An Analysis of Geometric Modeling in Database Systems

ALFONS KEMPER and MECHTILD WALLRATH

Universität Karlsruhe, Institut für Informatik II, D-7500 Karlsruhe, West Germany

The data-modeling and computational requirements for integrated computer aided manufacturing (CAM) databases are analyzed, and the most common representation schemes for modeling solid geometric objects in a computer are described. The *primitive instancing* model, the *boundary representation*, and the *constructive solid geometry* model are presented from the viewpoint of database representation. Depending on the representation scheme, one can apply geometric transformations to the stored geometric objects. The standard transformations, scaling, translation, and rotation, are outlined with respect to the data structure aspects. Some of the more recent developments in the area of engineering databases with regard to supporting these representation schemes are then explored, and a classification scheme for technical database management systems is presented that distinguishes the systems according to their level of object orientation: *structural* or *behavioral object orientation*. First, several systems that are extensions to the relational model are surveyed, then the functional data model DAPLEX, the nonnormalized relational model NF^2, and the database system R^2D^2 that provides abstract data types in the NF^2 model are described.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types*; H.2.1 [**Database Management**]: Logical Design—*data models*; Languages—*data description languages (DDL)*; *data manipulation languages (DML)*; *query languages*; J.6 [**Computer Applications**]: Computer-Aided Engineering—*computer-aided manufacturing*; I.1.3.5 [**Computer Graphics**]: Computational Geometry and Object Modeling—*hierarchy and geometric transformation*

General Terms: Design, Languages

Additional Key Words and Phrases: Engineering database systems, geometric modeling, object-oriented database systems

INTRODUCTION

Motivation

The last few years have shown a rapid increase in the use of robots in mechanical assembly, and we predict an even larger trend toward computer-aided manufacturing (CAM) in the future, at least in the industrialized nations. As pointed out by Requicha [1980], the major breakthrough in fully automated assembly has yet to come. It is argued that software is the real bottleneck in robotics, very much as with other computerized systems. The technology of robots is much more advanced than the methods for programming them.

CONTENTS

The traditional approach to robot programming consists of manually leading the robot through all the assembly operations. This method could be called "programming by example." Every robot operation that is required for the assembly process is executed once and stored for repetitive execution during the actual assembly operation. The main disadvantage of this approach is that it ties a robot to the assembly environment during the development phase of the application.

A second approach, consisting of programming the robotic application off line [Wesley 1980], is at a much higher programming level than the traditional programming-by-example method. It is still in its infancy and is an active research issue. This approach frees the robot, as well as the workspace of the robot, from the development phase of the assembly process. The robot is programmed in an assembly-oriented language, where a sample instruction might be

mount cog wheel x on shaft y

This high-level assembly program has to be translated into a robot motion program that specifies exactly where to grasp the cog wheel x and where the shaft y is located in the workspace. Furthermore, a path has to be selected to get to the position of the object x and back to object y without colliding with any other fixtures in the workspace.

Because all these computations are performed off line, that is, the robot as well as the workspace is simulated, we need a precise model of the robot and its surrounding workspace in order to simulate the assembly operation. Figure 1 is a schematic rendering of an off-line integrated robotics programming system according to Wesley [1980] and Blume et al. [1983].

The central part of such an integrated robotics programming system forms a comprehensive database that stores the so-called world model by describing the physical and geometric properties of real objects. The physical data describe such aspects as the material of which an object is composed and are entered via a geometric design processor that allows the engineer to specify and manipulate real-world objects interactively.

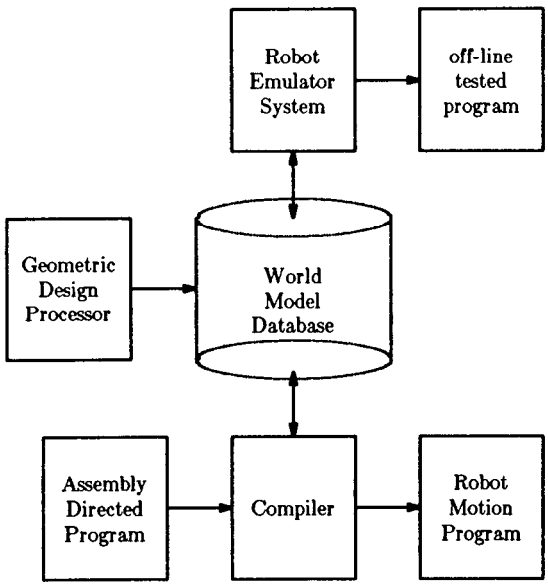


Figure 1. Integrated robotics database system.

Figure 1 shows the two other main modules that interface to the world model database:

(1) the robot emulator system;
(2) the compiler.

The robot emulator system is used to simulate existing robot motion programs for validation purposes. This is especially important if the robots are used in highly sensitive application areas, such as nuclear power plants, where any undetected program error could lead to very dangerous situations.

The second module, the compiler, gets an assembly-directed program as input. From the world model database the compiler deduces how to translate the assembly-oriented commands into robot motions. Of course, the world model is not a static database; rather, it is dynamic, that is, it is manipulated according to robot operations. The real-world assembly process is simulated by manipulating the database accordingly.

Even though the world model database forms the central part of the integrated robotics simulation system, it has traditionally received the least attention. All known commercially available CAD/CAM systems for robotics are based on a customized file system rather than a comprehensive database management system. Their main disadvantage is that there is no generally accepted format to which other modules can interface. To manipulate the data obtained by one CAD/CAM module by some other module generally requires tedious conversion of the data.

Why have database management systems not been employed? The answer to this question is manyfold. (1) Today's commercially available DBMSs, which are designed for highly structured commercial database applications, do not adequately support technical problem domains [Lockemann et al. 1985]. (2) It is not clear that we can achieve the same efficiency that is possible with a special-purpose file structure with currently available general-purpose DBMSs. (3) The CAD/CAM systems are usually designed and implemented by engineers who are not necessarily database experts. Database experts have traditionally ignored

technical problem domains. Only recently has there been a shift of research activities within the database community toward engineering applications.

Focus

We first want to analyze the requirements imposed on database management systems by computer-aided manufacturing applications. We begin by describing the more important representation schemes for solid geometric objects as they occur in the robotics world. This investigation is carried out primarily from a database point of view rather than by presenting a rigorous mathematical definition of the representation schemes. Section 2 describes the geometric transformations that can be applied to solid objects stored in the world model database. Section 3 presents a classification scheme for technical database systems and reviews some of the more recent proposals for engineering databases with respect to their suitability for integrated robotics databases. The first systems that we survey are extensions to the relational model. Then we investigate the functional data model DAPLEX as one representative of the object-oriented approach. The NF^2 model is a nonnormalized relational model that allows nested relations. R^2D^2 (*R*elational *R*obotics *D*atabase System with Extensible *D*ata Types) is a database system that is based on the NF^2 model and allows the database user to define application-specific data types and operations. The systems are described by defining a sample schema of some geometric representation model. Section 4 summarizes the main results of our investigation.

1. REPRESENTATION SCHEMES

Robots manipulate solid geometric objects. Thus the basis for any automated assembly operation by robots is a way of storing information about geometric objects in a computer. There are several quite different representation methods for solid objects. Some of them are investigated in this section. We do not attempt to give a formal or complete definition of all existing representation schemes for three-dimensional solid objects. Rather, we restrict ourselves to outlining the most important schemes. Only those aspects of importance to the design of database support of the particular representation are described. A more theoretical overview is provided by Requicha [1980].

There are three representation schemes for which database support is feasible [Maier 1985]:

(1) primitive instancing;
(2) constructive solid geometry (CSG);
(3) boundary representation (BR).

Our presentation is based primarily on the example geometric object of Figure 2, a bracket with four holes that frequently occurs in assembly operations. Even though this example object is a fairly simple one, it should suffice to demonstrate the main characteristics of the three representation schemes.

1.1 Primitive Instancing

In this approach every geometric object is defined as a special instance of a generic primitive object. In relational database terminology this means that one would create a relation for every generic object type. The attributes of the relation would correspond to the parameters that describe the geometric object. Each geometric object would then be stored as a tuple of the relation corresponding to the generic object class.

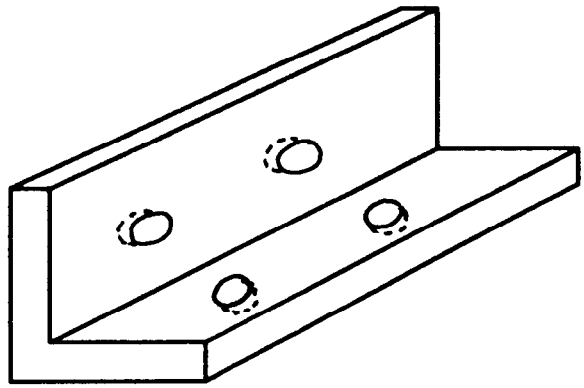


Figure 2. Bracket with four holes.

An example of a generic object class might be brackets with holes, as shown in Figure 2.

Now let us consider a generic record type that would describe the object class bracket with a variable number of holes. The record type would be defined as follows:

```
generic type BRACKET (#holes:integer)
  length: real
  width: real
  height: real
  material: {iron, copper, . . .}
  . . .
  HOLES: array [1 . . #holes] of
  record
    diameter: real
    location: array [1 . .3] of real
  end record
end generic type BRACKET
```

A particular object of type BRACKET with four holes is instantiated as follows:

```
create BRACKET(4)
```

The reader will note that this is a very simple representation for a number of well-known and highly structured assembly objects, such as brackets, nuts, cog wheels, and shafts. As is pointed out in the literature [Voelcker and Requicha 1977], however, the majority of mechanical objects are produced only in relatively small quantities, on the order of 500, say. This means that the number of instances of a particular object class is fairly small, whereas there is usually a large number of different generic object classes. The primitive instancing approach is not useful in such applications since it requires the specification of a generic record type for each different object class. In database terms this means that we would have to create an abundance of different relations, each consisting of only a small number of tuples. For this reason this approach is not always usable in a general-purpose CAM system.

1.2 Constructive Solid Geometry

Together with the boundary representation, the CSG scheme is the most widely used representation in existing CAD/CAM systems. It is possible to transform a CSG representation to a BR representation automatically. Many existing systems

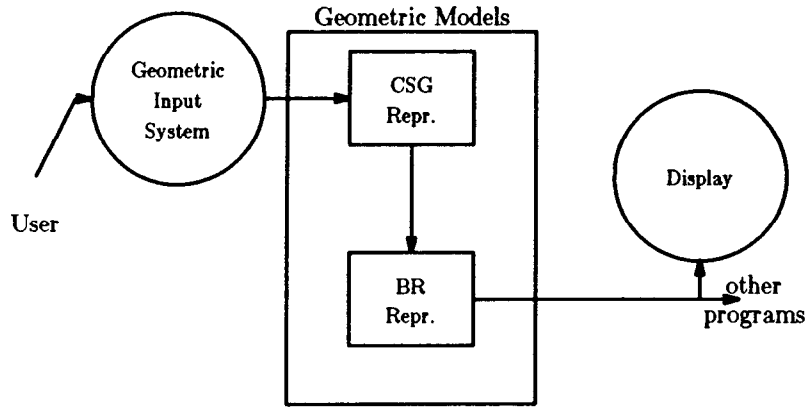


Figure 3. Typical architecture of a geometric modeling system.

[Requicha 1980] are organized as shown in Figure 3. The input to the geometric modeling system is usually via the CSG representation, which is much easier for the user, that is, the engineer, to handle than is the boundary representation. Internally, the CSG representation is automatically transformed into the boundary representation.

The CSG scheme is a volumetric representation of geometric objects, in which an object is described as a composition of a few primitive objects. The composition is achieved via motional or combinatorial operators. Example operators are the (regularized) union, intersection, and difference of two solid objects. Motional operators are, for example, "rotate" and "scale". The description of a geometric object in CSG format is a tree defined by the following context-free grammar:

```
<mechanical part> ::= <object>
<object>          ::= <primitive> |
                      <object> <motion op> <motion argument>|
                      <object> <set operator> <object>
<primitive>       ::= cube | cylinder | cone | ...
<motion op>       ::= rotate | scale | ...
<set operator>    ::= union | intersection | difference | ...
```

In Figure 4 we show a CSG tree for our example object "bracket with 4 holes." In the CSG tree each nonterminal node represents an operation, either a rigid motion or a combinatorial (set) operator. Terminal nodes either represent a motion argument or a primitive object. Each primitive object is described by its parameters, such as length, width, and height, as well as its relative position.

In our example we have only two primitive objects: cuboid and cylinder. A cuboid is defined by its length, width, and height. A cylinder is defined by its radius and length.

We notice that, in contrast to the primitive instancing scheme, the CSG representation requires only a few primitive objects. Therefore the CSG tree of complex objects can become very deep, which might lead to inefficient data retrieval if there is no suitable data access support.

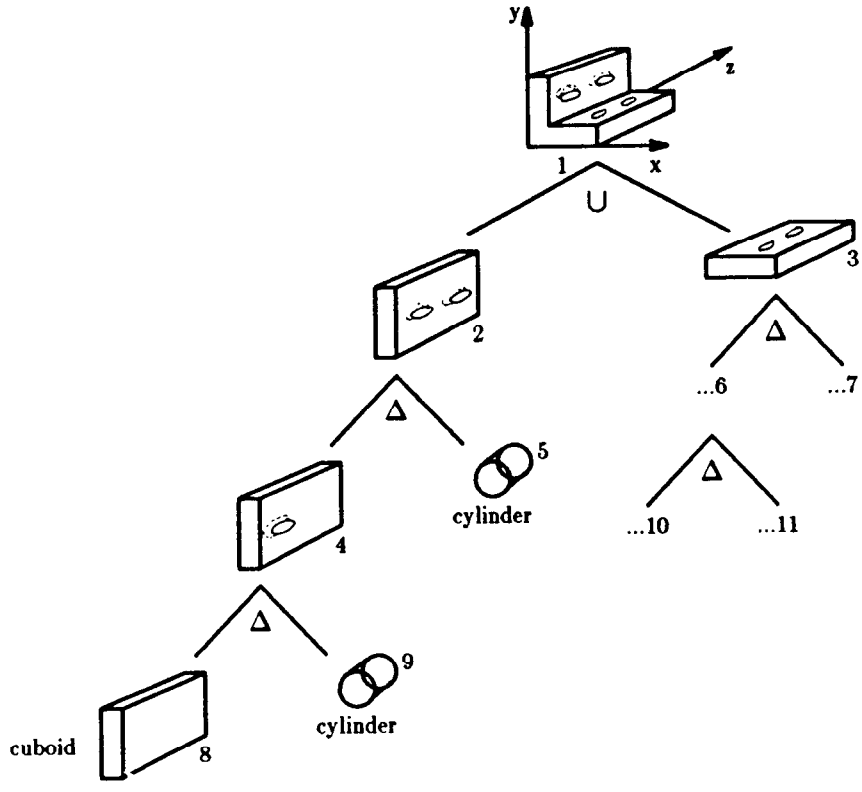


Figure 4. CSG tree of the bracket.

1.3 Boundary Representation

In this representation scheme a solid object is segmented into its nonoverlapping faces. Each face in turn is modeled by its bounding edges and vertices. Again we present the representation of our bracket in Figure 5.

From a database point of view we note that this representation scheme consists of different abstraction levels, that is, faces, edges, and vertices. In contrast to the CSG scheme, the depth of the tree is constant, that is, 3. A more complex solid object just leads to more nodes in the tree without increasing the depth.

The lowest level of the tree stores the metric information, that is, three-tuples (x_i, y_i, z_i) for vertex v_i, for i in $\{1, \ldots, m\}$. The second level of the tree stores the edges as combinations of vertices. Edge e_i is represented by the tuple (v_{i1}, v_{i2}), where i in $\{1, \ldots, n\}$. On the topmost level of the tree each node describes a *variable* number of edges which represent the boundaries of one face of the rigid object.

2. GEOMETRIC TRANSFORMATIONS

The two most important representation models for rigid solids are the constructive solid geometry model (CSG) and the boundary representation (BR). To display the edges of a three-dimensional solid on a computer display, the boundary representation is much easier to handle. Many commercially available

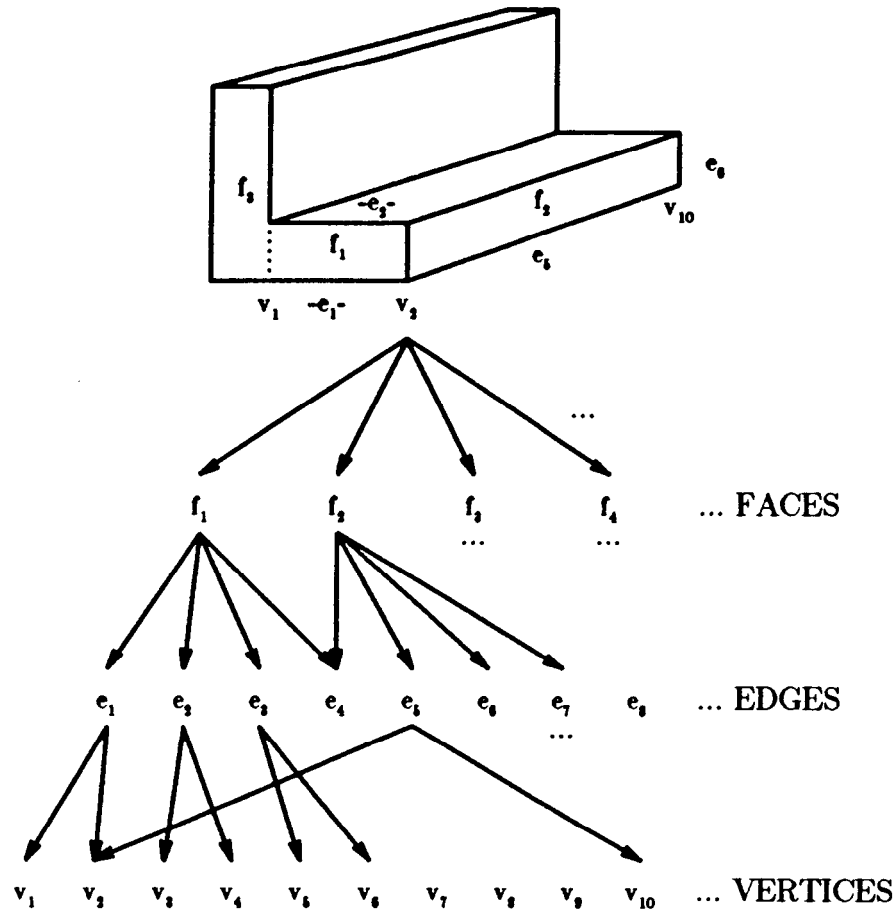


Figure 5. Boundary representation of the bracket.

three-dimensional modelers employ the CSG method for inputting an object, but can automatically transform the representation to BR format, as was shown in Figure 3.

In this section we give the reader a brief introduction to the area of computer geometry. Unfortunately this presentation does not allow us to give a detailed treatment of this problem domain. We refer the reader who wants more details to the book by Foley and van Dam [1983]. In this section we merely outline the computational requirements imposed on the geometric modeling system by geometric transformations.

A graphical display of a geometric object, in this case a cuboid, is shown in Figure 6. Except for references to faces and edges, the only data that are stored in the boundary representation are vertices in the three-dimensional space, that is, vectors of the form (x, y, z). To uniquely describe the cuboid, one has to store the eight vertices $v_1, \ldots, v_8$. The corresponding *boundary representation* of this cuboid is depicted in Figure 7.

In order to be able to view an object from different perspectives (angles) and to zoom in and out on the particular object, one can apply three geometric

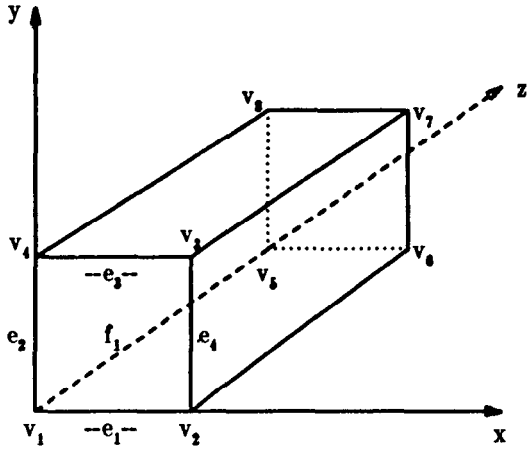


Figure 6. Projection of a cuboid on a display.

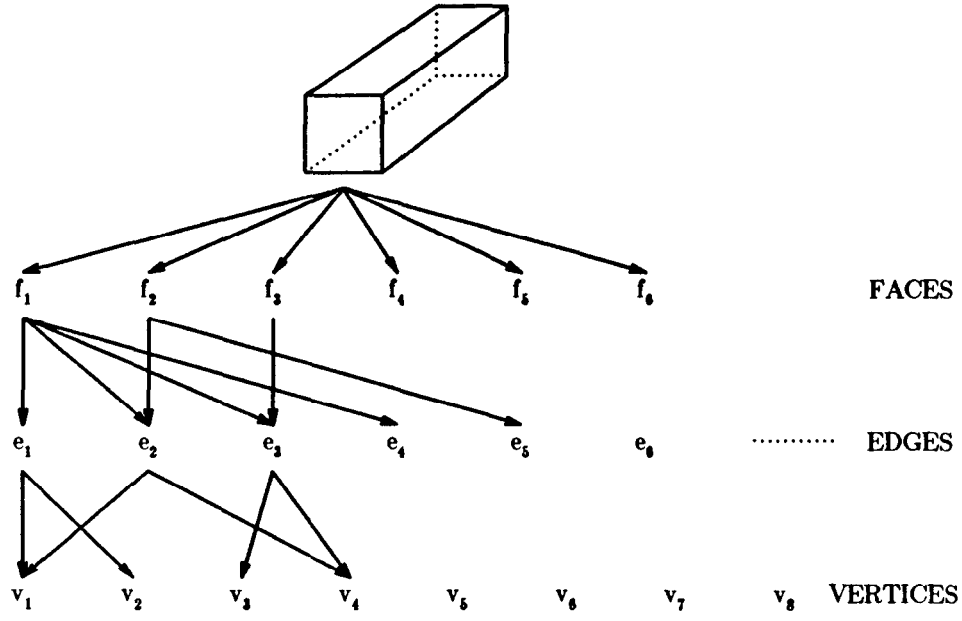


Figure 7. Boundary representation of the cuboid.

transformations to the object stored in BR representation:

- translation;
- rotation;
- scaling.

We briefly explain each of these transformations in turn.

2.1 Translation

Translation corresponds to moving the graphical object within the three-dimensional coordinate system relative to the origin without altering the object's orientation. This is achieved by rotations, which are explained later. A translation

is defined by the translation vector $T = (D_x, D_y, D_z)$. A single vertex is translated by adding the translation vector to the vector representing the vertex in the three-dimensional system:

$$v_i = (x_i, y_i, z_i),$$
$$T = (D_x, D_y, D_z),$$
$$T(v_i) := v_i + T = (x_i + D_x, y_i + D_y, z_i + D_z).$$

To translate a geometric object represented in boundary representation requires translating all vertices of the object that are stored in the BR schema. Thus, for the example of the cuboid, one would have to carry out the following computation:

```
...
for all v_i in {v_1, ..., v_8} do
  v_i := v_i + T;
```

2.2 Homogeneous Coordinates

The other two transformation operations, that is, scaling and rotation, can be defined naturally as multiplications of the vertex (vector) with a corresponding transformation matrix, as we show below. In order to be able to combine different transformations of the same object, for example, rotation and translation, we would like to also represent translation as a matrix multiplication. Then we would be able to combine different transformation matrices by multiplying them.

In order to represent the translation also as a matrix multiplication, the concept of homogeneous coordinates has to be employed, as is done in many graphics packages [Foley and van Dam 1983]. This concept requires a vertex to be stored as a four-element, rather than a three-element, vector. Then vertex v_i is represented as

$$v_i = [x_i, y_i, z_i, 1].$$

Now the translation matrix T looks as follows:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ D_x & D_y & D_z & 1 \end{bmatrix}.$$

The translation of the vertex v_i is then defined as

$$T(v_i) = [x_i, y_i, z_i, 1] * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ D_x & D_y & D_z & 1 \end{bmatrix} = [x_i + D_x, y_i + D_y, z_i + D_z, 1].$$

Translation of the cuboid would then result in the following program fragment:

```
...
for all v_i in {v_1, ..., v_8} do
  v_i := v_i * T;
```

2.3 Scaling

An important concept in viewing geometric objects on a computer display is varying the size in form of scaling (or stretching). Vertices (as endpoints of vectors) can be scaled by S_x along the x-axis, S_y along the y-axis, and S_x along

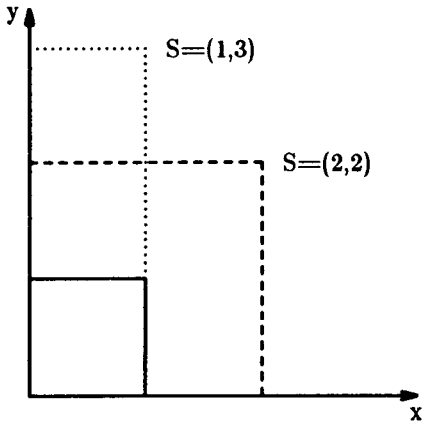


Figure 8. Scaling of a two-dimensional object.

the z-axis, according to the scaling matrix S as follows:

$$S(v_i) = [x_i, y_i, z_i, 1] * \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = [S_x * x_i, S_y * y_i, S_z * z_i, 1].$$

Scaling of a two-dimensional object is shown in Figure 8. Scaling of a geometric object is carried out by scaling each surrounding edge of the object representation in BR format, which is equivalent to scaling each vertex of the BR representation. Thus the following program would scale the cuboid of Figure 2:

```
...
for all v_i in {v_1, ..., v_8} do
  v_i := v_i * S;
```

where

$$v_i = [x_i, y_i, z_i, 1],$$

$$S = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

and the centered asterisk ($*$) corresponds to matrix multiplication.

2.4 Rotation

Rotations are used to change the orientation of geometric objects in the three-dimensional space. This way one can provide a view of the objects from all different angles. Graphically, rotation of a cuboid is shown in Figure 9.

In three dimensions we have to distinguish three kinds of rotations:

- rotation about the z-axis;
- rotation about the x-axis;
- rotation about the y-axis.

Here we show only briefly the definition of rotation about the z-axis. The interested reader is referred for more detail to Foley and van Dam [1983].

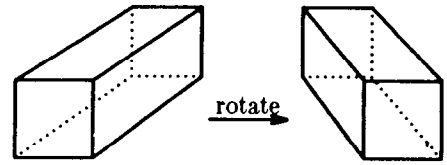


Figure 9. Rotation of the cuboid.

A rotation about the z-axis is defined by the rotation angle Φ. Corresponding to this angle the rotation matrix $R_z(\Phi)$ is constructed as

$$R_z(\Phi) = \begin{bmatrix} \cos\Phi & \sin\Phi & 0 & 0 \\ -\sin\Phi & \cos\Phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

The rotation of vertex v_i is then given as

$$[x_i, y_i, z_i, 1] * \begin{bmatrix} \cos\Phi & \sin\Phi & 0 & 0 \\ -\sin\Phi & \cos\Phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= [x_i * \cos\Phi - y_i * \sin\Phi,\ x_i * \sin\Phi + y_i * \cos\Phi,\ z_i,\ 1].$$

Rotation of a geometric object stored in boundary representation is carried out analogously to scaling and translation; that is, each vertex has to be rotated. The program fragment is shown below:

```
...
for all v_i in {v_1, ..., v_8} do
  v_i := v_i * R_z(Φ);
```

2.5 Simulation of Assembly Operations as Geometric Transformations

As described in Figure 1, the world model database forms the central part of a robot programming system. One major task in such a system is to simulate off-line robotics operations, for example, assembly operations of the form

mount cog wheel x on shaft y

The standard geometric transformations described above are the operations used to model such an operation. We assume that object x (the cog wheel) exists at some location in the world of this robot application, that is, it exists in the world model database. The same holds true for object y, the shaft onto which x has to be mounted. Simulating this assembly operation means, in terms of the world model database, changing the location of object x. In our particular case this is achieved (not regarding the problem of collision handling) by the following (standard) geometric operations:

1. Translate by T_1 (*pick up object* x).
2. Rotate about the z-axis by $R_z(\Phi)$.
3. Rotate about the x-axis by $R_x(\Theta)$ (*rotate* x).
4. Rotate about the y-axis by $R_y(\Gamma)$.
5. Translate again by T_2 (*mount object* x *on* y).

The following program fragment would achieve this transformation:

```
for all v_i in {v_1, v_2, v_3, ...} do
  begin
  v_i := v_i * T_1;
  v_i := v_i * R_z(Φ);
  v_i := v_i * R_x(Θ);
  v_i := v_i * R_y(Γ);
  v_i := v_i * T_2;
  end
```

This results in altogether $5 * j$ matrix multiplications, where j is the number of vertices used to describe object x. For example, if 8 vertices are used to model the object, the above method would result in 40 multiplications. A much more efficient way combines the transformation matrices T_1, $R_z(\Phi)$, $R_x(\Theta)$, $R_y(\Gamma)$ into one matrix M:

```
M := T_1 * (R_z(Φ) * (R_x(Θ) * (R_y(Γ) * T_2)))
for all v_i in {v_1, v_2, v_3, ...} do
  v_i := v_i * M;
```

This method results in $5 + j$ matrix multiplications; that is, for an object with 8 vertices, one needs 13 multiplications.

3. SURVEY OF PROPOSALS FOR ENGINEERING DATABASES

In this section we first sketch the relational database model for those readers who are not familiar with databases. Then we introduce the notion of object orientation in database systems, which is applied to characterize the engineering database systems investigated in the remainder of this section.

3.1 The (Pure) Relational Database Systems

In the introduction to this paper we cited a few reasons why database management systems have not been extensively used in technical applications. The main reason is that the data-modeling capabilities of the traditional database systems are insufficient for engineering applications. For example, in the relational database model [Codd 1970] technical objects usually have to be decomposed onto different relations. Let us illustrate this on a relational BR schema that is described below.

Mech_Part

ID	FACES
cuboid	f1
cuboid	f2
pyramid	f101

FACES

ID	EDGES
f1	e1
f1	e2
...	...

EDGES

ID	VERTICES
e1	v1
e1	v2
e2	v1

VERTICES

ID	X	Y	Z
v1	0	0	0
v2	...	...	...
v3	...	...	...

We notice that this representation is broken up into four different relations, where the relationship of the tuples of the various relations is achieved via user-generated attribute values.[1] This makes the model difficult to use by the database user, that is, the engineer, in order to retrieve and manipulate the data because it requires an intrinsic knowledge of the underlying schema definition. In order to retrieve all the bounding vertices of the mechanical part "cuboid," one could formulate the following SQL [IBM 1981] or QUEL [Stonebraker et al. 1976] queries:

```
select Mech_Part.ID,X,Y,Z
from Mech_Part,FACES,EDGES,VERTICES
where Mech_Part.FACES = FACES.ID
  and FACES.EDGES = EDGES.ID
  and EDGES.VERTICES = VERTICES.ID
  and Mech_Part.ID = "cuboid"
```

```
range of m is Mech_Part
range of f is FACES
range of e is EDGES
range of v is VERTICES
    retrieve (m.ID,v.X,v.Y,v.Z)
    where m.FACES = f.ID
          and f.EDGES = e.ID
          and e.VERTICES = v.ID
          and m.ID = "cuboid"
```

These queries involve joining the four relations Mech_Part, FACES, EDGES, and VERTICES. Adequately supporting such frequent join operations seems to be the major issue in extending the (pure) relational model for engineering use [Lorie 1982; Lorie and Plouffe 1983].

3.2 Object Orientation: A Classification Scheme for Engineering Databases

In summary one could state that the problems with traditional database management systems stem from the fact that they do not allow the modeling of engineering objects in a natural way—or at least they do not support the retrieval and manipulation of such objects in a way that is familiar to engineers. In particular, they do not handle technical objects as a whole database entity; rather, they require a schema design that is imposed by the underlying data model but does not necessarily constitute a natural mapping of technical objects on database structures.

Object-oriented database systems have been proposed by many authors as a new concept for supporting technical applications. In the database area in particular, two kinds of object orientation should be distinguished [Dittrich 1986]: the *structural* and the *behavioral* object orientation.

The structural approach has originated from database technology and is essentially motivated on technical grounds. The central notion here is that of a "complex object" [Lorie and Plouffe 1983] or of a "molecule" [Batory and Kim 1985], reflecting the fact that objects in the engineering world are composed of parts that may among themselves undergo a variety of other relationships. Typical approaches are based on hierarchical extensions to the relational model, such as XSQL [Haskin and Lorie 1982] or the NF^2 data model [Schek and Pistor 1982; Lum et al. 1985; Dadam et al. 1986], and extensions to the entity–relationship model [Zaniola 1983; Glinz et al. 1985; Dittrich et al. 1986].

[1] The attributes named ID do not constitute keys of the relation. For example, in the relation FACES the attribute ID is just used to uniquely identify an object that represents a face of the mechanical part.

Structurally object-oriented data models provide facilities for mapping complex objects onto database structures and for retrieving these objects as entities, but they usually lack constructs to define manipulations of these objects in a manner that is familiar to engineering users.

The behavioral approach has a more application-oriented flavor. The identification of an object is largely determined by what a user perceives to be an entity that, at least at times, can be manipulated as a whole. In such an abstract view, data manipulation is object-type specific by necessity. Take as examples a geometric object that is to be rotated in space or attached to another such object, or an image that is to be searched for the occurrence of a particular pictorial pattern or overlaid with another image. The behavioral approach to databases has its origins in programming languages, particularly the notion of abstract data type. Lately, considerable work in this area has been reported in the database literature [Maier et al. 1985; Atwood 1985; Zdonik and Wegner 1986; Zaniola et al. 1986].

One approach for a behaviorally object-oriented CAD system is reported in Eastman [1981, 1986]. GLIDE is a Pascal extension that incorporates permanent data and provides language constructs for geometric modeling, graphical input and display functions, and a user-oriented command language. Thus GLIDE allows the user to manipulate permanent data objects by application-specific operators. The concept of data abstraction is even more central in the successor system FORM:ULAE: [Eastman 1986; Eastman and Kulay 1985], which is an extension of GLIDE to the extent that it allows the embedding of external abstract data types. In particular, the system supports the development of abstraction hierarchies by stepwise refinement of abstract data types. Restricting the manipulation of abstract data objects to those operations that are predefined for the abstract data type provides a powerful tool for integrity management since it avoids any inconsistent manipulations.

Whereas Eastman's work concentrates on the programming language aspects of CAD systems, we analyze several recent proposals for object-oriented database systems with respect to geometric modeling, all of which evolved out of the relational database model [Codd 1970] and were intended for the so-called nontraditional applications, that is, applications that do not belong to the traditional business domain. We apply our classification scheme to each proposed system and discuss what level of object orientation the particular model provides.

3.3 QUEL as a Datatype

"QUEL as a Datatype" was proposed by Stonebraker et al. [1983b], as an extension to the database management system INGRES [Stonebraker et al. 1976]. It allows attributes of relations to be of type QUEL; that is, the attribute consists of a QUEL query that retrieves tuples from one or more different relations. The purpose of this extension is to provide a very general referencing mechanism. The database designer could define new objects, such as vectors, cubes, and arrays, in separate relations and access them from the parent relation via an attribute of type QUEL.

3.3.1 Constructive Solid Geometry

We define a CSG schema in "QUEL as a Datatype" as shown in Figure 10 [Lee and Fu 1983]. Mechanical_part is the root relation and contains information about the assembly part as a whole. In our case the assembly part is the bracket. The mechanical part is then divided into its constituent objects according to the

```
mechanical_part(id,name,composition:QUEL)
object(id,parent:QUEL,belonging_to,kind,description:QUEL)
moved_object(id,object:QUEL,arg:QUEL,op)
composed_object(id,left:QUEL,right:QUEL,op_code)
primitive_object(id,type,reference:QUEL)
cylinder(id,radius,length,loc:QUEL)
cuboid(id,width,height,length,loc:QUEL)
motion_arg(id,old:QUEL,new:QUEL)
location(id,x,y,z)
```

Figure 10. CSG scheme in "QUEL as a Datatype."

CSG tree of Figure 4. An object is further described in one of the relations moved_object, primitive_object, and composed_object, respectively. Primitive objects are distinguished between cylinders and cuboids, the only primitive CSG elements that we consider at this point.

The process of inserting data into this schema turns out to be quite tedious, since each attribute of type QUEL requires an explicitly inserted query. Only a small fraction of the insertion commands for our example geometry object "bracket" is shown in the program of Figure 11.

Figure 12 shows a few data-filled relations that store the example object in CSG representation. For simplicity we show the respective query for each attribute of type QUEL. In an actual implementation the column would probably store the query in a preprocessed form, or even in the form of pointers to the result tuples of the query.

3.3.2 *Extended Query Language*

To give the reader an idea of the extended query capabilities of "QUEL as a Datatype," let us consider the following very simple query.

Example

Find the locations of all primitive objects that are constituents of the object #5, that is, the bracket.

range of o **is** object
retrieve o.description.reference.loc.all
where o.belonging_to = 5 **and** o.kind = po

The subclause "o.description" in the retrieve command references a tuple of the relation primitive_object. If the same query is stated with "where o.id = 1," this same subclause would reference a tuple of the relation composed_object. Then the clause "o.description.reference" would make no sense, since reference is not an attribute in composed_object. We note that this causes problems with type checking since the validity of the query can only be determined at execution time. This means that the user needs to know stored attribute values in order to state a valid query. The clause "o.description.reference.loc" finally results in a tuple of the relation location that is returned as a result by this query.

We see that the "." operator can be nested in this extended query language. The ability to reference tuples of different relations via an attribute of type

```
append_ to mechanical_part(
        id=5,name="bracket",
        composition="range of o is object
                     retrieve o.all
                     where o.belonging_to=5")
append_ to object(
        id=1,belonging_to=5,
        kind="co",
        parent="range of o is object
                retrieve o.all
                where o.id=1"
        description="range of c is composed_object
                     retrieve c.all
                     where id=1")
append_ to object(
        id=5,belonging_to=5,
        parent="range of o is object
                retrieve o.all
                where o.id=1"
        kind="po",
        description="range of p is primitive_object
                     retrieve p.all
                     where p.id=5")
append_ to cylinder(
        id=5,
        radius=1.5,
        length=1,
        location="range of l is location
                  retrieve l.all
                  where l.id=5")
append_ to location(id=5,x=2,y=3,z=5)
```

Figure 11. Insertion into the relations of the CSG scheme.

QUEL results in significantly easier queries. This same query would have involved three explicit joins in the traditional relational model. The scope of this presentation does not allow us to give a more detailed description of the query language, and the interested reader is referred to Stonebraker et al. [1983b].

3.3.3 Boundary Representation

The boundary representation of a mechanical object could be stored in the following "QUEL as a Datatype" schema:

```
mechanical_part(id,name,faces:QUEL)
faces(id,parent:QUEL,edges:QUEL)
edges(id,vertices:QUEL)
vertices(id,loc:QUEL)
locations(id,x,y,z)
```

The insertion of data into this schema is shown in Figure 13.

mechanical_part

ID	NAME	COMPOSITION
3	...	...
4	...	...
5	bracket	'range of o is object retrieve o.all where o.belonging_to=5'
6	cog wheel	...
7	...	...

object

ID	PARENT	BELONGING_TO	KIND	DESCRIPTION
1	'range of o is object retrieve o.all where o.id=1'	5	co	'range of c is composed_object retrieve c.all where c.id=1'
2	'range ... retrieve o.all where o.id=1'	5	co	'range ... retrieve c.all where c.id=2'
3	'range ... retrieve o.all where o.id=1'	5	co	'range ... retrieve c.all where c.id=3'
4	...	5	...	co
5	'range ... retrieve o.all where o.id=2'	5	po	'range of p is primitive_object retrieve p.all where p.id=5'
6	...	5	co	...
7	...	5	po	...

composed_object

ID	LEFT	RIGHT	OP_CODE
...	...	...	...
1	'range of o is object retrieve o.all where o.id=2'	'range of o is object retrieve o.all where o.id=3'	union
...	...	...	...

primitive_object

ID	TYPE	REFERENCE
...	...	...
5	cylinder	'range of c is cylinder retrieve c.all where c.id=5'
...	...	...

cylinder

ID	RADIUS	LENGTH	LOC
...	...	...	...
5	1.5	1.0	'range of l is location retrieve l.all where l.id=5'
...	...	...	...

location

ID	X	Y	Z
...	...	...	...
5	2.0	3.0	5.0
5	...	...	...
...	...	...	...

Figure 12. Some data-filled relations.

```
append to mechanical_part(
    id=1,
    name="bracket",
    faces="range of f is faces
           retrieve f.all
           where f.parent=1")
append to faces(
   id=f1,
   parent=1,
   edges="range of e is edges
          retrieve e.all
          where e.id in {e1,e2,e3,e4}")
append to edges(
   id=e1,
   vertices="range of v is vertices
             retrieve v.all
             where v.id in {v1,v2}")
append to vertices(
   id=v1,
   loc="range of l is locations
        retrieve l.all
        where l.id=v1")
```

Figure 13. Insertion of BR data into the schema.

An example query is

Example Query. Find the bounding vertex locations of face f_1.

range of f **is** faces
retrieve f.edges.vertices.loc.all
where f.id = f_1

3.3.4 Discussion

"QUEL as a Datatype" is a very interesting proposal toward engineering databases. In summary one can say that although this approach introduces a very general reference type in the relational data model, there are still some problems with respect to integrated CAM databases. One very obvious problem is the extremely tedious insertion process, which is even more problematic in a dynamic problem area like robotics, where new objects have to be created on a very frequent basis. It seems that the additional insertion complexity is the penalty for the increased expressive power of the query language with the implicit join operation.

The second shortcoming can be seen in Figure 12, which shows the relations of the CSG representation of the bracket. Even though "QUEL as a Datatype" supports referencing between tuples of different relations, it is still the user's responsibility to uniquely identify the objects with some key identifier attributes. This might create consistency problems, especially if more than one engineer works on the database. It would be more suitable if the system were to support the generation of identifiers that could then be assured to be unique within the database.

The CSG data representation is a recursively defined tree. "QUEL as a Datatype" does not support recursion, which might lead to very complicated data manipulation algorithms. The boundary representation generates a constant depth tree, for which "QUEL as a Datatype" seems to work fairly well. Using attributes of type QUEL we can generate references to the lower level abstraction,

for example, faces to edges, fairly easily. But again we note that the data insertion process is extremely tedious. One would have to devise a way to simplify this if "QUEL as a Datatype" were to be used in practice.

Another problem seems to be that the representation is split up into very small partitions, down to vertex locations. This might lead to inherently inefficient data manipulation processes, unless we can manage to cluster data appropriately. This is also true for the CSG representation where you might have to traverse very deeply into the tree to retrieve some subobject.

In summary "QUEL as a Datatype" supports structural object orientation via a very general referencing mechanism, but the system does not provide any facilities for behaviorial object orientation; that is, the model does not allow the definition of application-specific operations.

3.4 ADT-INGRES

ADT-INGRES was proposed by Stonebraker et al. [1983a] and implemented as an experimental prototype on top of the existing DBMS INGRES [Stonebraker et al. 1976] by Fogg [1982]. ADT-INGRES provides a facility that allows the user to define his or her own data types. The representation of the new data type has to be specified in C [Ritchie 1978].

Let us now consider an example. We want to define a relation to store cuboids. For this purpose we specify an ADT for the attributes vertex, which consists of three decimal numbers, the x, y, and z coordinates:

```
cuboids(id,
        material:char(10),
        description:char(20),
        V1:ADT:vertex_type,
        V2:ADT:vertex_type,
              ...
              ...
        V8:ADT:vertex_type)
```

An example query using the ADT attribute vertex_type would look as follows:

```
range of c is cuboids
retrieve (c.material,c.description,c.V1)
where c.id=5
```

Depending on the implementation of the ADT, the output of this query could then look as follows:

material	description	V1		
		X	Y	Z
copper	massive	1.0	3.5	2.0

And a possible append command could look as follows:

```
append to cuboids(
        id=5,
        material="copper",
        description="massive",
        V1=(1.0,3.5,2.0),
        V2=( ... ),
          ...
        V8=( ... ))
```

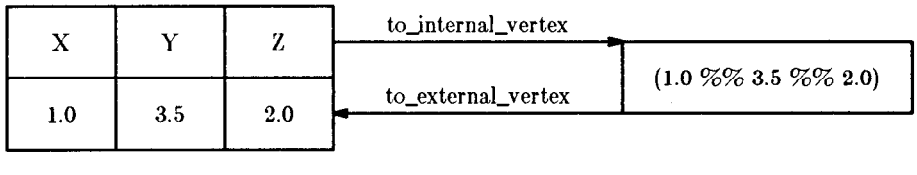
The user has to supply the implementation of such an abstract domain. For our example this would be

```
define ADT(
    typename="vertex_type",
    bytesin=9,
    bytesout=9,
    inputfunc="to_internal_vertex",
    outputfunc="to_external_vertex",
    filename="/usr/ingres/.../vertex")
```

Inputfunc and outputfunc are C subroutines that convert the data type to internal and external representation, respectively. Outputfunc, for example, would extract the X, Y, and Z coordinates from the internal representation and output them in the format shown above. For the implementation of these routines the user needs the knowledge of C.

An obvious disadvantage of ADT-INGRES is that each abstract data type has to be mapped onto one attribute. In our case this means that the three coordinates are mapped onto an attribute of type string. This is a very unnatural mapping. It would be much more convenient (and natural) to map the coordinates onto three attributes of type float.

Schematically the ADT mapping for our data type vertex_type is



In addition to such a data type, the ADT-INGRES user can define his or her own operators on these domains. As an example let us present the framework of the operator R_y, which takes as an argument a vertex and an angle. It returns a vertex that is rotated about the y axis by the given angle. Here we assume that the data type, which is just a numerical type, has been defined previously. The

implementation would look as follows:

```
define adtop(
   opname="R_y",
   funcname="rotate_about_y",
   filename="/usr/ingres/.../rotate_y",
   result=vertex,
   arg1=ADT:vertex_type,
   arg2=ADT:angle_type,
   prec like "+")
```

Once again the file rotate_y must contain a C program implementing this operator.

Similarly one can define the other possible geometric transformations, scaling, translation, and rotation, about the other two axes.

Let us now implement a QUEL program that rotates our previously inserted copper cuboid.

```
range of c is cuboids
   replace c(V1=R_y(c.V1,PHI),V2=R_y(c.V2,PHI), ... , V8=R_y(c.V8,PHI))
   where c.id=5
```

Discussion. ADT-INGRES provides a novel way of specifying new data types and corresponding operators in a database management system. The advantage of this approach lies in the fact that the operators can be arbitrarily complex. For example, we showed the framework for all the geometric transformations on three-dimensional objects, that is, scaling, translation, and rotation.

However, the additional flexibility of the system also has its penalty. The new data types have to be specified in the programming language C. Thus the ADT-INGRES user has to be familiar with two quite different systems: (1) the database language QUEL, and (2) the programmming language C.

Another shortcoming of this approach is inherent in the database management system INGRES: it only allows fields of up to 250 bytes.[2] Therefore we can only specify those objects as ADTs whose internal representation fits into 250 bytes. ADT-INGRES does not allow mapping an ADT onto different tuples (or relations); it requires mapping each ADT completely onto one attribute. Thus the internal representation of engineering objects does not reflect the external structure of the object (as the user perceives it). This usually results in a fairly tedious transformation process from external to internal representation, and vice versa. For example, the ADT vertex_type had to be mapped into a character string rather than onto three attributes of type float, which would have been a much more natural mapping.

ADT-INGRES does not provide any additional support for handling hierarchical data structures that occur frequently in engineering applications. Whereas

[2] This is imposed by the UNIX file structure since each tuple has to fit entirely on one page. (UNIX is a trademark of AT&T Bell Laboratories.)

"QUEL as a Datatype" allows referencing tuples of the same or different relation by formulating an appropriate query as an attribute, the ADT-INGRES approach does not.

ADT-INGRES provides some facilities for behavioral object orientation by allowing the database user to define application-specific ADT operations. However, these operations are quite tedious to implement because internally the model is not structurally object oriented.

3.5 GEM

GEM was developed at Bell Laboratories by Carlo Zaniola [1983]. It is a general-purpose query and update language for the entity–relationship data model. GEM was designed as an extension to the database language QUEL.

The language GEM is very similar to Stonebraker's approach in "QUEL as a Datatype," with the following differences:

- GEM supports set type attributes, that is, sets of atomic types as attributes.
- GEM supports a sophisticated notion of null values.

Thus one could have the following schema in GEM:

item(name, price, {colors})

Each tuple of the relation consists of a value for name, price, and a set of colors. An example would look as follows:

ITEM		
NAME	PRICE	COLORS
Chevy	6000	white black red
Ford	7000	black
Pontiac	6500	black yellow

Except for the representation schemes of the CSG and BR models, the set-valued attributes would not yield any advantage, since GEM only allows sets of noncomposite types, that is, sets of integers, reals, characters, etc., and does not allow sets of tuples (or records) as would be needed to simplify the schema for the CSG or BR representation. For example, using a set of edge_id's, we could have combined the faces and the edges relations of the BR representation as follows:

faces(id, {edge_id})

Thus we could have saved the additional relation edges.

In summary we conclude that for robotics databases GEM is of the same expressive power as "QUEL as a Datatype." A major improvement, especially for modeling hierarchical data structures, would have been achieved by supporting sets of composite types as in the NF^2 model proposed by Schek and Pistor [1982] and Schek and Scholl [1983].

3.6 The Complex Object Data Model: An Extension to System R

System R [Astrahan et al. 1976] is a relational database management system developed at IBM in San Jose. Currently there are efforts under way to enhance the data model to support technical applications. Aside from a suitable transaction mechanism [Lorie 1982; Lorie and Plouffe 1983], the new type long field and the notion of complex object were introduced. Long fields, which are useful for storing unstructured data such as text, are only of minor interest for integrated robotics databases. The CSG, as well as the BR representations, constitute highly structured data schemes. Of course, one could store these data in long fields and then retrieve them using appropriately defined operators on the long fields. This would resemble the domain ADT approach of ADT-INGRES.

For modeling geometric data the concept of a complex object could be quite helpful. A complex object is a *hierarchical* cluster of tuples of different relations, that is, it corresponds to a 1:N relationship. The hierarchical relationship between tuples is expressed by attributes of type "component_of." General N:M relationships are expressed by attributes of type "reference." The main difference between the two reference types is that the data model provides built-in support to access all tuples belonging to a component_of relationship by physically clustering the data and maintaining pointers to the component tuples. The association of tuples is achieved via so-called surrogate attributes [Codd 1979].

3.6.1 Constructive Solid Geometry

The schema description of the CSG approach as a complex object is shown in Figure 14. The entity MP (mechanical part) forms the root of the complex object and is split up into objects that are either composed, moved, or primitive objects. Primitive objects are either cylinders or cuboids. We note already at this point that the notion of a complex object cannot really capture the semantics of the entity "composed object," which is composed of entities of type "object," that is, members of the parent entity. This type of reference cannot be modeled in the complex object approach. This shortcoming is explained further in the description of the boundary representation.

In Figure 15 we show part of the definition of the CSG database schema. We restrict our presentation to the attributes of type identifier, component_of, and reference. The other attributes of the relations are identical to those of Figure 10. An attribute of type identifier, for example, MP_ID, is automatically assigned an internally generated unique value, which might consist of two parts: the processor id, and the time the tuple was generated. This would ensure a worldwide unique identifier value. An attribute of type component_of references exactly one tuple of the parent relation via its (the parent tuple's) identifier value. Thus the component_of concept is used to model 1:N relationships among tuples of different relations. This is shown in the example relations of Figure 16.

In addition to the component_of references, we can also have attributes of type reference to model general N:M relationships. These attributes can reference tuples of a different relation, not necessarily the parent relation. Tuples associated with an attribute of type reference do not form a cluster, and therefore the access of these associated tuples is not particularly supported in the system. Attributes of type reference are used to define the relation CO (composed_object), where each tuple is composed of a left and right child of type OBJ (object).

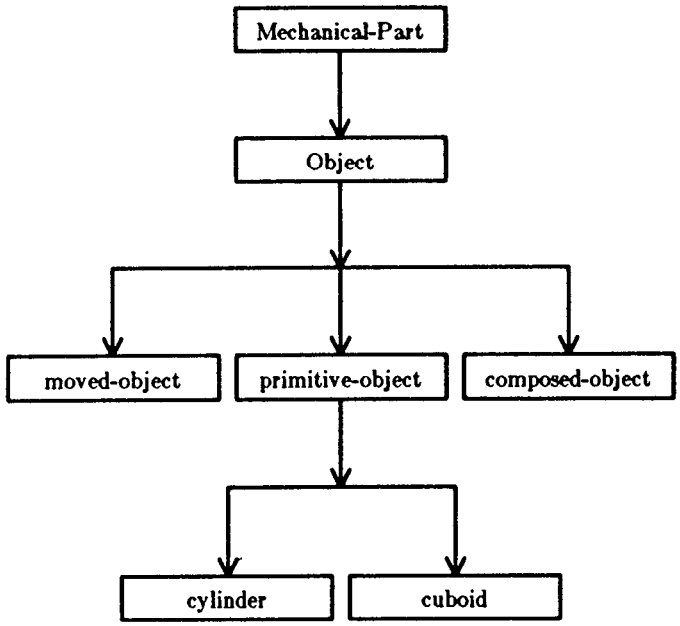


Figure 14. CSG representation as a complex object.

```
create table MP(
       MP_ID identifier,
       NAME ...,
       ...)
create table OBJ(
       OBJ_ID identifier,
       OBJ_COMP component_of(MP),
       ...)
create table MO(
       MO_ID identifier,
       MO_COMP component_of(OBJ),
       ...)
create table CO(
       CO_ID identifier,
       CO_COMP component_of(OBJ),
       LEFT reference(OBJ),
       RIGHT reference(OBJ),
       ...)
create table PO(
       PO_ID identifier,
       PO_COMP component_of(OBJ),
       ...)
create table CY(
       CY_ID identifier,
       CY_COMP component_of(PO),
       ...)
create table CU(...)
```

Figure 15. CSG schema in System R.

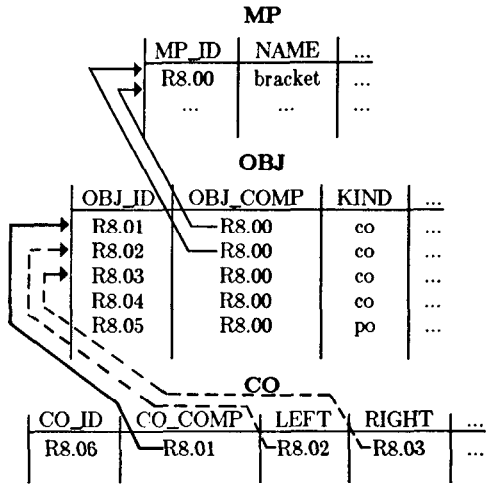


Figure 16. Some example tables.

3.6.2 Boundary Representation

The problem with the component_of concept is that each tuple has exactly one parent tuple in the parent relation and therefore supports only 1 : N relationships. This is a severe shortcoming with respect to modeling the boundary representation of geometric objects. A possible BR schema is shown in Figure 17.

We note that each edge belongs to two faces, and a vertex always belongs to at least two edges. Using the above schema one would have to include a lot of redundant information, since, for example, a vertex has to be stored for each edge for which it is an endpoint. It might even make data manipulation algorithms very complex. Consider the query:

find all faces that have vertex v_i in common

Because of the data redundancy this query involves an exhaustive search for all vertices $v_j = (x_j, y_j, z_j)$ such that $x_j = x_i$, $y_j = y_i$, and $z_j = z_i$, where (x_i, y_i, z_i) are the coordinates of vertex v_i.

3.6.3 Versions in System R

Modeling a geometric or even more complex technical object (e.g., a car) usually requires the development of several different versions of this object or its subobjects during the stages of the design process, and therefore in an engineering database system mechanisms for managing versions of objects must be available. Dittrich and Lorie [1985] provide a model to establish and manipulate versions in System R.

The authors propose a design object not to be uniquely identified by its design object identifier but by a pair (DO identifier, version number). One version of an object can be labeled CURRENT at each point of time, that is, this version is identified by the pair (DO identifier, CURRENT). Furthermore, a version can be declared FROZEN. A frozen design object cannot be updated or deleted until it is "thawed" again. Thus this mechanism is useful to protect an object that has reached a consistent design state from illegal updates. The last frozen object can also explicitly be identified by (DO identifier, LAST FROZEN). In order to define and manipulate design objects and versions in System R, several operators have been added to SQL.

```
create table MP(
    MP_ID identifier,
    NAME ...,
    ...)
create table FACES(
    F_ID identifier,
    F_COMP component_of (MP),
    ...)
create table EDGES(
    E_ID identifier,
    E_COMP component_of(FACES),
    ...)
create table VERTICES(
    V_ID identifier,
    V_COMP component_of(EDGES),
    ...)
```

Figure 17. BR schema in System R extension.

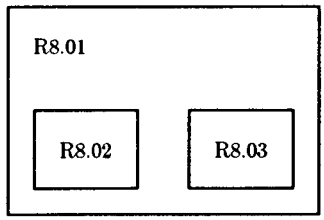


Figure 18. Hierarchical relationship of a design object.

As shown above, the extensions to System R allow a hierarchical design of objects using references. Let us demonstrate the interaction of references and versions on the example object with identifier R8.01 of table CO of Figure 16. This object consists of one instance of the composed object R8.02 and one instance of R8.03, as shown in Figure 18.

Assume R8.02 has two different versions and one user of R8.01 would like to have a view onto the object using version 1 of subobject R8.02; another user, however, would like to use version 2 of the same subobject. In order to solve this problem, Dittrich and Lorie [1985] propose that the references be generic, that is, that references to a design object be independent of its versions. The instantiations of the references are defined by so-called *environments*, which contain direct entries that explicitly specify pairs of (DO identifier, version number) for objects to be actual within the environment and several further entries, referencing specifications of direct entries within other environments. Furthermore, a default version can be specified. The environment a user wants to work with must be activated explicitly.

The problem, shown above, can now be solved by defining different environments for the design object R8.01, as shown in Figure 19.

When environment E1 is activated, the run-time view of object R8.01 is that of an object consisting of version 1 of composed object R8.02 and the current version of object R8.03; in the case of E2, version 2 of R8.02 is chosen by the system. If E1 is activated with default option for R8.01, the current versions of R8.02 and R8.03 are chosen. For further details see Dittrich and Lorie [1985].

3.6.4 Discussion

Lorie [1982] and Lorie and Plouffe [1983] solved one of the shortcomings of "QUEL as a Datatype" by devising a strategy to use system-generated identifiers to reference tuples of different relations. This is a very helpful mechanism that

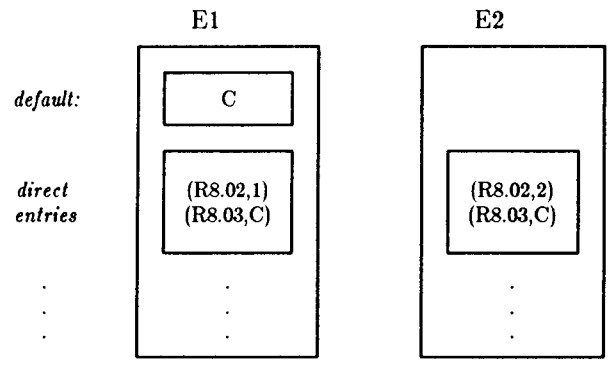


Figure 19. Design environments for different versions of a design object.

builds up abstraction hierarchies without back references. In summary, one can say that they have enriched the relational model to capture the semantics of the hierarchical model as well. This supports the structurally object-oriented modeling of hierarchical engineering objects. In addition to the referencing mechanism the System R extension also includes enhancements of the query language that support retrieval operations on the hierarchical data structures. This allows retrieval of complex objects as entities even though their representation may be segmented over different relations.

However, it is still doubtful that the type concept developed by Lorie and Plouffe is sufficient for technical applications. The only new *data* type (aside from identifier, reference, and component_of types, which could all be called referencing attribute types) is the long field, which is a sequential storage structure for data very much like a file system in traditional programming languages. In order to achieve some level of behavioral object orientation, a language has to be integrated to define powerful operations on this data type. This approach would be very similar to ADT-INGRES.

3.7 The Functional Data Model

A quite different approach to providing a conceptually natural database interface language is the data language DAPLEX [Shipman 1981], which is based on the functional data model. Although this data model has not been developed for nonstandard applications, it provides several useful mechanisms for those applications.

The basic constructs of DAPLEX are the entity and the function. Entities are intended to model real-world objects, whereas functions define their properties. In many cases properties of one object can be derived from those of another and can be modeled in DAPLEX by the notion of derived functions. Furthermore, DAPLEX provides for the construction of very sophisticated user views of a database, which are also specified in terms of derived functions.

3.7.1 Boundary Representation

In DAPLEX a schema for storing mechanical parts in boundary representation might be specified as follows:

```
DECLARE mechanical_part ( )              =>> ENTITY
DECLARE m_id(mechanical_part)            =>  STRING
DECLARE name(mechanical_part)            =>  STRING
DECLARE face(mechanical_part)            =>> faces

DECLARE faces ( )                        =>> ENTITY
DECLARE f_id(faces)                      =>  STRING
DECLARE edge(faces)                      =>> edges

DECLARE edges ( )                        =>> ENTITY
DECLARE e_id(edges)                      =>  STRING
DECLARE vertex(edges)                    =>> vertices

DECLARE vertices ( )                     =>> ENTITY
DECLARE v_id(vertices)                   =>  STRING
DECLARE loc(vertices)                    =>  locations

DECLARE locations ( )                    =>> ENTITY
DECLARE l_id(locations)                  =>  STRING
DECLARE X(locations)                     =>  REAL
DECLARE Y(locations)                     =>  REAL
DECLARE Z(locations)                     =>  REAL
```

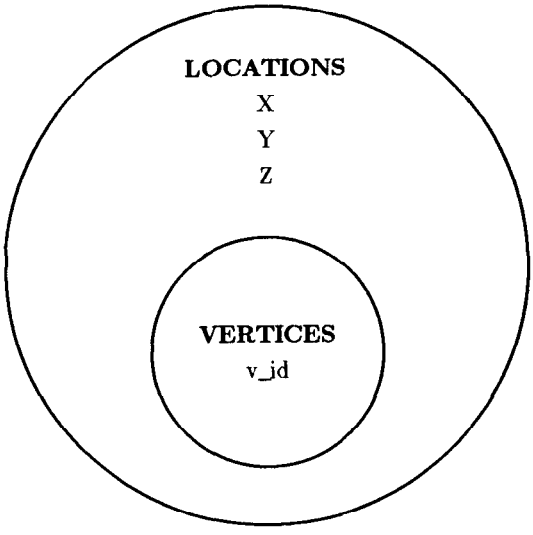
The first statement declares an entity, which is always expressed as a function without any argument. => denotes a single-valued and =>> a multivalued function. For example, the fourth statement defines faces to return a whole entity of type faces. Although all functions in this example have only one argument, multiple-argument functions may be defined too.

DAPLEX also allows the representation of generalization hierarchies. In the example above, location can be interpreted as a general case of vertices, since the set of vertices may be a special subset of the set of locations. In order to express this hierarchical relationship, the declaration of vertices can be changed as follows:

```
DECLARE vertices ( )                     =>> locations
DECLARE v_id(vertices)                   =>  STRING
```

Vertices is now a subtype of locations, and locations is called a supertype of vertices. The set of "vertices" entities is now a subset of "locations" entities. This implies that any "vertices" entity also inherits the functions X, Y, and Z.

Schematically this can be represented as follows:



So far, we have only discussed constructs of DAPLEX concerning data definition. We shall return to this point after the following discussion of some constructs of the data manipulation language.

The representation of a mechanical part, for example, a cuboid, can be inserted in the following way:

```
FOR A NEW mechanical_part
  BEGIN
  LET m_id(mechanical_part) = "m1"
  LET name(mechanical_part) = "cuboid"
  LET face(mechanical_part) = faces
      SUCH THAT f_id(faces) IN {f1,f2,f3,f4,f5,f6}
  END
FOR A NEW faces
  BEGIN
  LET f_id(faces) = "f1"
  LET edge(faces) = edges
      SUCH THAT e_id(edges) IN {e1,e2,e3,e4}
  END
 FOR A NEW edges
  BEGIN
  LET e_id(edges) = "e1"
  LET vertex(edges) = vertices
      SUCH THAT v_id(vertices) IN {v1,v2}
  END
```

```
FOR A NEW vertices
 BEGIN
 LET v_id(vertices) = "v1"
 LET loc(vertices) = THE locations
     SUCH THAT l_id(locations) = "v1"
 END
```

The concept of functions in DAPLEX is quite well integrated in the query language as shown in the following examples.

Query 1. Find the bounding vertex locations of face f1.

```
FOR faces SUCH THAT f_id(faces) = "f1"
PRINT     {X(vertex(edge(faces)))
           Y(vertex(edge(faces)))
           Z(vertex(edge(faces)))}
```

Query 2. Find the names of all mechanical parts that have at least one face with fewer than four edges.

```
FOR EACH mechanical_part
  SUCH THAT
    FOR SOME face(mechanical_part) COUNT(edge(faces)) < 4
  PRINT name(mechanical_part)
```

We now want to return to the data definition language. One construct that has not yet been discussed is the derived function. By means of derived functions, new properties of objects based on the values of other properties may be defined. In the specification of a mechanical part described above, the attribute parent that was introduced in the database schema of "QUEL as a Datatype" (see Figure 13) has been dropped from the mechanical_part schema. This was possible because DAPLEX supports function inversion. In order to find out the mechanical part to which a particular face belongs, one may define a derived function by

```
DEFINE object(faces) =>> INVERSE OF face(mechanical_part).
```

Further useful applications of derived functions are the definition of often used queries as derived functions, the definition of user views, constraints and triggers, as well as the application of system-provided functions like INVERSE OF or UNION OF.

3.7.2 Constructive Solid Geometry

In order to model a mechanical part we used the boundary representation. Another possibility would have been to choose the CSG representation, a schema

for which is given below [Stehle 1986]:

```
DECLARE object( )                    =>> ENTITY
DECLARE id(object)                   =>  STRING
DECLARE type(object)                 =>  STRING

DECLARE mech_part ( )                =>> object
DECLARE id(mechanical_part)          =>  STRING

DECLARE comb_o( )                    =>> object
DECLARE l_obj(comb_o)                =>  object
DECLARE r_obj(comb_o)                =>  object
DECLARE c_op(comb_o)                 =>  STRING

DECLARE mot_o( )                     =>> object
DECLARE arg(mot_o)                   =>  object
DECLARE m_op(mot_o)                  =>  object
DECLARE x(mot_o)                     =>  FLOAT
DECLARE y(mot_o)                     =>  FLOAT
DECLARE z(mot_o)                     =>  FLOAT

DECLARE prim_o( )                    =>> object
DECLARE id(prim_o)                   =>> STRING

DECLARE cubes( )                     =>> prim_o
DECLARE height(cubes)                =>  FLOAT

DECLARE cylinders( )                 =>> prim_o
```

In this schema, design object is a supertype for entity types mech_part, comb_o (combinatorial object), mot_o (motion object), and prim_o (primitive object). An interesting query is: What are the primitive objects a geometric object is composed of? In order to compute those objects directly, recursion is required. DAPLEX, unlike most other query languages, supports one recursive operation, the transitive closure. It can be defined in terms of derived functions. This means that we are able to retrieve all subobjects that compose a geometric object directly.

The DAPLEX query for computing the transitive closure is given as

```
DEFINE obj(object) =>> { l_obj(object AS comb_o),
                         r_obj(object AS comb_o),
                         arg(object AS mot_o) }
DEFINE all_obj(object) =>> TRANSITIVE OF obj(object)

FOR THE mech_part SUCH THAT id(mech_part) = '...'
     FOR EACH X IN all_obj(mech_part) AS prim_o
          PRINT id(X)
```

In this query first the derived function obj is defined, which retrieves the immediate subparts of motion and combination objects. By application of the operator AS to the argument object of the derived functions l_obj, r_obj, and arg, the set of objects considered is restricted to the subsets comb_o and mot_o, respectively. Then the transitive operator is applied to this derived function, returning the complete hierarchy as all_obj. Subsequently only those objects that constitute primitive objects are selected in the query.

The discussion of derived functions shows that they provide a powerful construct for the data definition language. For data manipulation, however, there remain some difficulties with this construct. The main problem is to update derived data.

3.7.3 Discussion

The functional data model and the language DAPLEX as proposed by Shipman seem to be a very interesting candidate for the development of nonstandard database systems. The main advantages are

- DAPLEX supports the representation of hierarchical relationships. Therefore DAPLEX constitutes a structurally object-oriented data model by providing the facility of nested functions to retrieve objects that are composed of several different entity types.
- Derived functions allow users to represent arbitrary relationships directly by defining them in terms of existing relationships; they allow application semantics to be encoded into the data description, and they allow the user to express conceptual abstractions.
- Programming in terms of functions is very similar to programming with a programming language instead of a database language. This may increase the acceptance of potential system users.

The limitations of the functional data model with respect to engineering database applications are

- DAPLEX does not allow the user to define computationally complex functions. In DAPLEX functions are merely used to define relationships. In order to define manipulations of complex objects, DAPLEX would have to be extended to provide user-defined operations on the entities, as is proposed in EDAPLEX [Stehle 1986].
- Inserting geometric data into a DAPLEX schema is extremely expensive because many functions have to be defined in order to provide constructs that allow simple handling of the database.

In comparison with other approaches to nonstandard database systems, as, for example, "QUEL as a Datatype," DAPLEX provides advantages as well as disadvantages. One advantage is the small number of entities that have to be defined, because several relationships can be expressed as multiple argument functions or derived functions. However, this requires a very large number of functions to be specified. DAPLEX allows queries, user views, and system-provided operations to be specified in terms of derived functions. The DEFINE statement for derived functions, though, is a declarative statement that means, in contrast to "QUEL as a Datatype," that these functions are to be defined within the schema specification. An analog to an attribute of type QUEL would therefore not be stored in a user's relation but as metadata in a system relation. This enables the system (unlike "QUEL as a Datatype") to do type checking at

data definition time; that is, the validity of a query does not depend on stored attribute values. An advantage of DAPLEX over other extended database languages like "QUEL as a Datatype" is some support of recursion.

3.8 The NF^2 Data Model

The NF^2 (non-first-normal form) model, as introduced by Schek and Pistor [1982], is based on the nonnormalized relational model. AIM-P [Dadam et al. 1986] is an implementation of the NF^2 model that is being developed at the IBM Scientific Center, Heidelberg. It supports composite attribute types, which can be either tuple valued, that is, one tuple (record), or relation valued, that is, a set of tuples. A composite attribute could also be a list of (possibly composite) elements. All these structures can be arbitrarily nested.

3.8.1 Data Definition Language

We now demonstrate a nested schema definition for the boundary representation of a mechanical part. The definition is shown in a syntax according to Pistor and Traunmüller [1986].

```
create Mechanical_Part {
    [ ID: integer,
      NAME: string(20),
      FACES: {
        [ ID: string(4),
          EDGES: {
            [ ID: string(4),
              VERTICES: {
                [ ID: string(4),
                  LOCATION:
                    [ X: ...,
                      Y: ...,
                      Z: ... ]
                ] }
            ] }
        ] }
    ] }
end
```

Mechanical_Part is a relation with the single-valued attributes ID and NAME and the relation-valued attribute FACES. The relation FACES has the single-valued attribute ID and the relation-valued attribute EDGES, which again is split up into ID and the relation VERTICES. The relation VERTICES consists of an attribute ID and an attribute of type tuple, LOCATION, which consists of the three coordinates X, Y, and Z.

3.8.2 Data Manipulation Language

The data manipulation language (DML) of AIM-P is similar to SQL [IBM 1981], but now the DML has to support the nested structure of relations. To insert data into such a nested structure requires the use of an extended insert command. For the first inserted BR tuple of our example bracket of Figure 4 this

would look as follows:

```
insert
  { [ ID: 5,
      NAME: 'bracket',
      FACES: {
        [ ID: 'f1',
          EDGES: {
            [ ID: 'e1',
              VERTICES: {
                [ ID: 'v1',
                  LOCATION:
                    [ X: 1,
                      Y: 0,
                      Z: 2 ]
                ] }
            ] }
        ] }
    ] }
into Mechanical_Part
```

Now we have to insert one more vertex v2 for the edge e1. This is achieved by the following command:

```
insert
  { [ ID: 'v2',
      LOCATION:
        [ X: 0,
          Y: 1,
          Z: 0 ]
    ] }
into v
from v in e.VERTICES,
          e  in f.EDGES,
          f  in mp.FACES,
          mp in Mechanical_Part
where e.ID = 'e1'
```

We see that the "." operator is analogously overloaded as in "QUEL as a Datatype." In particular, it can be nested arbitrarily deep in order to reference subrelations into which data have to be inserted (or from which data are retrieved or updated).

The relation Mechanical_Part is shown schematically in Figure 20.

3.8.3 Query Language

Pistor and Anderson [1986] and Pistor and Traunmüller [1986] propose the extended SQL [IBM 1981] query language HDBL (*H*eidelberg *D*ata*b*ase *L*anguage) for AIM-P. The query language still uses the basic SQL **select** ... **from** ... **where** ... construct, but, in order to facilitate queries over nested relations, the **select** ... **from** ... **where** ... construct can be nested in AIM-P. An example query over the relation Mechanical_Part of Figure 20 is shown below.

Mechanical_Part							
ID	NAME	FACES					
		ID	EDGES				
			ID	VERTICES			
				ID	LOCATION		
					X	Y	Z
...	...	...	...	...	...	...	...
5	bracket	f1	e1	v1	1	0	2
				v2	0	1	0
			e2	v3	1	2	3
				v1	1	0	2
			e3	v3	1	2	3
				v4	1	0	5
			e4	v2	0	1	0
				v4	1	0	5
		f2	e5	v5	...	...	...
				v6	...	...	...
			e6	...	...	...	...
			...	...	...	...	...
		...	...	...	...	...	...
...	...	...	...	...	...	...	...

Figure 20. The NF^2 relation mechanical_part.

Example Query. Find the vertices of edge e_1 of the face f_1 that belongs to the bracket #5 in the relation Mechanical_Part.

```
select [ m.NAME,
    (select [ f.ID,
        (select [ e.ID,
            (select { [ v.ID,
                (select [ l.X,l.Y,l.Z ]
                from l in v.LOCATION)
                  ] }
            from v in e.VERTICES)
                ]
        from e in f.EDGES
        where e.ID='e1')
            ]
    from f in m.FACES
    where f.ID='f1')
     ]
 from m in Mechanical_Part
where m.ID=5
```

The result of this query would then look as follows:

```
[ bracket,
    [ f1,
        [ e1,
            { [ v1,[1,0,2] ]
              [ v2,[0,1,0] ]
            }
        ]
    ]
]
```

This same query in a pure relational model and a relational scheme analogous to that described in Section 3.1 would have required about the same number of joins that the NF^2 query has as a degree of nesting.

3.8.4 Versions in AIM-P

In AIM-P different versions of a design object can be implemented by the concept of "time versions," a concept that allows the database user to specify objects on which updates are not performed by replacing the old values and generating a new representation of the affected object. This leads to a sequence of time versions; that is, the designer can retrieve the information about an object's state as it was at a specific point in time or within a time interval. The query

```
select m
from m in Mechanical_Part
asof 1985 June 13th 15:13:01
where m.ID = 5
```

for example, would retrieve the object as it was at the specified date.

3.8.5 Discussion

The NF^2 data model implicitly incorporates references to tuples of different relations. Thus it is really a hybrid of the relational and the hierarchical data model. Again, we note that there is a problem with data redundancy; for example, the same vertex v_i could be an endpoint of many different edges but has to be stored for each such edge.

If we do not consider the data redundancy problem, then in the case of a purely hierarchical data structure, such as the BR representation, the NF^2 schema is extremely concise, because we can nest relations arbitrarily deep. Even though abstraction hierarchies are an important issue in engineering applications, we note that the NF^2 model does not necessarily lead to a more concise representation in all CAM applications. For example, the CSG representation would still require most of the relations shown in Figure 6.

The example query that we implemented showed that the extended SQL query language of the NF^2 model is nontrivial because of its nested nature. But, as pointed out by Pistor and Traunmüller [1986], complex NF^2 queries would be at

least as complex in the pure relational model. For our example query this is quite obvious since the query would involve a join over four different relations.

In summary, the NF^2 model provides structural object orientation for hierarchically composed objects. Unlike the System R extension this is achieved by physically (and logically) clustering such complex objects via subrelations. HDBL does not provide for the definition of application-specific operations.

3.9 R^2D^2: Relational Robotics Database System with Extensible Data Types

R^2D^2 (*R*elational *R*obotics *D*atabase System with Extensible *D*ata *T*ypes) is a project currently under way at the University of Karlsruhe. It is an extension to the DBMS AIM-P [Dadam et al. 1986], which is an implementation of the NF^2 data model done at the IBM Scientific Center, Heidelberg. In R^2D^2 the user can define his or her own data types, which can then be used like any built-in type [Kemper 1987; Kemper et al. 1987]. The internal representation of these user-defined data types corresponds to normalized relations in the NF^2 model. In addition to new data types the R^2D^2 user can also define operations for these new types.

3.9.1 User-Defined Data Types

Language concepts for defining abstract data types have been integrated in programming languages for a long time. Abstract data types are used for objects that are frequently used and whose internal representation should be hidden from the user. Here we want to demonstrate the abstract data type facility of R^2D^2 from a user's perspective. We show how the data types and operations that are essential in computer geometry can be integrated into the NF^2 model, thereby making the data model easier to use for storing geometric data.

In Sections 1 and 2 of this paper we have seen that there are only a few data types and operations in a geometric modeling system into which all geometric objects and transformations can be decomposed. These data types are vectors of length 3 or 4, 4×4 matrices, and primitive solids, for example, cuboids, cylinders, and pyramids.

The NF^2 data model has, like most other models, only a limited set of built-in basic data types, such as character, numeric, and Boolean. Of these basic data types one can create structured objects, such as lists, tuples, and relations. The NF^2 model, unlike the pure relational model, allows nesting of structured objects; for example, a list could have elements that are lists to support hierarchical structures.

In R^2D^2 we allow the user to define his or her own data types. These data types can then be used like any built-in data type; that is attributes can be of a type that has been previously defined by the user as an ADT. User-defined data types can be much more complex than the basic NF^2 types. In R^2D^2 a user-defined data type can be any "structured" NF^2 object. The following syntax is used to define new data types:

```
create ADT <identifier> is
           <list_type>|
           <tuple_type>|
           <relation_type>
end <identifier>.
```

Since we allow an abstract data type to be any NF^2 object, it is, in particular, possible to define abstract data types that are nested NF^2 objects, for example, a relation as an attribute of another relation.

Let us now demonstrate how we can support in R^2D^2 some of the data types that are needed in geometric applications as described above. For this purpose we first define the ADT's vector and matrix:

```
create ADT vector is
          <4 FIX real>          ## exactly length 4
end vector.

create ADT matrix is
          <4 FIX vector>   ## exactly 4x4 (a list of a list)
end matrix.
```

In the definition of these data types we use the built-in NF^2 data structure *list*, which is denoted by the pair of brackets "⟨ ⟩". The NF^2 model allows the user to access elements of a list by their position in the list; for example, vector[i] returns the ith component of the list vector. Another built-in function on lists is INDL, which returns the index range of a list. In our example INDL(V) returns $1 \cdots 4$.

These user-defined data types can now be used like any other built-in data types of the database management system. Actually this is already shown in the definition of the ADT matrix, which consists of a list of vectors, where each vector is an ADT consisting of a list of four numeric values. Thus the nesting of the data type matrix is actually hidden from the user who might not be aware of the internal implementation of a matrix.

In addition to these basic types that are needed to implement geometric applications, one also needs to use more complex objects. For example, in the CSG representation the designer can combine primitive predefined objects, such as cylinders, cuboids, and pyramids, to form more complex objects. We could view these primitive objects as application-specific data types.

As an example let us now model the primitive object *cuboid* as an abstract data type in R^2D^2. In the definition of this new ADT we will make use of the previously defined ADT's vector and matrix:

```
create ADT cuboid is
     [VERTICES:
          < 8 FIX [ ID:string(16),
                    LOC:vector
                  ]                          ## exactly 8 vertices
          >
      TRANSFORM:matrix
     ]
end cuboid.
```

In this example we defined an abstract data type *cuboid*, which stores all the bounding vertices of a cuboid, as well as a transformation matrix. This transformation matrix can then be applied to all vertices of the cuboid. It could, for example, specify rotations and translations of the cuboid in the three-dimensional coordinate system.

This ADT cuboid can then be used to define the relation CUBOIDS that stores all the cuboids used in some applications as follows:

```
create CUBOIDS
       {[IDENTIFIER:string,
         MATERIAL:string,
         GEOMETRY:cuboid
        ]}
end
```

The degree of nesting in the relation CUBOIDS is now far greater than the user might be aware of. For the user, who is not the implementer of the ADTs, the relation just appears as defined above, that is, with the three attributes IDENTIFIER, MATERIAL, and GEOMETRY. The user is not aware of the internal representation of GEOMETRY.

In order to use the attribute GEOMETRY at all, we have to provide appropriate operations; otherwise the DBMS user would still have to "dig into" the internal representation of the ADT cuboid in order to retrieve, for example, the coordinates of vertex v_1. How operations on user-defined data types are defined in R^2D^2 is shown in the next section.

3.9.2 Definition of New Operations

In addition to new data types, we also have to be able to define new operations on these data types. In computer geometry the common operations on the data types "vector" and "matrix" are vector addition, vector multiplication, multiplication of a vector with a matrix, and multiplication of two matrices.

In R^2D^2 these operations are defined in an extended HDBL language. Instead of presenting the formal syntax of this language, let us demonstrate it from a user's point of view by implementing the example operations on the ADT's vector and matrix:

```
operation +v (V,W:vector) returns vector
     return                                          ## vector addition
        select V[i]+W[i]
        from i in INDL(V)
end +v

operation *v (V,W:vector) returns real
     return                                          ## vector multiplication
        SUM (select V[i]*W[i]
             from i in INDL(V))
end *v
```

The implementation of the following operations is similar to the implementation of those shown above:

```
operation *vm (V:vector,M:matrix) returns vector
     return .....                                    ## multiplication of a
                                                     ## vector with a matrix

operation row (k:real,M:matrix) returns vector
     return .....                                    ## returns the k^th row
                                                     ## of the matrix

operation transpose (M:matrix) returns matrix
     return .....                                    ## transposes the rows of
                                                     ## a matrix into columns

operation *m (M,N:matrix) returns matrix
     return .....                                    ## matrix multiplication
```

After having defined the necessary numeric operations on vertices and matrices, we can now define the geometric operations "rotate," "scale," and "translate." Since this presentation does not allow us to give a detailed implementation of all these operations, we restrict our discussion to the geometrical transformations rotate_z and c_rotate_z, which define the rotation of a vertex and of a cuboid about the z-axis. Let us first define the (trivial) operation "transform," which takes as arguments a vector and a transformation matrix and multiplies the two:

```
operation transform (V:vector,M:matrix) returns vector
     return V *m M

end transform
```

The operation R_z takes as an argument a numeric value representing an angle and returns a matrix that corresponds to the rotation matrix about the z-axis for this particular angle:

```
operation Rz (PHI:real) returns matrix
     return
          < < .... >
            < .... >                  ## returns a transformation matrix
            < .... >                 ## according to section 3.4
            < .... > >
end Rz

operation rotate_z (V:vector,PHI:numeric) returns vector
     return
          transform(V,Rz(PHI));
end rotate_z
```

```
operation c_rotate_z (C:cuboid,PHI:numeric) returns cuboid
    return
          [VERTICES: (select                    ## rotate them all
                        <[ ID: V.ID,
                           LOC: rotate_z(V.LOC,PHI)
                         ]>
                      from V in C.VERTICES),
           TRANSFORM: C.TRANSFORM               ## just leave it the same
          ]
end c_rotate_z
```

We now implement one last operation on the ADT cuboid in order to demonstrate a query that involves the operation c_rotate_z. The operation V takes as an argument a numeric value i from 1 through 8 and a cuboid and returns the coordinates of v_i. The skeleton implementation is as follows:

```
operation V (i:integer,C:cuboid) returns vector
                ....      ## impl. is straightforward
```

Let us now implement a somewhat contrived query that, nevertheless, demonstrates the integration of ADTs in the query language. Assume we want to retrieve those cuboids from the relation CUBOIDS whose vertex v_1 has the coordinates (x_1, y_1, z_1) when rotated about the z-axis by an angle of 90 degrees. This is achieved by the following query:

```
select c.IDENTIFIER
from c in CUBOIDS
where V(1,c_rotate_z(c.GEOMETRY,90))=<x1,y1,z1,1>
```

In this query we see that the database user applies previously defined operations, such as V and rotate_z, on the attribute GEOMETRY, thereby avoiding having to deal with the internal structure of this attribute and merely applying functions that are quite common in his or her area of expertise.

3.9.3 Discussion

R^2D^2 constitutes a symbiotic approach to object-oriented database systems by providing concepts for structural, as well as behavioral, object orientation. This is achieved by integrating the concept of abstract data types into the data definition and data manipulation language of a structurally object-oriented DBMS. Thus the database user can define data types that correspond to application-specific objects. The structural features are inherited by choosing a particular object-oriented data model, the NF^2 model, which allows nested relations whereby hierarchical relationships among subobjects can be modeled. The internal representation of an ADT corresponds in R^2D^2 to a (possibly) nested NF^2 relation. Thus it is guaranteed that the objects defined at the user level are also internally treated as clustered objects.

The behavioral object orientation in R^2D^2 is achieved by defining operations on these abstract data types. The behavioral aspect was demonstrated on some specific operations for geometric modeling. In R^2D^2 it is possible, as was shown in the vector and matrix example, to implement ADTs by stepwise refinement, that is, by making use of previously defined data types and operations.

4. Conclusions

In the first part of this paper we investigated the requirements imposed on database management systems by computer-aided manufacturing applications. We began by introducing the most important computer representation models for rigid solid objects in the form of a tutorial in order to present the aspects that are relevant for database designers. Special emphasis was put on the description of the data structures of these representation models rather than on a thorough theoretical framework. It was concluded that the CSG representation with the recursively defined tree structure, as well as the BR model consisting of an abstraction hierarchy, are promising candidates for storing solid objects in a CAM database. The third representation scheme that we investigated, the primitive instancing model, was found to create substantial problems for database support because it would require an abundance of different record types.

In the second part of this presentation we analyzed some of the more recent proposals for engineering databases. We only considered systems that evolved out of the relational database model. Our analysis was mostly restricted to schema support for CAM applications, and only a minor investigation of the data manipulation language was carried out. It was shown that the traditional data models do not adequately support technical applications because they lack object orientation. Most of the surveyed database proposals recently developed support structural object orientation. These systems are

- "QUEL as a Datatype" and GEM, which support modeling of structurally complex entities via a very general reference type;
- DAPLEX, which achieves this by allowing nested functions to retrieve complex objects;
- the System R extension and the NF^2 model, which provide constructs for modeling hierarchical relationships.

ADT-INGRES constitutes the first database system to provide some level of behavioral object orientation by allowing the definition of abstract data types with corresponding operations. R^2D^2 takes this idea one step further by integrating the abstract data type concept in a structurally object-oriented data model, the NF^2 model. Thereby the ADT implementer can map external objects on structured internal database entities.

Another very important concept is recursion, especially for manipulating CSG data. However, general recursive language features were not included in any of the proposals (with the exception of transitive closure in DAPLEX).

ACKNOWLEDGMENTS

This paper is a revised and extended version of "CAM Databases: Requirements and Survey" by A. Kemper, which appeared in the *Proceedings of the 19th Hawaii International Conference on Systems Sciences* (Honolulu, Jan., 1986). Western Periodicals, North Hollywood, Calif., pp. 453–463.

The work described in this paper was done within the R^2D^2 project. R^2D^2 is a cooperative project between the IBM Scientific Center, Heidelberg and the University of Karlsruhe, Fakultät für Informatik.

It is a pleasure to acknowledge the helpful suggestions by Thomas A. Bagli and Peter C. Lockemann on an earlier version of this paper. We are indebted to the referees for many helpful comments.

REFERENCES

ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J. N., GRIFFITHS, P. P., KING, W. F., LORIE, R. A., MCJONES, P. R., MEHL, J. W., PUTZOLU, G. R., TRAIGER, I. L., WADE, B. W., AND WATSON, V. 1976. System R: Relational approach to database management, *ACM Trans. Database Syst. 1*, 2 (June), 97–137.

ATWOOD, T. M. 1985. An object-oriented DBMS for design support applications. In *Proceedings of the IEEE Compint.* IEEE, New York, pp. 299–307.

BATORY, E., AND KIM, W. 1985. Modeling concepts for VLSI CAD objects. *ACM Trans. Database Syst. 10,* 322–346.

BLUME, C., MÜLLER, E., AND PODS, R. 1983. RODABAS—Eine Roboter Datenbasis für die implizite Programmierung. In *Höhere Programmiersprachen für Industrieroboter,* H. Wolters, Ed. Kernforschungszentrum Karlsruhe.

CODD, E. F. 1970. A relational model for large shared data banks. *Commun. ACM 13,* 6 (June), 377–387.

CODD, E. F. 1979. Extending the relational database model to capture more meaning. *ACM Trans. Database Syst. 4,* 4 (Dec.), 397–434.

DADAM, P., KÜSPERT, K., ANDERSON, F., BLANKEN, H., ERBE, R., GÜNAUER, J., LUM, V., PISTOR, P., AND WALCH, G. 1986. A DBMS prototype to support extended NF^2-relations: An integrated view on flat tables and hierarchies. In *Proceedings of the ACM SIGMOD Conference.* ACM, New York, pp. 376–387.

DITTRICH, K. R. 1986. Object-oriented database systems: The notion and the issues. In *Proceedings of the International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept.). IEEE Computer Society Press, pp. 2–6.

DITTRICH, K. R., AND LORIE, R. A. 1985. Version support for engineering database systems. Research Rep., IBM Research Laboratory, San Jose, Calif.

DITTRICH, K. R., GOTTHARD, W., AND LOCKEMANN, P. C. 1986. Complex entities for engineering applications. In *Proceedings of the 5th Entity-Relationship Conference* (Dijon, France). North-Holland, Amsterdam.

EASTMAN, C. M. 1981. Database facilities for engineering design. *Proc. IEEE 69,* 10 (Oct.), 1249–1263.

EASTMAN, C. M. 1986. The use of object-oriented databases to model engineering systems. In *Proceedings of the International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept.). IEEE Computer Society Press, pp. 215–216.

EASTMAN, C. M., AND KULAY, A. 1985. Specification of FORM:ULAE: A distributed engineering data management system. In *Proceedings of the ASCHE Conference* (Dallas, Tex.).

FOGG, D. 1982. Implementation of domain abstraction in the relational database system INGRES. Master's thesis, Electrical Engineering and Computer Science Dept., Univ. of California, Berkeley.

FOLEY, J. D., AND VAN DAM, A. 1983. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, Mass.

GLINZ, M., HUSER, H., AND LUDEWIG, J. 1985. SEED—A database system for software engineering environments. In *Informatik-Fachberichte, 94.* Springer-Verlag, Berlin, pp. 121–126.

GUTTMAN, A., AND STONEBRAKER, M. 1982. Using a relational database management system for computer aided design data. *IEEE Database Eng. 5,* 2 (June).

HASKIN, R. L., AND LORIE, R. A. 1982. On extending the functions of a relational database system. In *Proceedings of the International Conference on the Management of Data* (Orlando, Fla., June 2–4). ACM, New York, pp. 207–212.

IBM 1981. SQL/Data system, concepts and facilities. Rept. GH 24-5013, IBM Corp., Jan.

KEMPER, A. 1986. CAM databases: Requirements and survey. In *Proceedings of the 19th Hawaii International Conference on System Sciences* (Honolulu, Jan.). Western Periodicals, North Hollywood, Calif., pp. 363–378.

KEMPER, A. 1987. Abstract datatypes in geometrical databases. In *Proceedings of the 20th Hawaii International Conference on System Sciences* (Kona, Jan.), pp. 453–463.

KEMPER, A., WALLRATH, M., AND LOCKEMANN, P. C. 1987. An object-oriented system for engineering applications. In *Proceedings of International Conference on the Management of Data* (San Francisco, Calif., May 27–29). ACM, New York.

LEE, Y. C., AND FU, K. S. 1983. A CSG based DBMS for CAD/CAM and its supporting query language. In *Proceedings of ACM SIGMOD Conference on Engineering Design Applications* (San Jose, Calif., May), ACM, New York.

LOCKEMANN, P. C., ADAMS, M., BEVER, M., DITTRICH, K. R., FERKINGHAFF, B., GOTTHARD, W., KOTZ, A., LIEDTKE, R. P., LÜKE, B., AND MÜLLE, J. 1985. Anforderungen technischer Anwendungen an Datenbanksysteme. In *Informatik-Fachberichte, 94.* Springer-Verlag, Berlin, pp. 1–26.

LORIE, R. 1982. Issues in databases for design applications. In *File Structures and Databases for CAD,* J. Encarnacao and F. L. Krause, Eds. North-Holland, Amsterdam.

LORIE, R., AND PLOUFFE, W. 1983. Complex objects and their use in design transactions. In *Proceedings of ACM SIGMOD Conference on Engineering Design Applications* (San Jose, Calif., May), pp. 115–121.

LUM, V., DADAM, P., ERBE, R., GÜNAUER, J., PISTOR, P., WALCH, G., WEMER, H., AND WOODFILL, J. 1985. Design of an integrated DBMS to support advanced applications. In *Proceedings of the International Conference on Foundations of Data Organization* (Kyoto, Japan, May 22–24), pp. 21–31.

MAIER, D., OTIS, A., AND PURDY, A. 1985. Object-oriented database development at Servio Logic. *IEEE Database Eng. 8,* 4 (1985), 58–65.

MEIER, A. 1985. Applying relational database techniques to solid modelling. In *Informatik-Fachberichte, 94.* Springer-Verlag, Berlin, pp. 50–67.

PISTOR, P., AND ANDERSEN, F. 1986. Designing a generalized NF² data model with an SQL-type language interface. In *Proceedings of the 12th International Conference on Very Large Databases* (Kyoto, Japan). VLDB Endowment, Saratoga, Calif., pp. 278–285.

PISTOR, P., AND TRAUNMÜLLER, R. 1986. A data base language for sets, lists, and tables. *Inf. Syst. 11*, 4, 323–336.

REQUICHA, A. A. G. 1980. Representations for rigid solids: Theory, methods, and systems. *ACM Comput. Surv. 12*, 4 (Dec.), 437–463.

RITCHIE, D. 1978. *The C Programming Language.* 1978. Prentice-Hall, Englewood Cliffs, N.J.

SCHEK, H.-J., AND PISTOR, P. 1982. Data structures for an integrated data base management and retrieval system. In *Proceedings of the 8th International Conference on Very Large Databases* (Mexico City). VLDB Endowment, Saratoga, Calif.

SCHEK, H.-J., AND SCHOLL, M. 1983. Die NF²-Relationenalgebra zur einheitlichen Manipulation externer, konzeptueller und interner Datenstrukturen. In *Informatik Fachberichte 72.* Springer-Verlag, Berlin, pp. 113–133.

SHIPMAN, D. 1981. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1 (Mar.), 140–173.

STEHLE, H. 1986. EDAPLEX: An extension of the functional data model DAPLEX for computer-geometry applications (in German). Master's thesis, Univ. Karlsruhe, Karlsruhe, Germany.

STONEBRAKER, M., AND ROWE, L. 1986. The design of POSTGRES. In *Proceedings of the International Conference on Management of Data* (Washington, D.C., May 28–30). ACM, New York, pp. 430–355.

STONEBRAKER, M., WONG, E., KREPS, P., AND HELD, G. 1976. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept.), 189–222.

STONEBRAKER, M., RUBENSTEIN, B., AND GUTTMAN, A. 1983a. Application of abstract data types and abstract indices to CAD databases. In *Proceedings of ACM SIGMOD Conference on Engineering Design Applications* (San Jose, Calif., May). ACM, New York.

STONEBRAKER, M., ANDERSON, E., HANSON, E., AND RUBENSTEIN, B. 1983b. QUEL as a datatype. Memo. UCB/ERL M83/73, Univ. of California, Berkeley, Dec.

VOELCKER, H. B., AND REQUICHA, A. A. G. 1977. Geometric modelling of mechanical parts and processes. *Computer 10*, 12 (Dec.).

WESLEY, M. A. 1980. Construction and use of geometric models. *Springer Lecture Notes in Computer Science*, vol. 89, J. Encarnacao, Ed. Springer-Verlag, Berlin.

ZANIOLA, C. 1983. The database language GEM. In *Proceedings of the International Conference on Management of Data* (San Jose, Calif., May 23–26). ACM, New York, pp. 207–218.

ZANIOLA, C., AIT-KACI, H., BEECH, D., CAMMARATA, S., KERSCHBERG, L., AND MAIER, D. 1986. Object-oriented database systems and knowledge systems. In *Proceedings of the 1st International Workshop an Expert Database Systems*, L. Kerschberg, Ed. Benjamin Cummings, Menlo Park, Calif., pp. 49–64.

ZDONIK, S. B., AND WEGNER, P. 1986. Language and methodology for object-oriented database environments. In *Proceedings of the 19th Hawaii Conference on System Sciences* (Honolulu, Jan.). Western Periodicals, North Hollywood, Calif., pp. 378–388.

BIBLIOGRAPHY

BAUMGART, B. G. 1975. A polyhedron representation for computer vision. In *AFIPS Conference Proceedings*, vol. 44. AFIPS Press, Reston, Va., pp. 589–596.

EASTMAN, C. M. 1980. System facilities for CAD databases. In *Proceedings of the 17th ACM/IEEE Design Automation Conference* (Minneapolis, Minn., June 1980), pp. 50–56.

JAESCHKE, G., AND SCHEK, H.-J. 1982. Remarks on the algebra of non-first-normal form relations. In *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Los Angeles, Calif., Mar. 29–31). ACM, New York, pp. 124–138.

KIM, W., LORIE, R., MCNABB, D., AND PLOUFFE, W. 1984. A transaction mechanism for engineering design databases. In *Proceedings of the 10th International Conference on Very Large Databases* (Singapore, Aug.). Very Large Database Endowment, Saratoga, Calif., pp. 355–362.

LÜKE, B., AND BEVER, M. 1985. Ein prozedurorientiertes Datenmodell für CAD/CAM Anwendungen und seine Realisierung mittels konventioneller Datenbanksoftware und Ada. In *Informatik-Fachberichte, 94.* Springer-Verlag, Berlin, pp. 127–146.

MCLEOD, D., NARAYANASWAMY, K., AND BAPA RAO, K. V. 1983. An approach to information management for CAD/VLSI applications. In *Proceedings of ACM SIGMOD Conference on Engineering Design Applications* (San Jose, Calif., May). ACM, New York, pp. 39–50.

MEIER, A. 1986. *Methoden der Graphischen und Geometrischen Datenverarbeitung.* Teubner, Stuttgart, 1986.

Received June 1986; final revision accepted May 1987.