applications because they seem to require re-writing the application in the database language. A self-contained system, however, usually can interface to existing applications by writing the application input to a file external to the DBMS. This file can then be used as input to the application. They provide other virtues that will be described later.

The sophistication of these facilities vary greatly, as do the cost of the systems that provide them. 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 investments “on top of” a DBMS.

B. Alternative Data Models

The complexity of database organization has led to alternative conceptual models for describing it to different classes of user. These models define a database in terms that are largely independent of its physical implementation. Data models focus on ways of understanding a DBMS’s functional capabilities. The principal data models are:

1) The CODASYL Report is a proposed standard for database systems 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 capabilities to those specified in the CODASYL Report [13]. The Digital Equipment Corporation’s DBMS, SEED, IDMS and TOTAL are four database management systems based on the CODASYL Report [62, ch. 9]. The description presented here is consistent with the CODASYL Report.

2) Relational database structure is another major data model. It is composed of relations, consisting of a set of data values, called components. Relations are usually laid out in tabular format, in which columns hold components for a particular attribute, and rows correspond to record instances and are called tuples. References between records are resolved by formal set theoretic operators. The Relational data model has gained strong advocates because it allows formal analysis of database structure and operations [14], [15]. For a survey of implementations of the relational model, see [35].

3) ANSI/X3/SPARC is a second recently formed 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 [11]. The terms used in the three models are shown in Fig. 1.

This review of database system concepts has been cursory, at best. But with these concepts, we can begin to investigate some detailed issues regarding support for design databases.

C. Example Development of a Simple Design Database

The above facilities can be used for design databases. A set of examples is developed in this section that will be used later to elucidate a variety of functional and implementation issues. Even simple databases are large and complex in their variety, with many diverse design and implementations issues.

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

1) piping system topology design; this application supports interactive definition of end node and source node requirements and their locations and connectivity;
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 inplace construction cost of piping and fittings;
5) a program for the production of numerical control tapes, for automatic pipe cutting, bending, welding, threading, etc.

Input to application programs can be classified into three types of data: 1) specification of the engineering system being analyzed; 2) application constants, such as computational coefficients, safety or efficiency factors or design constants, and 3) control instructions regarding the execution of the analysis [6]. In the preceding suite of applications, most user input regarding the system specification is in the first program. Later
D. Project Dependent Databases

Such a set of applications involves two types of databases: one or more databases describing the engineering project (in this case, the piping system) and a second set of databases that hold general information used in many different engineering projects.

The first type of database is called a project dependent database and these are denoted DB\(_1, 2, 3\) in Fig. 2. The diagram shows three different project dependent databases; these could be separate, or they could be different 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. Between each DB is a translator that reformats the information in a form most usable for the next set of applications. If an application can access multiple databases during its execution, then the distinction between having one or several databases is insignificant.

The database shown in Fig. 2 supports a fixed development sequence, because of the sequential ordering of applications. It allows iterative design cycles, because the result of any application can be thrown away and regenerated until one is found that is satisfactory. While the ability to throw away results and iterating seems general and sufficient, but it isn’t. Suppose that during shop drawing application, it was found that some fittings spatially conflict with the elements of other subsystems and that one or more nodes must be moved. In such a case, iteration is expensive, in that probably two earlier applications would have to be iterated (topology design, sizing). More desirable would be the ability to make a change directly in DB\(_3\) and check that the change is acceptable to the sizing application. But in order to make such a change directly, means must be provided to externally modify DB\(_3\) and also another application must be added to check that the change is consistent, e.g., a backward mapping.

This example drives to the heart of a number of database issues regarding the consistency of data generated at various stages of design. 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 as the spatial conflict example, then the existing data is not acceptable and its computation must be iterated. This is because the dependencies are all in the forward direction. The management of more complex dependencies among applications will become significant as capabilities are added. For the present level of discussion, these issues are human management problems; such inconsistencies are either tolerated or the set of applications iterated. Later, more elaborate alternatives will be discussed.

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 manageable within current software development practices. Thus this example and ones similar to it can be realized in many design areas [11]. 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 in one direction. Relaxing these limitations will be taken up in the following sections.

E. Project Independent Databases

The second type of database used in CAD holds catalogs of parts, such as standard fittings, cost information, material properties and other information external to the context of a single project. These are called project independent databases. Project independent databases hold information that is used by the project dependent databases, either by referring to it or by copying it. This choice of copying or referring is a crucial one and affects many other considerations.

The advantages of referring to project independent data is the great savings in space gained in each project dependent database by sharing the common information. It also increases control over the data used in the project databases. If referring is used, then care must be taken in strongly linking the versions of the project independent databases used in each project database. Modification to project independent databases requires archiving of past versions so that they may be re-used later.
with the project dependent databases that used them. Over time, project independent databases that are used by referring become a significant management problem.

Copying of the project independent data into projects adds storage requirements, but usually allows faster accessing. It also allows special elements to be added to a single project, without adding them to the project independent databases. For special purposes, it also allows modifying the properties of elements (which is not usually desirable) for exploring special performance problems. This is also desirable in recursive design environments, where the possibility exists to take a component “off the shelf” or to design a custom one. As space requirements are relaxed, the advantages of copying project independent data are expected to move efforts strongly in this direction.

**III. DEVELOPMENT OF MULTIDISCIPLINARY DESIGN DATABASES**

The above example ignores the many other functions to which any engineering product must finally respond. 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 multidisciplinary design.

A typical structural design development sequence might be as shown in Fig. 3. The development sequence is: generation of the structural network, determination of section properties, member selection and proportioning and cost estimation, and joint design and detailing, followed by production of shop drawings. Again, the DBT and DBP could be merged into a single database that is their union or kept separate, with mappings between them. 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 project independent DB’s for the material properties and joint detailing assumptions. Steel sections available from industry are provided in a catalog along with their properties. This development sequence is also limited to forward development only; a change in topology based on detailing considerations cannot be propagated backward. Complete iteration is required.

The structural and piping databases could be separate from each other. However, their definition in a common database would allow management of important interactions of several forms 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 elements are initially located. The geometry of a shape can be represented explicitly, as a set of bounded surfaces [5], [10] or in some approximated form. Means for handling spatial conflicts are described in [26] and are based on methods for accessing information by the location of the element being described 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 could 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.

**A. The Ordering of Multifunctional 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 Figs. 2 and 3 quickly become too complex and do not incorporate needed information.

An 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 preconditions 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 loads or 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 the edge's 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 [22], [28].

A transaction is an operation within which logical relations within a database temporarily are not valid, due to data modifications. Before and after the transaction, the logical relations are assumed to hold [27]. Thus a transaction is a basic database operation, defined by the logical relations that the data must satisfy to be useful. In CAD, it has been shown that applications add to the pool of logical relations that are satisfied as well as maintain existing ones [22]. 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.

In Fig. 4, the two subsystem databases are structured in corresponding transaction graphs and combined. Three new applications are shown that relate the two subsystems. The definition of loads for the structure has as its pre-condition the rough sizing of the piping (so as to determine the pipe's weight and the linear weight of its contents), as generated in DB sub 2. It is needed as input for the application that analyzes beam stresses; thus it is a precondition of the state defined by DB sub 1. The application of the spatial conflict program can go between a variety of database conditions. Here, we have limited interaction to two places in the development sequence: between DB sub 1 and DB sub 2 and DB sub 2 and DB sub 3. While there is a strict precedent ordering for the load generation, spatial conflict has no precedent dependency, but a joint one.

Transaction graphs can be used to organize the integration of 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 [22].

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 everywhere they exist within the database.

B. Alternative Design Development Applications

Alternative technologies exist for realizing an engineering function and it is desirable to support such multiple capabilities within a design DB. The design programs supporting the entry and definition of these technologies vary, though the resulting systems serve the same overall purpose.

In the organization of a design database, alternative technologies take the form of multiple parallel development sequences. For example, there can be multiple graph definition programs for the 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 Fig. 5(a).

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

While this kind of mixing of applications is easily supported within the organization of a database, it makes defining the completion of a database staff difficult. The state conditions are not guaranteed by completion of any one program. Instead, they 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 Fig. 5(b).

The checking program can be thought of as a post-processor to the preceding applications that determines the existence of the required database state for the completion of the transaction. There is some advantage in maintaining the checking programs separately from their application, especially when there is also a choice of several succeeding applications.

C. Automatic Management of Precedence Ordering

Up to this point, the sequential management of the execution of applications has been assumed to be 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 used for input. Examples might be 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. With a database of even medium complexity, there is a need to automate the management of transactions. Automatic management of transactions disallows the execution of a transaction for which pre conditions are not satisfied. Two types of approaches have been discussed in the research literature:

1) those based on a snapshot evaluation of the database state;
2) a state transition approach specifying the legal transitions from one state to another.

The first approach has been developed by McLeod [44], among others. The conditions a database must satisfy in order to accept some transactions are defined atomically as "integrity constraints." Integrity constraints can be used to check the validity of data at READ time (when the ratio of writes-to-reads is large) and at WRITE time (otherwise). State snapshot approaches have been developed for small transactions, but not for larger applications. A major problem with its use is in defining at the outset the necessary integrity constraints.

The state transition approach, on the other hand, does not require definition of all atomic integrity constraints, but only the conditions produced by the completion of the high level applications. This is both a virtue and the weakness of this approach. The state conditions created by large applications can be defined intuitively and associated with a database state. This is the way applications are combined normally. The weakness is in not characterizing these states rigorously. Eventually, both methods require rigorous definition of the integrity constraints needed to start a transaction.

In the state transition approach, 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, e.g., the flag 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 post-state 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 [25].

One example of the data that could be used in such a central control program is shown in Fig. 6. It shows an edge-to-edge precedence matrix of the transaction graph shown in Fig. 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-to-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. Other graph representations, such as node-to-edge, may be used. Notice that transactions 3 and 11 are bidirectional, with pre-states and post-states for both directions.

Both approaches to automatic management of transactions requires further research development before they can be widely applied.

IV. DESIGN DATABASES SUPPORTING CONCURRENT USERS

Most design projects are of such magnitude and under such time constraints that parallel development by multiple designers is 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 a data item at a time. The
data items used for locking are usually files or records [27]. This mechanism gives adequate protection if the information entities for which locks are allowed are small, such as a record. READ–WRITE locks respond to conflicts in simultaneous 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.

A. 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 or different set of completed pre-condition 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 Fig. 4, the application to size structural members cannot be initiated until the program that rough sizes the piping is completed. Notice that, with these rules, shared data is only READ. If data already has 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 imposes requirements on record structures of the database schema. 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 onto 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 VII.

B. Managing Concurrent Use Spatially

Another means of separating 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, e.g., 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 Fig. 7, where each story and each space of a building are distinct and organized so that the components within them can be assessed separately.

These partitions are fixed; story as a partitioning scheme is defined in the database schema. Ideally, one would like to identify such partitions dynamically, without preorganization. Some turnkey CAD systems provide capabilities suggestive of a means to accomplish this (but are not able to realize it). By choosing a graphic window that is a small portion of the total physical layout, the user defines the subset of space he is working with. Windowing suggests a means for identifying the subset of the design being worked on. However, applications would have to be written to operate totally within such spatially derived subschemas. Some structures for making such dynamic spatial accesses are presented in Bentley and Wood [8], [26]. The current use of layers in turnkey CAD systems is another example of schema defined spatial partitioning. Concurrent use can be based on many forms of logical partitioning.

C. 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 not completed, and the assumptions they made in certain decisions. These communication requirements are similar to those needed among software development teams.

Communication methods should be available to support “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 communication messages. Talk mode should be a monitor level capability, that is managed so that it will not halt the current operation (as could happen if a text message was sent to a terminal while display operations were going on). Thus queuing is required for even immediate communication. Attached messages should be read for all fields used in a transaction at its beginning. It is important to recognize that a design database is a communication system as well as a database system.

V. Localized Integrity Management

If we consider engineering designing as a problem-solving task, then the maintenance of the logical relations within a design can be recognized as a major task. By logical relations, we mean consistency across all duplicated data, consistency between data that is redundant, i.e., completely derived from other data. Logical relations are based on definitions, physical laws and engineering models. In the vocabulary of the database literature, such logical conditions are called integrity relations.

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 structures must be modified to maintain the
integrity of the database. This includes embedding newly created data in accessing structures. It also includes making the required corollary modifications to maintain integrity when one piece or a small set of data is changed.

A. 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 data. Software engineering emphasizes the need to define data structures and their operations jointly, while hiding implementation details [16], [49]. This merging of data and operations is commonly referred to as data abstraction.

In most DBMS's, there is no means to define operators within the database. It is assumed that every application programmer will learn the requisite bookkeeping required to update parts of the database. This policy is both inefficient and dangerous. If the DBMS has an active, procedural capability, then the schema definition can include the procedures for modifying a database and these can be used by all application programmers. There have been several calls for such capabilities within database systems [37], [39], [61], and a few research database languages have been developed that incorporate processing capabilities [24], [55]. Self-contained DB's also have the facilities to support localized integrity management.

Data abstraction in database systems can be achieved with current host-based 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.

McLeod [44] has identified the need for localized procedures for the following operations:

1) creation: to correctly initialize its value(s) and install it in bookkeeping and access structures;
2) reading: to check the status of data being accessed 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.

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 [2], [40]. 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. Yasky [63] has proposed two additional operations to those defined above:

5) updating: to re-compute the various data derived from the data entity;
6) extracting: to compute data that is not stored but recomputed when it is accessed, e.g., a function.
Update allows the separation and delaying of some integrity management operations. 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. Some guidelines for selecting integrity relations to be embedded in the database include:

1) if the integrity relation is associated with data entities manipulated by more than one application program. This guideline includes, for example, all intersubsystem relations;
2) if the integrity relation is associated with data entities that are part of a transaction that requires management of its pre- and post-states, as described in Section III-C.

It is clear that database integrity checks are especially useful for managing intersubsystem 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 [23]. Procedures that maintain integrity that are defined in abstract data types may be nested. One procedure that manages some high level integrity relations is likely to read, write or update other lower level entities leading to their integrity management. An outside integrity operation, that calls lower level ones, must guarantee the integrity of the lower level one by calling all the required updates. With highly nested integrity operations, it is likely that updates will proliferate, so as to deal with the management of different lower level updates, at various levels of nesting.

Data abstraction is only a partial answer to integrity management, because of its reliance on the nesting of abstraction updates. The resulting tree cannot cope with many-to-one and many-to-many relations. Even so, localized integrity checking, as allowed by appropriate use of data abstraction, greatly eases the task of application programming.

B. Integrating Data Abstraction with Transactions

In the earlier discussion of transactions, they were considered as being application programs. There are several costs associated with transactions of this size. 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 and prohibit its concurrent use by others. Basically, the granularity of the transactions is too coarse.

Both transactions and data abstractions were conceived to respond to the same issue of integrity management. Transactions enclose operations that modify data that, during their modification, violate integrity relations. The operations associated with an abstract data type also are transactions, because they modify data such that the integrity relations are satisfied after the operation is complete, but not within them. The operations in abstract data types provide a central place for managing integrity relations and also provide the potential for verifying their correctness. These two concepts can be combined, leading to both an improved structure for managing integrity relations and in implementing their maintenance operations.

A high level abstract data type can be defined that includes immediate or delayed lower level integrity operations. The lower level operations are called by and completely controlled by the high level abstract data type. Notice that these operations include both integrity maintenance and checking. These operations could be defined in a transaction graph, showing the various nested transactions. This structure would then both relate existing applications and also break down the applications into finer grained transactions. The resulting small transactions can support high degrees of concurrency, if the constituent transactions within an application incrementally access and update the DB. The decomposition also allows immediate feedback on the validity of a transaction, by doing the necessary appropriate-level integrity checking. This is in contrast to an application level transaction, that holds all checking until the mapping of data changes is sent to the database as its completion.

But many design actions are not under automatic control and involve user-defined changes of data at any level in the nesting of abstract data types. According to the management of transactions described earlier, the update of any transaction requires re-executing all its successors, including any checking tasks of the higher level transactions that bound the lower level ones. Because a low level transaction may be called by several others, it may be necessary to re-execute the successors along several different paths within the transaction graph.

In the cases where such updates are invoked by the user, the outer level integrity operations have no special knowledge of the cause of any failure and the user must be prepared to "patch up" the problems with all outer level integrity relations. Of course, many normally entered transactions will require their integrity relations to be maintained by user actions and won't be "smart enough" for automatic maintenance. The resulting structure can support automatic maintenance of integrity relations, entered from above in their normal manner and automatic checking (but without maintenance) when entered manually.

The effect of this merging is smaller transactions, resulting in the possibility of them becoming 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 [38], [44]. The development of database languages that support both data abstraction and other special needs of CAD are reviewed in Section VIII.

This model of a well structured engineering design database is a research goal only being realized now in research projects. Some of these apply artificial intelligence efforts to CAD. In general, more powerful programming techniques are needed if these capabilities are to become common features of design database systems. Local integrity checking, with user control, has also been shown to be useful in low level design decision-making, as a design aid [51], [58].

C. Delayed Integrity Checking

The introduction of update operations emphasize that it may not be effective to check immediately the integrity of all data items that violate integrity. It is not only expensive in CPU cycles, but also may result in much needless testing. Many changes involve sets of modifications that only should be checked after all are completed. For example, moving a column, if full structural integrity was applied, would require

---

3 This requires major re-writing of the application.
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 harmony. In interactive design, when to check integrity is often best controlled by user.

Integrity requires that all the potential violations of previous actions be acted upon. Instead of leaving the management of these updates calls to users (either end-users or application programmers), some people have developed proposals for a central manager of delayed integrity operations. Lafue has proposed putting integrity checks on a list to be evaluated later [36]. This allows holding of tests, then flushing them according to alternative strategies, e.g., by function or area. Currently, such capabilities are only research ideas and have not been extensively tested.

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. We have also moved from a database structure using normally sized application programs as transactions to nested transactions that break down normal applications to primitive actions of the lowest level abstract data type. This analysis of applications poses a research objective only now being approached. In this transition, we have also moved from practical and realizable systems to those that reflect major research endeavors. The boundary of current production practices seems to be at the simplest levels of concurrent operation, with coarse scale transactions and loose interaction between designers.

VI. VARYING DESIGN DEVELOPMENT USING NORMATIVE DATA

In Fig. 4, DB² must be developed before executing the structural analysis program that results in DB₂, 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 occurrence. Designers estimate a likely value that has not yet been analytically determined for some critical data item, so that decisionmaking can proceed. Later, the estimated data 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 recalculated. (If the estimate is too conservative, computations may be redone 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 preconditions for an application are met can be extended to ask the user for missing data or can use stored, 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 guarantee 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 the information accessed within a transaction. The "worst case input" is used to determine the status 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" [25], [28]. Completing a design requires that the final database state's status is VALID VALUE.

VII. 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 data corresponding to the root, i.e., top, of a directed graph. At the bottom are the leaves of the graph holding the most detailed data. In between are data that detail the top and are abstractions of the bottom leaves [20], [58]. Several classes of abstraction have been identified, suggesting the outlines of a theory of representation [57].

In terms of an engineering artifact, the development of an abstraction hierarchy consists of decomposing the system definition into abstract data types for each engineering sub-system. The types must separate the data into single design applications. Each project subsystem is described in a sequence of record and other types, defined 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 Fig. 7. In the context of applications, earlier higher level data type instances provide the context for defining low-level data type instances. Levels of detail can serve as reference marks in progress development, for project management.

An abstraction hierarchy orders the transactions in a design database. 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 [20]. This ordering corresponds to the precedent ordering of transactions allowed within the CAD system. The data manipulated at any one level of detail is assumed to be consistent with other data using the same pre-condition information. When a user adds information to the model, he is elaborating the data describing a subsystem previously de-
defined more grossly. The gross description defines the assumed performances and resources that the subsystem is to achieve. If the subsystem can achieve the goals set by the gross description, then evaluating them in light of other subsystems is not necessary, as these have been resolved previously. Thus the higher descriptions in the hierarchy 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 aggregate description must be reconsidered 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 [3], [20], [23], [58].

Throughout this paper, we have approached the forming of transactions using a decompositional approach. That is, we have started with application programs at transactions, then have broken them down. In database theory, this is one of two ways to approach the structuring of a database. It is contrasted with the synthetic approach that identifies the data associated with the most primitive integrity relations, then composes these into nested levels of transactions. Derivation of the proper units of data, using synthesis or decomposition, is called normalization and is an intense area of research, particularly by those working within the relational model [7].

VIII. Special System Requirements of Engineering Design Databases

The earlier sections have presented the need for design DBMS's that incorporate procedural capabilities for implementing abstract data types. As the size of transactions are reduced, with the corresponding increase in degree of concurrency and integrity checking, the type of system facilities needed look more like a programming language with database accesses and less like the current models of a DBMS. Its address space must be adequate for the task; it should support the various features responding to data abstraction; its language constructs should support both database integrity checks and user interaction; it should support concurrent users. Other features of a CAD oriented DBMS are reviewed in this section.

A. Dynamic Schema Extension

In Section III-B 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.

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 [24].

B. 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 especially 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.

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 subrecords or fields that can be dynamically instantiated during execution [13];
2) priority sets that affect the physical organization of disk. Systems with this feature group the instances in the priority set, when possible, in contiguous locations on disk, 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. Another feature that can alleviate accessing problems are the use of special hardware for data access.

C. Unique Support of Multiple Disjoint Alternatives

It is common practice in manual design to develop multiple alternatives for subsystems and to 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 any records that have been modi-
fied and stores these separately instead of overwriting the previous version of the records [18], [24]. Multiple sets of changes can be generated, each considered as a design alternative. Only one check-point file may be merged back into the database however.

D. Conclusion

A shopping list of features desirable for a DBMS for design applications should include support for:

1) interfacing to Fortran and other application languages;
2) defining abstract types (or for emulating this capability);
3) dynamic schema extensions, so as to allow design development to dynamically define subschemas;
4) grouping of diverse data for fast access and/or use of special hardware;
5) fast execution, so as to support small and frequent transactions;
6) support for multiple design alternatives.

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 CAD 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 [18], [56].

IX. GRAPHICS AND DISTRIBUTED PROCESSING

Design databases rely heavily on interactive graphics for user communication. Data structures for the management of the graphic display data are presented in [47]. Special graphics structures are needed to support user interaction with application programs. They allow identification of operations and operands and execute the resulting processes. Graphics, then, is an additional system structure that should be considered in an overall system implementation. The result is a three-part system design for applications, as shown in Fig. 8.

The parts consist of display structures, application subschemas and database schema.

The three parts may be allocated to computer hardware in a variety of ways, but there seem to be four principal options, currently. The first option, and until recently the clearly dominant one, is shown as Fig. 8(a). All operations take place in a single processor with strict sequential control. While the display may have hardware graphics 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 input-output (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 necessarily time-shared in a concurrent environment. The result is that both graphics speed and host CPU cycles are compromised, with a relatively expensive processor being relied on heavily for I-O operations.

A second option is to pass graphics control to a microprocessor that manages all communication with the graphics device. See Fig. 8(b). The microprocessor takes high level operations from the host and expands them into multiple graphic operations. An example might be 'return a sequence of points'. Since the graphics processor is dedicated, it can respond at its full capability. Correspondingly, the host processor is not interrupted on every operation and the speed of graphics interaction is enhanced.

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 local workstation processor (that also acts to control graphics) (see Fig. 8(c)). 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 combined 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.

The last major option is shown in Fig. 8(d). It considers graphics as the major transaction process for the database and thus is organized so that graphics directly communicates with it. Applications are treated as (separate) support tasks, running on the same or different processor. This configuration emphasizes high speed interaction with the DB using small transactions and supporting high degrees of concurrency [18].

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 inter-
action with larger amounts of program and data, reducing the frequency of communication results in aggravating the man-
agement 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 atten-
tion has been given to the management of dependencies within
totally distributed databases. For that reason, it seems premise
to consider them for design use. But see [32], [54].

X. 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 enter-
prise. Though they hold promise for great quality and produc-
tivity benefits, the development of design databases is best
approached cautiously, in a fashion commensurate with the
development of design databases is best approached cautiously,
in a fashion commensurate with the
development of a new

tion has been given to the management of dependencies within
additional fields and record types. The system design must
applications will be proposed and added, with the need for
projects. They provide communication within an engineering
tory for methods of analysis, optimization and other forms
(b) A. Bijl and G. Shawcross, "Housing site layout system," Com-
puter Aided Design, vol. 7, no. 1, Jan. 1975.

REFERENCES
[1] ANSI, "Interim report from the study group on database manage-
ment systems," Bull. SIGMOD, vol. 7, no. 2, 1973.
[2] W. E. Armstrong, "Dependency structures for database relation-
ships," Information Processing '74, IFIP (1974). Amsterdam,
The Netherlands: North-Holland Press, pp. 998-1006.
[3] A. E. Bandurski and D. Jefferson, "Enhancements to the DBTG
model for computer-aided ship design," in Proc. ACM Workshop
on Databases for Interactive Design (Univ. Waterloo, Water-
loo, Ont.), 1975.
[4] A. E. Bandurski and D. Jefferson, "Data description for computer
aided design," in Proc. ACM-SIGMOD Conf. Management of Data
(San Jose, CA), 1975.
[5] A. Baer, C. Eastman, and M. Henrion, "Geometric modeling: A
survey," Computer Aided Design, vol. 11, no. 5, pp. 253-272,
Sept. 1979.
[6] A. Baer and C. Eastman, "The consistency of integrated databases
for computer aided design," in Proc. Workshop on Computer
Representation of Physical Systems (Carnegie-Mellon Univ., Pitts-
burgh, PA), Aug. 1976.
[7] C. Beerl, F. Bernstein, and N. Goodman, "A sophisticate's intro-
duction to database normalization theory," in Proc. 4th Int. Conf.
Very Large Databases (ACM, NY), pp. 113-124, 1977.
[8] J. Bentley and D. Wood, "An optimal worst-case algorithm for re-
porting intersections of rectangles," Dep. Comput. Sci., Carnegie-
Mellon Univ., Res. Rep. CS-79-122, 1979.
[9] L. A. Bjork, "Generalized audit trail requirements and concepts
for database applications," IBM Syst. J., vol. 14, no. 3, pp. 229-
245, 1975.
[10] J. W. Boyce, "Interference detection among solids and surfaces," Comm.
un. ACM, vol. 22, no. 1, pp. 3-9, Jan. 1979.
[11] R. R. Breland and R. A. Frieden, "Designer's workbench: The
user environment," Bell Telephone Tech. J., vol. 59, no. 9, pp.
1767-1792, Nov. 1980.
[12] G. A. Butlin, "Interactive graphics for finite elements," presented at
Finite Element Symp., Atlas Computer Lab., Chilton, U.K.,
Mar. 1974.
[13] CODASYL, Database Task Group April 1971 Report. New York:
ACM, 1971.
[14] E. F. Codd, "A relational model of data for large shared data
bank," Commun. ACM, vol. 13, no. 6, pp. 377-387, 1970.
[15] E. F. Codd, "Further normalization of the data base relational
model," in Proc. 2nd Annual Conf. Comput. Sci. Symp. 6: Database
Systems (New York), 1971.
[16] O. J. Dahl and C. A. R. Hoare, "Hierarchical program structures," in
Structured Programming, O. J. Dahl, E. W. Dijkstra, and C. A. R.
Hoare, Eds. New York: Academic Press, 1972.
[17] J. C. Date, Introduction To Database Systems. Reading, MA: Addison-
Wesley, 1975.
[18] C. Eastman, "System facilities for CAD databases," in Proc. 17th
ACM Design Automation Conf., June 1980.
[19] ——, A prototype building description system," in Proc. CAD 80
Conf., Brighton, England: IPC Press, 1980.
[20] ——, "Representation of design problems and maintenance of
their structure," in Proc. Working Conf. Applications of Artificial
Intelligence to CAD (Grenoble, France), 1978.
[21] ——, General purpose building description systems," Computer
Aided Design, vol. 8, no. 1, pp. 17-26, Jan. 1976.
[22] C. Eastman and G. Lafue, "Semantique integrity transactions in
design databases," in Proc. Building Sci., Carnegie-Mellon Univ.,
Pittsburgh, PA, Res. Rep. 13, Mar. 1981.
[23] C. Eastman and Y. Yasky, "The integrated building model and
database schema for the second phase of integrated CAEADS,"
CAD-Graphics Laboratory, Inst. Building Sci., Carnegie-Mellon
Univ., Pittsburgh, PA, Res. Rep., Jan. 1981.
[24] C. Eastman and R. Thornton, "A report on the GLIDE2 language
definition," preliminary draft, Inst. Physical Planning, Carnegie-
Mellon Univ., Pittsburgh, PA, Mar. 22, 1979.
[25] S. M. Elmanian and S. G. H. 71, "Presentation and consis-
tency maintenance needs in engineering databases," presented at
Conf. Engineering and Scientific Data Management, Hampton,
VA, May 1978.
[26] C. Eastman and J. Lividini, "Spatial search," Inst. Physical
Planning, Carnegie-Mellon Univ., Pittsburgh, PA, Res. Rep. 55,
May 1979.
[27] K. P. Eswaren, J. Gray, R. Lorie, and I. Traiger, "The notions of
consistency and predicate locks in a data base system," IBM Res.
Rep. RJ47, San Jose, CA, Dec. 1975.
[28] S. Fenvs, "Integration of database management and project control
for engineering design," in Proc. ACM Workshop on Databases
for Interactive Design, New York, 1975.
[29] W. E. Fischer, "PHIDAS—A database management system for
CAD/CAM application software," Computer Aided Design,
vol. 11, no. 3, pp. 146-150, May 1979.
[30] B. Fredrikson, J. Mackere, and B.G.A. Persson, "Finite-element
programs in integrated software for structural mechanics and
CAD," Computer Aided Design, vol. 13, no. 1, pp. 27-39, Jan.
1981.
[31] E. M. Hoskins, "Computer aids in building," Computer Aided
Design, J. Vlletstra and R. F. Wellins, Eds. New York: Ameri-
can Elsevier, 1973.
[32] H. R. Johnson, D. Comfort, and D. Shull, "An engineering data
management system for IPAD," in Proc. IPAD Nat. Symp.,
NASA Conf. Publ. 2143, pp. 145-178, Sept. 1980.
[33] H. A. Kanel and Shanta, "A solid mesh generation and result
display package," J. Pressure Vessel Tech., ASME, vol. 96, no. 3,
pp. 207-312, Aug. 1974.
[34] J. H. Kim and D. Siewiorek, "Lauises in IC implementation of high
level, abstract designs," in Proc. 17th Design Automation Conf.,
p. 85-91, June 1-4, 1981.
[35] W. Kim, "Relational database systems," ACM Comp. Surveys,
vol. 11, no. 3, pp. 185-212, Sept. 1979.
[36] G. Lafue, "An approach to automatic checking of semantic
integrity in design databases," Ph.D. dissertation, School of
Urban and Public Affairs, Carnegie-Mellon Univ., Pittsburgh,
PA, 1980.
[37] ——, "Integrating language and database for CAD applications,
Computer Aided Design, vol. 11, no. 3, pp. 127-130, May 1979.
[38] M. F. Leesley, A. Buchmann, and D. Mulraey, "An approach to
a largely integrated system for computer aided design of chemical
process plants," in Proc. Int. Congr. Computers to the Develop-
ment of Chemical Engineering and Industrial Chemistry (Paris, France), Mar. 1978.

[39] P. C. Lockemann, H. C. Mayr, W. H. Weil, and W. H. Wohllebe, "Data abstractions for database systems," ACM TODS, vol. 4, no. 1, pp. 60-75, Mar. 1979.

[40] D. Maier, A. Mendelson, and Y. Sagiv, "Testing the implications of data dependencies," ACM TODS, vol. 4, no. 4, pp. 455-469, Dec. 1979.

[41] G. S. Malenson and S. A. Spurulin, "A fundamental approach to database implementation for design automation," in Proc. ACM Workshop on Databases for Interactive Design, pp. 120-129, Sept. 1975.

[42] J. Martin, Computer Database Organization. New York: McGraw-Hill, 1975.

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

[44] D. J. McLeod, "High level expression of semantic integrity specifications in a relational database system," M.I.T. Lab. Computer Science, Cambridge, MA, Tech. Rep. TR-165, 1976.

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

[46] R. E. Miller, J. Southall, and S. Wahlstrom, "Requirements for management of aerospace engineering data," Comput. Structures, vol. 10, pp. 45-52, 1979.

[47] W. Newman and R. Sproull, Principles Of Interactive Computer Graphics, 2nd Ed. New York: McGraw-Hill, 1979.

[48] L. A. O'Neil et al., "The designer's workbench," Bell Syst. Tech. J., vol. 59, no. 9, Nov. 1980.

[49] D. L. Parma, "Information distribution aspects of design methodology," in Proc. IFIP Congress 1971. Amsterdam, The Netherlands: North-Holland, 1971, pp. 339-344.

[50] R. L. Patrick, Security System Review Manual. Montvale, NJ: AFIPS Press, Oct. 1974.

[51] K. Pless, "Engineering design viewed as an activity in artificial intelligence," SRI Int., Palo Alto, CA, Tech. Note 167, July 24, 1978.

[52] P. Reisner, R. Boyce, and D. Chamberlin, "Human factors evaluation of two database query languages—Square and Sequeal," in Proc. 1975 AFIPS NCC, pp. 447-452, 1975.

[53] D. T. Roos, Ices Systems Design. Cambridge, MA: M.I.T. Press, 1967.

[54] J. B. Rothnie, P. Bernstein, S. Fox, et al., "Introduction to a system for distributed databases (SDD-1)," ACM TODS, vol. 5, no. 1, pp. 1-17, Mar. 1980.

[55] L. Rowe and K. Shoens, "Data abstraction, views and updates in RIGEL," in Proc. 1979 ACM SIGMOD Conf. Management of Data, May 1979.

[56] T. W. Sidle, "Weaknesses of commercial database management systems in engineering applications," Proc. 17th ACM Design Automation Conf., pp. 57-61, June 1980.

[57] J. M. Smith and D. C. Smith, "Data abstractions: aggregation and generalization," ACM TODS, vol. 2, no. 2, pp. 125-133, June 1977.

[58] G. J. Sussman and G. L. Steele, "CONSTRAINTS—A language for expressing almost hierarchical descriptions," Artificial Intell., vol. 14, no. 1, pp. 1-40, Aug. 1980.

[59] R. W. Taylor and R. Frank, "CODASYL database management systems," ACM Comp. Surveys, vol. 8, no. 1, pp. 67-104, Mar. 1976.

[60] M. Tsubaki, "DPLS: Database, dynamic program control and open-ended POL support," in Proc. ACM Workshop on Databases for Interactive Design, pp. 146-160, Sept. 1975.

[61] H. Weber, "A software engineering view of database systems," in Proc. 4th IEEE VLDB Conf., pp. 36-52, 1978.

[62] G. Wiederhold, Database Design. New York: McGraw-Hill, 1977.

[63] Y. Yasky, "A consistent database for an integrated CAAD system: Fundamentals for an automated design assistant," Ph.D. dissertation, Dep. Architecture, Carnegie-Mellon Univ., Pittsburgh, PA, 1981.

[64] M. Zloof, "Query by example," in Proc. 1975 AFIPS NCC, pp. 431-438, 1975.