# Toward a General Spatial Data Model for an Object-Oriented DBMS

**Frank Manola, Jack A. Orenstein**
**Computer Corporation of America**
**Cambridge, Massachusetts U.S.A.**

## Abstract

This paper describes the development of a general spatial data model for PROBE, a knowledge-oriented DBMS being developed at CCA [DAYA85, DAYA86]. The data model, called PDM, is an extension to the Daplex functional data model [SHIP81, FOX84]. The paper first describes the approach taken to defining spatial semantics in the model, and how these semantics were incorporated into the non-spatial aspects of the model. Second, some implementation aspects are discussed.

## 1. Introduction

It is widely recognized that existing database management systems do not address the needs of many "non-traditional" applications such as geographic information systems and computer-aided design. The underlying data models, query languages, and access methods were designed to deal with simple data types such as integers and strings, while the new applications are characterized by spatial data, temporal data, and other forms of data having both complex structure and semantics. While spatial data can usually be stored in conventional DBMS data types, it is extremely difficult to specify even the simplest spatial operations in such a DBMS. Moreover, the implementation will have poor performance because the query will be complicated and difficult to optimize, because the access paths were not designed for spatial data, and because the clustering of data normally provided by a DBMS (e.g. on a numeric attribute) is a poor choice for spatial data. In order to deal with spatial data, more support is needed at all levels of the DBMS.

One approach that has been taken to address this problem is to define specific extensions for various nontraditional data types, and add them to conventional DBMSs, in many cases borrowing from the extensive literature on abstract data types (from which we also borrow, e.g., [MALL82]). For example, various special-purpose extensions to DBMSs have been proposed for dealing with text [STON82, SCHE82], images ([IEEE77, CHAN81] contain many relevant papers), and geographic data [IEEE77, MORE85].

The problem with this approach does not lie in starting with conventional DBMS facilities. In any real application for databases of spatial data there are databases of non-spatial data that must be dealt with, and for this data conventional DBMS facilities are often ideal. Instead, the difficulty is that in each case the specific extensions added are application-specific, and limited in generality. For example, the spatial capabilities required for geographic data would be at best of limited use in a mechanical CAD application. Moreover, even for a single *type* of data, e.g. geographic data, there are many different ways to represent and manipulate the data, and each way may be the best in some specific application. It does not seem possible to select one approach to build in and maintain generality. At the same time, it is clearly impossible to provide all useful approaches in the same DBMS.

One of the goals of the PROBE DBMS being developed at CCA [DAYA85, DAYA86] is to efficiently process a variety of spatial and temporal data types. The approach being taken in PROBE is to design an "extensible" object-oriented DBMS. This allows the inclusion of specific object classes that support the required spatial data types, while maintaining generality by allowing additional types to be defined as applications grow or change.

An extensible DBMS stores and manipulates members of object classes. The set of operations in the data model includes conventional database operations like select, and operations supplied with the object class definitions. Certain common object

This work was supported by the Defence Advanced Research Projects Agency and by the Space and Naval Warfare Systems Command under Contract No. N00039-85-C-0263. The views and conclusions contained in this paper are those of the authors and do not necessarily represent the official policies of the Defense Advanced Research Projects Agency, the Space and Naval Warfare Systems Command, or the U.S. Government.

classes (e.g. numeric and string object classes) are needed by all users. More specialized object classes can be added as needed. The definition of an object class includes implementations of operations, the representation of object class instances, a description of the algebraic properties of the operations and information about the cost of the operations. These last two items will be used by an extensible query optimizer.

This paper describes the basic concepts supporting spatial data handling in PROBE, and the facilities of a specific object class being implemented in PROBE for handling spatial data. This object class is intended primarily to support spatial query processing. To allow wide applicability, the object class is not tailored to a particular dimension or representation. This object class will implement "approximate geometry". Approximate geometry (AG) is based on the idea that approximate answers to spatial queries can be calculated much more quickly than exact answers. As developed here, AG can be used in conjunction with a wide variety of established representations that would be hidden in other object classes. The algorithms and data structures we will describe are well-supported by the facilities of conventional DBMS implementations.

The rest of the paper is organized as follows. Section 2 briefly describes the characteristics of the PROBE data model. Section 3 describes the concepts underlying PROBE's spatial data model. Section 4 presents a short example illustrating the use of these concepts. Approximate geometry is discussed in section 5. Section 6 contains concluding remarks.

## 2. Data Model Overview

Like others who have investigated the problems of DBMSs for spatial data, we begin with an existing data model. The model we have chosen is the Daplex functional data model [SHIP81, FOX84]. The extension is referred to here as PDM (for PROBE Data Model). Due to lack of space, and our intent to concentrate on spatial data handling, we can only present here a brief description of the basics of PDM. A more detailed description of PDM is found in [MANO86].

As in Daplex, there are two basic types of objects in PDM, *entities* and *functions*. An *entity* is a database object that denotes some individual thing. It may be though of as being denoted by a surrogate value (system-generated unique identifier). The basic property of an entity in the model is its distinct identity. Attributes and relationships of entities are represented by functions (see below). Entities are grouped into classes called *entity types*. The same entity may be associated with one or more entity types in the database, as defined by metadata specifications.

In Daplex, a *function* is a mapping from entities either to other entities, to scalar values, or sets of entities or scalar values. PDM generalizes this concept by defining a function as a mapping from collections of entities and scalar values (the parameters of the function) to other collections of entities or scalar values. Thus, a function is defined over one or more input arguments (of specified types), and returns one or more output arguments, also of specified types. This is indicated by the notation:

$$\text{function-name}(\text{input}_1, \ldots, \text{input}_n): (\text{output}_1, \ldots, \text{output}_m)$$

Each entity type has defined for it (via metadata specifications) a collection of functions that may be applied to entities of that type. There are two classes of these functions. The first class consists of generic data model operations that apply to all entities in the database, such as "selection". These functions have been defined in the form of a "PDM algebra" that plays the same role in PDM as the relational algebra does in the relational model. The second class consists of functions defined by users using the data description facilities of PDM. These functions are used to represent entity attributes and relationships. PDM makes no distinction between functions that have explicit stored representations (similar to relations) and functions whose output is computed procedurally on demand. Thus, arbitrarily-complex functions may be specified in a database, and referenced in queries. Entity types may be defined as subtypes of other entity types. In such cases, entities of a subtype may inherit functions from their supertypes. The use of entities and functions in spatial modeling is discussed in the next section.

## 3. Spatial Data Model

### 3.1 Point Sets and Spaces

Given a data model supporting conventional data types, such as PDM, it is necessary to do several things to define spatial enhancements. The first of these is to find a way to represent spatial characteristics of the entities defined in the data model, and to associate these characteristics with the non-spatial characteristics. The second is to define the precise semantics of the various spatial characteristics. Finally, implementations of the defined semantics must be provided.

A spatial representation of an entity in a given space can be modeled by a function that maps from the entity to one or more points in that space. Intuitively, this function says, for the entity, what points in the space it "occupies". Similarly, non-spatial attributes of entities that vary over the spatial representation of the entity, such as the color of a mechanical part, can be modeled by a function that maps from the attribute value to one or more points in the spatial representation of the entity. Note that, in the absence of constraints to the contrary, several entities may occupy the same space.

A problem that must be considered is how to generate the point set values of such functions. Many applications involve a continuous (non-discrete) space, in which most useful point sets contain an uncountable number of points. In order to be practical, however, it

must be possible to construct a value of interest in a finite number of operations. To do this, we first consider these functions as mapping from real entities to special entities that denote point sets, rather than mapping to actual (enumerated) sets of points (although this will still be permitted in practice). These special entities are defined as entities of type PTSET.

With the development so far, the basic elements of our approach to incorporating spatial data in the model can be identified with reference to Figure 3.1. (The arrows denote entity-valued functions; double-headed arrows denote entity-set-valued functions).

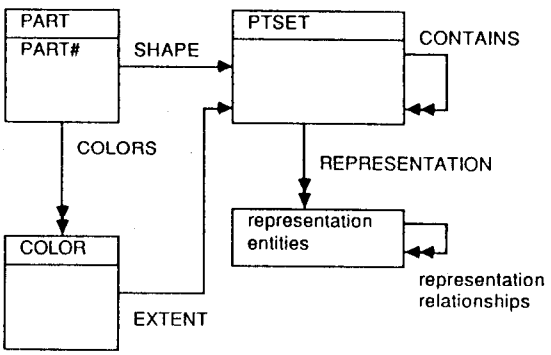


Figure 3.1 Elements of Spatial Data Model Extensions

Entities of type PTSET that have the semantics of points or point sets (such as lines, areas, or volumes) are included in the model, and serve as the values of spatial attributes, such as "shape" or "boundary", of ordinary database entities, such as "parts". Using PTSET entities allows both spatial and non-spatial attributes (such as "PART#") to be associated with the same database entities in a straightforward way, as shown in Figure 3.1. Attributes, such as COLOR or DENSITY, that vary over the shape of the part may be handled in two ways in PDM. First, the attribute, e.g. COLOR, can be defined as one or more multiargument functions, such as COLOR(PART,EXTENT). Alternatively, a separate entity can be defined, as shown in the figure.

It must be possible to specify entities of type PTSET that denote required point sets in a finite number of operations. The usual solution (and the one adopted here) is to provide specific entity subtypes of the general entity type PTSET that denote shapes needed in a particular range of applications, together with operations for combining entities of type PTSET (and its subtypes) to produce new entities of type PTSET. A given entity could then be specified by specifying one of the specialized entity subtypes, together with values for its various parameters, e.g., "CONE (RADIUS=>12, HEIGHT=>40)". The set of points denoted by this entity would be implicit in the underlying definition of "cone" (e.g., an equation, possibly defined in terms of a default coordinate system and origin) together with the specified parameters. More complex shapes could be built by combining such specialized shapes. In graphics, the entity subtypes would typically include boxes, points, line segments, and other shapes (including text). In 3D solid modeling, the entity subtypes could include the various solid shapes, such as blocks, spheres, cones, etc., found in solid modelers. In the case of boundary representations, subtypes of PTSET would be formed from other subtypes of PTSET with the special semantics of boundaries, using special operations for forming structured objects from entities of these types [MANT82].

Only the most general point set semantics are defined for the PTSET type. The detailed behavior and characteristics of spatial entities required for particular applications are defined in the various specialized subtypes of PTSET. For example, [REQU80] identifies "r-sets" (bounded, closed, and regular sets) as having the required characteristics for representing 3-dimensional solids (r-sets, for example, are finite and have well-formed boundaries). In PROBE, we anticipate adding these specialized entity subtypes using PROBE's extensibility features. The new types would either inherit the definitions of operations from the PTSET type, or would provide specialized versions of such operations. For example, a subtype 3DSOLID of type PTSET might be provided for representing 3-D solid objects using "r-sets" as its representation. However, [REQU80] notes that ordinary point set operations (such as union) are not closed for these objects (they can create "dangling edges" of zero thickness). Thus, it would be necessary to use "regularized set operations" (described in [REQU80]) that preserve the properties of "r-sets" for subtype 3DSOLID instead of the generic point set operations provided for its supertype PTSET (described below). Specialized types could also have additional specialized functions that apply to them (such as a "boundary" function), as well as specialized predicates.

In addition to dealing with PTSET entities as individual objects, there are many situations in which it is necessary to deal with PTSETs contained within other PTSETs. For example, a map feature might have a PTSET describing its shape. The PTSET for the containing map would have to contain all the PTSETs of features contained within the map (Figure 3.2).

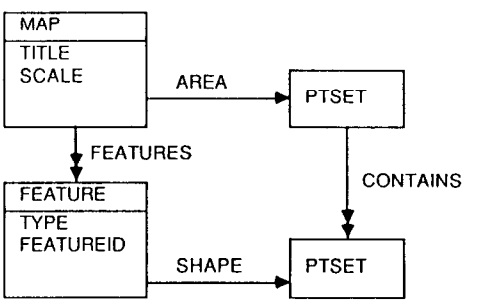


Figure 3.2 A Map and its Component Features

Similarly, PTSET entities that represent individual parts within an assembly may be grouped as components of the PTSET entity that represents the entire assembly.

When we deal with a PTSET in its role as a "container" of other PTSETs, we refer to the "container" PTSET as a "space". Since a PTSET contained in one space can itself contain other PTSETs, PTSETs naturally exhibit a hierarchical structure. We represent the hierarchical structure in the model by a set-valued CONTAINS function from the space to the spatial entities contained within the space. Multiple decompositions of the same set of points (such as a geographic area) can be defined using multiple PTSET entities denoting the same set of points.

A database entity may be related to multiple PTSET entities (e.g. in different spaces) in a straightforward way, using the normal capabilities of the data model to support 1-n relationships. This allows an entity to be associated with any number of different versions of its "shape". For example, a bridge might be represented as a point in one map, as a line in a map showing greater detail, as a space frame in its design data, etc. This provides a method for associating all representations of the bridge (assuming they are known). Also, each of the 2-D point sets representing the bridge in a particular map, for example, would be associated with the point set representing the area covered by the entire map, enabling the bridge to be associated (and located) with respect to the other features in the same map.

Finally, figure 3.1 shows that the model also allows aspects of the *implementation* of PTSET entities to be visible in the database, if this is appropriate, via "representation entities" (and relationships).

### 3.2 Operations

Since entities of type PTSET are first class PDM entities, they can be used as arguments of generic PDM functions in the same way as conventional PDM entities. In addition, specialized operations associated specifically with entities of type PTSET are defined. The operations provided for operating on generic PTSET entities fall into two categories, *point set operations* and *structural operations*.

The point set operations include set operations on PTSET entities, spatial selection, overlay, and geometric transformations. The point set operations intersection, union, and difference, provide the primary means for combining PTSET entities into new PTSET entities. These operations are defined for entities P1 and P2 of type PTSET as follows:

- *Point set union* -- The point set union ptunion(P1,P2) is an entity Pr of type PTSET that denotes the set of points belonging to either P1 or P2 (or both).
- *Point set intersection* -- The point set intersection ptintersect(P1,P2) is an entity Pr of type PTSET that denotes the set of points belonging to both P1 and P2.
- *Point set difference* -- The point set difference ptdiff(P1,P2) is an entity Pr of type PTSET that denotes the set of points belonging to P1 and not to P2 (note that difference is not symmetric).

Also defined as point set operations are special variants of generic PDM functions that are tailored to operate with PTSET entities. Specifically, predicates are added to functions such as selection that test various spatial conditions, such as whether a point set is empty, contains another point set, or intersects another point set. A whole range of other spatial relationships (e.g. "left-of", "above", "adjacent-to") can be added in the same way.

Given a space containing objects that may overlap with one another, it is often useful to identify maximal subspaces that do not contain any object boundaries. For example, a crucial operation in geographic information systems is "polygon overlay". This operation superimposes two maps of the same area (e.g. land usage and political districts) and creates all the regions due to the intersection of regions from the input maps. PROBE's spatial data model includes an *overlay* operator to facilitate this kind of processing.

In discussing overlay it is useful to have the concept of a *uniform region*. Let obj(p,S) be the set of objects in a space, S, where p is a point of S. Then a uniform region is a maximal subspace u, in a space S, such that for every point p in u, obj(p,S) is the same. I.e. u is a uniform region if $\forall$ p1, p2 $\in$ u: obj(p1,S) = obj(p2,S) and no subspace containing u has this property. To support operations such as polygon overlay, it is useful to be able to turn uniform regions into first-class objects. This is the finest partitioning that can be obtained given a set of objects (using only object boundaries to define partitions.) Any desired partitioning can be created from the uniform regions. From the point of view of the data model, a space containing objects is indistinguishable from a space containing the uniform regions derived from a set of objects. They are both represented by a space containing spatial objects.

Based on this discussion of uniform regions, we can now define *overlay*: Overlay(S) returns a space containing a spatial object for each uniform region of space S. The overlay operation can be used to compute polygon overlay as follows. Each input map is represented by a space containing a PTSET for each polygon of that map. The PTSETs from the two maps are placed in a single space by the obunion operation (discussed below). The overlay operator is applied to the output from obunion.

It is useful to be able to compute attributes of uniform regions from attributes of the objects (e.g. area). An approach to this problem is discussed in [OREN85].

A *geometric transformation* is an operation that moves the points of a PTSET entity without changing its identity (in effect, the transformation changes the definition of the set of points denoted by the entity); thus, any geometric transformation can be characterized by a function between points. Application of a geometric transformation to a PTSET entity S can be denoted:

```
transform(T,S)
```

where T is a specification of the transformation to be performed. Syntactically, transformation specifications can be defined "on the fly" in a query or PDM algebra expression, or declared in the database

and stored for later access. Transformations can be supported directly through the use of multiargument functions. Transformations between entire spaces may be defined in the same way as those between individual spatial entities, since both individual spatial entities and whole spaces are denoted by entities of type PTSET.

The point set operations defined above form an algebra on point sets. As a result, given these operations, PTSET entities denoting complex "shapes" can be constructed by specifying "algebraic expressions" of point set operations applied on (possibly transformed) PTSET entities.

The *structural operations* are concerned with the hierarchical structure of spaces described earlier. In general, these are convenient "macros", as they can be defined in terms of the non-spatial operators of the PDM algebra. The *object set operations* are defined for spaces S1 and S2 denoting the *same* point set, but having possibly different contained PTSETs (i.e. S1 and S2 register different information about a single point set). The definitions are as follows:

- *Object union* -- The object union obunion(S1,S2) is a space S3 denoting the same point set as S1 and S2 that contains the set of PTSET objects contained in S1, S2, or both. (The objects in S3 may not be spatially distinct although their identities are retained.)
- *Object intersection* -- The object intersection obintersect(S1,S2) is a space S3 denoting the same point set as S1 and S2 that contains the set of PTSET objects contained in both S1 and S2.
- *Object difference* -- The object difference obdiff(S1,S2) is a space S3 denoting the same point set as S1 and S2 that contains the set of PTSET objects contained in S1 and not in S2 (again, difference is not symmetric).

The operation sinsert(P1,P2) takes an entity P1 of type PTSET and inserts an entity P2 of type PTSET into it (P2 must be capable of being fully contained within P1). The semantics of sinsert can be described in terms of operations on the CONTAINS function described above. In its most primitive form, sinsert(P1,P2) simply adds P2 to entity P1's CONTAINS function. More complete information may be captured by allowing P2 to be the result of some spatial transformation operation on another PTSET entity, as in sinsert(P1,transform(P2,spec)), where "spec" denotes the specification of of the transformation to be performed on P2 prior to inserting it in P1. This captures not only the fact that P2 is contained in P1, but *where* within P1 entity P2 is actually located. Such specifications may be more or less precise, depending on the subclass of PTSET entities involved.

The *expand* and *reduce* operators provide additional control over the CONTAINS relationship in spaces. If S is a space, X is in S's CONTAINS function, and Y is in X's CONTAINS function, expand(S) produces a space S' denoting the same point set, having moved Y into S's CONTAINS function without altering X's CONTAINS function. That is, for each immediate child X of S, expand(S) effectively copies each child of X so that it is also an immediate child of S. Note that placing an object in a new space (Y in the space of S in the above example), requires computation of the position of the object within the space. If the position is specified as a transformation, then a composition of transformations is necessary (e.g. multiplication of 4 x 4 matrices). Reduce is, in some sense, the inverse of expand. Reduce(S) produces a space S' having no immediate children that are also contained in some other (immediate or indirect) child object of S.

Finally, since the substructure of a particular spatial representation is structured hierarchically, it is possible to use recursive processing techniques to search this hierarchical structure, by traversing the CONTAINS relationship. Such recursive processing techniques are also being developed in the PROBE DBMS [DAYA85, ROSE86].

## 4. Example

This example illustrates the type of definitions possible within the model, once the appropriate subtypes are defined. It shows the definition of the shape of a simple missile in 3 dimensions, using a "constructive solid geometry" approach, in which primitive 3D shapes are combined using point set operations to give a complex result. The entire shape might then be assigned as the value of the SHAPE function of a PART entity defining the missile.

```
create new C in CONE (NAME=>NOSE,R=>12,H=>40)
create new CY in CYLINDER (NAME=>BODY,R=>12,H=>180)
create new W in RECTANGLE (NAME=>WING,Z=>20,X=>2,Y=>180)
create new H in RECTANGLE (NAME=>HORIZ,Z=>20,X=>2,Y=>80)
create new V in RECTANGLE (NAME=>VERT,Z=>20,X=>80,Y=>2)
create new G in SPHERE (NAME=>GYROSCOPE,R=>8)

create new CS1 in GTRANS (OP=>translate(0,0,40),WRT=>C)
create new CS2 in GTRANS (OP=>translate(0,0,60),WRT=>CS1)
create new CS3 in GTRANS (OP=>translate(0,0,90),WRT=>CS2)
create new CS4 in GTRANS (OP=>translate(0,0,10),WRT=>C)

C := C sinsert transform(G,CS4)

create new S in 3DSOLID (NAME=>MISSILESHAPE,
 DEFINITION=> C ptunion transform(CY,CS1) ptunion
             transform(W,CS2) ptunion transform(H,CS3)
             ptunion transform(V,CS3))
```

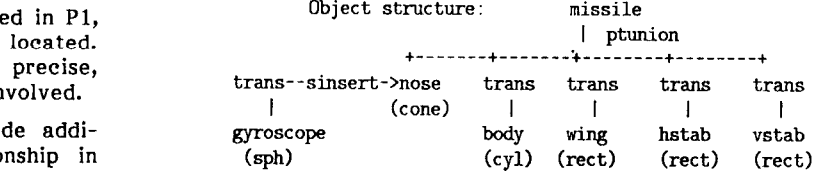


Figure 4.1 Cruise Missile Definition

The definition is shown in Figure 4.1.

Each primitive shape is predefined as an entity type (a subtype of the specialized spatial data type "3DSOLID", which is itself a subtype of the general spatial data type PTSET). Thus, when new entities of these types are created with specific values of their parameters (e.g. a specific radius and height), 3D objects are actually being created. For example, CY is a cylinder defining the missile body.

In subtype 3DSOLID, each primitive object is defined with its own default coordinate system. In order to combine objects, they must be transformed into the same coordinate system (as opposed to requiring all objects to be defined absolutely with respect to the same coordinate system). The necessary transformations are defined as objects of type GTRANS. For example, CS1 translates the body with respect to the nose cone. The shape of the missile is then defined by performing combinations of transformations and set operations on the primitive shapes. Syntactically, the definition of this shape is assigned to the DEFINITION function of the generic 3DSOLID entity representing the shape, since, unlike a CONE, the definition is not implicit in the type of entity involved.

The insertion of an independent spatial object (a gyroscope) into the set of points defined by the missile's shape (specifically, into the missile's nose cone) is also illustrated. In this case, the nose cone is considered as a "space", into which other objects might be inserted. The operator used for this purpose is the "sinsert" operator. The resulting missile shape is shown in Figure 4.2 inside a "shipping box".

To illustrate show such spatial data might be used in queries, suppose we've defined a shipping box shape and want to see if certain versions of the missile will fit in it. This is a simple example of "interference checking", performed quite often in CAD systems.

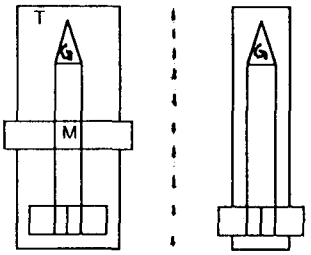


Figure 4.2 Missile (partially) in Shipping Box

The shape of the box would be defined, using a declaration such as "create new B in RECTANGLE (NAME=>BOX, Z=>250, X=>160, Y=>75) (this box is a solid representing the inside space of the shipping box).

A query would then be specified to access the shapes of both the box and specific missile versions and test them to see whether the missile shape can be entirely contained within the box shape. If not, an indication that the particular version does not fit in the box would be printed. The query (in the PDM Daplex query language) would be:

```
for each M in PART where NAME(M) = "missile"
  for each V in VERSIONS(M) where
          RELEASE-DATE(V) > "09-26-82"
    for each T in PART where NAME(T) = "shipbox"
      if SHAPE(V) is not contained in SHAPE(T) then
       print(REP(RELEASE-DATE(V)),"does not fit");
end; end; end;
```

While this example is necessarily rather simple, it illustrates the basic ideas involved. Quite complex shapes can be constructed, and spatial relationships tested, using the basic set operators.

## 5. Supporting the spatial data model

This section is concerned with the implementation of the spatial data model. As discussed above, the structural operations can be defined and implemented in terms of PDM algebra. In this section we concentrate on the point set operations since they cannot be handled using "conventional" DBMS facilities. It is possible to represent point sets using known techniques (e.g. boundary representation), implement algorithms for each geometric operator, and encapsulate all this in an object class. There are two problems with this approach. First, it is difficult to do. The algorithms would have to be concerned with sets of spatial objects (and with attendant problems relating to secondary storage and buffering). This complicates the implementation of the spatial object class. Secondly, it is likely to be very slow. With common representations such as constructive solid geometry and boundary representation, it is difficult to avoid doing work that is "obviously" unnecessary (i.e. obvious if you look at a picture.) For example, to find all polygons in a set, S, that overlap a given polygon P (in 2d), a boundary representation algorithm would have to compare edges of polygons in S against edges in P. Any optimizations would have to be explicitly coded. (For example, ignore a polygon p of S if a box containing p and a box containing P do not overlap.)

Ideally, the DBMS would take care of collections of objects while the spatial object class would take care of individual objects and interactions among them. This would reduce the amount of work required to extend the DBMS with a new object class. Taking this approach means that optimization involving collections of objects must take place in the DBMS.

This is the motivation behind including "approximate geometry" (AG) in the DBMS. AG can "take care of collections of objects", implement optimizations on collections of objects, and make use of an "object-at-a-time" interface to spatial object classes. Generally, algorithms can be optimized using AG if they rely on 1) iteration over the objects in one or two spaces, 2) a spatial predicate to detect "interesting" objects or pairs of objects, and 3) a procedure to handle these objects or pairs. Parts (1) and (2) can be done very efficiently using widely applicable AG techniques. The spatial object class would have to supply parts (2) and (3). (Part (2) is done approximately in AG and precisely in the spatial object class.)

Our approach to AG is based on a "grid" or "raster" representation of spatial objects. Many complex spatial operations can be implemented with very simple algorithms given a grid representation. The algorithms usually involve iteration over all cells or pixels of the grid, performing the same basic step for each pixel. In practice, it is not feasible to store grids explicitly. At high resolution, the space requirements are too high and iteration over all pixels is too slow. The techniques to be presented can be thought of as methods that optimize the handling of grid or raster representations. Support for the geometric operations using approximate geometry will be discussed in section 5.2.

### 5.1 Supporting the structural operations

The structural operations create new spaces and manipulate the CONTAINS function of spaces. These operations can be implemented using the operations of PDM algebra. The implementation of the object set operations is fairly straightforward. For expand and reduce, it is necessary to compute the "square" of the containment relationship (as reflected in the CONTAINS function). That is, for all spaces, find objects that are contained in contained objects. To compute expand, these indirect containment relationships are added to the current set; for reduce they are removed.

### 5.2 Supporting the geometric operations

In this section we will discuss the basic ideas of approximate geometry and show how these ideas are used in supporting the geometric operations of the spatial data model. The details of the algorithms are beyond the scope of this paper; see [OREN85, OREN86] for more complete discussions.

The essential idea behind approximate geometry is the decomposition of PTSET entities into box-shaped elements; each entity is approximated by the ptunion of its elements. The approximation covers at least the space occupied by the PTSET entity. Thus, approximate geometry provides a filter. For example, if the approximations of objects A and B do not overlap, then A and B definitely do not overlap. If the approximations do overlap, then A and B do not definitely overlap.

The decomposition of PTSET entities is carried out in a highly constrained way. The decomposition strategy used leads to 1) a very concise representation of the elements, 2) very simple spatial relationships between elements, and 3) a useful ordering of the elements, "z order". For any two elements, either one contains the other, or one precedes the other in z order.

The absence of overlap (other than containment) and the presence of a total ordering allows the use of very simple algorithms based on the merging or traversal of z-ordered sequences of elements. All the geometric operations discussed in section 3 can be supported in AG by such algorithms. Spatial selection can be supported by the "spatial join". Given two sets of objects, S and T, the spatial join detects all pairs of objects (s, t), such that s is in CONTAINS(S), t is in CONTAINS(T), and the approximations of s and t overlap.

A very attractive feature of the spatial join is that it can be incorporated into existing DBMSs with very little effort. The access methods and buffering strategies in current use (e.g. B-trees and LRU page replacement) provide exactly the right foundation for the implementation of AG. Furthermore, it appears that performance comparable to the best obtainable with "custom" algorithms can be obtained.

Note that the refinement of the approximate results by an "exact geometry" object class is simple. The interface to the DBMS (implementing AG) is "instance-at-a-time", not "set-at-a-time". Therefore, adding such an object class to the PROBE DBMS should be simpler than what would be required in [STON83].

Overlay is a more difficult spatial operation to support in a DBMS because, unlike spatial selection, it does not resemble any common database operation (e.g. from PDM algebra or relational algebra). However, a short, "one-pass" AG algorithm for computing overlay of an AG representation is known [OREN85]. Following the computation of overlay on the AG representation, the exact version can be computed quickly - the AG version identifies which objects participate in which non-empty uniform regions.

Ptunion, ptintersect, and ptdiff are easy to compute by merging sequences of elements. The details are in [OREN85]. With the spatial selection and overlay operations, the computation of the AG version of the operation reduced the amount of work that had to be carried out with the exact representation. That does not appear to be the case here. For example, computing the AG version of ptunion does not appear to be useful in computing the exact version. However, it is imporant to provide AG versions of PTSET operations because later processing may benefit. For example, if solids are described using constructive solid geometry, then AG representations of the objects can be built also (using the point set operations). Having constructed the AG representations, interference detection (for example) can be optimized using AG. This would not have been possible had the AG versions of the point set operations not been computed.

## 6. Current Work

Current PROBE activities include further development of the ideas described above. For example, we are working on augmenting the "containment" relationships described above with other relationships, such as adjacency, that are important in dealing with spatial data. We are also working to incorporate temporal data as a special case within the spatial data framework. Current results indicate that this is valid from both the modelling and implementation viewpoints. Finally, we are developing a "breadboard"

imlementation that will demonstrate some of the PROBE facilities, including approximate geometry, using a VLSI CAD application.

*Acknowledgements*

We gratefully acknowledge the contributions of Alex Buchmann, Umesh Dayal, David Goldhirsch, Sandra Heiler, and Arnie Rosenthal to the ideas described here, and the contributions of one of the referees to the presentation.

## 7. References

[CHAN81]
S.-K. Chang, ed., "Pictorial Information Systems", special issue, *Computer, 14*, 11 (November 1981).

[DAYA85]
Dayal, U., et.al., "PROBE - A Research Project in Knowledge-Oriented Database Systems: Preliminary Analysis", Technical Report CCA-85-03, Computer Corporation of America, July 1985.

[DAYA86]
Dayal, U., and J.M. Smith, "PROBE: A Knowledge-Oriented Database Management System", to appear in M.L. Brodie and J. Mylopoulos (eds.), *On Knowledge Base Management Systems: Integrating Artificial Intelligence and Database Technologies*, Springer-Verlag, 1986.

[FOX84]
S. Fox, T. Landers, D. R. Ries, R. L. Rosenberg, "Daplex User's Manual", CCA-84-01, Computer Corporation of America, November 1984.

[IEEE77]
*Proc. IEEE Workshop on Picture Data Description and Management*, 1977.

[MALL82]
W.R. Mallgren, "Formal Specification of Graphic Data Types", *ACM Trans. Prog. Languages and Systems*, 4,4, October 1982.

[MANO86]
Frank Manola and Umeshwar Dayal, "PDM: An Object-Oriented Data Model", submitted for publication, April 1986.

[MANT82]
M. Mantyla and R. Sulonen, "GWB: A Solid Modeler with Euler Operators", *IEEE Computer Graphics and Applications*, September 1982.

[MORE85]
Scott Morehouse, "ARC/INFO: A Geo-Relational Model for Spatial Information", *Proc. Seventh Intl. Symp. on Computer-Assisted Cartography*, American Congress on Surveying and Mapping, 1985.

[OREN85]
Jack A. Orenstein, "Spatial Query Processing in PROBE", CCA Working Paper, December 1985.

[OREN86]
Jack A. Orenstein, "Spatial Query Processing in an Object-Oriented Database System", *Proc. 1986 ACM-SIGMOD Int'l Conf. on Management of Data.*

[REQU80]
Aristides A.G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems", *Computing Surveys 12*, 2 (December 1980).

[ROSE86]
A. Rosenthal, S. Heiler, U. Dayal, F. Manola, "A DBMS Approach to Recursion", *Proc. 1986 ACM-SIGMOD Intl Conf. on Management of Data.*

[SCHE82]
H.-J. Schek, P. Pistor, "Data Structures for an Integrated Data Base Management and Information Retrieval System", *Proc. VLDB 8*, (1982), 197-207.

[SHIP81]
Shipman, David, "The Functional Data Model and the Data Language DAPLEX", *ACM Trans. Database Systems*, 6,1 (March 1981).

[STON82]
M. Stonebraker, et. al., "Document Processing in a Relational Data Base System", *ACM TOOIS 1*, 2 (1983), 143-158.

[STON83]
Michael Stonebraker, Brad Rubenstein, and Antonin Guttman, "Application of Abstract Data Types and Abstract Indices to CAD Databases", *Proc. Database Week: Engineering Design Applications*, IEEE Computer Society, 1983.