RSD-TR-21-87

A NEW CSG TREE RECONSTRUCTION ALGORITHM FOR FEATURE UNIFICATION

Yung-Chia Lee

Kuen-Fang Jack Jea

October 1987

CENTER FOR RESEARCH ON INTEGRATED MANUFACTURING

Robot Systems Division

COLLEGE OF ENGINEERING

THE UNIVERSITY OF MICHIGAN

ANN ARBOR, MICHIGAN 48109-1109

ABSTRACT

By feature unification we mean the transformation of a specific feature into a desirable and isolated representation. Due to the global nature and non-uniqueness properties of the Constructive Solid Geometry (CSG) scheme, a feature on an object may be represented by different sets of primitives, which may yet scatter around the CSG tree. A tree reconstruction algorithm which groups these primitives as close as possible is therefore needed before the feature can be unified into a desirable format suitable for applications (such as process planning). Lee and Fu earlier proposed such a tree reconstruction algorithm which works well in most cases, but fails to move primitives in some difficult situations. In this paper, we investigate thoroughly the formal properties of moving nodes in CSG trees, and propose a new tree reconstruction algorithm based on a well-defined single-step move-up operation. This new table-driven algorithm is more powerful because the cases failed by the previous algorithm are either overcome or elaborated. It is also simple to understand, easy to implement, and very efficient.

Contents

1 Introduction

In the areas of computer-aided design and manufacturing (CAD/CAM), the Constructive Solid Geometry (CSG) scheme is one of the most widely used solid modeling schemes for representing 3-D objects. It represents an object as a Boolean combination or construction of solid primitives (e.g., cylinders, cubes, cones, spheres, and tori) via the regularized set operators *unions, differences*, and *intersections*. The resulting representation of an object is a tree (called CSG tree) with the solid primitives as terminal nodes and the regularized set operators as non-terminal nodes. The CSG scheme has many advantages [2,12,17]: it is *informationally complete* (i.e., unambiguous) and *concise*; it has a wide range of *geometric coverage*; objects are easy to construct and their *validity* is automatically ensured by the syntax; application algorithms based on the tree structure of CSG are easy to program, relatively numerically stable, inherently parallel, and efficient for some operations. Therefore, many contemporary solid modeling systems use the CSG scheme; for example, PADL-1 [15], PADL-2 [4], GMSOLID [3], TIPS [11], GDP [16], SYNTHAVISION [6], UNISOLIDS, Applicon SOLIDS MODELING II, Catronix CATSOFT, Control Data ICEM SOLID MODELER, Sperry SOLID MODELER, etc.

While the CSG scheme has many advantages, it has two problems with feature applications such as feature recognition and extraction [10,18]. First, its *global nature* may scatter the solid primitives of a feature throughout a CSG tree. Extracting a feature must go through the whole CSG tree to find these primitives and check their relationships. Second, its *non-uniqueness nature* may result in multiple CSG representations for an object, that is, a physical object can be represented by many different CSG trees of different structures and primitives. The task of defining feature models for matching is thus very difficult, if not impossible.

The problems resulting from both the global and non-uniqueness natures of the CSG scheme can, however, be alleviated if there exist some reliable *tree reconstruction* algorithms which can move the solid primitives around in a CSG tree and/or replace a CSG subtree by another equivalent one. The CSG trees after reconstruction must of course represent the same objects and, hopefully, are more suitable for the underlying applications.

The tree reconstruction technique has been employed by researchers in different applications.

Kakazu and Okino [9], in their Group Technology Code generation, first convert an arbitrary CSG tree into a canonical form (i.e., the union of all *real existence* primitives difference the union of all *removal* primitives) and are then able to apply a pattern recognition procedure to obtain the GT code. Goldfeather *et al.* [7] also use a tree reconstruction algorithm to transform a CSG tree into a *normal form* (i.e., a union of simpler subtrees) which can then be traversed to generate quadratic coefficients and operation codes for fast displaying the object by their Pixel-Powers Graphics System. In a similar work, Sato *et al.* [14] also reconstruct CSG trees for fast image generation.

Feature unification [10] is another application domain where the tree reconstruction technique seems to be useful. By *feature unification*, a specific feature is transformed into an isolated, desirable representation. Due to the global and non-uniqueness nature of CSG, a feature on an object may be represented by different sets of primitives, which may yet scatter around the CSG tree in a number of different ways. A tree reconstruction algorithm which groups these primitives as close as possible is therefore needed before the feature can be unified or mapped into a desirable format suitable for applications (such as process planning).

Lee and Fu [10] earlier proposed a tree reconstruction algorithm based on the *move-up* and the *shuffle* operations. The move-up operation either moves a node up one level in the CSG tree or changes the node's environment so that it can be moved up later. The shuffle operation performs the exchange of a node with its uncle twice so as to pair together two "cousins-german" in the CSG tree. Figure 1 illustrates how to pair together two nodes A and B in a CSG tree rooted at R. Nodes A and B are respectively raised level by level through the move-up operation until becoming the "cousins-german" under the same grandparent R'. They are then brought together under the same parent Y by the shuffle operation.

This algorithm sometimes must stop because the move-up operation fails to move primitives. To solve this problem and make it more powerful, we propose in this paper a new tree reconstruction algorithm which follows the same ideas of *moving-up* and *shuffling* nodes but is based on a well-defined single-step move-up operation. All the cases failed by the previous algorithm are either overcome or elaborated. The new algorithm is simple to understand, easy to implement, and very efficient.

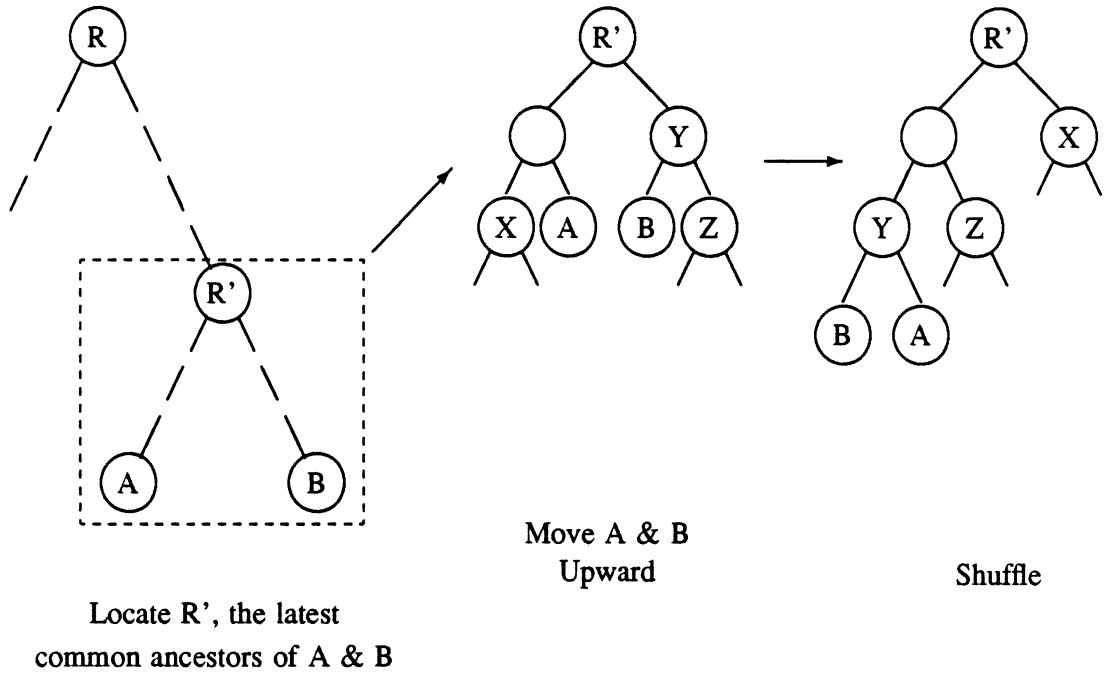


Figure 1: Pairing of Two Nodes.

This paper consists of six sections. The next section defines the problem to be studied and describes the proposed set-theoretical approach to solving this problem. In Section 3, basic terms necessary for this study are defined and various ways to move nodes are discussed. Four useful properties are observed from this section. In Section 4, two state tables are derived and then optimized by the state reduction technique. A new table-driven algorithm is also developed to move nodes for feature unification. In Section 5, a comparison of this new algorithm with the previous one is made with respect to their primitive operations, control strategy, number of inputs used, and their power of moving up nodes. Finally the last section concludes this paper and describes two future works.

2 Problem and Proposed Approach

As mentioned before, a tree reconstruction algorithm which groups primitives for feature unification may be based on the move-up and the shuffle operations. Since the existing algorithm [10] does not guarantee the success of moving up nodes in all cases, it appears essential to closely

reexamine all possibilities that a node in different environments can be moved up. The problem can be reduced to the problem of one-level move-up, as it is the basic step for moving a node up several levels to a desirable position. If an operation with well-formed behavior can be defined, it can then be plugged into any tree reconstruction algorithm which needs to move up nodes. In sum, the main problem to be studied in this paper can be stated as follows:

Given a CSG tree in which solid primitives are constructed by the regularized set operators **unions** *and* **differences** *only, how can a node in different situations be moved up* **one level** *yet without duplicating the node itself?*

In this problem, the *intersection* operator is not included because it can be implemented by the *difference* operation (e.g., $A \cap B = A - (A - B)$) and it is less important than the *union* and *difference* operations which are sometimes analogous to the welding of assembly operation and the machining operation [8], respectively. In addition, a criterion we must insist in this study is that no duplication of the nodes being moved should be made. The purpose of feature unification is to group together all the primitives of a feature and convert them into another form, while the duplication of these nodes will ultimately leave their copies ungrouped and unconverted. Note however that the sibling of the node being moved is allowed to be duplicated if necessary.

Our approach to solving this problem is purely set-theoretical. A CSG tree can be one-to-one mapped to a mathematical *set expression* where variables are the solid primitives, operators are the regularized set operators[1] unions and differences, and parentheses enforce the CSG tree structure. Since this mapping can be isomorphic, a tree reconstruction can be achieved by applying *set-theoretical identities* to a set expression and transforming it to an equivalent one. The study of moving a node up one level in the CSG tree is therefore equivalent to the search of appropriate set-theoretical identities which can be applied to the corresponding set expression to yield a new expression in which the node is effectively one level higher than it was before.

[1]Hereafter we will omit the word *regularized* in the context.

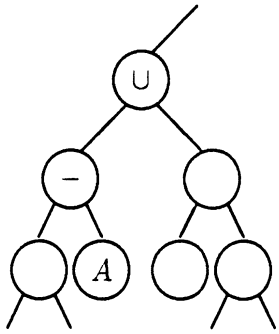


Figure 2: Up-two-level Environment of A.

3 One-level Move Up

3.1 Basic Terms

We will first define some necessary terms.

A *working node* is defined as the node or subtree to be moved in a CSG tree. The *level* of a node or a subtree is defined as the number of ancestors along its parent chain to the root of the CSG tree. The level of the root is defined as 0. A *move* is an application of one set-theoretical identity to the CSG tree which results in the changes of the environment (defined below) of a working node. *One-level move-up* is defined as the result of one or several moves such that the level of a working node is one less than it was before the move or moves (i.e., the working node is at the same level as its original parent).

An *environment* of a working node is characterized by, along its parent chain, the positions of itself to its ancestors and the operator types of these ancestors. A *up-two-level* environment of a working node is thus defined as a state which specifies the operator types of the working node's parent and grandparent, and the relationship of this node to its parent (either leftchild or rightchild) and that to its grandparent (either in leftsubtree or in rightsubtree). This term is so named because only two levels of ancestors (i.e., parent and grandparent) are referenced. In the example of Figure 2, the up-two-level environment of the working node A is characterized by the $\cup$, $-$ operators, and A as in the leftsubtree of its grandparent and a rightchild of its parent.

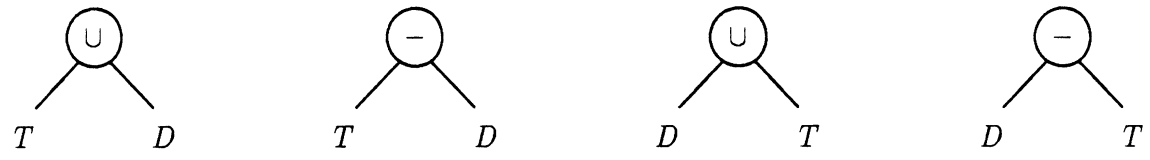
From the Up-two-level environment point of view, a working node can only be in one of the 16 possible environments because the operators of its parent and grandparent can be either unions

State	Set-theoretical Representation
1	$(A \cup B) \cup C$
2	$(B \cup A) \cup C$
3	$C \cup (A \cup B)$
4	$C \cup (B \cup A)$
5	$C - (A \cup B)$
6	$C - (B \cup A)$
7	$(A - B) - C$
8	$(B - A) - C$
9	$(A \cup B) - C$
10	$(B \cup A) - C$
11	$(A - B) \cup C$
12	$(B - A) \cup C$
13	$C \cup (A - B)$
14	$C \cup (B - A)$
15	$C - (A - B)$
16	$C - (B - A)$

Table 1: Definition of 16 States.

or differences and it can be either in rightsubtree or in leftsubtree to its parent and grandparent ($2 * 2 * 2 * 2 = 16$). Since these 16 possible environments are important in forming the basis of the moving-up analysis, they are defined as 16 states shown in Table 1 where A is the working node to be moved, B is its sibling, and C its uncle.

The 16 states determine how a working node is moved up. However, information on some of these states is not enough for this decision so that more environmental information must be used. The environmental information beyond up-two-level is thus considered as the *inputs* for moving up the working node. The environmental information (i.e. operator type, and left- or



Input a Input b Input c Input d

Figure 3: Definition of Inputs

right-subtree) of each level beyond the up-two-level is regarded as an input, and thus collectively forms an input string starting from the upward third-level to the root. Since each environmental information consists of the operator type (either union or difference) and the position of working node (either in left- or in right- subtree), there are 4 possible combinations for each input. We use a, b, c, d to represent these combinations, as listed in Figure 3 where T is the subtree containing the working node, and the subtree D is the sibling of T.

3.2 Node Move-up

In this section, we will investigate how a working node in the 16 different possible states be moved up one level in a CSG tree. Some of these states do not need to reference any input and the working node can be directly moved up, but some of them needs one or more inputs. The results are summarized by several properties.

Consider *State 1* (i.e., $(A \cup B) \cup C$) in Table 1. According to the precedence rule of the parenthesis, the working node A must operate with B first and then operate with C. Therefore A is at the same level as B and one level below C. However, if adequate set-theoretical identities (e.g., $(A \cup B) \cup C = A \cup (B \cup C)$) can be applied to transform *State 1* into another expression ($A \cup (B \cup C)$, in this case) in which A is outside the parenthesis, and B, C are enclosed in a pair of parenthesis, then A is one level higher than both B and C, which means that A is already moved up one level by the set-theoretical identities.

Similarly, a working node in *State 2* to *State 8* in Table 1 can be directly moved up one level because the following set-theoretical identities hold. (Note that each expression on the left side corresponds to a state, from *State 1* to *State 8*, in Table 1.)

1. $(A \cup B) \cup C = (C \cup B) \cup A$

2. $(B \cup A) \cup C = (B \cup C) \cup A$

3. $C \cup (A \cup B) = A \cup (C \cup B)$

4. $C \cup (B \cup A) = A \cup (B \cup C)$

5. $C - (A \cup B) = (C - B) - A$

6. $C - (B \cup A) = (C - B) - A$

7. $(A - B) - C = A - (B \cup C)$

8. $(B - A) - C = (B - C) - A$

Property 1: *For one-level move-up in a CSG tree, State 1 to State 8 in Table 1 can directly move the working node up one level without duplicating nodes and referencing any inputs.*

Consider *State 11* (i.e., $(A - B) \cup C$) in Table 1. Since there is no simple set-theoretical identities which can be applied to move up A, input information from larger environment will be used. For *State 11*, the first input (i.e., the upward third-level environment) is enough to decide the movement.

1. Input is *a*. The environment of the working node A becomes $((A - B) \cup C) \cup D$. Since the set-theoretical identity $((A - B) \cup C) \cup D = (D \cup C) \cup (A - B)$ holds and the A in the expression at right side is one level higher than that at left side, A is moved up one level by this set-theoretical identity.

2. Input is *b*. The environment of the working node A becomes $((A - B) \cup C) - D$. It can be easily observed that the following set-theoretical identities hold:

$$\begin{aligned} ((A - B) \cup C) - D &= ((A - B) - D) \cup (C - D) \\ &= (A - (B \cup D)) \cup (C - D) \end{aligned}$$

Comparing the last and first expressions, A has been moved up one level.

Present State	Input			
T	$T \cup D$	$T - D$	$D \cup T$	$D - T$
$(A - B) \cup C$	$(D \cup C) \cup (A - B)$	$(A - (B \cup D)) \cup (C - D)$	$(A - B) \cup (D \cup C)$	$(D - C) - (A - B)$
$(B - A) \cup C$	$(D \cup C) \cup (B - A)$	$((B - D) - A) \cup (C - D)$	$(B - A) \cup (D \cup C)$	$(D - C) - (B - A)$
$C \cup (A - B)$	$(D \cup C) \cup (A - B)$	$(C - D) \cup (A - (B \cup D))$	$(A - B) \cup (C \cup D)$	$(D - C) - (A - B)$
$C \cup (B - A)$	$(C \cup D) \cup (B - A)$	$(C - D) \cup ((B - D) - A)$	$(B - A) \cup (C \cup D)$	$(D - C) - (B - A)$

Table 2: Move up A in *State 11, 12, 13, 14.*

3. Input is *c*. The environment of the working node A becomes $D \cup ((A - B) \cup C)$. Since the set-theoretical identity $D \cup ((A - B) \cup C) = (A - B) \cup (D \cup C)$ holds, A can be moved up one level by applying this set-theoretical identity.

4. Input is *d*. The environment of the working node A becomes $D - ((A - B) \cup C)$. Since the set-theoretical identity $D - ((A - B) \cup C) = (D - C) - (A - B)$ holds, A can be moved up one level.

Similarly, *State 12, 13,* and *14* can also move up nodes using only one input. The results are shown in Table 2 and *Property 2.*

Property 2: *For one-level move-up in a CSG tree, State 11, 12, 13 and State 14 in Table 1 can directly move the working node up one level by only referencing one input.*

Now let us consider the remaining states. For *State 9*, $(A \cup B) - C$, no simple set-theoretical identities can be applied to directly move A up. However, since the identity $(A \cup B) - C = (A - C) \cup (B - C)$ holds, *State 9* can be converted by this identity to *State 11* (A is a leftchild, in the left subtree of its grandparent, its parent is $-$, and grandparent is $\cup$). Once converted to *State 11*, the working node A can be moved up one level as shown in Table 2. By the same token, *State 10* can be first converted to *State 13* by the identity $(B \cup A) - C = (B - C) \cup (A - C)$, and A can then be moved up as the same way in *State 13*. For *State 15*, i.e., $C - (A - B)$, it can be converted to *State 12* by the identity $C - (A - B) = (C - A) \cup (C - (C - B))$. A can then be moved up in the way shown in Table 2 although more nodes (i.e., C) are duplicated. The results

Present State	Input			
T	$T \cup D$	$T - D$	$D \cup T$	$D - T$
$(A \cup B) - C$	$(D \cup (B - C))$ $\cup (A - C)$	$(A - (C \cup D))$ $\cup ((B - C) - D)$	$(A - C) \cup$ $(D \cup (B - C))$	$(D - (B - C))$ $-(A - C)$
$(B \cup A) - C$	$((B - C) \cup D)$ $\cup (A - C)$	$((B - C) - D)$ $\cup (A - (C \cup D)$	$(A - C) \cup$ $((B - C) \cup D)$	$(D - (B - C))$ $-(A - C)$
$C - (A - B)$	$(D \cup (C - (C - B)))$ $\cup (C - A)$	$((C - D) - A) \cup$ $((C - (C - B)) - D)$	$(C - A) \cup$ $(D \cup (C - (C - B)))$	$(D - (C - (C - B)))$ $-(C - A)$

Table 3: Move up A in *State 9, 10, 15.*

State	Set-theoretical Representation
16.*a*	$(C - (B - A)) \cup D$
16.*b*	$(C - (B - A)) - D$
16.*c*	$D \cup (C - (B - A))$
16.*d*	$D - (C - (B - A))$

Table 4: Elaboration of *State 16.*

of moving up A are shown in Table 3 and *Property 3.*

Property 3: *For one-level move-up in a CSG tree, State 9, 10 and State 15 in Table 1 can be converted to State 11, 13, 12, respectively, and the working node in these states can then be moved up one level by only referencing one input.*

Finally let us consider the *State 16*, $C - (B - A)$. This state is the most difficult one where neither can simple set-theoretical identities be directly applied nor can it be converted to another states in order to move up the working node one level. Since a larger environment and more inputs must be used, we define four new states by elaborating *State 16* with the first input augmented. These new states are defined in Table 4. We will analyze them in detail.

Case *State 16.a* $((C - (B - A)) \cup D)$: Now if we temporarily consider $(B - A)$ as a working

Present State	Input			
T	$T \cup E$	$T - E$	$E \cup T$	$E - T$
$(C - (B - A)) \cup D$	$(D \cup E) \cup$ $(C - (B - A))$	$((C - E) - (B - A))$ $\cup (D - E)$	$(C - (B - A))$ $\cup (E \cup D)$	$(E - D) -$ $(C - (B - A))$
$D \cup (C - (B - A))$	$(D \cup E) \cup$ $(C - (B - A))$	$(D - E) \cup$ $((C - E) - (B - A))$	$(C - (B - A))$ $\cup (D \cup E)$	$(E - D) -$ $(C - (B - A))$

Table 5: Move up A in *State 16.a, 16.c.*

node, then *State 16.a* becomes *State 12*, and the operations listed in Table 2 can be applied. Once $(B - A)$ is moved up, so does the A. Since *State 12* needs an input, we need the second input to decide the movement of *State 16.a*. Assume E, from the second input, is the sibling of the subtree denoted as *State 16.a*. The results are listed in Table 5.

Case *State 16.b* $((C - (B - A)) - D)$: This state is the easiest one in *State 16* because, without referencing any input further, its working node can be moved up by the set-theoretical identity: $(C - (B - A)) - D = (C - D) - (B - A)$.

Case *State 16.c* $(D \cup (C - (B - A)))$: Similar to *State 16.a*, if we temporarily consider $(B - A)$ as a working node, then *State 16.c* becomes *State 14*, and the operations listed in Table 2 can be applied. Once $(B - A)$ is moved up, so does the A. Since *State 14* also needs an input, we need the second input to decide the movement of *State 16.c*. The results are listed in Table 5.

Case *State 16.d* $(D - (C - (B - A)))$: This is the most difficult case because no simple set-theoretical identities can be used. Even though we temporarily consider $(B - A)$ as a working node, *State 16.d* become *State 16* again. It means there is recursion here. Assume the second input is used, by the analysis of *State 16.a, 16.b* and 16.c, it is enough to move the working node if this input is b, or it may need one more input if the input is a or c. However, if the second input is d again, then the recursion occurs and more inputs are needed until the input is not d.

Property 4: *For one-level move-up in a CSG tree, State 16 needs to reference one, two or more inputs in order to move up the working node. If the first input is b, then this input is enough. If the input is a or c, then an extra one is enough. However, if the input is d, then more inputs are*

needed until the input is not d.

4 New Algorithm for Feature Unification

Last section has shown how to move up a working node one level in different states. However, for the purpose of feature unification, a working node must be kept on moving up until a desirable position. This means that we need to know not only how to move up a node in different states but also what the next states are after the node has been moved. In this section, we will first derive two tables: one is the action table (or output table) for actually moving the working node, the other is the next state table which decides the working node's next state after moving. We will then apply the *state minimization* technique to reduce the states and also the sizes of these two tables. A driver equipped with these two tables becomes our new algorithm for moving nodes up. Finally, this new move-up algorithm is used to develop a new feature unification algorithm.

4.1 Output Table and Next State Table

Two tables will be created in this section for continuously moving nodes up. One is the output table (or action table) which specifies what actions should be taken to move up one level the working node at different states. The other is the next state table which specifies what the next state is after the working node is moved up one level. The next state will determine the next move-up action.

In Section 3, we have studied how to move nodes up one level. The results are listed in Table 2, Table 3, Table 5 and those set-theoretical identities in Section 3.2. By combining these tables and identities, we then have an output table, which is shown in Table 6. Although *State 1* to *State 8* do not need any input for moving nodes, the results after their movement are augmented with the first input for consistency with other states and for the derivation of next states. (Note that, in Table 6, *State 16.d* needs more inputs until the incoming input is not *d*, then it is converted to one of the *State 16.a, 16.b,* or *16.c* and the associated action can be performed.)

The next state table can be built by observing the states of the working node after it is moved by Table 6. For example, for the *State 11*, if the input is *a*, according to the operation specified

Present State	Input			
T	$T \cup D$	$T - D$	$D \cup T$	$D - T$
$(A \cup B) \cup C$	$(C \cup B) \cup A) \cup D$	$(C \cup B) \cup A) - D$	$D \cup ((C \cup B) \cup A)$	$D - ((C \cup B) \cup A)$
$(B \cup A) \cup C$	$((B \cup C) \cup A) \cup D$	$((B \cup C) \cup A) - D$	$D \cup ((B \cup C) \cup A)$	$D - ((B \cup C) \cup A)$
$C \cup (A \cup B)$	$(A \cup (C \cup B)) \cup D$	$(A \cup (C \cup B)) - D$	$D \cup (A \cup (C \cup B))$	$D - (A \cup (C \cup B))$
$C \cup (B \cup A)$	$(A \cup (B \cup C)) \cup D$	$(A \cup (B \cup C)) - D$	$D \cup (A \cup (B \cup C))$	$D - (A \cup (B \cup C))$
$C - (A \cup B)$	$((C - B) - A) \cup D$	$((C - B) - A) - D$	$D \cup ((C - B) - A)$	$D - ((C - B) - A)$
$C - (B \cup A)$	$((C - B) - A) \cup D$	$((C - B) - A) - D$	$D \cup ((C - B) - A)$	$D - ((C - B) - A)$
$(A - B) - C$	$(A - (B \cup C)) \cup D$	$(A - (B \cup C)) - D$	$D \cup (A - (B \cup C))$	$D - (A - (B \cup C))$
$(B - A) - C$	$((B - C) - A) \cup D$	$((B - C) - A) - D$	$D \cup ((B - C) - A)$	$D - ((B - C) - A)$
$(A \cup B) - C$	$(D \cup (B - C))$ $\cup (A - C)$	$(A - (C \cup D))$ $\cup ((B - C) - D)$	$(A - C) \cup$ $(D \cup (B - C))$	$(D - (B - C))$ $-(A - C)$
$(B \cup A) - C$	$((B - C) \cup D)$ $\cup (A - C)$	$((B - C) - D)$ $\cup (A - (C \cup D)$	$(A - C) \cup$ $((B - C) \cup D)$	$(D - (B - C))$ $-(A - C)$
$(A - B) \cup C$	$(D \cup C) \cup (A - B)$	$(A - (B \cup D)) \cup (C - D)$	$(A - B) \cup (D \cup C)$	$(D - C) - (A - B)$
$(B - A) \cup C$	$(D \cup C) \cup (B - A)$	$((B - D) - A) \cup (C - D)$	$(B - A) \cup (D \cup C)$	$(D - C) - (B - A)$
$C \cup (A - B)$	$(D \cup C) \cup (A - B)$	$(C - D) \cup (A - (B \cup D))$	$(A - B) \cup (C \cup D)$	$(D - C) - (A - B)$
$C \cup (B - A)$	$(C \cup D) \cup (B - A)$	$(C - D) \cup ((B - D) - A)$	$(B - A) \cup (C \cup D)$	$(D - C) - (B - A)$
$C - (A - B)$	$(D \cup (C - (C - B)$ $)) \cup (C - A)$	$((C - D) - A) \cup$ $((C - (C - B)) - D)$	$(C - A) \cup (D$ $\cup (C - (C - B)))$	$(D - (C - (C - B)$ $)) - (C - A)$
$C - (B - A)$	(see *State 16.a* in Table 5.)	$(C - D) - (B - A)$	(see *State 16.c* in Table 5.)	(need more input to resolve.)

Table 6: The Output Table (Action Table) for Move-Up.

by the table, the working node A will be moved to the state $(D \cup C) \cup (A - B)$, which is the *State 13* as far as A's environment is concerned. Therefore, the next state of *State 11* with input a is *State 13*. *State 16* is the one where its next states can not be immediately observed. If the input is b, the next state is *State 16* from Table 6. If its input is a or c, Table 5 must be used because another input is required. According to the Table 5, however, each entry in the row for *State 16.a* or *State 16.c* specifies that the next state for A is always *State 16* even after two inputs are used. For *State 16.d*, it must wait until an input is not d. Once that input occurs, the states will become one of $16.a, 16.b$, or $16.c$ in order to move up the subtree which contains A. As we just mentioned, the next state of all these three is *State 16*. Therefore, the next state of $16.d$ is *State 16*. The next state table is shown in Table 7.

4.2 State Minimization of Tables

If we look into Table 6 and Table 7 carefully, we will soon find that several states seems to be equivalent. Consider *State 1* and *State 2*, for example. Their associated set expressions $((A \cup B) \cup C$ and $(B \cup A) \cup C)$ are set-theoretical equivalent. Their corresponding entries in the next state table are identical, and their corresponding expressions in the output table are also equivalent. It seems that we may merge them into a single state. In this section, we will apply the *state reduction* [5] or *state minimization* [1] techniques to reduce the number of states in both tables.

In the *state reduction* method, there is a *partitioning algorithm* which can effectively partition states into blocks of equivalent states. The algorithm first partitions all states into blocks in which every state has the same output for each input symbol. It then applies each input to the states of the same block to see if their next states are in the same block. If true, put them in the same block for next partition; otherwise, put them into different blocks in next partition. This process is iterated until two consecutive partitions are equivalent. Then all the states in the same block in final partition are considered as equivalent and merged to one state.

Now we apply this partition algorithm. Consider the Table 6. Two outputs are equivalent if their expressions are set-theoretical equivalent. Therefore, we may partition the 16 states into nine blocks: {1, 2, 3, 4}, {5, 6}, {7}, {8}, {9, 10}, {11, 13}, {12, 14}, {15}, {16}. Then we consider the next state table Table 7, and apply each input to the states in same block to see

Present State		Input			
State	T	$T \cup D$	$T - D$	$D \cup T$	$D - T$
1	$(A \cup B) \cup C$	2	10	4	6
2	$(B \cup A) \cup C$	2	10	4	6
3	$C \cup (A \cup B)$	1	9	3	5
4	$C \cup (B \cup A)$	1	9	3	5
5	$C - (A \cup B)$	12	8	14	16
6	$C - (B \cup A)$	12	8	14	16
7	$(A - B) - C$	11	7	13	15
8	$(B - A) - C$	12	8	14	16
9	$(A \cup B) - C$	13	11	11	15
10	$(B \cup A) - C$	13	13	11	15
11	$(A - B) \cup C$	13	11	11	15
12	$(B - A) \cup C$	14	12	12	16
13	$C \cup (A - B)$	13	13	11	15
14	$C \cup (B - A)$	14	14	12	16
15	$C - (A - B)$	14	12	12	16
16	$C - (B - A)$	16	16	16	16

Table 7: The Next State Table for Move-Up.

State	Corresponding Original States
$\tilde{1}$	{ 1,2,3,4 }
$\tilde{2}$	{ 5,6 }
$\tilde{3}$	{ 7 }
$\tilde{4}$	{ 8 }
$\tilde{5}$	{ 9,10 }
$\tilde{6}$	{ 11,13 }
$\tilde{7}$	{ 12,14 }
$\tilde{8}$	{ 15 }
$\tilde{9}$	{ 16 }

Table 8: Definition of New States.

whether new partitions are generated. After the application, we find that no block needs to be split. This means the initial partition is also the final partition, and all the states in each of the nine blocks are equivalent so that they can be merged into single state. Therefore we define these nine new states in Table 8, and create the reduced output table and next state table as shown in Table 9 and Table 10.

4.3 The New Algorithm

With the given output table and next state table in Table 9 and Table 10 respectively, it is very easy to write a driver routine to move up nodes to a desired level in a CSG tree. Table 9 and Table 10 can be implemented by array data structures of two indices. The first index specifies one of the nine states in Table 8, and the second one specifies one of the four possible inputs. Each entry in these two tables can then be addressed (accessed) through the two indices. Listed below is the driver routine *MoveUp*, which uses the table-lookup technique to move a working node A to a desired level L as specified by its caller.

Present State	Input			
T	$T \cup D$	$T - D$	$D \cup T$	$D - T$
$\tilde{1}$	$(A \cup (B \cup C)) \cup D$	$(A \cup (B \cup C)) - D$	$D \cup (A \cup (B \cup C))$	$D - (A \cup (B \cup C))$
$\tilde{2}$	$((C - B) - A) \cup D$	$((C - B) - A) - D$	$D \cup ((C - B) - A)$	$D - ((C - B) - A)$
$\tilde{3}$	$(A - (B \cup C)) \cup D$	$(A - (B \cup C)) - D$	$D \cup (A - (B \cup C))$	$D - (A - (B \cup C))$
$\tilde{4}$	$((B - C) - A) \cup D$	$((B - C) - A) - D$	$D \cup ((B - C) - A)$	$D - ((B - C) - A)$
$\tilde{5}$	$(D \cup (B - C))$ $\cup (A - C)$	$(A - (C \cup D))$ $\cup ((B - C) - D)$	$(A - C) \cup$ $(D \cup (B - C))$	$(D - (B - C))$ $-(A - C)$
$\tilde{6}$	$(D \cup C) \cup (A - B)$	$(A - (B \cup D)) \cup (C - D)$	$(A - B) \cup (D \cup C)$	$(D - C) - (A - B)$
$\tilde{7}$	$(D \cup C) \cup (B - A)$	$((B - D) - A) \cup (C - D)$	$(B - A) \cup (D \cup C)$	$(D - C) - (B - A)$
$\tilde{8}$	$(D \cup (C - (C - B)))$ $\cup (C - A)$	$((C - D) - A) \cup$ $((C - (C - B)) - D)$	$(C - A) \cup$ $(D \cup (C - (C - B)))$	$(D - (C - (C - B)))$ $-(C - A)$
$\tilde{9}$	(see *State 16.a* in Table 5.)	$(C - D) - (B - A)$	(see *State 16.c* in Table 5.)	(need more input to resolve.)

Table 9: The New Output Table (Action Table) for Move-Up.

Present State	Input			
T	$T \cup D$	$T - D$	$D \cup T$	$D - T$
$\tilde{1}$	$\tilde{1}$	$\tilde{5}$	$\tilde{1}$	$\tilde{2}$
$\tilde{2}$	$\tilde{7}$	$\tilde{4}$	$\tilde{7}$	$\tilde{9}$
$\tilde{3}$	$\tilde{6}$	$\tilde{3}$	$\tilde{6}$	$\tilde{8}$
$\tilde{4}$	$\tilde{7}$	$\tilde{4}$	$\tilde{7}$	$\tilde{9}$
$\tilde{5}$	$\tilde{6}$	$\tilde{6}$	$\tilde{6}$	$\tilde{8}$
$\tilde{6}$	$\tilde{6}$	$\tilde{6}$	$\tilde{6}$	$\tilde{8}$
$\tilde{7}$	$\tilde{7}$	$\tilde{7}$	$\tilde{7}$	$\tilde{9}$
$\tilde{8}$	$\tilde{7}$	$\tilde{7}$	$\tilde{7}$	$\tilde{9}$
$\tilde{9}$	$\tilde{9}$	$\tilde{9}$	$\tilde{9}$	$\tilde{9}$

Table 10: The New Next State Table for Move-Up.

procedure *MoveUp* (A : CSGnode; L :integer);
begin
    S := current state of A;
    repeat
        I := the input from A's environment;
        perform the action specified in
            OutputTable(S, I);
        S := *NextStateTable*(S, I);
    until L = level of A;
end; { *MoveUp* }

To illustrate this algorithm, let us consider the example $(((w - x) - y) \cup z) - t$. The working node w, at level 4 in the tree, is to be moved up two levels. w is at the state $\tilde{3}$, thus $S = \tilde{3}$. The environment at the upward third level determines the input, which is a, and thus $I = a$.

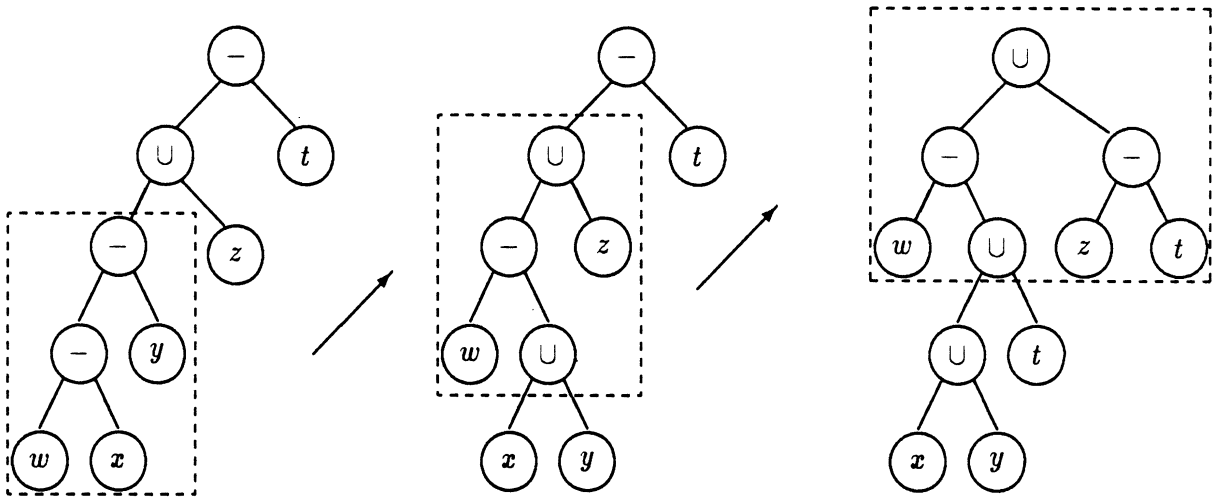


Figure 4: Example of Moving Up w Two Levels.

To perform the action specified in $OutputTable(\tilde{3},a)$, the original subtree $((w - x) - y) \cup z$ is converted to $(A - (B \cup C)) \cup D$ where the working node $A = w$, its sibling $B = x$, its uncle $C = y$, and ancestor $D = z$. The result is a new tree $((w - (x \cup y)) \cup z) - t$, in which the w, at level 3, is moved up one level. Now according to $NextStateTable(\tilde{3},a)$, the next state is $\tilde{6}$, that is, $S = \tilde{6}$. Then the loop in $MoveUp$ is iterated again. The new input $I = b$, and the CSG tree is replaced by $(A - (B \cup D)) \cup (C - D)$, specified in $OutputTable(\tilde{6},b)$, where $A = w$, $B = (x \cup y)$, $C = z$, and $D = t$. The final result is the tree $(w - ((x \cup y) \cup t)) \cup (z - t)$, and the working node, now at level 2, is moved up two levels already. This example is shown in Figure 4.

With the *MoveUp* procedure above, we may construct the algorithm for feature unification. As mentioned before, the most important task for feature unification is to group together relevant primitives after they are recognized by a feature recognizer or feature extractor. Once they are grouped together under a subtree, they are ready to be directly converted to other desirable formats. Listed below is the algorithm to group nodes. Assume that all the primitives recognized by a

feature recognizer are collected in a set, call *Bag*. This algorithm starts with the *Bag* and, pair by pair, groups together all the nodes in the *Bag* under a CSG subtree. The $Shuffle$'s function includes two consecutive exchanges: A's sibling is first exchanged with A's uncle, and then B's sibling is exchanged with B's uncle.

procedure *GroupNode* (Bag : **set** of CSG subtrees);
begin
    repeat
        pick up two elements a and b from Bag;
        r = the latest common ancestor of a and b;
        lv = level of r + 2;
        $MoveUp(a, lv)$;
        $MoveUp(b, lv)$;
        $Shuffle(a, b)$;
        put the parent of a and b back into Bag;
    until only one element in Bag;
    return the only element in Bag;
end; { *GroupNode* }

5 Comparison to the Existing Algorithm

In this section we will compare this new algorithm and the previous one [10] for feature unification. Both algorithms employ the same paradigm to group nodes: first moving them up and then shuffling them together. The real differences between these two algorithms are the operations of moving nodes. Therefore, we will briefly review the moving-up operation, called *Up*, of the previous algorithm before comparing it with its counterpart, *MoveUp*, of the new algorithm.

The previous algorithm also differentiates the working node's environment as 16 states (as we did in Section 3, Table 1), and the procedure *Up* performs adequate operations in response to these 16 states. The basic operations of *Up* are the switch of nodes with their uncles provided

that appropriate set-theoretical identities are applicable. For those states that can not be supported by the basic operations, they are converted to other states so as to apply the basic operations. States can be converted either by duplication of nodes (e.g., $(A \cup B) - C = (A - C) \cup (B - C)$) or by *escaping* the working node to its parent and trying to move up the new working node. If an escaped working node needs another escape (which is determined by the states of the working node), it is then *released* back to the original working node so as to keep on moving. The algorithm fails, unfortunately, if the next operation after a release is still an escape, since it then leads to a cyclic situation where the escape and the release simply alternate the working node and its parent.

In the following, we compare the procedures *Up* and *MoveUp* from several perspectives: primitive operations, control strategy, number of inputs, power.

primitive operations. The primitive operations that *Up* uses are *switch, duplication, escape,* and *release.* The *switch* exchanges a working node with its uncle with possible update of tree structure and operator types. The *duplication* copies nodes and, if necessary, updates operator types too. The *escape* and the *release* simply modify the pointers to tree nodes. All primitive operations are simple and efficient, but the movement of a node up one level might require several primitive operations. On the other hand, the primitive operations of *MoveUp*, which are defined in the output table Table 9, are a little bit more complicated. They might also update the tree structure, operator types, and duplicate nodes. However, as far as moving nodes up one level is concerned, each primitive operation always succeeds by itself—no cooperation of others is needed. It is more efficient than its counterpart in which several operations may be involved and the working node is possibly escaped and released back and forth.

control strategy. The control strategy that is used in *Up* for governing the continuation of moving is basically a state-transition paradigm which identifies the current state and performs corresponding actions. This simple control strategy is, however, complicated by the introduction of the escape-release mechanism. In comparison, the control strategy used in *MoveUp* is a table-driven mechanism which is defined by the next state table and the output table. It is very simple, uniform, and easy to understand and to implement.

number of inputs. The actions of the procedure *Up* are determined by the current state and

the status of the control flag for escape and release. After the node is moved, a new current state is formed by its upward two level environment. A new input (i.e., the upward second level information) is augmented here to form the state before operation. Therefore, the procedure *Up* uses only upward two level information each time to decide operations. On the other hand, the procedure *MoveUp* uses more input information to determine its operations. It is obvious from Table 9 and Table 10 that more than upward two level information is used. Besides the upward two level information which forms the current state, one, two, or more inputs must be used.

power. By the meaning of *power* we mean the ability of moving a node up at different environment. The new procedure *MoveUp* is definitely more powerful than the previous procedure *Up*. The less power of procedure *Up* is because it does not use enough environmental information to make decisions so that the escape and release control mechanism abandons too early on states which are possible to succeed. There are two categories of states which are failed by *Up* due to cycles. The first one is those states which need two consecutive escapes. It includes $(C - (A - B)) \cup D$, $D \cup (C - (A - B))$, $D - (C - (A - B))$, $(C - (B - A)) \cup D$, $D \cup (C - (B - A))$, and $D - (C - (B - A))$ (assume that A is the working node). The second category is those states in which the working node A', which was previously escaped from A, now needs another escape. It includes $(X - A') \cup Y$, $Y \cup (X - A')$, and $Y - (X - A')$ where X, Y are CSG subtrees. However, just from the output table Table 6, or Table 9 in *MoveUp*, the working node in all these failure cases can actually be moved up except the $D - (C - (B - A))$, i.e. *State 16.d*. In the procedure *MoveUp*, a working node in *State 16.d* can still have a chance to be moved up if one of its inputs is not *d*. Therefore, the new procedure *MoveUp* is more powerful than the previous procedure *Up*.

6 Conclusion and Discussion

6.1 Conclusion

In this paper, we have studied the problem of how to move a node up one level in a CSG tree without duplicating itself. Mathematical set theory is employed to solve this problem. All 16 possible states in which a working node may reside are identified, and set-theoretical identities are

applied to move the node up. Several significant and interesting results are discovered (Section 3): on the *feasibility* issue, a node at any of the 16 different states is possible to be moved up one level; on the *implementation* issue, the ways of moving up a node at different states are realized and an algorithm is suggested.

Based on this study, the state tables (i.e., output table and next state table) for moving a node up one level are created. They are further refined by the state reduction (minimization) technique so as to have a minimal number of states, i.e., 7 , instead of 16. These two tables plus a simple driver routine constitute a new procedure, called *MoveUp*, which can move up a node several levels. Our new algorithm thus relies on the procedure *MoveUp* to move nodes around a CSG tree for grouping primitives and unifying features.

Finally we compare the new algorithm with its predecessor with respect to moving up nodes. They— Procedure *MoveUp* and its counterpart, Procedure *Up*— are compared from four different points of view: primitive operations, control strategy, number of inputs, and power. Although the new procedure *MoveUp* has slightly complex primitive operations, it is usually more efficient in time because *Up* sometimes needs several of its primitive operations to move a node up one level while *MoveUp* needs only one. *MoveUp* also has a simple and uniform control strategy (i.e., a table-driven control). Since it references more inputs, Procedure *MoveUp* is more power than Procedure *Up* in moving up nodes. All the cases failed by *Up* are identified and overcome in *MoveUp* except for the case *State 16.d*. The case *State 16.d* is elaborated because a working node in that state can still be moved up as long as one of its incoming inputs is not the input *d*.

6.2 Discussion

Although all the cases failed in the previous algorithm are resolved, the new algorithm still can not move up a node if it is at *State 16.d* and all of its incoming inputs are *d*. In practice, such a situation seldom happens because designers rarely design features in this way. Nevertheless, for theoretical completeness, we are still in search of adequate set identities that can either directly move the node or convert this state to other states.

While the new algorithm *MoveUp* is designed originally for feature unification, it can be used in other applications where the CSG primitives also need to be moved. Feature recognition

can be such an example. Since a feature on an object may be represented by different sets of primitives which may yet scatter around the CSG tree, the tasks of defining feature models and matching the models with CSG trees would be difficult and complex. However, if some rough information about a feature is available (e.g., a feature must be composed of certain types of primitives or certain primitives that have specific relationship in their principal axis directions), then these candidate primitives can be moved and grouped together under a CSG subtree before they are matched or recognized. This would reduce the complexity of feature recognition because the feature matching can be performed locally and feature models can be defined with less variety considered. At present, we are also investigating the application of the new algorithm *MoveUp* along this direction.

References

[1] A. V. Aho, and J. D. Ullman, *Principles of Compiler Design,* Addison-Wesley, Reading, Mass., 1977, pp 99.

[2] G. Allen, "An Introduction to Solid Modeling," *Computer and Graphics*, Vol.8, No.4, 1984, pp. 439-447.

[3] J. W. Boyse and J. E. Gilchrist, "GMSolid: Interactive Modeling for Design and Analysis of Solids," *Computer Graphics and Applications*, Vol.2, No.2, March 1982, pp. 27-40.

[4] C. M. Brown, "PADL-2: A Technical Summary," *Computer Graphics and Applications*, Vol.2, No.2, March 1982, pp. 69-84.

[5] P. J. Denning, J. B. Dennis, and J. E. Qualitz, *Machines, Languages, and Computation,* Prentice-Hall, Englewood Cliffs, New Jersey, 1978, pp. 110.

[6] R. Goldstein and L. Malin, "3D Modeling with the SYNTHAVISION System," Proc. 1st Ann. Conference on Computer Graphics in CAD/CAM Systems, April 1979, pp. 244-247.

[7] J. Goldfeather, J.P.M. Hultquist and H. Fuchs, "Fast Constructive Solid Geometry Display in the Pixel-Powers Graphics System," *Computer Graphics*, Vol.20, No.4, August 1986, pp. 107-116.

[8] R. H. Johnson, *Solid Modeling: A State-of-the-art Report,* Second Edition, CAD/CIM ALERT, Management Roundtable, Inc., Massachusetts, 1985.

[9] Y. Kakazu and N. Okino, "Pattern Recognition Approach to GT Code Generation on CSG," Proc. 16th CIRP International Seminar on Manufacturing Systems, Tokyo, 1984, pp. 10-18.

[10] Y.C. Lee and K.S. Fu, "Machine Understanding of CSG: Extraction and Unification of Manufacturing Features," *Computer Graphics and Applications*, Vol.7, No.1, January 1987, pp. 20-32.

[11] N. Okino, Y. Kakazu and H. Kubo, "TIPS-1: Technical Information Processing System for Computer-aided Design, Drawing, and Manufacturing," *Computer Language for Numerical Control*, Edited by J. Hatvany, North-Holland, Amsterdam, 197 3.

[12] A. A. G. Requicha, "Representations for Rigid Solids: Theory, Methods, and Systems," *Computing Surveys*, Vol.12, No.4, December 1980, pp. 437-464.

[13] A. A. G. Requicha and H. B. Voelcker, "Solid Modeling : A Historical Summary and Contemporary Assessment," *Computer Graphics and Applications*, Vol.2, No.2, March 1982, pp. 9-24.

[14] H. Sato *et al.*, "Fast Image Generation of Constructive Solid Geometry Using A Cellular Array Processor," *Computer Graphics*, Vol.19, No.3, July 1985, pp. 95-102.

[15] H. B. Voelcker *et al.*, "The PADL-1.0/2 System for Defining and Displaying Solid Objects," *Computer Graphics*, Vol.12, No.3, August 1978, pp. 257-263.

[16] M. A. Wesley, T. Lozano-Perez, L. I. Lieberman, M. A. Levin and D. D. Grossman, "A Geometrical Modeling System for Automated Mechanical Assembly," *IBM Journal of Research and Development*, Vol.24, No.1, 1980, pp. 64-74.

[17] J. R. Woodwark, *Computing Shape,* Butterworths, Inc., England, 1986.

[18] J. R. Woodwark, "Some Speculations on Feature Recognition," International Workshop on Geometric Reasoning, Keble College Oxford, England, June 30 - July 3, 1986.