APPLICATION OF ABSTRACT DATA TYPES AND ABSTRACT INDICES TO CAD DATA BASES

Michael Stonebraker
Brad Rubenstein
Antonin Guttman

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
UNIVERSITY OF CALIFORNIA
BERKELEY, CA.

ABSTRACT

This paper explores the use of one form of abstract data types in CAD data bases. Basically, new data types for columns of a relation, such as boxes, wires and polygons, become possible. Also explored is the possibility of secondary indices for new data types that can support existing and user-defined operators. The performance and query complexity considerations of these features are examined

I. INTRODUCTION

It has been pointed out [HASK82, KATZ82] that Computer Aided Design (CAD) applications are not particularly well suited to current relational data base management systems. Extensions or modifications appear desirable to deal with the following issues:

a) Support for new data types such as polygons, rectagons, text strings, etc.

b) Support for efficient spatial searching.

c) Support for complex integrity constraints.

d) Support for design hierarchies and multiple representations.

The first issue arises because CAD applications are not well served by the integers, floating point numbers and character strings prevalent in business data processing applications. Moreover, spatial searching is needed for design operations that involve objects which fall in a specific area, such as the display of a portion of a VLSI design on a CRT screen. Spatial searching is not effectively supported by existing general purpose DBMSs. The third issue arises because CAD designers often wish complex integrity constraints, such as integrated circuit layout rules, to be enforced for their data.

The last issue arises because many design environments have hierarchical levels of detail. For example, a VLSI integrated circuit might have several intermediate levels of detail between one containing the whole chip as a single black box and the one containing detailed spatial masks for circuit cells at the lowest level. These intermediate levels suppress details irrelevant to that particular level, and a designer can use whatever level of detail fits his particular needs. In addition, more than one view of the design may exist simultaneously, giving multiple overlapping representations for data base objects. An example would be a bit-slice design for a CPU. For purposes of describing its physical construction, the design is made up of several parallel bit-slices, but a functional block diagram may consist of separate boxes for the ALU, register file, etc.

In [KATZ82] several approaches are suggested for various of the above issues. In this paper we report on the success observed using one approach, abstract data types, as a solution to issue a) and issue b) above.

The remainder of this paper is organized as follows. In Section II we briefly review our use of abstract data types. A more complete discussion appears in [STON82]. Then in Section III we describe extensions to secondary index facilities to support abstract data types. Extended secondary indices can provide efficient spatial searching as well as other kinds of indexing. Next, in Section IV we apply abstract data types to a data base of VLSI design information used in [GUTT82]. Lastly, in Section V we report on the performance implications of abstract data types by redoing the performance comparison between a relational data base system and a special purpose CAD system reported in [GUTT82] and performing other experiments.

II. ABSTRACT DATA TYPES

Abstract data types (ADTs) [LISK74, GUTT77] have been extensively investigated in a programming language context. Basically, an ADT is an encapsulation of a data structure (so that its implementation details are not visible to an outside client procedure) along with a collection of related operations on this encapsulated structure. The canonical example of an ADT is a stack with related operations: new, push, pop and empty.

ADTs have been considered extensively in the context of semantic data modeling and as a central theme in data base system implementation [LOCK79]. Moreover, the use of ADTs in a relational data base context has been discussed in [ROWE79, SCHM78, WASS79]. In these proposals a relation is an abstract data type whose implementation details are hidden from application level software. Then, allowable operations are defined by procedures written in a programming language that supports both data base access and ADTs. One use of this kind of abstract data type is suggested in [ROWE79] and involves an EMPLOYEE abstract data type with related operations hire-employee, fire-employee and change-salary. This use of ADTs can also limit access to a relation in prespecified ways, thereby guaranteeing a higher level of data security and data integrity. Also, a view [STON75] can be defined as an ADT. Consequently, the algorithm that transforms updates on views into updates on relations actually stored in the data base can be encapsulated in the ADT, thereby providing a high degree of data independence.

This section presents a different use of ADTs, in par-

This Research was supported by the Advanced Research Projects Agency under Contract #N00039-C-0235.

ticular [illegible] individual columns of a relation. The goal is to extend the semantic power of a relational data base system by providing definition of new data types and related operators on these data types by means of defined procedures obeying a specialized protocol. This use of ADTs is a generalization of data base experts [STON80]. We explain our use of ADTs with an extended example concerning geometric objects.

In computer aided design of integrated circuits, objects are often made up of small rectangular pieces called 'boxes". For a VLSI data base one would like to be able to define a column of a relation of type "box". For example, one might define a boxes relation as follows:

```
create boxes (owner = i4,
              layer = c15,
              box-desc = box-ADT)
```

Here, the boxes relation has three fields: the identifier of the circuit of which each box is a part, the processing layer for the box (polysilicon, diffusion, etc.) and a description of the box's geometry. All fields are represented by standard built-in types except box-desc which is a data type added by an ADT implementor.

Tuples can be added to this relation using QUEL [STON76] as follows:

```
append to boxes (owner = 99,
                 layer = "polysilicon",
                 box-desc = "0,0,2,3")
```

Clearly, all fields can be correctly converted to an internal representation and stored in a data base system with the exception of the string "0,0,2,3", which represents the box bounded by x=0, y=0, x=2, y=3. In order to be interpreted as the description of a box, special recognition code is required. Basically an input procedure must be available to the DBMS to perform the conversion of the character string "0,0,2,3" to an object with data type box-ADT. Such a routine is analogous to the procedure ascii-to-float which converts a character string to a floating point number. A DBMS has a collection of built-in conversion routines for standard types, and our proposal entails allowing additional conversion routines to be supplied by an ADT implementor.

One would also like to use standard DBMS operators on the box-ADT domain, e.g.

```
range of b is boxes
replace b (box-desc = b.box-desc * "0,0,4,1")
      where b.owner = 99
```

The intended effect of this command is to replace the box by its intersection with another box. The intended semantics of * are those of intersection; consequently, the data manager must be instructed how to interpret the multiplication of two objects of type box-ADT. In this case "0,0,4,1" will be converted to an object of type box-ADT to match the type of b.box-desc, and a procedure must be available to perform the appropriate multiplication.

Next one would like to define new functions on the box-ADT column. Numerical columns have sin, cos, log, etc. defined as built-in functions. Each of these accepts an integer or floating point number as input and returns a floating point number. Similarly, one might want to define a function that calculates the area of a box and use it in data manipulation commands, e.g.

```
retrieve (b.owner)
where area(b.box-desc) > 100
```

One must define for the data base manager the function "area" which accepts a box-ADT as input and returns an integer.

In addition, one might want to define new comparison operations. For example, one might wish to define the concept "overlaps" for boxes, and to have a corresponding operator, "||", defined for this purpose. The overlap operator could then be used to ask if there were any box overlapping the unit square based at the origin as follows:

```
range of b is boxes
retrieve (b.box-desc)
      where b.box-desc || "0,0,1,1"
```

Not surprisingly, a procedure defined by the ADT implementor for the || operator is required.

Lastly, one would like to be able to define aggregate functions for the new type. For example, one would like to be able to find the owner of the box with the largest vertical dimension on the polysilicon layer:

```
retrieve (b.owner) where b.box-desc =
      tallest(b.box-desc where b.layer = "polysilicon")
```

Again, a routine is required to define the "tallest" function for the new data type.

As a result an ADT is the following abstraction.

a) a registration procedure whereby a DBMS is informed of the existence of the new type and given the length of its internal representation.

b) a collection of routines which define the semantics of operators for this type and perform conversions to other types. These routines must obey a prespecified protocol for accepting arguments and returning results. Once defined by the ADT implementor, the new type and operators become available to other users of the DBMS.

c) small changes to the parser and query execution engine to correctly parse commands with new operators and call the routines defined by the ADT implementor when appropriate during execution.

This abstraction has been implemented in the INGRES relational data base system and is about 2500 lines of code Details on implementation issues concerning parser tables, overloading of operators, security and dynamic loading of routines are addressed in [FOGG82, ONG82]. It appears to execute with modest performance degradation [FOGG82]. It should be noted that the implementation also allows definition of new operators and functions for ordinary data types.

III. EXTENDED SECONDARY INDICES

The preceding section has indicated a mechanism for adding new data types and new operators to a relational data base system. This section explores the possibility of supporting secondary indices in this more general environment.

Traditionally, secondary indices provide a fast access path to required data items when a query specifies an exact match with a user specified value or a comparison operator applied with a specified value. For example, if secondary indices exist for the name and salary fields of an employee relation, then the queries

```
range of e is employee
retrieve (e.salary) where e.name = "Jones"
retrieve (e.name) where e.salary > 1000
```

can be answered by using indices.

Since new operators can be defined for normal data types and for new ADT data types, one would want the following capabilities.

1) Indices on ADT columns with existing operators.
For example, consider the situation where salaries of employees are stored as packed decimal numbers. Since this is not one of the built-in data types in many systems, an ADT is required. One would want to index salary so that the above query could be answered effectively In this case extending an indexed sequential access method to support the new data type will be adequate.

2) Indices on normal columns using new operators.
For example, consider the query:

```
range of e is employee
retrieve (e.name) where e.name !! 7
```

which requests the names of employees whose names are exactly 7 characters long. The new operator !! counts the number of non-blank characters in a name and compares the result to an integer operand. One would want a secondary index for the !! operator so that this query could be efficiently answered. Clearly, an index which provided a bin for each possible length would be appropriate.

3) Indices on ADT columns with new operators.
Consider the query from Section II to find all the boxes that intersect the unit square at the origin

```
range of b is boxes
retrieve (b.box-desc)
        where b.box-desc || "0,0,1,1"
```

We need an index that will allow retrieval of only the boxes that qualify, or will at least restrict the search to a small subset of the boxes relation.

The objective of this section is to propose a scheme which supports all three capabilities above. It has always been our position that an appropriate collection of access methods should be provided by any DBMS and that it should be easy to add new ones [STON76]. Hence, our goal is to allow any access method to operate in the more general context of ADTs. Hashing and indexed-sequential (ISAM) are the access methods currently supported by INGRES, and we focus the discussion on extending these. As a running example, we use the boxes relation defined above.

An index can be created using the INGRES index command:

```
index on boxes is b-index (box-desc)
```

This will create a relation of the following form which will be used as a secondary index:

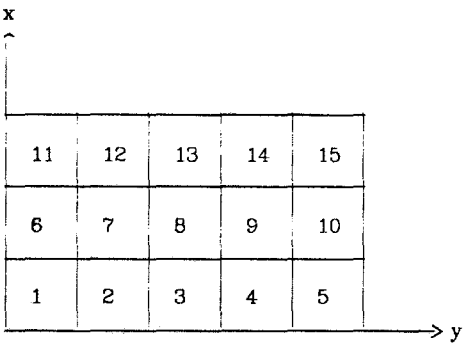
```
b-index (box-desc, pointer-to-tuple)
```

A row exists in b-index for each row in the boxes relation and contains the box-desc field along with a pointer to the given tuple in boxes. The index b-index is initially stored as a heap and must be modified to hash or ISAM to be useful. For example:

```
modify b-index to hash on box-desc using my-function
```

The only change to the current modify command is the inclusion of a "using" clause. INGRES normally builds hashed secondary indices by allocating a number of buckets, then reading the tuples one by one, calling its internal hash function to obtain a bucket number and storing the tuple in the correct bucket. In this context INGRES calls my-function instead of its built-in hashing function to obtain bucket numbers. My-function must be a valid function registered through the ADT registration facility which expects a box-ADT as an argument and returns an integer. No other modifications are required to the code if my-function returns a single integer.

However, suppose we have a grid in the x-y plane as shown in Figure 1, and we want my-function, when passed a box, to return the numbers of all the grid cells that it intersects.



A grid structure for my-function
Figure 1

Grid cell zero is reserved for boxes which extend outside the boundary of the above structure. In this situation my-function returns a list of buckets instead of a single bucket number and INGRES must insert a row in the appropriate bucket in b-index for each value in the list. The modify command for this structure is

```
modify b-index to hash on box-desc
        using my-function (param-list)
```

Here param-list is a character string containing necessary information such as the number and size of the grid squares and the location of the grid in the plane. These values could be hard-wired into my-function, but it is preferable that they be setable for each index.

We now illustrate how to use an ISAM structure with new columns and operators. Again, we could run the following modify command:

```
modify b-index to isam (box-desc) using <+
```

Normally, an ISAM structure is built by sorting the values for box-desc using the built-in operator < to define the sort order. In this case the index can be built in an analogous way by substituting the operator <+ to define some ordering on boxes, for example by comparing their areas. <+ would be expected to compare two box descriptions and return true or false if one was "less than" the other. The ISAM structure would then support the ordering determined by <+.

Once a hashed or ISAM secondary index is created for the boxes relation, one must specify to INGRES how the index can be used in processing queries. INGRES has a built-in function, FIND, for hashed structures which will return the hash buckets which must be inspected for tuples which satisfy a particular query. In the current implementation a hash bucket is identical to a UNIX page, so FIND returns a collection of pages. An analogous FIND function returns a collection of pages for an ISAM structure. These functions are called by specifying the value used in a qualification and the operator involved. For example, for the qualification

```
where e.salary > 1000
```

FIND is called with parameters > and 1000. In our extended environment, a FIND function must be provided for each possible operator for which the index can be used. We propose a new INGRES command for the purpose, i.e.

```
use b-index with find-function
        for (|| box-ADT, box-ADT ||)
```

This command specifies the circumstances under which the the routine, find-function, should be called to provide the required collection of ppages to search. The above example indicates that this function is appropriate when the intersection operator || is encountered connecting a variable and a value of type box-ADT. Moreover, the value can be on either side of the || operator. For example, suppose one submitted the query:

```
range of b is boxes
retrieve (b.all)
where b.box-desc || "0,0,1,1"
```

The string on the right is converted automatically to an object of type box-ADT because || is defined to take box-ADT arguments. After the conversion, the qualification is of the form

```
where b.box-desc || box-ADT
```

and therefore b-index can be used to process the query. The ADT function find-function is called to return a list of pages which must be examined. Then, INGRES simply iterates over the list examining each index entry, following the appropriate pointer, obtaining a tuple from boxes and finally evaluating the user's qualification to ascertain if it is satisfied for the tuple in question.

It is possible to define different FIND routines for different operators as illustrated below. Suppose one defines a new operator "#|" which compares a box and a line and returns true if the box is "to the left of" the line. The index b-index can be used to process queries involving the #| operator; however, a new FIND function must be used:

```
use b-index with second-fn for (#| line-ADT)
```

A user can submit a query such as

```
retrieve (b.all)
where b.box-desc #| "0,0,1,3"
```

whereby he wants to see all boxes which are to the left of the line from (0,0) to (1,3). If the grid structure for b-index from Figure 1 is one unit long on each side, then the boxes which qualify must lie in grid cells 1, 6, 11 or 0 and the others can be excluded. The function second-fn can provide the needed semantics.

When more than one index can be used to process an INGRES query, e.g.

```
where b.box.desc || "0,0,1,1"
    and b.box.desc #| "0,0,1,3"
```

then INGRES must choose which index to use in processing the query. This is currently done by a hard-wired strategy routine. To be able to choose in the above context, this routine must be generalized to call both find functions to obtain list of pages and then compare the sizes of the results, choosing the smaller list for iteration.

IV. APPLICATION OF ABSTRACT DATA TYPES

In [GUTT82] we described a CAD data base consisting of integrated circuit descriptions as stored by a special purpose graphics editor, KIC [KELL81]. A KIC data base consists of a collection of circuit "cells". Each cell can contain mask geometry and subcell references. Circuit designs are hierarchical; a complete design expands into a tree, with a single cell at the root and instances of other cells, used as subcells, for the non-root nodes. Cells are the building blocks used to construct a circuit and include such objects as buffers, NOR gates and at a higher level, PLAs and arithmetic logic units.

In [GUTT82] we also described a relational schema which models this data base consisting of five main relations:

cell-master (name, author, master-id)

box (owner, layer, x1, x2, y1, y2)

wire (owner, layer, wire-id, width, x1, y1, x2, y2)

polygon (owner, layer, polygon-id, vertnum, x, y)

cell-ref (parent, child, cell-ref-id, t11-t32)

In the *cell-master* relation, *name* is the textual name given to the cell and *author* is the name of the person who designed it. *Master-id* is a unique identification number assigned to each cell. It is used for unambiguous references to the cell within the data base.

The *box* relation describes mask rectangles. *Owner* is the identifier of the cell of which the box is a part. *Layer* specifies the processing layer, e.g "polysilicon" or "diffusion". *X1* and *x2* are the x-coordinates of the left and right sides of the box while *y1* and *y2* are the y-coordinates of the top and bottom.

A "wire" is a set of lines that serves to make an electrical connection between different parts of a circuit. Each tuple in the *wire* relation describes one line segment, giving the coordinates of its centerline *(x1, y1, x2, y2)* and its *width*. *Wire-id* is a unique identifier for a particular wire. *Owner* and *layer* mean the same as in the *box* relation.

A polygon is a closed figure with any number of vertices. One vertex is stored in each tuple of the *polygon* relation. *X* and *y* are the coordinates of the vertex, and *vertnum* orders the vertices (tuples) within one polygon.

Each *cell-ref* tuple describes the use of one cell as a part of another, i.e. as a subcell. For example, suppose that the cell REGISTER contained several LATCH subcells. Then, there would be several *cell-ref* tuples, each containing the identifier of the REGISTER cell in the *parent* field, and the identifier of the LATCH cell in the *child* field. *T11* through *t32* are a 3 X 2 matrix of floating point numbers specifying the location, orientation and scale of each subcell with respect to its parent. This representation of a spatial transform is the one generally used in computer graphics [NEWM79].

To apply abstract data types to this application, we suggest the following:

a) A data type "box-ADT".
The internal representation of a box can be four integers representing the locations of the top, bottom and side boundaries. The external representation can be a character string consisting of numbers separated by commas, e.g. "0,0,1,1"

b) A data type "polygon-ADT".
The internal representation can be a fixed length string of integers if a maximum number of vertices is specifiable. For example, if 25 vertices are allowable, then 50 integers can represent any polygon. If no upper bound is possible, then ADTs can still be used. One can allocate a polygon file external to the data base system which will be used to store polygon descriptions. When a new polygon is inserted, it will be physically placed in the external file (say using a first fit or best fit placement algorithm). The input conversion routine will then return a byte offset and length which will be stored as a fixed length object in the data base.

c) A data type "wire-ADT".
If all segments of a wire are the same width, then the internal representation can be one integer for the width and four integers for each segment giving the coordinates of the endpoints.

d) a data type "array of floats"
The internal representation is the obvious one for the required 3x2 matrix, t11, .. t32.

With these new data types we can simplify the schema above to:

cell-master (name, author, master-id, defined)

box (owner, layer, box-desc)

wire (owner, layer, wire-desc)

polygon (owner, layer, polygon-desc)

cell-ref (parent, child, cell-ref-id, orientation)

Notice that box-desc, wire-desc, polygon-desc and orientation are all new types.

The performance experiments in [GUTT82] involved three common operations on VLSI data, namely retrieval of the top level geometry for a given circuit cell, full expansion of a design tree and retrieval of the top level geometry which falls in a particular geographic area. We express the first and third queries for the original INGRES schema below. Then, we introduce new operators for our abstract data types which will simplify the description of these queries.

In the following set of queries which retrieve the top level geometry, CELLID identifies the cell to be displayed. The first two queries simply retrieve all the boxes and wire segments belonging directly to the designated cell in any order. The third query, which retrieves polygon vertices, is more complicated because the vertices must be produced in the correct order and grouped by polygon-id and layer in order to simulate the operation of KIC. All polygon data belonging to the given cell is first gathered into a temporary relation, which is then sorted, and finally the data is retrieved from the temporary and passed to the user. In the actual performance tests data was retrieved separately for each layer to accurately emulate the operation of KIC.

```
range of b is box     /* repeated for each layer */
retrieve (b.x1, b.y1, b.x2, b.y2)
where b.owner = CELLID and b.layer = value

range of w is wire    /* repeated for each layer */
retrieve (w.layer, w.x1, w.y1, w.x2, w.y2)
where w.owner = CELLID and b.layer = value

range of p is polygon
retrieve into ptemp
      (p.layer, p.polygon-id, p.vertnum, p.x, p.y)
where p.owner = CELLID and b.layer = value

modify ptemp to heapsort
      on layer, polygon-id, vertnum

range of pt is ptemp  /* repeated for each layer */
retrieve (pt.layer, pt.polygon-id, pt.x, pt.y)
```

The query below retrieves polygons from the top level geometry which fall in a specific geographic window. LEFT, RIGHT, BOTTOM and TOP are numbers giving the boundaries of the window. Again, the modify command is required to correctly order polygon vertices

```
range of p is polygon
retrieve into ptemp
      (p.layer, p.polygon-id, p.vertnum, p.x, p.y)
where p.owner = CELLID

modify ptemp to heapsort
      on layer, polygon-id, vertnum

range of pt is ptemp
retrieve (pt.layer, pt.polygon-id, pt.x, pt.y)
where max(pt.x by pt.polygon-id) > LEFT
and   min(pt.x by pt.polygon-id) < RIGHT
and   max(pt.y by pt.polygon-id) > BOTTOM
and   min(pt.y by pt.polygon-id) < TOP
```

With our abstract data types, we can rewrite the queries for top-level geometry as

```
range of b is box
retrieve (b.layer, b.box-desc)
where b.owner = CELLID and b.layer = value

range of w is wire
retrieve (w.layer, w.wire-desc)
where w.owner = CELLID and b.layer = value

range of p is polygon
retrieve (p.layer, p.poly-desc)
where p.owner = CELLID and b.layer = value
```

This version will run faster because the detailed representation of each kind of geometric object will be handled by special routines instead of the general purpose query interpreter. Polygon retrieval will be much faster, since it is no longer necessary for the data base system to put the vertices in the correct order.

If we also use a polygon overlap operator <> similar to the box overlap operator defined above, then we can rewrite the query for polygons in a specific window as

```
range of p is polygon
retrieve (p.layer, p.polygon-id, p.poly-desc)
where p.owner = CELLID
and p.polygon-desc <>
      make-poly("LEFT,RIGHT,BOTTOM,TOP")
```

The new query has several advantages over the original one. First, it is much clearer once the meaning of <> is understood, because We avoid the awkward collection of clauses in the qualification. Moreover, it will be faster because the test for overlap with the window can be done more efficiently in a special routine. Also, if an index using the <> operator exists for polygon-desc then INGRES can use it automatically to limit the number of polygons inspected. Since the original form of the qualification contained aggregates, no index could be effectively used and a search of all polygons was required.

V. PERFORMANCE COMPARISON

As noted in Section IV, performance improvement from the use of ADTs can be realized from four sources:
1) Manipulation of a smaller number of columns. For example, a box can be retrieved as a single column rather than as four constituent parts.
2) Manipulation of fewer tuples, for example when polygons are represented by single fixed-size tuples with vertex lists stored externally.
3) Simplification of queries due to the introduction of new operators. This is especially noticeable in spatial windowing.

4) Use of abstract indices.

In this section we report on three different experiments. First, we redo the performance comparison between KIC and INGRES noted in [GUTT82]. This shows the effect of sources 1, 2 and 3. Since abstract indices are not yet operational, our second experiment simulates abstract indices in INGRES in order to predict the performance improvement which we expect from source 4 with a full implementation. Our last experiment consists of retrieval of VLSI data represented as polygons instead of boxes. This shows the improvement from sources 2 and 3 and puts our current implementation in its best light.

In [GUTT82] a performance comparison was reported between KIC and INGRES for the operations mentioned in Section IV This comparison was performed for data bases corresponding to two VLSI circuits under development at Berkeley.

KIC stored the layout in virtual memory on a VAX 11/780 computer system using its own specially designed data structures. The test machine had enough main memory so that the layout could be entirely resident in primary memory. On the other hand INGRES stored the design as disk resident relations with only small portions in a main memory buffer pool. Hence, the performance comparison was between a system using special data structures mostly in main memory and a system using general purpose data structures mostly on disk. The two systems also differed in that KIC clipped geometries to fit an appropriate window on a graphics terminal while INGRES did not simulate this operation.

Figure 2 summarizes the results of the first experiment. The first two columns show the performance difference between KIC and INGRES. KIC is assigned unit response time and unit CPU usage while the performance of INGRES is indicated relative to the KIC time Note that INGRES performs about a factor of 3 worse in CPU time and 5 worse in response time for the top level geometry and the tree expansion queries. For the spatial retrieval it is a factor of 20 worse in CPU time and 45 slower overall.

The reason for the poor performance on spatial windowing is that KIC contains a geographic bin structure for spatial indexing similar to that in Figure 1. No such indexing is present in INGRES. Consequently KIC does a restricted search while INGRES must perform an exhaustive one.

Figure 2 also shows the performance of ADT-INGRES, a version of INGRES with the addition of the abstract data types mentioned in Section IV and an overlap operator.

	KIC	INGRES	ADT-INGRES
Top level geometry			
-CPU	1	3.2	3.6
-response	1	5.5	5.8
Tree expansion			
-CPU	1	3.5	3.9
-response	1	5.1	5.6
Spatial window			
-CPU	1	20	24
-response	1	45	51

Relative INGRES Performance
Figure 2

Notice that ADT-INGRES is about 10 percent slower than regular INGRES on the top level and tree expansion tests. This represents about half of the 20 percent extra overhead needed to run the more general environment according to the results reported in [FOGG82]. Source number 1 apparently accounts for the 10 percent difference. We anticipated that the addition of an overlap operator would improve performance considerably on the spatial windowing test. However the results show no improvement, probably because the extra overhead incurred by loading the overlap routine dynamically at run time cancels out the time saved during the actual processing of tuples. Source 2, the processing of fewer tuples for ADTs, has almost no effect because the design data consists mostly of boxes, where the number of tuples is the same. Unfortunately, the test data included only one polygon in the top level geometry, and the polygon portion of the benchmark consumed almost no time. The designers of the circuit in question chose to rely primarily on boxes as a representation vehicle and not on polygons. All complex shapes in the circuit except one are made up of overlapping boxes.

The second experiment is intended to predict the improvement we can expect from abstract indices. We preprocessed the box, wire and polygon data to compute a spatial bin for each object. This bin was stored explicitly in the data base and used as a key for a normal INGRES secondary index. The spatial windowing benchmark was redone with this simulated bucketing and the results are presented in Figure 3. Simulated bucketing should closely mimic an actual implementation of abstract indices and a performance improvement of nearly a factor of 2 should be realisable

	KIC	INGRES	INGRES with bins
Spatial window			
-CPU	1	20	15
-response time	1	45	25

Performance of Simulated Spatial Index
Figure 3

The third experiment illustrates the performance improvement that can be realized with ADTs when the data is in the form of polygons instead of boxes. Polygons offer an important advantage over boxes for representation of complex geometric objects, namely that each object can be stored explicitly as a single unit instead of being made up of many apparently separate boxes which may overlap. This advantage is clearly illustrated in design rule checking. When large shapes are composed of many boxes, an overlap may or may not be an error, but with single polygons an overlap is a clear error.

For the third experiment we converted the boxes of the design data into appropriate polygons, and compared the performance of normal INGRES with ADT-INGRES when retrieving top level polygons for a single cell. Without ADTs the query is

```
range of p is polygons
retrieve into ptemp (p.all) where p.owner = CELLID
range of pt is ptemp
modify ptemp to heapsort
        on layer, polygon-id, vertnum
retrieve (pt.x, pt.y)
```

With ADTs the query becomes

```
range of p is polygons
retrieve into ptemp
     (p.polygon-desc) where p.owner = CELLID
range of pt is ptemp
modify ptemp to hash on layer
retrieve (pt.polygon-desc)
```

The results of the test are shown in Figure 4.

	INGRES	ADT-INGRES
Top level geometry		
-CPU	1	.57
-response	1	.64

Polygon Retrieval
Figure 4

These latter two tests suggest that our tactics can save nearly half of the INGRES overhead for CAD data that is polygon-rich. It is entirely possible that a more efficient version of ADT-INGRES coupled with abstract indices could be made attractive for CAD data from a performance viewpoint.

VI. CONCLUSIONS

We have identified several issues that are important in the the ongoing effort to improve the usefulness and performance of data base systems for use in CAD applications, and have shown how ADT columns in relations and abstract secondary indices can solve some of these problems. We have described an implementation of ADT columns in INGRES and have presented measurements of performance improvement. Further work is in progress in the area of access methods to support multi-dimensional spatial searching and the implementation of abstract data type secondary indices.

REFERENCES

[FOGG82] Fogg, D., "Implementation of Domain Abstraction in the Relational Database System, INGRES", Masters Report, EECS Dept, University of California, Berkeley, CA Sept. 1982.

[GUTT77] Guttag, J., "Abstract Data Types and the Development of Data Structures," CACM, June 1977.

[GUTT82] Guttman, A. and Stonebraker, M., "Using a Relational Database Management System for Computer Aided Design Data", Data Base Engineering, June 1982.

[HASK82] Haskings, R. and Lorie, R., "On Extending the Functions of a Relational Database System," Proc. 1982 ACM-SIGMOD Conference on Management of Data, Orlando, FL, June 1982.

[KATZ82] Katz, R. (editor) Special Issue on CAD Data Bases, Data Base Engineering, June 1982

[KELL81] Keller, K., "KIC, A Graphics Editor for Integrated Circuits" Masters Report, University of California, EECS Dept, June 1981.

[LISK74] Liskov, B. and Zilles, S., "Programming With Abstract Data Types," ACM-SIGPLAN Notices, April 1974.

[LOCK79] Lockmann, P. et al. "Data Abstractions for Data Base Systems," TODS, 4, 1, March 1979.

[NEWM79] Newman, W. and Sproul, R., "Principles of Interactive Computer Graphics," McGraw-Hill, N.W. 1979.

[ONG82] Ong, J., "The Design and Implementation of Abstract Data Types in the Relational Database System, INGRES," Masters Report, EECS Dept, University of California, Berkeley, CA Sept 1980.

[ROWE79] Rowe, L. and Schoens, K., "Data Abstraction, Views and Updates in RIGEL," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass May 1979.

[SCHM78] Schmidt, J., "Type Concepts for Database Definition," Proc. International Conference on Data Bases, Haifa, Israel, August 1978.

[STON75] Stonebraker, M., "Implementation of Integrity Constraints and Views by Query Modification," Proc. 1975 ACM-SIGMOD Conference on Management of Data, San Jose, Ca., June 1975.

[STON76] Stonebraker, M. et al., "The Design and Implementation of INGRES," TODS 2, 3, September 1976.

[STON80] Stonebraker, M. and Keller, K., "Embedding Hypothetical Data Bases and Expert Knowledge in a Data Manager," Proc. 1980 ACM-SIGMOD Conference on Management of Data, Santa Monica, Ca., May 1980.

[STON82] Stonebraker, M., "Adding Semantic Knowledge to a Relational Database System," Proc. NSF Workshop on Semantic Modeling, Intervale, N.H. June 1982 (to appear as Springer-Verlag book edited by M. Brodie).

[WASS79] Wasserman, A.I., "The Data Management Facilities of PLAIN," Proc. 1979 ACM-SIGMOD Conference on Management of Data, Boston, Mass., May 1979.