Information Systems Vol. 15, No. 6, pp. 653–662, 1990
Printed in Great Britain.

0306-4379 90 $3.00 + 0.00

BUILDING EFFICIENT AND FLEXIBLE FEATURE-BASED INDICES

KUEN-FANG JACK JEA† and YUNG-CHIA LEE‡
Robotics Research Laboratory, Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109-2122, U.S.A.

(*Received* 8 *August* 1989; *in revised form* 12 *February* 1990; *received for publication* 26 *July* 1990)

Abstract—If database management systems are to play an important role in CAD/CAM technologies, building engineering indices must be a primary task even though it is beyond conventional database practice. Information regarding design semantics or functionalities is often embedded in the geometric description of design objects, and is therefore not directly available for indexing. Presented in this paper is an efficient and flexible indexing mechanism for retrieving design objects that possess similar design features as described by the user. The underlying database is composed of rotational objects represented by constructive solid geometry (CSG). Although domain-specific representation schemes and algorithms are involved, the main objective of this paper is to emphasize the importance of engineering indices and to illustrate the effort required to build as well as to use such indices.

Key words: Engineering database management, engineering indices, feature extraction, constructive solid geometry, multiple key access, pattern matching, design retrieval

1. INTRODUCTION

To support efficient retrieval, a successful database usually relies heavily on a proper indexing mechanism. While selecting an adequate set of indexing attributes for a conventional business database is rather straightforward [1–4], it is unfortunately very difficult for an engineering database. Often, values chosen for indices in an engineering database are simply not directly available and must be, if possible, reasoned or computed.

1.1. Importance of engineering indices

Arguably, design retrieval by names or identifiers should not be the only way for a user to locate an existing design object from a database. A more useful retrieving facility should allow its user to retrieve the designs for all engineering objects which satisfy a given set of descriptive measures, be they quantitative, qualitative, or both. *Why are such queries important*? There are many different aspects of concerns associated with each existing design. The database of existing designs is always so important that consulting with it would eliminate a great deal of duplication effort and thus enhance the design productivity, quality, manufacturability, etc.

However, if there exists no index built on these descriptive measures, the database management system must perform an exhaustive search through the entire database to locate all desirable objects. Such an exhaustive search is even worse if those descriptive measures must be matched against each existing design, those values for these measures are yet to be computed.

1.2. Problems in constructing engineering indices

There are in fact many problems involved in building up such indices. First, *what are these descriptive measures to be indexed*? For a new design, a designer usually starts with some rough ideas in mind. These rough ideas may be an approximate global shape, some significant secondary shapes, or some qualitative or quantitative feature descriptions that are associated with part shapes or design functions. An even more challenging problem is: "*How can these measures be made directly available for each existing design in the database?*" An obvious answer is to precompute them, or to ask the designer to assign them when the database is first created. Certainly, there are problems of efficiency and flexibility.

As far as efficiency is concerned, its solutions are mostly available through database technologies. However, if queries are to be specified in various input formats such as textual descriptions or graphical sketches, the capability to efficiently compute index values so as to utilize existing indices become a new issue related to pattern recognition or, specifically, feature extraction [5, 6]. With regard to flexibility, the question is "*How flexible are the existing indices in adapting to newly defined measures?*" A new index, once defined, should be constructed automatically. Again, when techniques such as feature extraction are required to *compute* index values, multiple representations may have to be derived in advance from the principal representation. And, the flexibility

Current affiliations: †Cell Communications Research; ‡AT & T Bell Laboratories.

will rely heavily on how these representations facilitate the extraction of index values [7, 8]. Figure 1 highlights these two concerns.

This paper consists of five sections. Described in Section 2 are the database we have been concerned with and its problems in supporting engineering indices. In Section 3, we will describe the various representation schemes and algorithms that are needed to build indices based on feature extraction techniques. Section 4 presents a global view of the indexing mechanism so as to illustrate how the whole retrieving mechanism works through the proposed methods. Section 5 concludes this paper.

2. A CONSTRUCTIVE SOLID GEOMETRY (CSG) DATABASE OF ROTATIONAL PARTS

The database considered in this paper is a database of rotational parts described by CSG. The CSG scheme is one of the prevailing solid modeling techniques which completely represents objects as constructions or combinations, via the regularized set (Boolean) operators, of solid components [9]. Unfortunately, while the Boolean operations are concise and flexible in constructing and verifying objects, they also lead the CSG scheme to an unevaluated and nonunique representation scheme. The CSG scheme is unevaluated in the sense that geometric entities such as vertices, lines and surfaces are not explicitly represented and must be computed by traversing the whole CSG tree. It is also nonunique in the sense that a physical object might be represented by two or more distinct CSG trees. Consequently, rather sophisticated methods are required to rebuild CSG trees so as to extract specific features [10, 11].

2.1. The principal axis representation (PAR) scheme

Recognizing the above problems, an *internal* representation scheme PAR for rotational parts has been proposed as an internal representation scheme for CSG [12]. The idea of the PAR scheme is to *uniquely* represent a rotational object by its principal axis along with a set of boundary curves. As an internal representation scheme, it is basically a derived data structure which is more efficient in computing various geometric properties of objects. In order to justify the correctness of its derivation from CSG, the PAR scheme has also been proved to be equivalent to the CSG scheme as far as rotational objects are concerned. A simple example of PAR will be presented in the next section.

A slight modification of Fig. 1 is shown in Fig. 2 to highlight this particular environment where a derived PAR database also exists in parallel to the CSG database. It is hoped that indices will be built more efficiently and flexibly from the PAR database. Yet, there remain problems in using shape features for the indexing purpose. In the following, a simple example is given to describe the PAR scheme and to illustrate why PAR is still not an adequate scheme for specifying or extracting shape features.

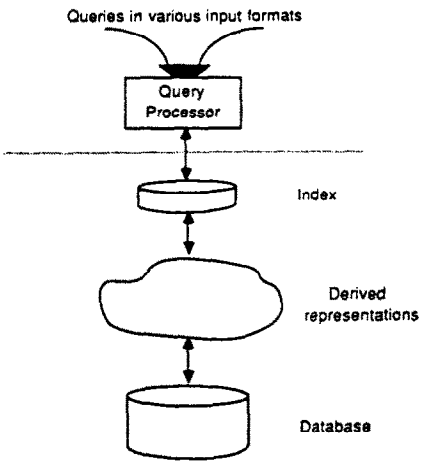


Fig. 1. Basic diagram for design retrieval.

2.2. Querying the CSG database through PAR

In Fig. 3, (a) shows a half circle curve c_{11}, which is defined as a shape feature to be indexed. Its corresponding PAR is $\{S_1, \{c_{11}, c_{12}\}\}$, where S_1 is the principal axis segment and c_{11} and c_{12} together are a pair of boundary curves associated with S_1. The PAR representation for the more complicated object in Fig. 3(b) is $\{(S_1, \{c_{11}, c_{12}\}), (S_2, \{c_{21}, c_{22}, c_{23}, c_{24}\}), (S_3, \{c_{31}, c_{32}, c_{33}, c_{34}\}), (S_4, \{c_{41}, c_{42}, c_{43}, c_{44}\}), (S_5, \{c_{51}, c_{52}, c_{53}, c_{54}\}), (S_6, \{c_{61}, c_{62}\}), (S_7, \{c_{71}, c_{72}\})\}$. While the half circle in (a) is completely represented in one segment, the corresponding one in (b) has been decomposed into four sections, each being associated with one segment. The reason for different PAR representations of the same feature (the half circle, in this case) is that the set of principal axis segments are determined by all the intersecting points between boundary curves and lines.

To match the half circle in (b) against the one in (a) for the purpose of feature extraction, we need to scan through several segments and connect the four pieces of curves for comparison. Clearly, this complexity in

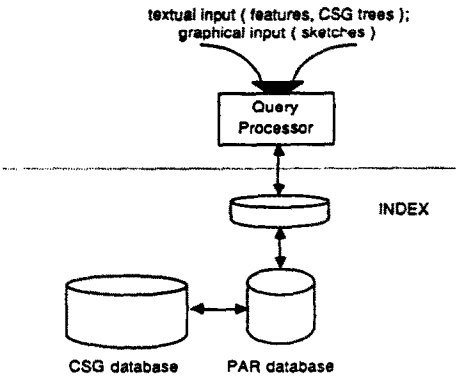


Fig. 2. Basic diagram for CSG design retrieval.

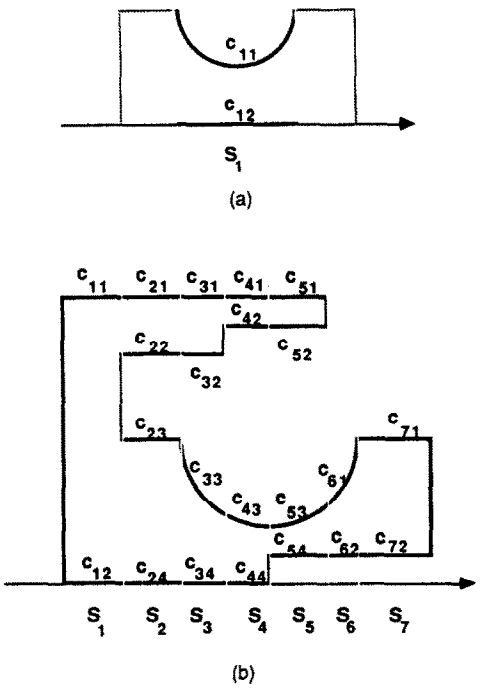


Fig. 3. The problem of matching two PAR's.

matching features at the PAR level arises directly from the fact that a feature may be decomposed and represented in several segments. Yet, such a decomposition is caused by the shape of other portions which are irrelevant to the feature itself. To overcome this problem, we need a representation scheme which can isolate a local feature and prevent it from being decomposed due to the shape of neighboring portions. Furthermore, it seems quite obvious that, in each pass, only one feature can be extracted if a direct match is performed on PAR. It becomes very inefficient and less desirable when a set of features are to be extracted for the indexing purpose because multiple passes of scanning the object's PAR would be required.

3. INDEXING BASED ON FEATURE EXTRACTION

Thus, as indicated that neither CSG nor PAR are representation schemes at the right level for supporting engineering indices, we need to identify yet another representation scheme which satisfies the following:

(1) allows shape features to be properly defined; it implies that, perhaps, the new representations should be as close as possible to profiles;
(2) allows shape features to be efficiently extracted; it would be best if an efficient algorithm already exists and is applicable to the new scheme;
(3) as a derived one, the new scheme can be efficiently transformed from either CSG or PAR.

Described first in this section is the scheme of pattern string representation (PS), which we believe qualifies the criteria above. The PS scheme relies heavily on Aho and Corasick's string matching algorithm [12], which not only extracts desirable features efficiently but also constructs the pattern matching machine (i.e. feature extractor) automatically. We will describe this algorithm only briefly and focus on how it can be used to extract shape features from a pattern string. Lastly, we will present an algorithm which converts a PAR into a pattern string. By then, all the proposed representation schemes and algorithms are complete and ready for the indexing purpose.

3.1. Pattern string representation

In the following, the pattern string representation for rotational parts is defined. The 16 pattern primitives which compose pattern strings are shown in Fig. 4. Each primitive is denoted by a character, and stands for either a line or an arc segment with a starting point, an ending point and a direction label. The first eight primitives (**a–h**) are line segments. Except for the horizontal (**a** and **e**) and the vertical (**c** and **g**) primitives, each of the rest line segments (**b, d, f** and **h**) may represent any vector (rooted at the origin) in that quadrant. This leaves room for an inexact match which will be explained later. However if two line segments of the same primitive type are head-to-tail connected, they are treated as two different primitives unless they are of the same slope. In that case, the two line segments are merged together and treated as one primitive. Accordingly, the length of a primitive is immaterial, and this leaves additional room for an inexact match. The other eight primitives are circular arc segments; rotational parts constructed from cylinders, cones, and tori have only circular arcs in their profiles. Similarly, each primitive may represent any circular arc within a specific quadrant. Unlike those primitives defined in Ref. [14] which may be arcs of half circle, all arc

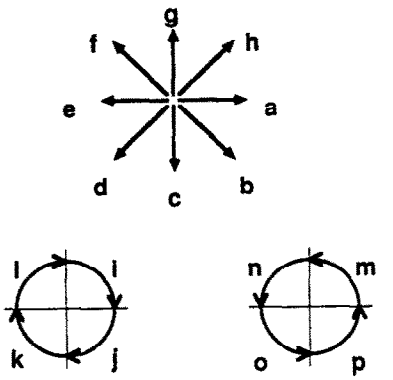


Fig. 4. 16 Pattern primitives.

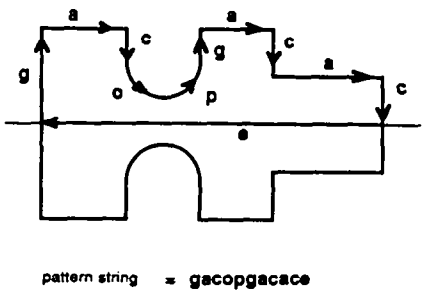


Fig. 5. Pattern string representation.

primitives defined here do not cover more than one quadrant because a feature or an object may be otherwise coded by different pattern strings.

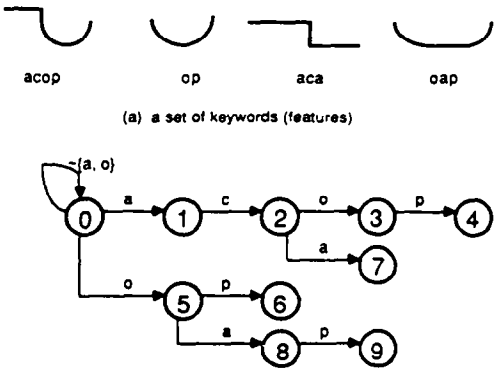
A pattern string representation for a rotational part is defined as a string of pattern primitives which describes the upper half of the object's 2-D profile in a *clockwise* order starting from the leftmost segment of the profile. The word *clockwise* means left-to-right for external boundary and right-to-left for internal boundary of the profile. If there is more than one leftmost segment, the one closest to the principal axis is selected as the starting segment. Figure 5 illustrates how to derive a pattern string from an object's profile. Starting from the leftmost segment **g** and proceeding around the upper half of the profile in a clockwise order, a pattern string "**gacopgacace**" can be identified for the object.

Unlike PAR, the pattern string representation is not intended to completely represent the object's geometry. Instead, simplifications are made for the purposes of efficiency, simplicity and inexact match. After all, design retrievals are to locate *similar* objects. The detailed information associated with each primitive is, however, recoverable if needed. Later, in the algorithm which transforms PAR into PS, such information is used to determine if two primitives should be merged into one and how two intersecting edges should be split.

3.2. Extracting features from PS

As indicated earlier, extracting features from PS becomes an extremely efficient and flexible task due primarily to the availability of Aho and Corasick's string matching algorithm [13]. The Aho and Corasick's string matching algorithm consists of two parts. The first part constructs a finite state pattern matching machine from a set of keywords (in this case, a set of features to be extracted). The second part of the algorithm runs the pattern matching machine against the input string (in this case, the pattern string of an object) to locate keywords. The machine signals whenever it has found a match of a keyword. All keywords embedded in the input string are located in a single pass of reading the input string.

3.2.1. Constructing a pattern matching machine. The Aho and Corasick's algorithm has been well documented in Ref. [13]. In the following, we will briefly review it through an example constructed by our own



i	1	2	3	4	5	6	7	8	9
f(i)	0	0	5	6	0	0	1	1	0

(c) failure function

i	λ(i)
4	{acop, op}
6	{op}
7	{aca}
9	{oap}

(d) output function

Fig. 6. Pattern matching machine: an example.

implementation. In Fig. 6, (a) shows the four features "**acop**", "**op**", "**aca**", "**oap**", and (b) shows the goto graph (goto function) g that is constructed from the four given keywords. Each circle in this graph represents a state and each edge denotes a state transition as the associated input symbol occurs. Note carefully that each path in this graph spells out a keyword. Shown in (c) and (d), respectively, are the failure f and the output λ functions for each state i, both computed from the goto function in (b). The output function for some states (i.e. states 0, 1, 2, 3, 5, 8) are missing from (d) because no outputs should be generated at those states. From the

Table 1. State transition function of the machine in Fig. 6

δ	Present State									
Input	0	1	2	3	4	5	6	7	8	9
a	1	1	7	8	1	8	1	1	1	1
b	0	0	0	0	0	0	0	0	0	0
c	0	2	0	0	0	0	0	2	2	0
d	0	0	0	0	0	0	0	0	0	0
e	0	0	0	0	0	0	0	0	0	0
f	0	0	0	0	0	0	0	0	0	0
g	0	0	0	0	0	0	0	0	0	0
h	0	0	0	0	0	0	0	0	0	0
i	0	0	0	0	0	0	0	0	0	0
j	0	0	0	0	0	0	0	0	0	0
k	0	0	0	0	0	0	0	0	0	0
l	0	0	0	0	0	0	0	0	0	0
m	0	0	0	0	0	0	0	0	0	0
n	0	0	0	0	0	0	0	0	0	0
o	5	5	3	5	5	5	5	5	5	5
p	0	0	0	4	0	6	0	0	9	0

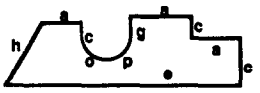
```
Algorithm.  Pattern Matching (Feature Extractor).
Input. An input string x = a1a2...an where each ai is an
       input symbol and a pattern matching machine M with
       state transition function δ and output function λ.
Output. Locations at which keywords occur in x.
Method.
  begin
      state ← 0;
      for i ← 1 until n do
      begin
          state ← δ(state, ai);
          if λ(state) ≠ φ then
             begin
                 print i;
                 print λ(state);
             end
      end
  end
```

Fig. 7. The feature extractor.

goto and the failure functions, the state transition function δ can be computed as shown in Table 1. The pattern matching machine is thus constructed. Of the most importance is the fact shown in Ref. [13] that the cost of computing g, f, λ and δ together requires an amount of time linear to the sum of keyword lengths.

3.2.2. Extracting features by the pattern matching machine. As features and objects are now represented as keywords and strings, respectively, extracting features from an object is equivalent to locating all keywords in an input string. Figure 7 shows in principle how a pattern matching machine, or a feature extractor, works. The machine consists of a finite set of states. Starting at the start state, the machine processes the input string by successively reading input symbols in the input string, making state transitions and producing output. At each new state, if the output function is not empty, the output and the position of the current input symbol are reported. The machine terminates when there is no symbol left. Overall, the machine makes n state transitions in processing an input string length n, which is independent of the number of keywords as well as the size of each keyword. All keywords (i.e. features) are identified and located in a single pass of processing the input string.

In Fig. 8, (a) shows a rotational object by its upper half profile, whose pattern string is "**hacopgacace**". (b) Shows the sequence of state transitions as each input symbol is consumed. (c) Shows the features extracted and their positions after the input string is processed.



(a) an input object

input string: h a c o p g a c a c e

state transitions: 0 0 1 2 3 4 0 1 2 7 2 0

(b) a sequence of state transitions

position	state	features
5	4	{acop, op}
9	7	{aca}

(c) features extracted from input string

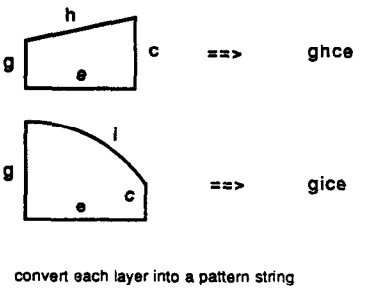
Fig. 8. Feature extraction using the machine in Fig. 6.

3.3. Deriving pattern strings from PAR

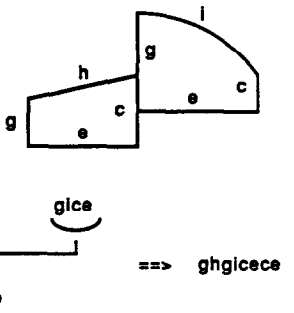
Remember that one of the criteria for the new representation scheme is to be derivable from either CSG or PAR. Described in this section is a one-pass algorithm which efficiently converts a PAR into a pattern string. The conversion algorithm consists of two main steps: first, convert each layer in the PAR into a pattern substring, and second, merge these substrings into a complete string.

The first step is to scan through each axis segment of the PAR and convert every layer associated with that segment into a pattern substring. Each layer in the PAR is defined by a pair of curves and bounded by two vertical line segments (may degenerate to a point in some cases) at the bound points of that segment. Thus, the pattern substring for a layer, which consists of no more than four pattern primitives, can be easily derived according to the types of the curves. For example, the two layers in Fig. 9(a) can be converted into two substrings "**ghce**" and "**gice**".

The second step is to, for those neighboring layers (or compositions of layers) that have intersecting edges, insert the right pattern substring into the left one at the position where their edges meet. In



(a) convert each layer into a pattern string



(b) insert the right pattern string into the left one at the position of intersection

Fig. 9. Two major steps in the pattern string conversion.

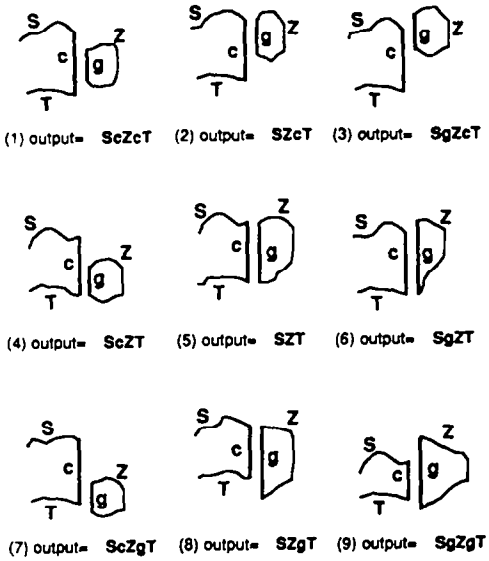


Fig. 10. Nine cases of inserting a pattern substring.

Fig. 9(b), for example, the edge c of the left substring meets the edge g of the right substring. The pattern string for the composition of these two layers, "**ghgicece**", can be generated by inserting the right substring "**gice**" into the left one "**ghce**" at the position c. Note that the two intersecting edges must be split appropriately. In general, there are nine cases in total that must be resolved. Figure 10 shows these nine cases as well as their solutions. Note that, in Fig. 10, a left substring is represented by "**ScT**" and a right substring is represented by "**gZ**", where c (pointed downward) and g (pointed upward) are the two intersecting edges. The capital letters S, T, and Z are pattern substrings.

A one-pass conversion algorithm based on the above two steps is presented in Fig. 11. Assume that a given PAR has n segments and, within each segment S_i ($1 \leqslant i \leqslant n$), there are m_i layers. The algorithm proceeds as follows: first, each layer in Segment 1 is converted into a pattern string and its rightmost edge is stored into a queue, called *NextEdgeQueue* (because it may intersect the leftmost edges of the layers in the next segment); then the algorithm goes iteratively to the next segment, converts each layer, stores the rightmost edge, and inserts the pattern string just converted into the pattern string according to the cases shown in Fig. 10. As each layer and each segment is processed only once, the conversion algorithm is a one-pass algorithm.

We have implemented all the representation schemes and algorithms described in this section. The next section will illustrate them in terms of several CSG objects.

4. DESIGN RETRIEVAL THROUGH MULTIPLE ENGINEERING INDICES

Presented in this section is the proposed mechanism for retrieving CSG objects using multiple

```
Algorithm PAR2PS.  Convert PAR to Pattern String.
Input. a PAR, which has n segments and each segment S_i
       has m_i layers.
Output. a pattern string PS.
Method.
  begin
    NextEdgeQueue ⟵ nil;
    for j ⟵ 1 until m_1 do
       convert the layer L_1j to a pattern string and
       store its rightmost edge r_1j into NextEdgeQueue;
    for i ⟵ 2 until n do
    begin
      CurrentEdgeQueue ⟵ NextEdgeQueue;
      NextEdgeQueue ⟵ nil;
      x ⟵ get an edge from CurrentEdgeQueue;
      for j ⟵ 1 until m_i do
      begin
        convert the layer L_ij to a pattern string and
        store its rightmost edge r_ij into NextEdgeQueue;
        y ⟵ the leftmost edge of L_ij;
        while (x.head.height > y.head.height) and
              not empty(CurrentEdgeQueue) do
           x ⟵ get an edge from CurrentEdgeQueue;
        if (x ∩ y) ≠ φ then
           insert the pattern string of L_ij into the string
                containing x;
      end
    end
    PS ⟵ the remaining pattern string;
  end
```

Fig. 11. A one-pass algorithm which converts PAR to PS.

indices. Although some aspects of this mechanism are rather domain specific, it is in general of interest to most engineering database management systems. We will first present a number of test examples to illustrate all the algorithms and representation schemes described earlier. We will then explain the indexing mechanism through a detailed block diagram. Finally, the related issue of multiple-key access will be discussed.

4.1. *Illustrative examples*

In the following, a number of CSG objects are used to test the algorithms developed thus far. The profiles of these examples are shown in Fig. 12. A summary of various results is listed in Table 2. All examples follow the same assumptions and procedures listed below:

1. All CSG objects are constructed by set operators *union* and *difference* from primitives including cylinders, cones and tori. These CSG objects are axis-symmetric and of single principal axis.
2. The algorithm *CSG-to-PAR* is applied to each CSG object to generate a PAR. For an illustrative purpose, some geometric properties that can be easily derived from PAR such as length, maximum diameter and profile are computed.
3. The algorithm *PAR-to-PS* is applied to transform each PAR into a pattern string.

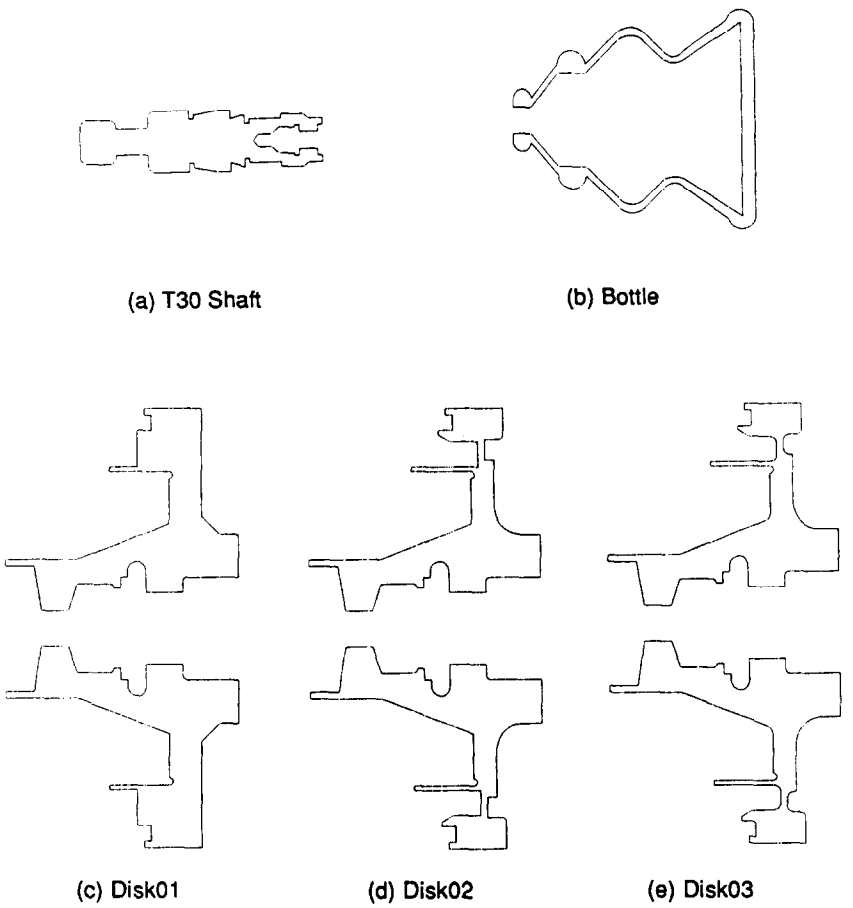


Fig. 12. Illustrative examples: (a) T30 shaft [14]; (b) bottle; (c) Disk01; (d) Disk02 and (e) Disk03 [13].

4. Independent of objects, a set of 13 features is defined, as shown in Fig. 13. These features are fed into Aho and Corasick's algorithm to construct a pattern matching machine.
5. The pattern matching machine is used to process the pattern string associated with each object. The extracted features are reported with their positions on the pattern string.

4.2. *The indexing mechanism*

There are two major tasks the indexing mechanism must achieve: building indices off-line and computing index values on-line for accessing the database. With all algorithms ready, let us examine how they can be used to accomplish these two tasks. A block diagram is shown in Fig. 14.

To build indices off-line, each CSG object in the database is processed first by the algorithm *CSG-to-PAR*. From PAR, geometric properties which are to be indexed are computed. Meanwhile, the algorithm *PAR-to-PS* is applied to convert each PAR into a pattern string, from which shape features previously defined are extracted. Indices are then built on these shape features.

To use indices on-line, all index values must be on-line evaluated regardless of the input format. As discussed earlier, a user, namely, a designer, could have asked any of the following:

(1) a simple textual query such as "retrieve all design objects that have length-to-diameter ratio between 0.1 and 1 and possess features *pgm* and *nco*";
(2) a more complicated textual query such as "retrieve all design objects that possess similar features as the object specified in the CSG data file *template*";
(3) through some interface such as a drawing pad, the query is to locate all design objects that possess similar feature as the object just sketched by the user.

As indicated in Fig. 14, textual input of shape features and geometrical properties can be used directly as index values for access; input CSG trees must go through exactly the same route that builds indices off-line; and, graphical sketches must be converted into pattern strings before feature extraction can be performed.

Again, let us examine why Fig. 14 indeed suggests an efficient and flexible indexing mechanism. The

Table 2. PS and extracted features of test objects

Design Object	T30 Shaft	Bottle	Disk01	Disk02	Disk03
CSG tree (#nodes)	53	53	55	79	115
CSG primitives	27	27	28	40	58
Dimension (length)	530	530	510	510	510
Dimension (diameter)	140	490.5	980	980	980
Pattern String	ghabcagha-caghachca-gagabacec-egecededede	glihlihli-bophliceg-djkfmndede	gahgpmega-gagegacba-cencegmnc-ecemedefe	gahgpmega-gefgagega-cdecacoac-encegmnce-cemedefe	gahpgpmeg-apgmefgap-gegacdenc-oacoacenc-egmnceccem-edefe
Features (Positions)	cagh (8) cagh (13) aga (20) aga (22) gec (31)	ih (4) ih (7) op (12) gd (19)	pm (6) aga (11) age (13) enc (22) emed (33)	pm (6) age (11) age (16) enc (30) emed (41)	pm (7) pgm (13) enc (27) nco (28) enc (36) emed (47)

efficiency is in part attributed to Aho and Corasick's algorithm. Based on this algorithm, the feature extractor extracts all the features in an object in a single pass. The efficiency is also dependent on how indices are maintained. Although not the focus of this paper, we will briefly discuss this issue as a problem of multiple-key access in the next subsection. As for the flexibility, note that defining a new feature will be accommodated by simply reconstructing the feature extractor and then rebuilding the indices; both may require significant computation effort but can be automatically done without human intervention.

4.3. *Multiple-key access*

Most queries refer to an arbitrary number of geometric properties and/or features. Therefore, this section deals with the problem of multiple-key access, as far as the indexing strategy in concerned.

While geometrical properties are usually associated with a range of values, shape features are not necessarily described quantitatively or qualitatively. At least within our current focus, queries are mainly to locate design objects that possess certain features. In other words, an index on a particular feature is to determine if the feature appears in each design object. The index value is either *yes* or *no*.

A straighforward way to maintain all these indices is presented in Fig. 15. For each shape feature index, there are two entries; one points to a bucket containing the identifiers of those designs which possess the specific feature while the other points to a bucket for those objects not possessing the specific feature. Some may argue that the *no* bucket should not exist because of space efficiency. However, it deserves to be there if queries that refer to nonexistency of features also happen quite frequently. For each geometric property index, there are multiple entries, each carrying a value or a

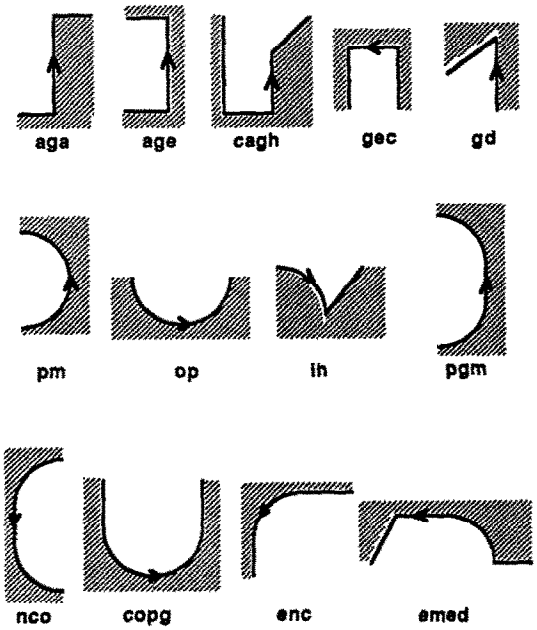


Fig. 13. A set of 13 features.

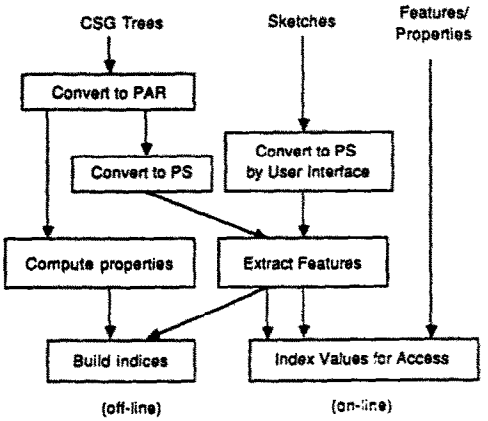


Fig. 14. Detailed block diagram of the indexing mechanism.

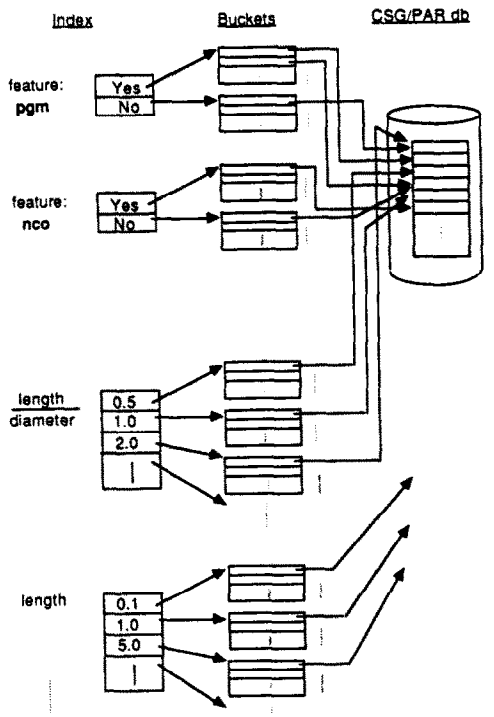


Fig. 15. Multiple indices structures.

range of values and pointing at a bucket which stores the identifiers of those qualifying objects. Given a query, the access strategy is then to collect all qualified buckets and perform adequate set operations (e.g. intersection, union, not) on their identifier sets. For simplicity, the intersection might be the only operation needed.

The operations on identifier sets sometimes cause great concern on efficiency, particularly when some of the identifier sets are overwhelmingly larger than others. The *grid structure* [3] provides one of the solutions to this problem by restructuring the indices. A grid structure is in general an n-dimensional array with each dimension corresponding to an index. The subscript range of each dimension is the range of the index values in each index. Each entry of the array (i.e. the grid structure) contains identifiers of those objects whose index values are consistent with the subscript values addressed to it. For example, if there are two indices A and B with the possible index values a_1, a_2, a_3 for A and b_1, b_2, b_3, b_4 for B, the corresponding grid structure is a 2-D array with $3 \times 4 = 12$ elements. For the query "**find** records **with** $A = a_2$ **and** $B = b_3$", the qualified records can be accessed by the identifiers stored in the grid element *grid* (a_2, b_3).

Assume that there are n feature indices defined in the database. Because each feature index has only two index values, the grid structure for all n feature indices requires only an n-bit addressing vector, each bit corresponding to an index, to access the 2^n buckets that are needed. Given a query, the access strategy is to determine the n-bit address according to the features specified or extracted. The advantage is that the retrieval is speeded up because there are no intersections performed at runtime. The disadvantage, on the other hand, is the space overhead attributed to the 2^n buckets that are needed. But, if necessary, the techniques used in dynamic hash functions [3] that split and coalesce buckets as the database grows and shrinks can be adopted to alleviate the problem of space overhead.

5. CONCLUSIONS

While selecting an adequate set of indexing attributes for a conventional business database is rather straightforward, it is unfortunately very different and, in fact, difficult for an engineering database. Often, values chosen for indices in an engineering database are simply not directly available and must be, if possible, reasoned or computed. The objective of this paper is to illustrate the effort required to build as well as to use engineering indices in a CAD/CAM database management system.

Using a CSG database as an example, this paper identifies the need for deriving intermediate representation scheme and for extracting shape features. Both efficiency and flexibility have been taken into account when building and using indices. Regarding efficiency, the pattern matching machine extracts all features from an input pattern string in a single pass (independent of the total number of predefined features), and the algorithm deriving a pattern string from PAR has a linear time complexity with respect to the number of axis segments in PAR. As for the flexibility, Aho and Corasick's algorithm automatically constructs the pattern matching machine for any new features, and the pattern matching machine extracts multiple features and builds multiple indices automatically. Accordingly, the indexing mechanism proposed in this paper is both efficient and flexible.

There are a number of limitations specifically related to the representation schemes and algorithms described in this paper for the CSG database. The CSG database is composed of 3-D, rotational design objects. The database stores nothing but part geometries in CSG representations. The objects shown in this paper are constructed by cones, cylinders and tori only. As such, their derived, 2-D representations consist of straight line segments and circular arcs. Obviously, other CSG primitives such as blocks or more complicated free-form solids can not be handled directly. To overcome such limitations, many algorithms are needed and some of them have recently been described elsewhere [11]. Note, however, that the proposed feature extraction techniques are much more suitable for rotational parts than those based on boundary representations (B-reps). First, deriving from CSG to B-rep is very time-consuming and, second, extracting arbitrary

features from B-rep, if possible, is not trivial at all [8, 15]. As for the features illustrated in this paper, we have made no attempt to select them from a list of most commonly used features. Nevertheless, if any feature is to be used in a database, the same indexing mechanism can be adopted. As long as the desirable features are specified in CSG trees, sketches or even pattern strings, their corresponding indices will be efficiently and automatically constructed against the whole database. It is also not mentioned in this paper whether the database should be a relational database or an object-oriented database. We would rather consider this issue less significant because the main difficulty lies in the unavailability of design semantics rather than in the way each design object is physically stored and managed.

The proposed mechanism employs Aho and Corasick's string matching algorithm to construct feature extractors. Although this algorithm is very efficient, there is still room for improving its performance. Algorithms presented in Refs [16, 17] are such examples. However, the performance gains in Ref. [16] (with the same order of time complexity as Ref. [13] are compromised by the more complex and larger (feature extractor) construction algorithm. And the algorithm in Ref. [17] does not allow the modification of keywords (i.e. features) and, thereby, results in the loss of flexibility in accommodating new features.

The retrieving mechanism described, albeit domain-specific, suggest in general how feature extraction and multiple-key access can be brought together for more efficient and flexible design retrieval. Though beyond the practice of conventional database design, building engineering indices is an important task in engineering database design.

Acknowledgements—Many thanks to the referees for their useful suggestions.

REFERENCES

[1] C. J. Date. *An Introduction to Database Systems*, 3rd edn. Addison-Wesley, Reading, Mass. (1981).
[2] C. J. Date. *An Introduction to Database Systems*, Vol. II. Addison-Wesley, Reading, Mass. (1982).
[3] H. F. Korth and A. Silberschatz. *Database System Concepts*. McGraw-Hill, New York (1986).
[4] J. D. Ullman. *Principles of Database Systems*, 2nd edn. Computer Science Press, Rockville, Md (1982).
[5] J. Earley. An efficient context-free parsing algorithm. *Communs ACM* **13**(2), 94–102 (1970).
[6] K. S. Fu. *Syntactic Pattern Recognition and Applications*. Prentice-Hall, Englewood Cliffs, N.J. (1982).
[7] R. Jakubowski. Syntactic characterization of machine parts shapes. *Cybernetics Systems* **13**(1), 1–24 (1984).
[8] S. M. Staley, M. R. Henderson and D. C. Anderson. Using syntactic pattern recognition to extract feature information from a solid geometric data base. *Computers Mech. Engng.* **2**(2), 61–66 (1983).
[9] A. A. G. Requicha and H. B. Voelcker. Solid modeling: a historical summary and contemporary assessment. *Computer Graph. Applic.* **2**(2), 9–24 (1982).
[10] Y. C. Lee and K. S. Fu. Machine understanding of CSG: extraction and unification of manufacturing features. *Computer Graph. Applic.* **7**(1), 20–32 (1987).
[11] K. F. Jea. An efficient and flexible design-retrieval mechanism for CAD/CAM databases. Ph.D. Thesis, The University of Michigan, Ann Arbor, Mich., Nov. (1989).
[12] Y. C. Lee and K. F. Jea. PAR: a representation scheme for rotational parts. *IEEE Trans. Systems Man, Cybernetics* **SMC-17**(11), 1039–1049 (1987).
[13] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communs ACM* **18**(6), 333–340 (1975).
[14] R. K. Li. A part-feature recognition system for rotational parts. *Int. J. Prod. Res.* **26**(9), 1451–1475 (1988).
[15] H. P. Wang and R. A. Wysk. AIMSI: a prelude to a new generation of integrated CAD CAM systems. *Int. J. Prod. Res.* **26**(1), 119–131 (1988).
[16] J. Aoe, Y. Yamamoto and R. Shimada. A method for improving string pattern matching machines. *IEEE Trans. Sftwre Engng* **10**(1), 116–120 (1984).
[17] J. Aoe. An efficient implementation of static string pattern matching machines. *IEEE Trans. Sftwre Engng* **15**(8), 1010–1016 (1989).