M. Tech. (Computer Science) Dissertation Series

A VISUAL OBJECT REPRESENTATION SCHEME

a dissertation submitted in partial fulfilment of the requirements for the M. Tech. (Computer Science) degree of the Indian Statistical Institute

by

Krishnendu Chakraborty

under the supervision of

Mr. Amarnath Gupta

and

Dr. Aditya Bagchi

INDIAN STATISTICAL INSTITUTE

203, Barrackpore Trunk Road

Calcutta - 700 035

ACKNOWLEDGEMENT

Bouquet of thanks to

Dr. A. Bagchi and Mr. A. Gupta for guidance and encouragement.

My classmates for co-operation.

Krishnendu Chakraborty.

LIST OF CONTENTS

Section 1.

PROBLEM SPECIFICATION

Visual object recognition problem is a well known problem in the field of computer vision. Many people has contributed different propositions to answer the question : how to store an object (i.e., knowledge) and how to recognise an input object or retrieve informations from that storage (i.e, knowledge base). One simple proposition is to extract sufficient amount of features (known as feature vector) for each object and store them in a flat file. An input to this system is compared with each stored object linearly to get a suitable match. Such a system faces problem when the number of stored object increases sufficiently. A search procedure will perform poorly in such a situation. Also there is chance of huge amount of information repetition espacially when all the objects or objects in groups has some structural uniformity. In this dissertation work an in-memomy storage scheme for visual objects has been presented. Here we have basically exploited the decomposablity property of a visual object along with a mechanism for avoiding repetition of information to some extend.

The internal representation of a complete object is based upon the following intuitive observations :

Visual objects are normally decomposable into a hierarchy of coarse to fine descriptions. At any level there is one descriptor for each visual entity (for example for each subpart at that level of resolution), connected to other entities through a predefined set of spatial relations. These spatial relations are parametric, allowing a tolerance range on the values of their parameters.

The attributes belonging to one descriptor of a part or subpart may often constrain an attribute of another part or subpart of the same object.

Different objects of the same structural / functional class often share a set of part names in addition to descriptions . However, it is not imperative that two objects sharing the same name will also share the corresponding description.

Accordingly, the internal representation proposed in this work consists of two spaces :- the name plane, and the description structure. The name plane contains, for each class of objects, a one-level **IS A** tree, and a **PART OF** tree for each object belonging to that class. The description structure for each object is, loosely, a tree of graphs. A tack hammer, for example, contains a head descriptor, and a handle descriptor at the next level of detail with an **IS PERPENDICULAR** relation between them. In the next level, the head splits into the striker end, an unnamed connector, and a chipper end, mutually related by **ADJACENT TO** and **TO THE END OF** relations. In the tree thus constructed, a non-leaf entity, that expands to a graph at the next level is called a higher-level semantic entity, (**HLSE** for short). The feature vector of an HLSE will, in general, be different from that of a primitive. In the example used in the dissertation, they are chosen to be a subset of the feature set used for the primitives. Also, in our present implementation a feature vector is small subset of the set of all possible features.

Next, the representations of the name plane and the description structures are explained as more than one object of the same class get stored in the system. For the name plane, only the **PART OF** structure is affected. At any arbitrary time, suppose there are k objects of the same class. For the i-th object the structure is a tree. However, some nodes of the tree (each node designating a name) would also be shared by the j-th object. It is assumed that a node at a certain level in the i-th PART OF tree will not appear at a different level for the j-th PART OF tree. If a color is assigned to each object, and the arcs of the PART OF tree of that object is assigned that color, then the general PART OF structure becomes an edge-colored multigraph . The description structure for a single object was loosely referred to as a tree of graphs. But a formal treatment requires it to be defined as a variant of a directed hypergraph . The reason is simply that, here a single element (an HLSE) of a subset of nodes (a single description plane) connects to a complete subset (another description plane) by a single arc (formally, ahyperarc). If the subset is called a hypernode, then the hypernodes and the hyperarcs constitute a tree. Now if colors are introduced as in the name plane,the following transformations take place. Some new hypernodes with different colors are introduced. Some old hypernodes expand, such that a part of a description plane of one color gets shared by a description plane of a different color. That is an union operation occurs between the graphs of different colors at any level. There is also a difference operation : a case where two description planes, otherwise equivalent in structure, have a mismatch at one node. Here, a dummy node is created to enforce a structural match of the two graphs, and a fork of bifurcating colors is created alongwith to depict the specialization. Just as in the name plane, strict levels of hierarchy will be maintained for each color, forcing in dummy hypernodes if necessary. Knowledge acquisition for the above structure is clearly, equivalent to the operation of a tree merging into a graph. There is an additional task of the acquisition process. It also updates the "generalization" tree after each object insertion. The generalization of the objects of the same class is defined as follows : In the name plane it is a tree, rooted at the general name, (say hammer) and consists of all those nodes in the PART OF structure that are visited by arcs of all colors. Within one hypernode, it is the largest common subgraph formed by nodes visited by all colors. In other words, it is the intersection of the graphs at any description plane. Between the description planes it is defined just as in the name plane :a tree, rooted at a common (may be dummy) HLSE, and consisting of all those hypernodes visited by hyperarcs of all colors.

Thus introducing an unique color for each object we achieve sufficient sharing of informations among the objects.

Section 2 presents à data structure for the above concept while section 3 gives the procedural detail of the work. The results has been discussed in section 4.

Section 2.

DATA STRUCTURE

The structure of the name plane and the description proposed in section 1 is shown in fig. 1. In this section a suitable data structure for the implementation of the foresaid scheme is presented.

□ Constants ::

1. MAX_FANOUT : Maximum number of decomposed sub-part of any part.
2. MAX_COLOR : Maximum number of specializations allowed.
3. MAX_RELATION : Maximum number of spatial relationship allowed.

□ Level_2 Node ::

# of parts	Name set	Parent_ptr	Offset	Desc._plane_ptr

- # of parts :: Total number of parts in the bucket.
- Name set :: Array of size MAX_FANOUT of Name_def_record.
- Parent_ptr :: Pointer to parent level_2 bucket (For root level_2 bucket it's a pointer pointer to level_0 bucket)
- Offset :: Index of the parent name in parent bucket.
- Desc._plane_ptr :: Pointer to the associated description plane.

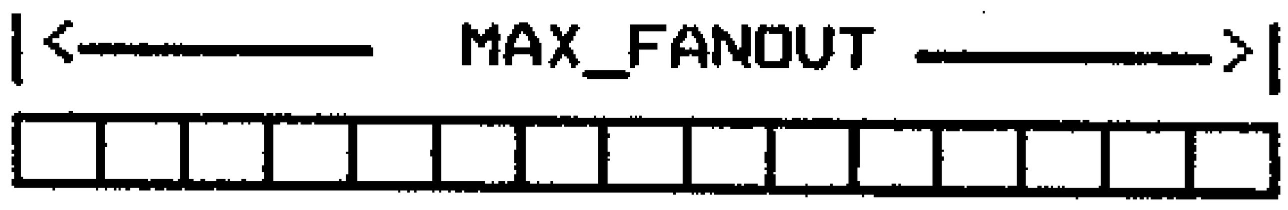
□ Name_def_record ::

Name	Child_ptr	In_color_reg	Out_color_matrix	In_g_bit	Out_g_bit

- Name :: Name of the part.
- Child_ptr :: Pointer to child level_2 bucket.

▸ In_color_reg :: A bit register of size MAX_COLOR. The i-th bit is ON iff the part is shared by a specialization having color_no i.

▸ Out_color_matrix :: A bit matrix of size MAX_COLOR × MAX_FANOUT. Each row represents a color and is a bit register as shown below.



The i-th bit is ON iff the i-th part in the child **level_2** bucket is a member in the named decomposition of the current part.

▸ In_g_bit :: A flag bit which is ON iff the part is visited by all the existing colors.

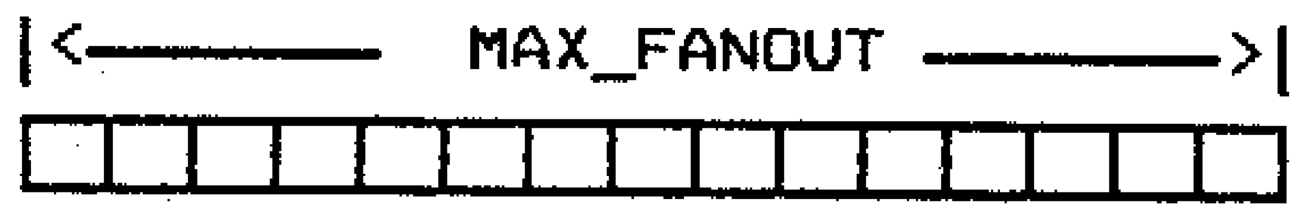
▸ Out_g_bit :: A flag bit. It's ON iff the part undergoes a decomposition for all the participating colors.

▫ <u>Level_0 Node</u> ::

Name	G_code	Specialization set	# of variant	Child_ptr

▸ Name :: Name of the object class.

▸ G_code :: A bit register of size MAX_FANOUT. The i-th bit is ON iff the i-th name in the first level_2 node (Root level_2 bucket) do participate in the definition of the class.



- **Specialization set** :: Set of currently available specializations.It's a collection of **level_1** node.
- **# of variant** :: # of currently available specializations of the class.
- **Child_ptr** :: Pointer to the root **level_2** node.

□ **Level_1 Node** ::

Name	Color_no	Child_code

- **Name** :: Name of the specialization.
- **Color_no** :: Unique integer assigned to the name.
- **Child_code** :: A bit register of size MAX_FANOUT. The i-th bit is ON iff the i-th name in the first **level_2** node (Root level_2 bucket) is a part in the 1-st level decomposition of the specialization.

All the data structures defined so far, collectively defines a **Name Plane** for the object class. It basically gives a named-description of each specialization in the object class.

The spatial relationships between the parts (we shall refer them as **named-node**) may involve some nodes which do not have any significant name, but playes an important roll in the semantic definition (we shall refer them as unnamed-node). A graphical representation of the spatial relationships among the named and unnamed nodes is reflected in a **Description Plane** in the form of a **Node_Table & Arc_table** combination. These Tables are presented below.

- <u>Description_Plane</u> ::

Parent_ptr	Node_table	Arc_table	Hash_table

- Parent_ptr :: Pointer to parent description plane.
- <u>Node_table</u> ::

Id	Type	F_vector	Child_plane	Part_color_reg	G_bit	Out_color_mat

- Id :: Node identifier.
- Type :: (Named-node, Unnamed-node).
- F_vector :: Feature vector of the node.
- Child_plane :: Pointer to child Description Plane.
- Part_color_reg :: A bit register of size MAX_COLOR. The i-th bit is ON iff the node is shared by a specialization having color_no i.

- G_bit :: A flag bit which is ON iff the node is visited by all the existing colors.
- Out_color_matrix :: A matrix of row size MAX_RELATION. Each row represents a color.Each entry of the matrix is a list of arc Id's. Thus the (i,j)-th entry of the matrix gives the set of arcs having label j which emanates

* **Hash_Table** :: Hash table for hashing the nodes in the plane with one or more elements of the feature feator as a key.

Section 3.

DETAIL OF THE WORK

In this section we present an algorithm for the creation of VISUAL SEMANTIC NETWORK (VSN). The total creation procedure has two parts.

1. creation of NAME PLANE.
2. creation of DESCRIPTION PLANE.

The insertion of specializations into the name plane involves the following steps .

Step 0. Initialise a level_0 node with a given object class (say, Hammer).
(Needed for the first specialization only)

Step 1. Insert a new specialization name as a level_1 node into the specialization set of header level_0 node.

Step 2. Insert its first level decomposed parts into the first level_2 node (we call it root level_2 bucket).

step 3. For each part node at root level_2 bucket input its children (decomposed subparts) recursively upto any required depth.

The first step (i.e. step0) is trivial. So we will focus our attention to step1, step2 & step3 only. We can also notice that the intermediate spatial relationship among the parts in any level_2 bucket is depicted in the description plane associated to it. A description plane is a graph G = (V , E), where the vertex set V is a set of named & unnamed nodes and the edge set E contains labelled colored edges. A sharing of informations stored in the existing description plane by a new description is obtained by a proper merging of the later with the former. Our algorithm will do such merging for each level_2 bucket only after receiving all subpart names for it.

A named or unnamed node may undergo decomposition involving only unnamed nodes. In such a case an heuristic way of merging has been used.

The complete algorithm is formulated by a number of interrelated procedures. The detail of some of the auxilliary procedures are omitted. For the time being, let us assume that the following procedures do exists.

1. Procedure Input_relationship(temp_node_table, temp_arc_table) :-

```
Node_Table          temp_node_table;
Arc_Table           temp_arc_table;
```

It inputs a relationship graph associated with some decomposition.The graph is returned in the form temp_node_table & temp_arc_table pair.

2. Procedure Merge_named_graph(temp_node_table, temp_arc_table, bucket, color_no) :-

```
Node_Table          temp_node_table;
Arc_Table           temp_arc_table;
Level_2 node        bucket;
Integer             color_no;
```

This procedure merges an input graph of color color_no and given in the form of temp_node_table & temp_arc_table to the model graph of the the description plane bucket.desc_plane_ptr. Here the vertex set of both the graphs may contain unnamed nodes along with named nodes.

3. Procedure Merge_unnamed_graph(parent_plane, node_name, color_no) :-

```
Description_plane   parent_plane;
String              node_name; /* A node in parent plane */
Integer             color_no;
```

This procedure is same as Merge_named_graph() except that the Input_relationship() is called inside it whenever needed and the description plane of interest is the child description plane of the node node_name.

4. Procedure Compare_feature(f_vector1, f_vector2) :-

Feature_vector f_vector1, f_vector2;

The f_vector1 is compared with f_vector2 (as reference) and is returned TRUE iff they are 'comparable'. The term 'comparable' is problem dependant. The detail of this procedure is not given.

5. Procedure Try_to_fork(node1, node2) :-

Node node1, node2 ; /* i.e., Node_table entries */

This procedure compares node2.f_vector with node1.f_vector and do forking at node node1 if the two feature vectors are comparable to a predefined degree. The detail is also skipped for this procedure.

A formal description of the procedures is given below.

<u>Procedure</u> Insert_specialization(class_head, sp_name, color_no)

```
Level_0 node    class_head; /* Header node of a class */
String          sp_name;    /* Name of a spacialization */
Integer         color_no;   /* color associated with sp_name */
```

<u>Begin</u>

1. <u>If</u> sp_name ∈ class_head.specialization_set <u>then</u> omit it and stop.
   <u>else</u> <u>goto</u> step 2.
   <u>EndIf</u>

2. Include sp_name in class_head.specializatin_set.

```
3.     Input feature vector of sp_name.

4.     call Insert_first_level_decomposition( class_head.child_ptr, sp_name,
                                                                color_no);
End.


Procedure Insert_first_level_decomposition( root_bucket, sp_name, color_no )
Level_2 node            root_bucket; /* root level_2 bucket */
String                  sp_name;     /* Name of a spacialization */
Integer                 color_no     /* Color associated with sp_name */
Var
   String               part_name;
   Node_Table           temp_node_table;
   Arc_Table            temp_arc_table;
Begin
1.    Let d_list be the list of named decomposed parts of sp_name.
      part_name ←—— first name in d_list;
2.    While part_name ≠ NULL do
2.1.     If part_name ∈ root_bucket.name_set then
2.1.1.      Let part_name be the j-th name in root_bucket.name_set.
         Else
2.1.2.      Let j be index of the free location in the bucket ( i.e., in
            root_bucket.name_set );
            Insert part_name at the j-th free location of the bucket;
         EndIf
2.2.     Set color_no -th bit of part_name.in_color_reg;
```

```
2.3.    Set j -th bit of sp_name.child_code;
2.4.    part_name <---- next name in d_list;
     EndWhile
3.   Call Input_relationship( temp_node_table, temp_arc_table );
4.   Call Merge_named_graph( temp_node_table, temp_arc_table,
                                            root_bucket, color_no );
5.   For each part N in root_bucket for which color_no is a
     participating color do
5.1.      If any unnamed decomposition of N is required then
5.1.1.       call Merge_unnamed_graph( temp_node_table, temp_arc_table,
                                  N, bucket.dsc_plane_ptr, color_no );
          Else
5.1.2.       call Input_children( N.child_ptr , N, color_no ).
          EndIf
     EndFor
End.

Procedure Input_children( bucket, part_name, color_no )
Level_2 node      bucket;     /* Child bucket of the part part_name */
String            part_name;
Integer           color_no;
Var
String            subpart_name;
Node_Table        temp_node_table;
Arc_Table         temp_arc_table;
Begin
1.   Let d_list be the list of named decomposed parts of part part_name.
     subpart_name <---- first name in  d_list;
```

```
2.    While subpart_name ≠ NULL do
2.1.    If subpart_name ∈ bucket.name_set then
2.1.1.      Let subpart_name be the j-th name in bucket.name_set.
        Else
2.1.2.      Let j be index of the free location in the bucket ( i.e., in
            bucket.name_set );
            Insert subpart_name at the j-th free location of the bucket;
        EndIf
2.2.    Set color_no -th bit of subpart_name.in_color_reg;
2.3.    Set ( color_no, j )-th bit of part_name.out_color_matrix;
2.4.    If subpart_name is not a leaf node then
2.4.1.      Call Input_children(subpart_name.child_ptr,subpart_name,color_no );
        EndIf
2.5.    subpart_name ←—— next name in d_list.
2.6.    If subpart_name = NULL then
2.6.1.      Call Input_relationship( temp_node_table, temp_arc_table );
2.6.2.      Call Merge_named_graph( temp_node_table, temp_arc_table,
                                                 bucket, color_no );
        EndIf
      EndWhile
3.     For each node N in bucket.desc_plane_ptr , such that N is visited by the
       color color_no   do
3.1.        If any more unnamed decomposition of N is required then
3.1.1.          Call Merge_unnamed_graph( temp_node_table, temp_arc_table,
                           N, bucket.dsc_plane_ptr, color_no );
```

```
        EndIf
    EndFor
End.
```

The two graph merging algorithms mentioned earlier may require multiple passes. For example, the procedure Merge_named_graph() works as follows : in the first pass each node N of the input graph is visited. In such a visit the algorithm try to map N to some model graph node. In the next step it try to map all one-arc-length neighbor of N. In this step each neighbor node M (both named and unnamed) of N is either successfully matched to some model graph node M_1, or a new equivalent node M_1 is created in model graph and the spatial relationship between N & M is preserved in the model. In the first step a mamed node in matched (or created)easily. But a new unnmaed node (one which is not already visited in the second step) is not created and leave the pass incomplete. Thus the algorithm forces all unnamed nodes of the input graph to be mapped in step 2 only. A boolean flag not_completed has been introduced for this purpose. The other procedure Merge_unnamed_graph() is mostly same as the former except that an initial heuristic search is performed to get a match between an input graph node and a model graph node. The detail of both the algorithms are presented below.

```
Procedure  Merge_unnamed_graph( parent_ptr, node_name, color_no )
Dsc._plane_pointer              parent_ptr;
String                          node_name;
                                /* A single entry of parent_ptr.Node_Table */
Integer                         color_no;
```

EndIf

1.7. not_completed ←—— FALSE ;

1.8. Start fron the node N (first matched node) of the input graph and consider each node in some order to do the following

1.9. If N is already matched to the model graph node N_k then

1.9.1. 'mark' N ; /* A marking indicates that the node is visited from the outer loop i.e. as a result of step 1.8*/

1.9.2. For each edge e in input graph, such that, e = (N, M) and e.label = 1 (say), and M is not marked do

1.9.2.1. If M is already matched to some the model graph node M_k then

1.9.2.1.1. If e1 = (N_k, M_k) and e1.label = 1 is not in the model then

1.9.2.1.1.1. create the same;

EndIf

1.9.2.1.2. Set color_no-th bit of e1.part_color_reg ;

Else

1.9.2.1.3. Find one-arc-length neighbor set (nb_set) of N_k;

1.9.2.1.4. If M is matched to some M_k ∈ nb_set then

1.9.2.1.4.1. Call Try_to_fork(M_k, M);

1.9.2.1.4.2. Set color_no-th bit of M_k.part_color_reg;

1.9.2.1.4.3. If e1 = (N_k, M_k) and e1.label = 1 is not in the model

1.9.2.1.4.3.1. then create the same ;

EndIf

1.9.2.1.4.4. Set color_no-th bit of e1.part_color_reg ;

Else

1.9.2.1.4.5. Create a new node M_k equivalent to M in the model graph ;

1.9.2.1.4.6. Create an edge e1 = (N_k, M_k) with e1.label = 1;

1.9.2.1.4.7. Set color_no-th bit of e1.part_color_reg;

```
            EndIf

          EndIf

      EndFor

   Else
1.9.3.  not_completed ←—— TRUE ;
            /* So one more pass is required */

   EndIf

1.10. Repeat through step 1.7 until not_completed = False;

1.11.  For each node N in  node_name.child_ptr.node_table,  such  that  N  is
       visited by the color color_no do

1.11.1.   If any more unnamed decomposition of N is required then
1.11.1.1.    Call Merge_unnamed_graph( N, bucket.dsc_plane_ptr, color_no );

          EndIf

       EndFor

End.
```

```
Procedure Merge_named_graph( temp_node_table, temp_arc_table,bucket,color_no )
Node_Table                  temp_node_table;
Arc_Table                   temp_arc_table;
Level_2 node                bucket;
Integer                     color_no;
```

Var

Boolean not_completed;

Begin

1. not_completed ⟵ FALSE;

2. For each node N of temp_node_table do

3. Switch (N.type)

Case 'named-node' :

3.1. If N already exists in the model graph as N_1 then

3.1.1. Call Try_to_fork(N_1, N);

Else

3.1.2. Create a new node N_1 equivalent to N in the model graph (i.e., in bucket.desc_plane_ptr.node_table)

EndIf

3.2. Set color_no-th bit of N_1.part_color_reg ;

3.3. For each edge e in input graph, such that, e = (N, M) ,e.label = 1 (say), and M is not marked do

/* A marking indicates that the node is visited from the outer loop i.e. as a result of step 2*/

3.3.1. If M is a named node then

3.3.1.1. If M already exists in the model graph as M_1 then

3.3.1.1.1. Call Try_to_fork(M_1, M);

Else

3.3.1.1.2. Create a new node M_1 equivalent to M in the model graph (i.e.,in bucket.desc_plane_ptr.node_table);

EndIf

3.3.1.2. If e1 = (N_1, M_1) and e1.label = 1 is not in the model then

```
3.3.1.2.1.          create the same;
            EndIf
3.3.1.3.    Set color_no-th bit of e1.part_color_reg ;
         Else /* M is an unnamed node */
3.3.1.4.    If M is already matched to the model graph node Mk then
3.3.1.4.1.        If e1 = ( Nk, Mk ) and e1.label = 1 is not in the model then
3.3.1.4.1.1.          create the same;
                  EndIf
3.3.1.4.2.        Set color_no-th bit of e1.part_color_reg ;
            Else
3.3.1.4.3.        Find one-arc-length neighbor set ( nb_set ) of Nk;
3.3.1.4.4.        If M is matched to Mk ∈ nb_set then
3.3.1.4.4.1.          Call Try_to_fork( Mk, M );
3.3.1.4.4.2.          Set color_no-th bit of Mk.part_color_reg;
3.3.1.4.4.3.          If e1 = ( Nk, Mk ) and e1.label = 1 is not in the model
3.3.1.4.4.3.1.        then  create the same ;

                      EndIf
3.3.1.4.4.4.          Set color_no-th bit of e1.part_color_reg ;
                  Else
3.3.1.4.4.5.          Create a new node Mk equivalent to M in the model graph ;
3.3.1.4.4.6.          Create an edge e1 = ( Nk, Mk ) with e1.label = 1;
3.3.1.4.4.7.          Set color_no-th bit of e1.part_color_reg;
                  EndIf
            EndIf
         EndIf
       EndFor
3.4.   Mark N;
```

```
      Case 'unnamed-node' :

3.5.           If N already exists in the model graph as N1 then
3.5.1.           For each edge e in input graph, such that, e = ( N, M ) ,e.label = 1
                 ( say ), and M is not marked do
                  /* A marking indicates that the node is visited from the
                     outer loop i.e. as a result of step 2*/
3.5.1.1.            If M is already matched to the model graph node M1 then
3.5.1.1.1.             If e1 = ( N1, M1 ) and e1.label = 1 is not in the model then
3.5.1.1.1.1.              create the same;
                       EndIf
3.5.1.1.2.             Set color_no-th bit of e1.part_color_reg ;
                    Else
3.5.1.1.3.             Find one-arc-length neighbor set ( nb_set ) of N1;
3.5.1.1.4.             If M is matched to M1 ∈ nb_set then
3.5.1.1.4.1.              Call Try_to_fork( M1, M );
3.5.1.1.4.2.              Set color_no-th bit of M1.part_color_reg;
3.5.1.1.4.3.              If e1 = ( N1, M1 ) and e1.label = 1 is not in the model
3.5.1.1.4.3.1.               then  create the same ;
                          EndIf
3.5.1.1.4.4.              Set color_no-th bit of e1.part_color_reg ;
                       Else
3.5.1.1.4.5.              Create a new node M1 equivalent to M in the model graph ;
3.5.1.1.4.6.              Create an edge e1 = ( N1, M1 ) with e1.label = 1;
3.5.1.1.4.7.              Set color_no-th bit of e1.part_color_reg;
                       EndIf
                    EndIf
                 EndFor
```

```
3.5.2.    Mark the node N ;
      Else
3.5.3.    not_completed ←—— TRUE ;
      EndIf
       EndSwitch
       EndFor
End.
```

Section 4.
RESULTS

In this dissertation work our object class of interest is the hammer from which three specializations (namely tack hammer, ball-pein hammer, double hitting end hammer) have been chosen. These are shown in the figures (fig 2., fig 3., fig 4.) where different parts are pointed out.

As stated in the first section, a feature vector (of an HLSE as well as of a primitive entity) in the current implementation is a subset of the set of possible features. The feature informations are extracted from the 2D image of the object where each entity (both HLSE and primitive) is a closed contour.The feature elements chosen here are as follows :-

1. <u>No. of Zero Crossing Points</u> (n) :-

Cuavature of a point on the contour varies from point to point and changes sign whenever the curve curves significantly. A zero crossing point of curvature (known as ZC point) is defined by the point on the curve where the curvature changes sign either from -ve to +ve or zero, or from +ve to -ve or zero . Thus the number of ZC points of a contour gives the number of significant turning in the path along a contour.

2. <u>$Perimeter^2$ / Area</u> :-

This unitless ratio gives information about elongatedness of the contour.

3. <u>Normalised Length Array</u> :-

The distance of each ZC point from the centroid of the contour is normalised with respect to the maximum distance . Thus, this is a variable length feature of size n (# of ZC point).

4. Angle Array :-

For each ZC point a line joining the point and the centroid of the contour is drawn. The line segment having maximum length is taken as reference.The angle formed by each line segment with the reference line measured in some predetermined direction (say, counterclockwise) is taken as a feature element. Once again this feature array is of size n.

5. Spine Length :-

Length of the largest spine of a closed contour gives information about the elongation of the contour.

For example, the contour hitting_end is represented by a feature vector given below.

1. # of ZC point = 8.

2. Perimeter^2 / Area = 14.45.

3. Normalised length array (of size 8) :

0.750	0.875	1.000	1.000	0.750	1.000	1.000	0.875

4. Angle array (of size 8 and in degree) :

0.000	40.00	55.00	125.0	180.0	135.0	305.0	320.0

5. Spine length = 24 unit.

Now consider the tack hammer (fig 2.) as the first representative of the class hammer. Its name plane structure is shown in fig 5a. and the description structures at different levels are presented in tabular form in Table 1. The same table also depicts the resulting description structure of the object class hammer. The input description structure is self explanatory. For example, the level 2 description consists of two named entities, namely head and handle, where head IS_PER_TO (is perpendicular to) handle.In the resulting description at each level three new columns are introduced. The *Mult #* gives the multiplicity of forking at a node. If a dummy node be a representative of i (i > 1) different nodes then that dummy node is said to have a multiplicity i. All non-dummy nodes do have Mult # = 1. The column *color grouping* refers to the type of forking at a node. If a node N is visited by a number (k : k > 1) of colors but N.Mult # = 1 then we can say that N is a non-dummy node. But if 1 < N.Mult # < k then the question arises how the different participating colors are grouped during forking at N. This information is displayed in the column *color grouping* by separating the colors in the same group by ','s and the groups by ';'. Thus *Mult # & Color grouping* bears information about the source and destination node of an arc.The third new column *Participating color* is simply the color numbers which visit the relational arc given in the corresponding row. Also notice that a '*' mark appears before a node (both source and destination) as well as before a relational arc. A '*' mark implies generalization. A node or an arc is '*' marked iff it is visited by all the existing colors. Insertion of the first specialization of a class marks all nodes and arcs introduced in the system as shown in Table 1. A subsequent insertion of a new specialization (ball-pein hammer) unmarks some of the nodes and arcs (refer Table 2, for example, node tack_end, arc no 2 at level 3). A separate name plane structure for this hammer is shown in fig 5b. Insertion of this new structure modifies the existing structure (fig 5a.) to form a new name plane (fig 6a.). In this system state the third specialization (double hitting end hammer) is introduced (see informations displayed in Table 3.). On a careful observation of the description structure at level 3 (decomposition of head), we notice that two arcs (no 0 & 2) are duplicated. This is because, for a double hitting end hammer the hitting_end IS_ADJ_TO (is adjacent to) both sides of the unnamed node (i.e., connector) 1 such that the three entities (hitting_end, 1, hitting_end) are COAXIAL. This is merely a convention followed in the system. The resulting name plane is shown in fig 6b.

The resulting description graph at different levels shown in Table 3 gives a complete description structure of the object class hammer with three specializations. Information about a specilization having color no i can be extracted by traversing through this structure and examining only those nodes and arcs which are visited by the color i. Also notice that this structure gives a generalised definition of hammer as follows : *a hammer has two main parts, namely a head and a handle which are perpendicular to each other. A head consists of a hitting_end and an unnamed node 1 (call it u1) which are adjacent and coaxial. The unnamed node u1 is a combination of three contours 1, 2, 3 (call them u11 , u12 and u13 respectively) such that u11 is adjacent to u13 and u12 in inside u13.*

An important characteristic of a learning procedure is data independence which says that learning should be independent of the order of the data items. Our present system is not a learning system in the true sense, but it builds a knowledge structure independent of the input ordering. Fig 7. shows the description structure (in the form of graph) at different levels for 6 (3!) possible ordering of the specializations.

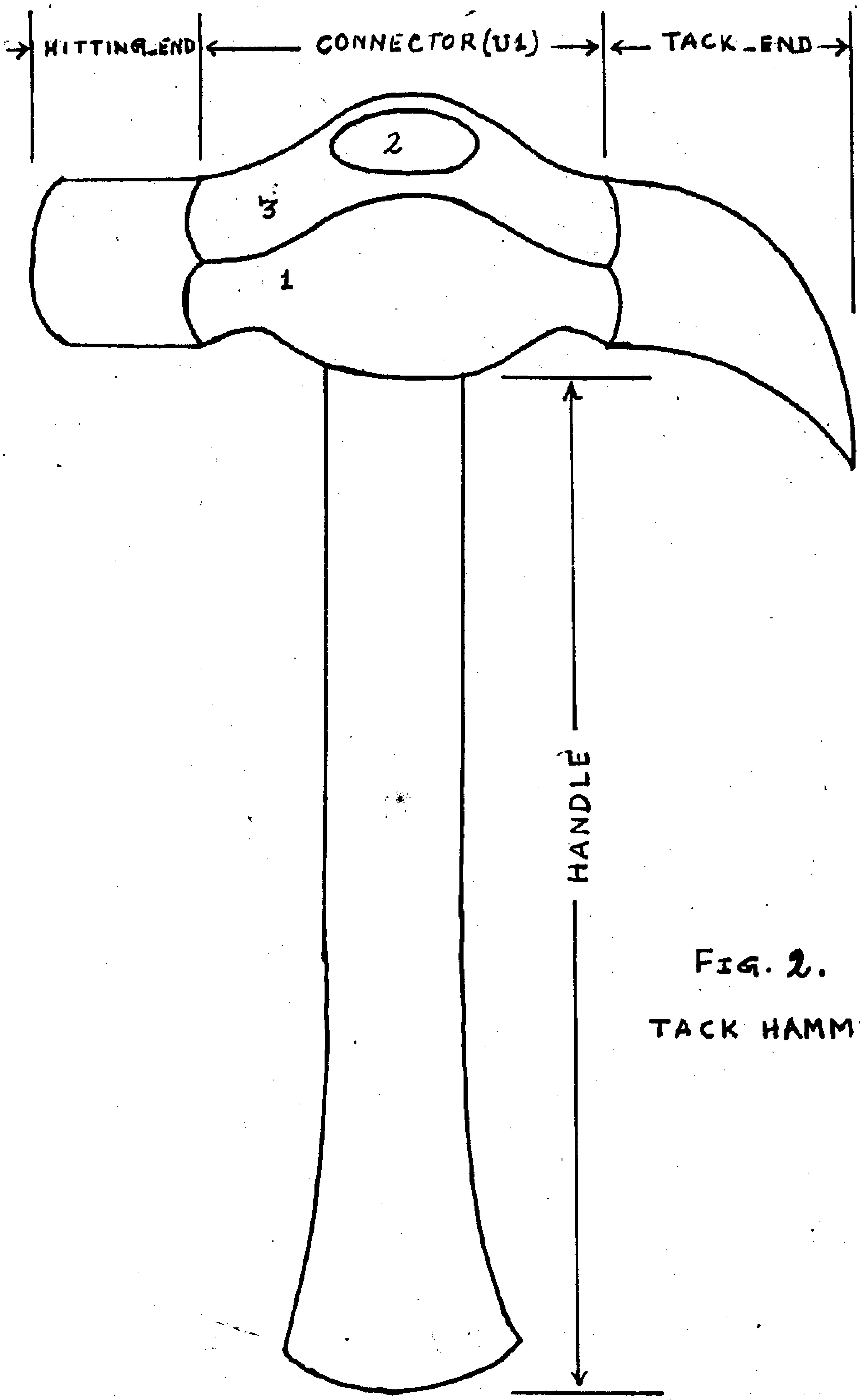


FIG. 2.

TACK HAMMER

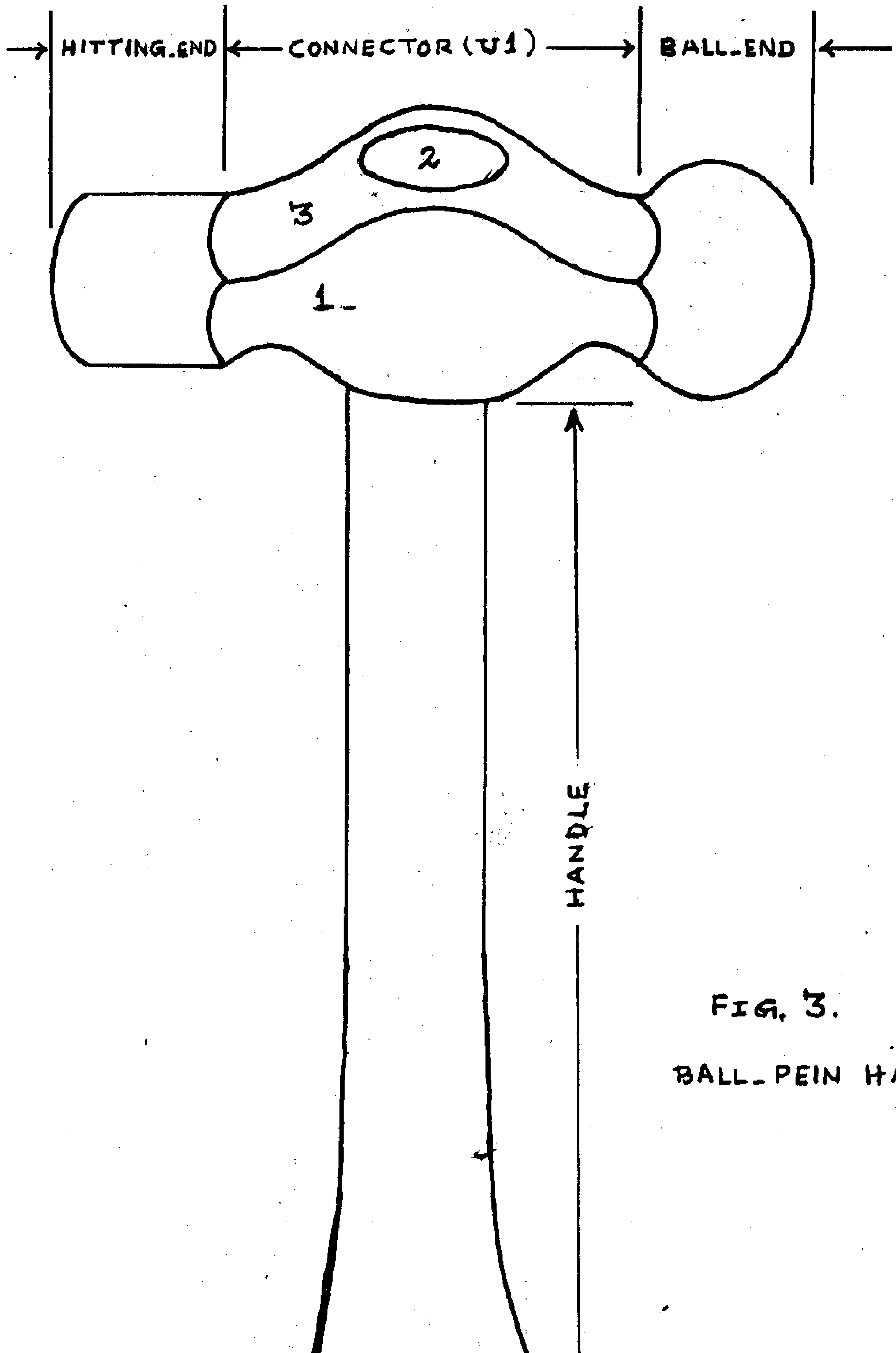


FIG. 3.

BALL-PEIN HAMMER

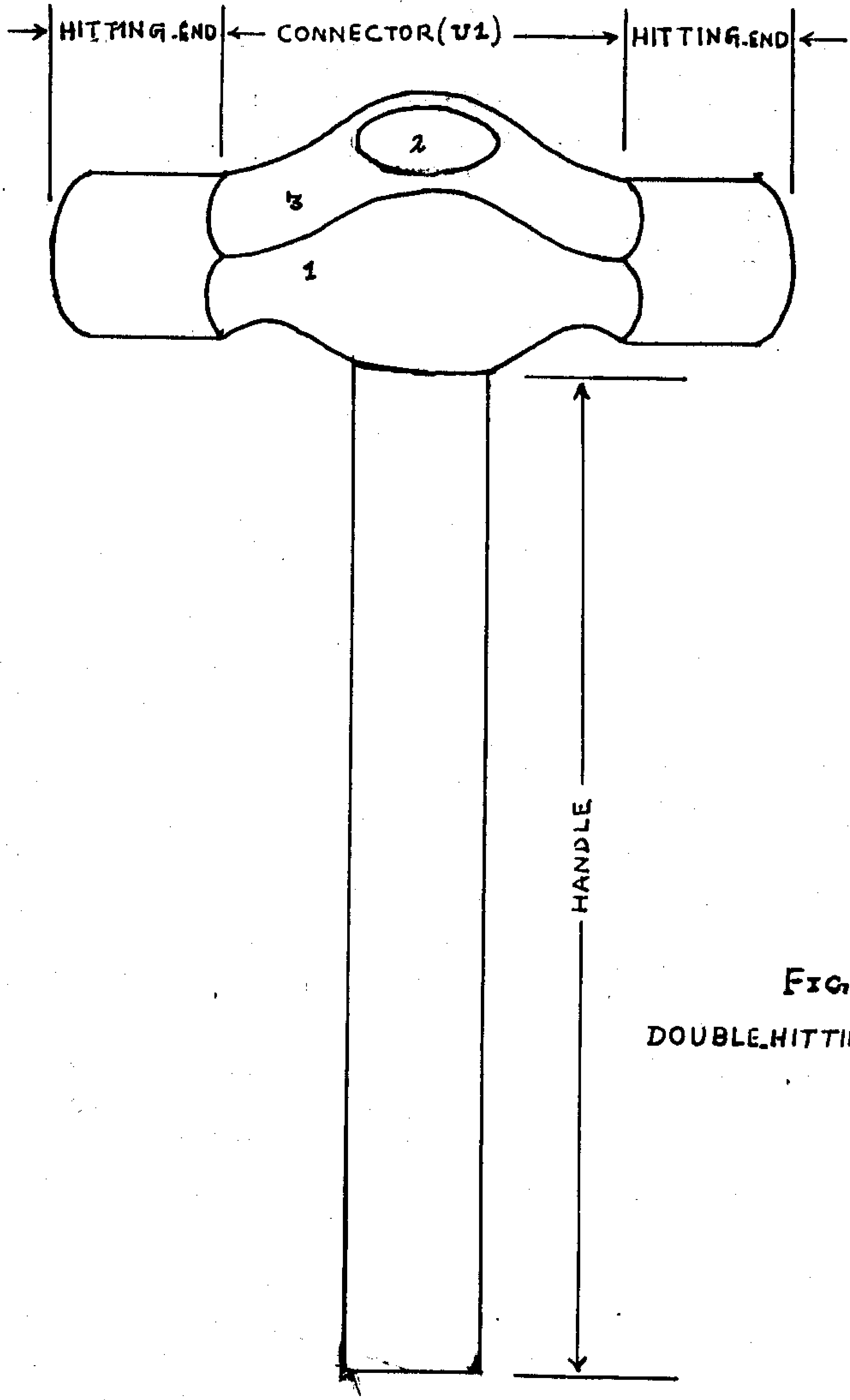


FIG. 4.

DOUBLE-HITTING-END HAMMER

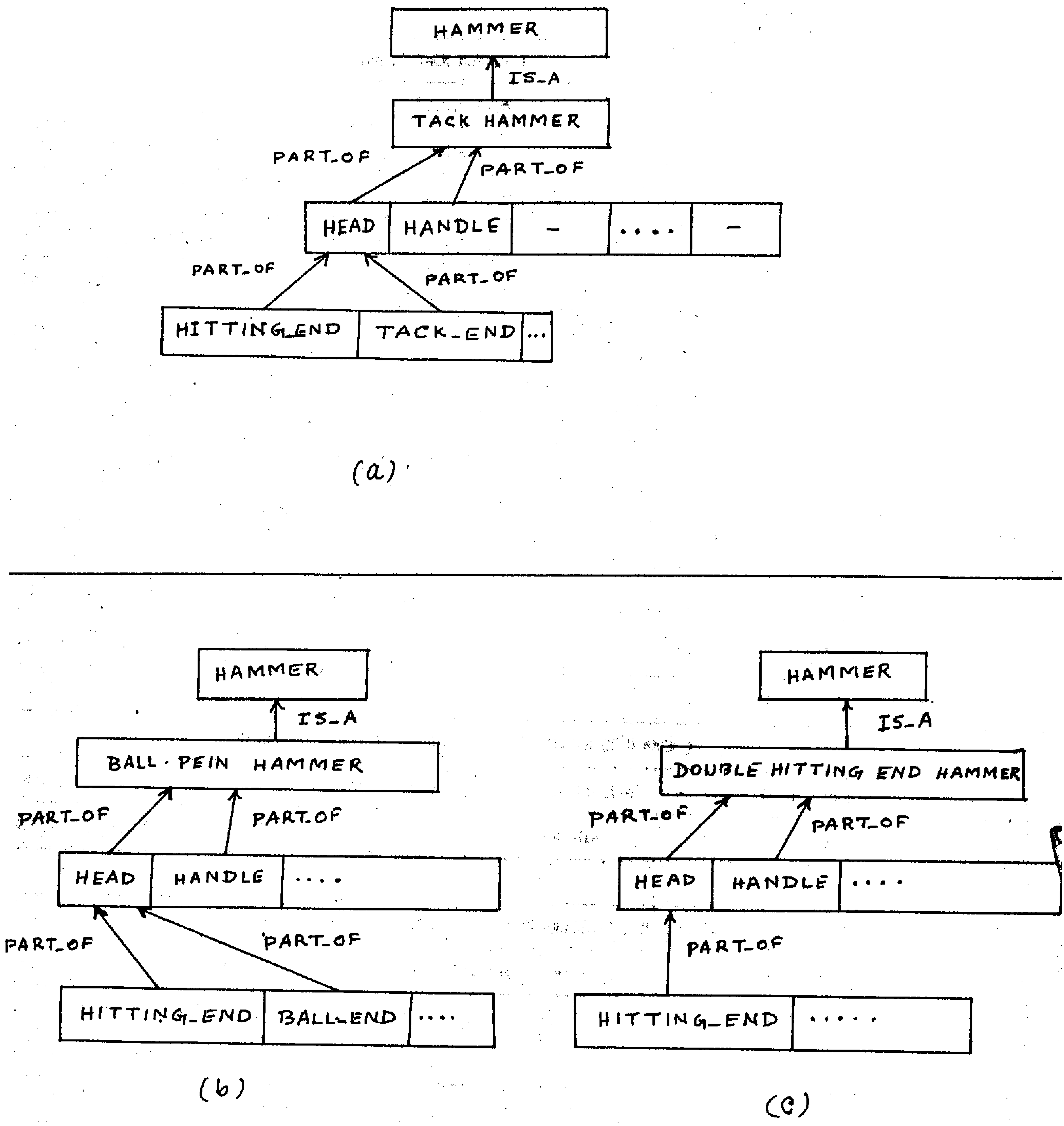


Fig 5. NAME PLANE STRUCTURE FOR DIFFERENT HAMMERS

(a) TACK HAMMER

(b) BALL - PEIN HAMMER

(c) DOUBLE HITTING END HAMMER

1. TACK HAMMER

INPUT GRAPH AT LEVEL 2 (DECOMPOSITION OF TACK HAMMER)

No.	Label	Source	Destination
0.	IS_PER_TO	head	handle

end

INPUT GRAPH AT LEVEL 3 (DECOMPOSITION OF head)

No.	Label	Source	Destination
0.	IS_ADJ_TO	hitting_end	1
1.	COAXIAL	hitting_end	1
2.	IS_ADJ_TO	tack_end	1

end

INPUT GRAPH AT LEVEL 4 (DECOMPOSITION OF 1)

No.	Label	Source	Destination
0.	IS_ADJ_TO	1	3
1.	IS_INSIDE	2	3

end

AFTER MERGING

RESULTING GRAPH AT LEVEL 2 (DECOMPOSITION OF HAMMER)

No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_PER_TO	*head	1	1;	*handle	1	1;	1,

end

RESULTING GRAPH AT LEVEL 3 (DECOMPOSITION OF head)

No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_ADJ_TO	*hitting_end	1	1;	*1	1	1;	1,
* 1.	COAXIAL	*hitting_end	1	1;	*1	1	1;	1,
* 2.	IS_ADJ_TO	*tack_end	1	1;	*1	1	1;	1,

end

RESULTING GRAPH AT LEVEL 4 (DECOMPOSITION OF 1)

No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_ADJ_TO	*1	1	1;	*3	1	1;	1,
* 1.	IS_INSIDE	*2	1	1;	*3	1	1;	1,

end

Table 1.

2. BALL-PEIN HAMMER

INPUT GRAPH AT LEVEL 2 (DECOMPOSITION OF BALL-PEIN HAMMER)

No.	Label	Source	Destination
0.	IS_PER_TO	head	handle

end

INPUT GRAPH AT LEVEL 3 (DECOMPOSITION OF head)

No.	Label	Source	Destination
0.	IS_ADJ_TO	hitting_end	1
1.	COAXIAL	hitting_end	1
2.	COAXIAL	hitting_end	ball_end
3.	IS_ADJ_TO	ball_end	1
4.	COAXIAL	ball_end	1

end

INPUT GRAPH AT LEVEL 4 (DECOMPOSITION OF 1)

No.	Label	Source	Destination
0.	IS_ADJ_TO	1	3
1.	IS_INSIDE	2	3

end

AFTER MERGING

RESULTING GRAPH AT LEVEL 2 (DECOMPOSITION OF HAMMER)

No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_PER_TO	*head	2	1;2;	*handle	1	2,1;	2, 1,

end

RESULTING GRAPH AT LEVEL 3 (DECOMPOSITION OF head)

No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_ADJ_TO	*hitting_end	1	2,1;	*1	1	2,1;	2, 1,
* 1.	COAXIAL	*hitting_end	1	2,1;	*1	1	2,1;	2, 1,
2.	IS_ADJ_TO	tack_end	1	1;	*1	1	2,1;	1,
3.	COAXIAL	*hitting_end	1	2,1;	ball_end	1	2;	2,
4.	IS_ADJ_TO	ball_end	1	2;	*1	1	2,1;	2,
5.	COAXIAL	ball_end	1	2;	*1	1	2,1;	2,

end

RESULTING GRAPH AT LEVEL 4 (DECOMPOSITION OF 1)

No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_ADJ_TO	*1	1	2,1;	*3	1	2,1;	2, 1,
* 1.	IS_INSIDE	*2	1	2,1;	*3	1	2,1;	2, 1,

end

Table 2.

3. DOUBLE HITTING END HAMMER

INPUT GRAPH AT LEVEL 2 (DECOMPOSITION OF DOUBLE HITTING END HAMMER)

No.	Label	Source	Destination
0.	IS_PER_TO	head	handle

end

INPUT GRAPH AT LEVEL 3 (DECOMPOSITION OF head)

No.	Label	Source	Destination
0.	IS_ADJ_TO	hitting_end	1
1.	IS_ADJ_TO	hitting_end	1
2.	COAXIAL	hitting_end	1
3.	COAXIAL	hitting_end	1

end

INPUT GRAPH AT LEVEL 4 (DECOMPOSITION OF 1)

No.	Label	Source	Destination
0.	IS_ADJ_TO	1	3
1.	IS_INSIDE	2	3

end

AFTER MERGING

RESULTING GRAPH AT LEVEL 2 (DECOMPOSITION OF HAMMER)

No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_PER_TO	*head	3	1;2;3;	*handle	2	2,1;3;	3, 2, 1,

end

RESULTING GRAPH AT LEVEL 3 (DECOMPOSITION OF head)

No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_ADJ_TO	*hitting_end	1	3,2,1;	*1	1	3,2,1;	3, 2, 1,
* 1.	COAXIAL	*hitting_end	1	3,2,1;	*1	1	3,2,1;	3, 2, 1,
2.	IS_ADJ_TO	tack_end	1	1;	*1	1	3,2,1;	1,
3.	COAXIAL	*hitting_end	1	3,2,1;	ball_end	1	2;	2,
4.	IS_ADJ_TO	ball_end	1	2;	*1	1	3,2,1;	2,
5.	COAXIAL	ball_end	1	2;	*1	1	3,2,1;	2,
6.	IS_ADJ_TO	*hitting_end	1	3,2,1;	*1	1	3,2,1;	3,
7.	COAXIAL	*hitting_end	1	3,2,1;	*1	1	3,2,1;	3,

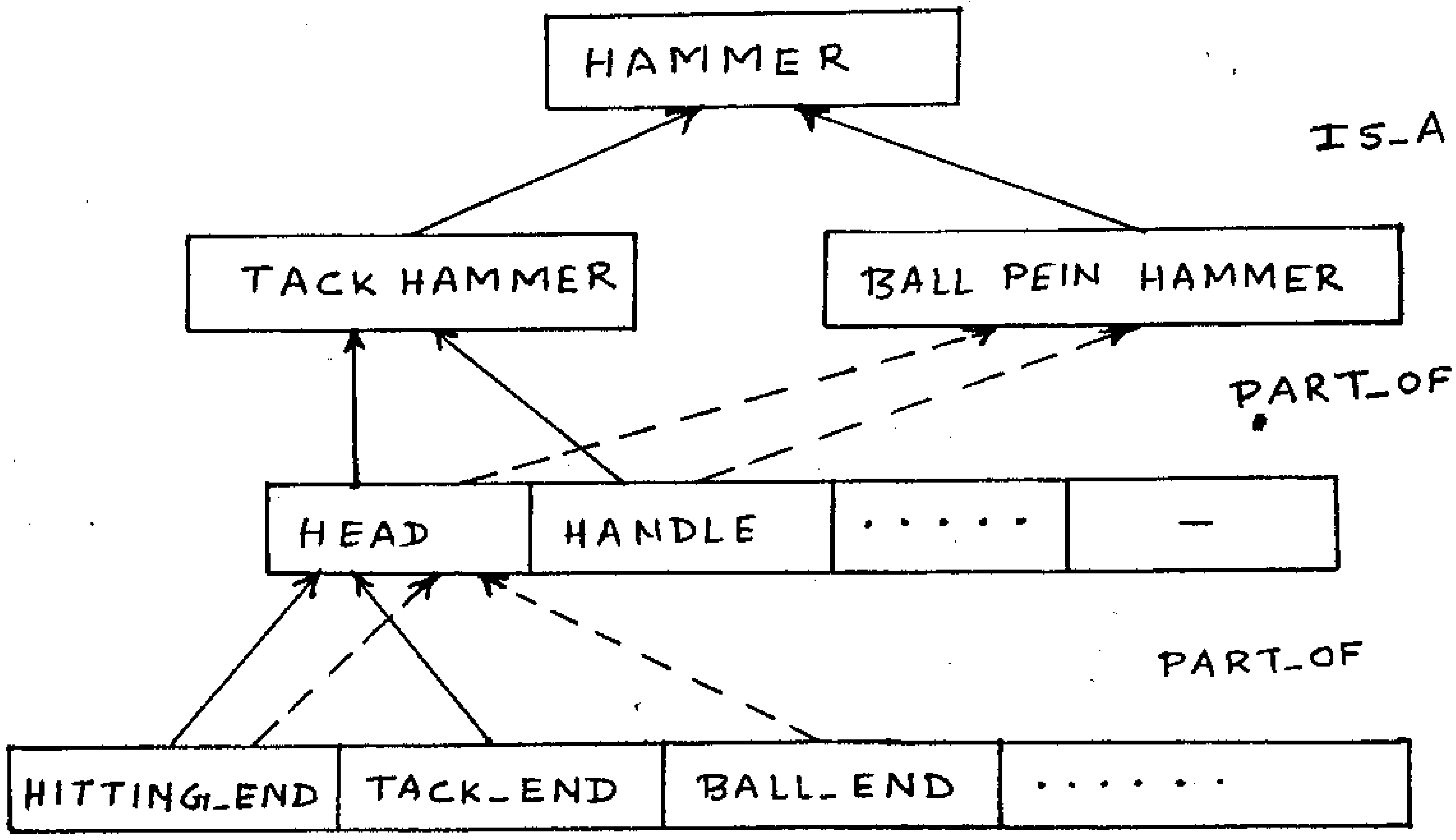
end

RESULTING GRAPH AT LEVEL 4 (DECOMPOSITION OF 1)

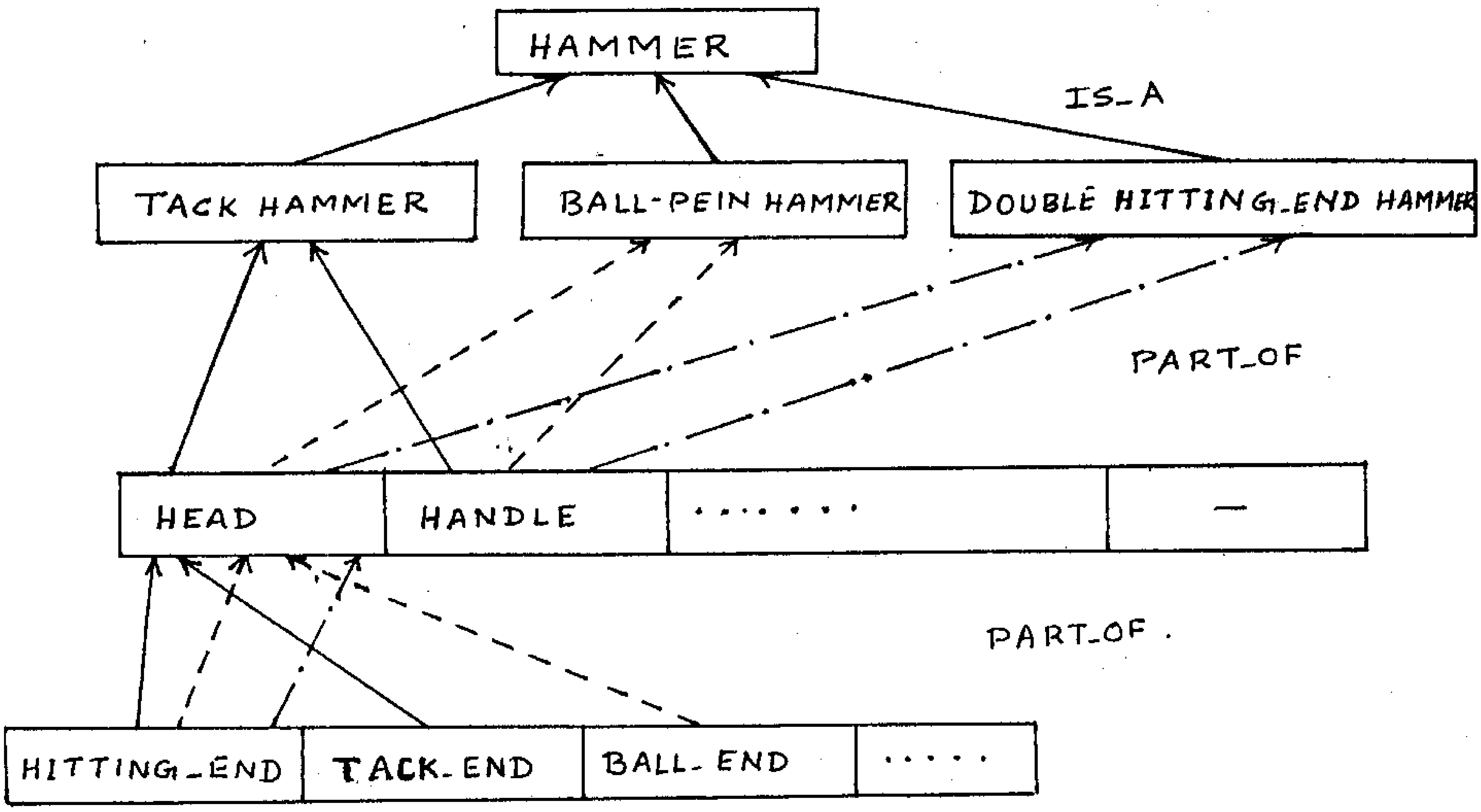
No.	Label	Source	Mult.#	Color grouping	Destination	Mult.#	Color grouping	Participant color
* 0.	IS_ADJ_TO	*1	1	3,2,1;	*3	1	3,2,1;	3, 2, 1,
* 1.	IS_INSIDE	*2	1	3,2,1;	*3	1	3,2,1;	3, 2, 1,

end

Table 3.

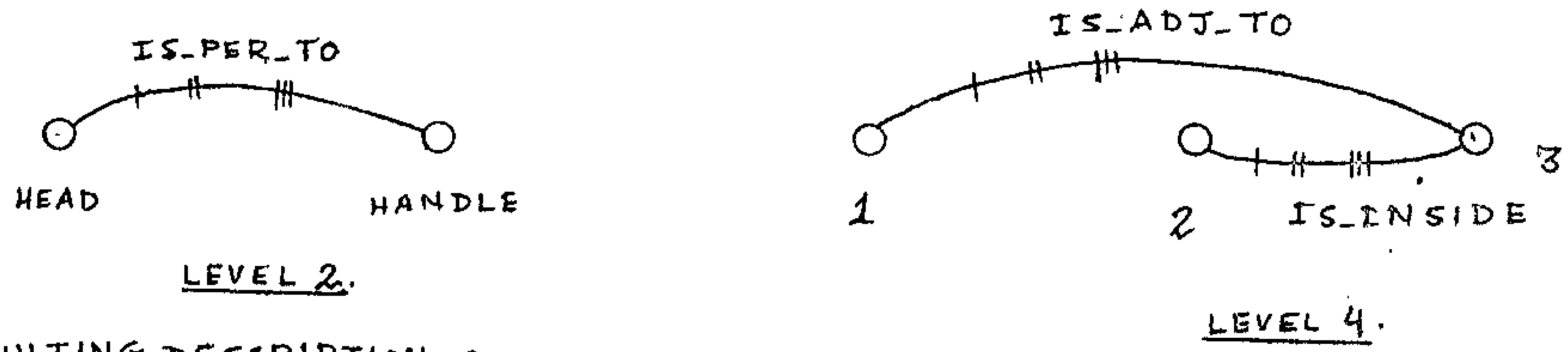


(a)

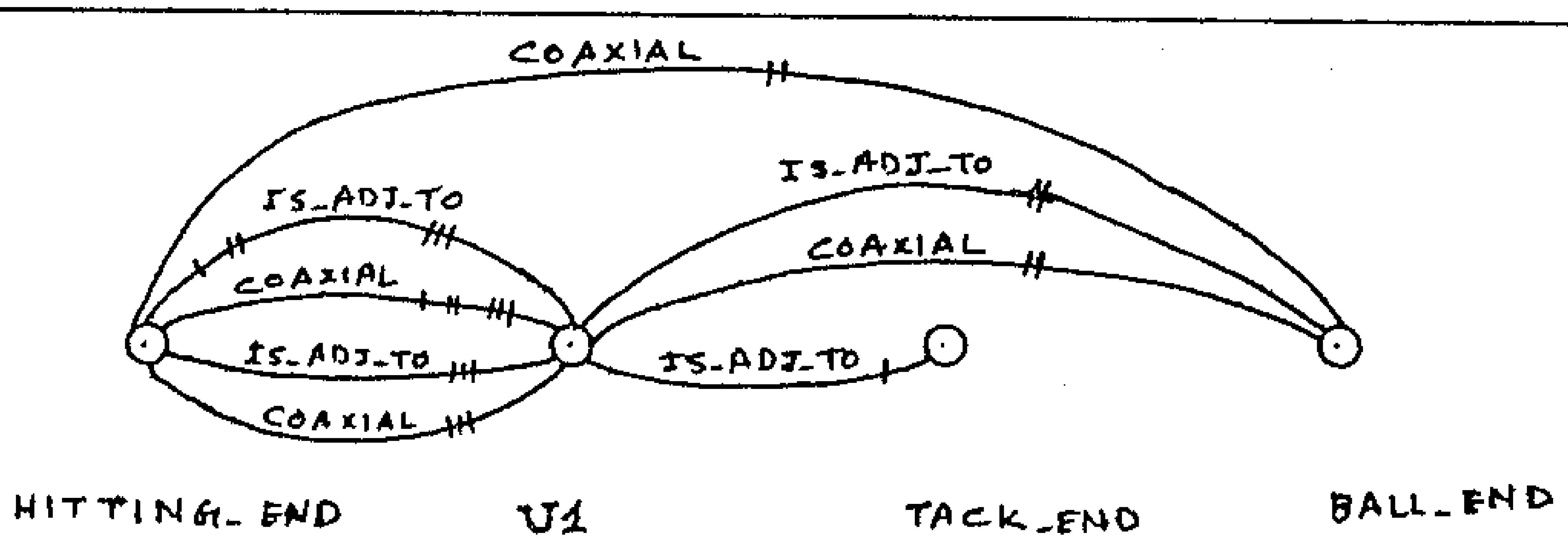


(b)

Fig 6. DEVELOPMENT OF NAME PLANE
(a) AFTER INSERTION OF BALL PEIN HAMMER INTO Fig 5a.
(b) AFTER INSERTION OF DOUBLE HITTING_END HAMMER INTO 6(a)

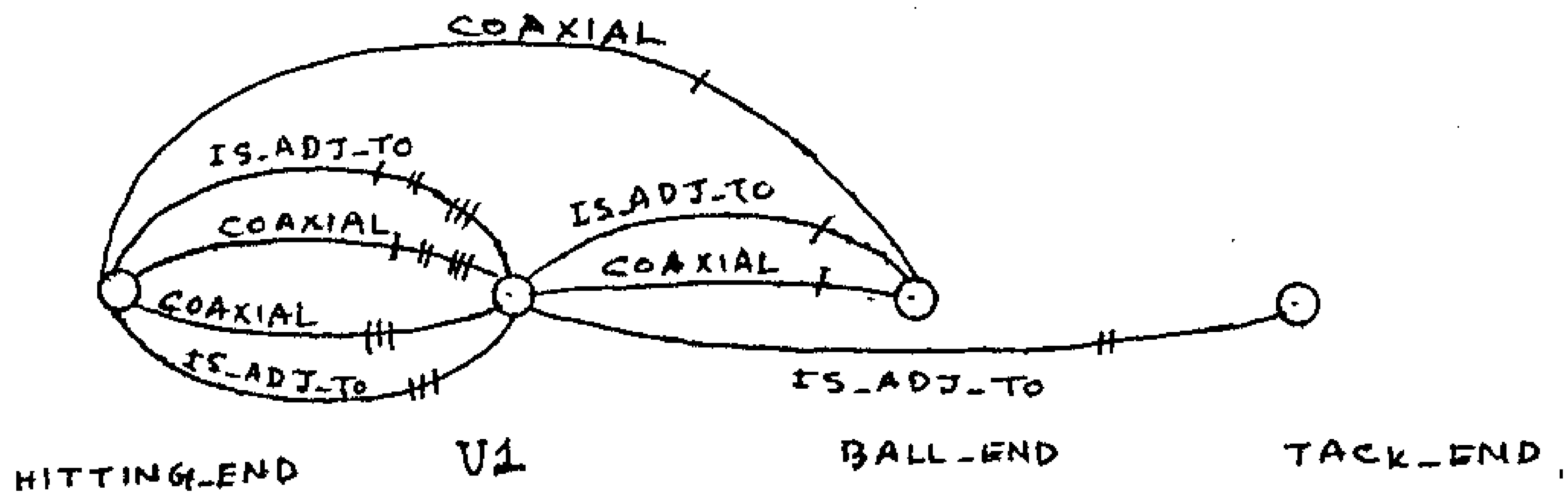


7(a) RESULTING DESCRIPTION GRAPH AT LEVEL 2 & 4.
(INDIPENDENT OF ORDERING)



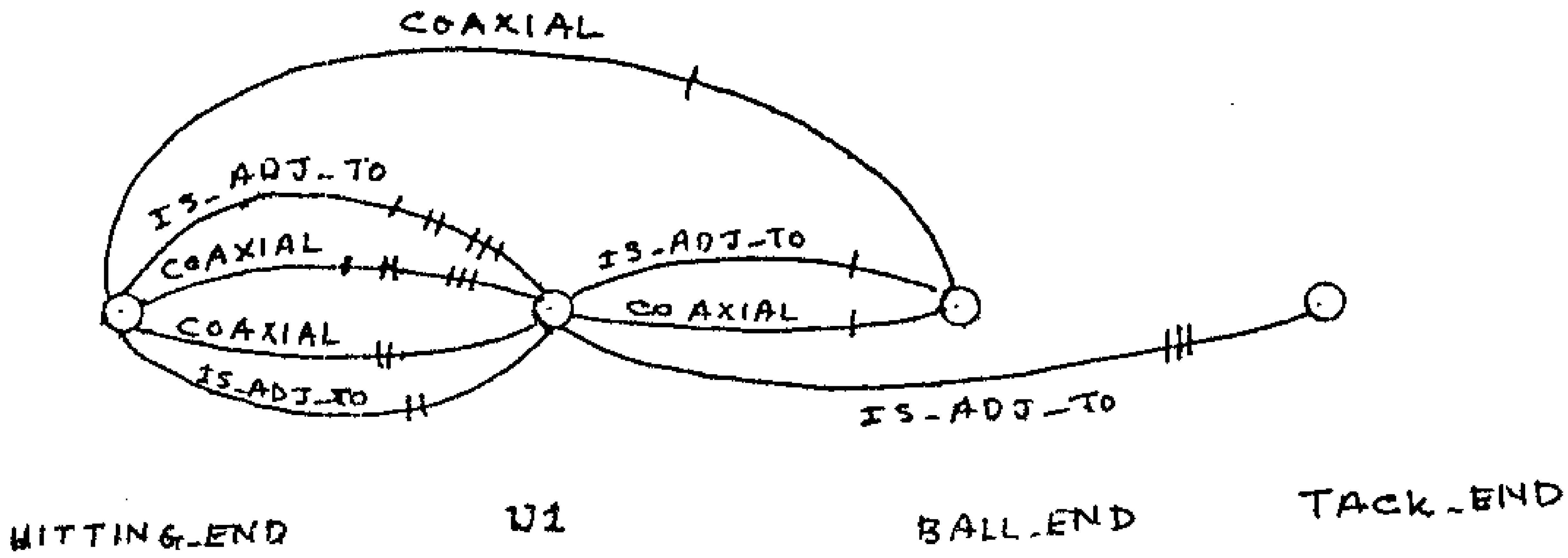
ORDERING : TACK HAMMER , BALL_PEIN HAMMER , DOUBLE HITTING_END HAM.

(b)



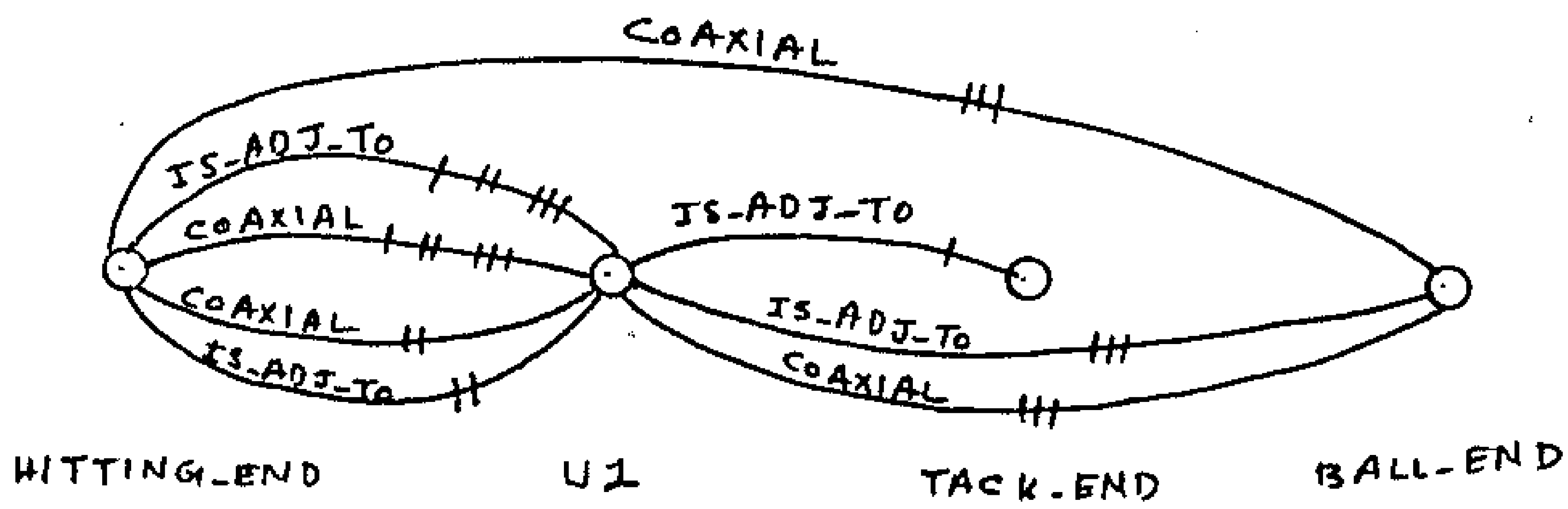
ORDERING : BALL_PEIN HAM. , TACK HAM. , DOUBLE HITTING_EN HAM.

(c)



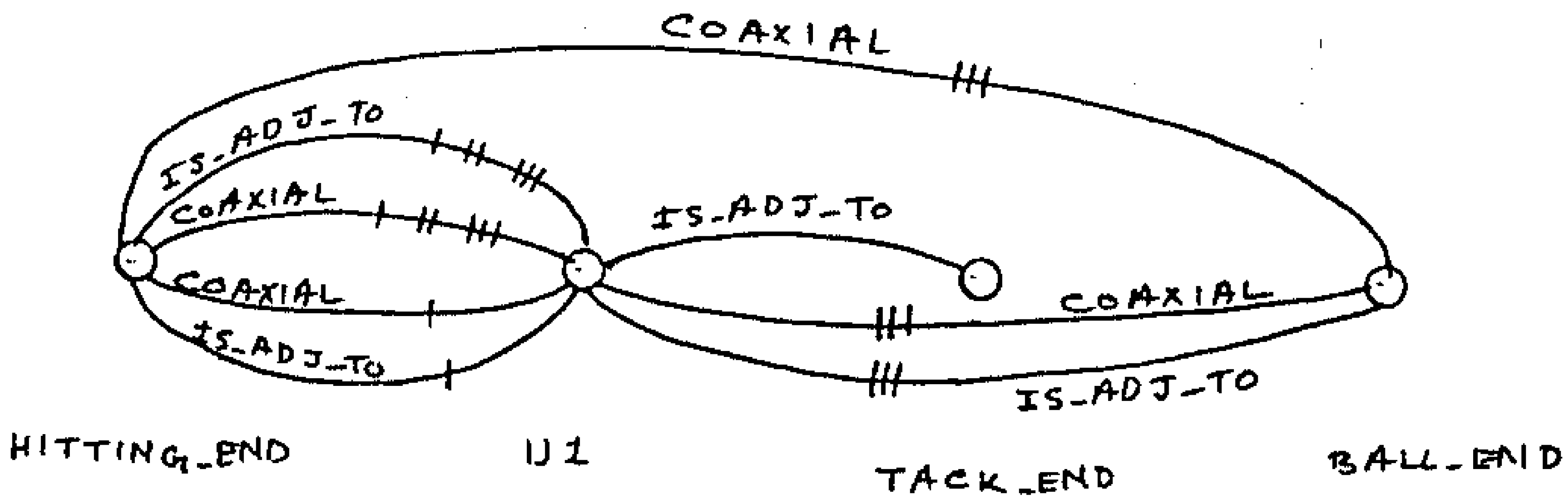
ORDERING : BALL_PEIN HAM., DOUBLE HITTING_END HAM. , TACK HAM.

(d)



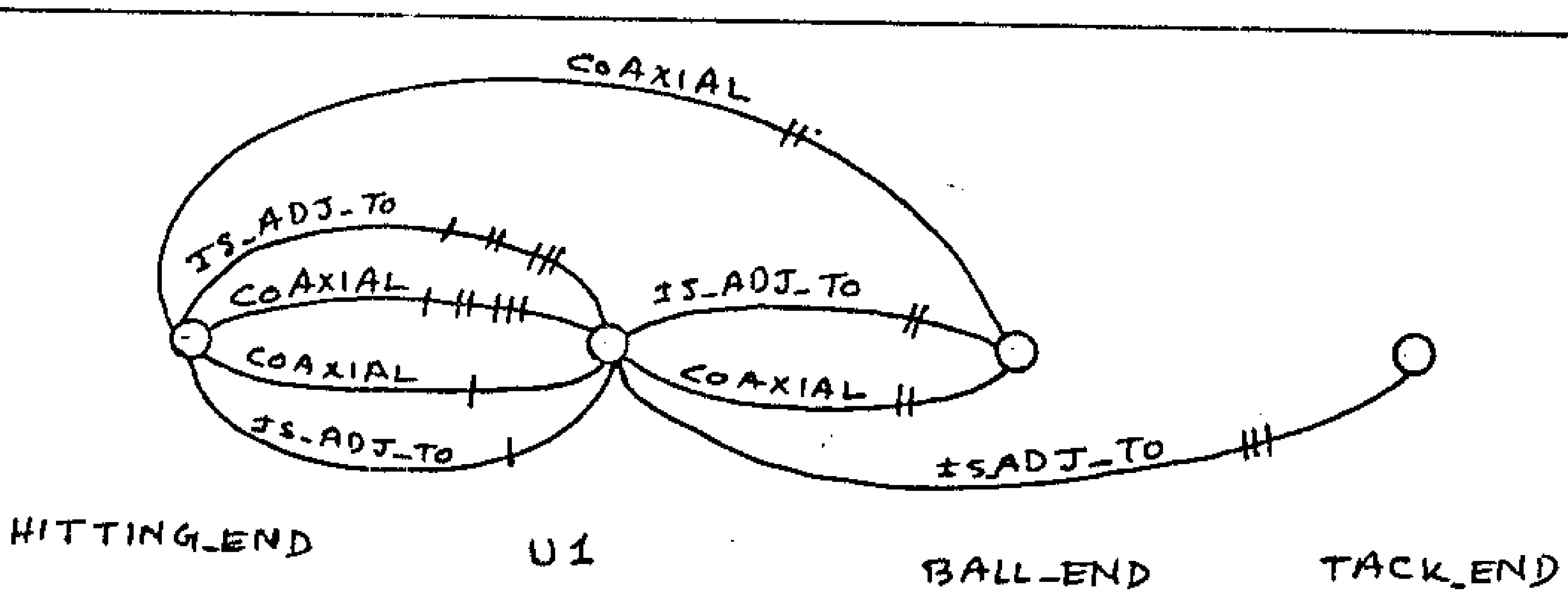
ORDERING : TACK HAM., DOUBLE HITTING_END HAM. , BALL_PEIN HAM.

(e)



ORDERING : DOUBLE HITTING_END HAM., TACK HAM., BALL_PEIN HAM.

(f)



ORDERING : DOUBLE HITTING_END HAM., BALL_PEIN HAM., TACK_HAM.

(g)

Fig 7. (b) – (g) DESCRIPTION GRAPH AT LEVEL 3 FOR DIFFERENT INPUT ORDERING.

(I → COLOR_NO 1
II → COLOR_NO 2
III → COLOR_NO 3)

Section 5.

CONCLUSION

In the present work feature extraction part ,is overlooked. In fact, from the practical point of view, this preprocessing part is very much important. The system can be made more efficient and intelligent by introducing a feature extraction module along with a shared file for communication. Also feature selection is to be done carefully in order to avoid wrong recognition of an object.