# A Scheme for Single Instance Representation in Hierarchical Assembly Graphs

Ari Rappoport

Institute of Computer Science, The Hebrew University, Jerusalem 91904, Israel.
http://www.cs.huji.ac.il/∼arir. `arir@cs.huji.ac.il`.

**Abstract:** The Hierarchical Assembly Graph (HAG) is a common representation for geometric models. The HAG is a directed acyclic graph. Nodes in the graph represent objects, and arcs denote the sub-part relation between objects. Affine transformations and other instantiation parameters are attached to the arcs. An *instance* of an object in a HAG is defined as a *path* ending at its node. Information common to all instances whose paths end at a given node can be attached to this node. Data associated with a *single* instance cannot be attached to any single node or arc in the graph. Such private data can be stored in an external list, hash table, or a partial expansion of the graph into a tree, but all of these schemes have severe drawbacks in terms of storage, access efficiency, or update efficiency.

In this paper we present a scheme for representing single instances in the assembly graph itself, by identifying an instance with the last node in its path when the only way of reaching the last node is through a unique path starting at the first node of the path. We give an algorithm for *singling* an instance in the graph, i.e. transforming the graph into an equivalent one in which the instance can be identified with a node. We also show how to undo an instance's singling when its private data is no longer needed.

**Keywords:** Assembly, Hierarchical Assembly Graph (HAG), geometric data structures, single instance representation, singling algorithm.

**Publication:** *IFIP Conference on Geometric Modeling in Computer Graphics,* Genova, Italy, June 1993. Published in: Falcidieno, B., Kunii T.L. (eds), Geometric Modeling in Computer Graphics, pp. 213-224, Springer, 1993.
Note: the present copy is an updated version.

## 1 Introduction

One of the most important activities in computer-aided design and computer graphics is the design of geometric models [Hoffmann89, Mäntylä88]. The preferred way of designing a model is hierarchically, by composing objects or parts into more complex objects. A common method of representing such a model is by a hierarchical assembly graph (HAG), a directed acyclic graph in which nodes denote objects and arcs denote the sub-part relation between objects. Geometric and other parameters related to the model can be attached to the nodes or the arcs. The most common example is attaching affine transformations to arcs to denote relative placement and scale of part and sub-part [Braid78].

An important observation regarding the HAG is that internal nodes do not represent *instances* of objects in the final model, but 'generic objects'. The generic object appears in as many instances as there are *paths* leading to it from the root of the graph. The HAG has two notable advantages: space efficiency – information common to all instances generated from the same generic object (including their sub-graphs) is stored only once; and fast update – modification of parameters or information in the generic object is instantaneously reflected in all its instances in the graph.

In many cases it is desired to attach data to a single instance. For example, saying that the color of one chair in a meeting room is different from the default color of the other chairs; or that a specific screw in a machine has a unique mark on top. In this paper we call this type of data *private instance data*, and assume that there is no special structure imposed on it, i.e., private data of one instance is independent of private data that may be attached to other instances. Note that since our graph can be instantiated as a sub-part in another graph, we actually want to associate private data with a sub-path in this other graph (a *sub-instance*).

An instance is specified by a path in the graph, therefore private data cannot be attached to a single node or arc. There are some simple methods for single instance representation, which include an external list or hash table, an expansion of the graph into a tree, and storage of a partial expanded tree having only paths leading to instances with private data. Each of these schemes has disadvantages in terms of storage, access efficiency, or update efficiency, to be described later.

We are not aware of substantial previous work on the issue of single instance representation or even on the representation of assemblies. There is a rich literature on boundary representations (see the textbooks [Hoffmann89, Mäntylä88]) and some work on hierarchical boundary models, e.g. [Floriani88]. Braid [Braid78] and Lee and Gossard [Lee85] describe assembly data structures which are essentially hierarchical assembly graphs. A more complex assembly structure, including symbolic repetitions and recursions, is described in [Emmerik93]. A modeling system using sequences of parameterized transformations is described [Rossignac89], and a method for interactive editing of a node's affine transformation is detailed in [Rossignac90]. None of these papers deals with the general problem of associating private data to single instances in geometric hierarchies. Requicha and Chan [Requicha86] briefly discuss the fact that single instances correspond to paths, in the context of representing features and tolerances in CSG. Rossignac [Rossignac86] presents a technique for storing, at any node, lists to sub-node instances, using a relative path. These references are used to override the inherited attributes for that instance during evaluation. However, usage of private instance data is done by expanding the graph into a tree.

The single instance representation issue is extremely practical, and was probably solved ad-hoc in many systems. The lack of literature may be attributed to the existence of seemingly simple and obvious solutions. However, the issue is important enough to justify a separate discussion, and its elegant and efficient solution is not as simple as first imagined.

This paper has two main contributions. First, we discuss the single instance representation issue in a general manner, defining the problem and the requirements from a solution. We describe numerous obvious solutions and show that they are not efficient in terms of time and space.

Second, we present a scheme for representing single instances in the assembly graph itself, by identifying an instance with the last node in its path when the only way of reaching it is through a unique path from the first node. We give an algorithm for *singling* instances in the graph, i.e., transforming the graph into an equivalent one in which the instance can be identified with a node. We also show how to undo an instance's singling when its private data is no longer needed. The elegance of our scheme lies in that it enables private data to be stored uniformly within the graph itself, in a similar way to storage of common data and transparently to algorithms manipulating the graph.

Section 2 defines the problem and discusses the advantages and disadvantages of the simple solutions. Section 3 presents the singling algorithm and some of its properties, and also shows how to undo the effects of the algorithm in order to delete private instance data once it is not needed.

## 2 Instances in Hierarchical Assembly Graphs

In this section we motivate and define the problem of representing single instances in hierarchical assembly graphs. We define the terms strict instance and sub-instance, discuss simple solutions to the problem and show that they have severe disadvantages.

### 2.1 The Hierarchical Assembly Graph (HAG)

A geometric model is best designed hierarchically, by composition of simple objects into more complex ones. The natural way of representing such a model is by a directed acyclic graph (DAG) [1]. A node in the graph represents a *generic object*. We denote objects and nodes by capital letters ($A, B, N$), where 'object $A$' means the sub-graph rooted at node $A$. An arc in the graph from node $A$ to node $B$ means that object $B$ is one of the objects used in defining object $A$. We say that the meaning of the arc is an *instantiation* of the generic object $B$; we also refer to $B$ as a *sub-object* of $A$ and to $A$ as a *parent* of $B$. We denote arcs by small letters ($e_i, e_k$). Note that it is a mistake to denote an arc by the pair of nodes it connects, since there may be several arcs connecting the same two nodes; an object can utilize another object more than once.

Figure 1(a) gives a textual specification of a simple HAG, in the notation described in [Emmerik93]. The same HAG is visualized in Figure 1(b); bold, hatched arcs denote several arcs, numbered in the range shown to their right. Figure 1(c) shows a possible object represented by the HAG.

[1] By requiring that the graph be acyclic we rule out using it for representing fractal-like objects. This is not a practical limitation. See [Emmerik93] for a description of a system allowing cyclic graphs.

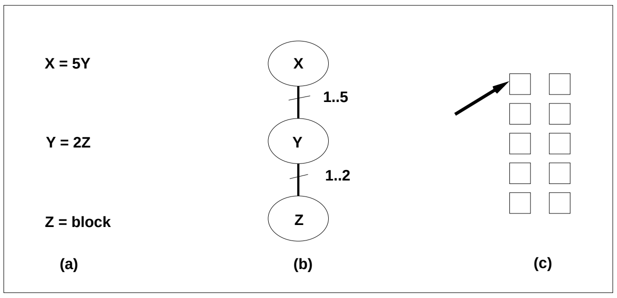

Figure 1: An example of a HAG.

Various parameters of an instantiation are attached to the corresponding arc. One such common parameter is an affine transformation expressing the placement and scale of a sub-object relative to those of its parent. There may be other parameters, for example, if the sub-object is an object parameterized by dimensions then an instantiation can supply the desired dimensions.

As an illustrative example, Figure 2 gives pseudo-code for displaying a model represented in a HAG, assuming: (1) all children of a node are combined with the set union operator, (2) every node has a color attached to it which is inherited by its children, (3) there are display functions available for the geometric primitives in the leaves of the graph.

```
Display (Node $N$) {
   UpdateCurrentColor (Color($N$))
   if $N$ is a leaf
      DisplayPrimitive (Geometry($N$))
   else
      for each arc $e \in OutArcs(N)$ {
         PushGraphicsState()
         MultCurrentTransformation (Transformation($e$))
         Display (DestinationNode ($e$))
         PopGraphicsState()
      }
}
```

Figure 2: A procedure for displaying a model represented by a hierarchical assembly graph.

## 2.2 Instances

The HAG is a graph and not a tree since one generic object may be instantiated more than once, by different objects or even by the same object. For example, in mechanical engineering there is a large number of standard parts which are commonly utilized in many of the components of a machine. In interior design, the same chair, lamp, or tile can be used many times in a building.

We define a *strict instance* of an object $A$ in the graph as a path in the graph starting from the root and ending in $A$. When the path can start in any node we say that the path is a *sub-instance*, because it corresponds to a strict instance in the sub-graph rooted at the path's first node. For simplicity, we will refer to both types as an *instance* and use sub- or strict-instance only when the differentiation is needed.

An instance $I$ is denoted by the list of the arcs on its path: $I = (e_1, ..., e_k)$. Note again that it is wrong to denote a path by a list of its nodes since this creates an ambiguity when there is more than a single arc connecting the same two nodes. Note also that an instance's path does not have to end in a leaf.

We say that an instance *contains* another if its path contains the other instance's path. Instances *overlap* if their paths have common arcs.

## 2.3 Single Instance Representation

Information common to all instances of a node is attached to the node. This information may include the object's basic geometry, default color and material, and so on. It may be desired to associate *private data* to an instance. As examples, pin number 1 in a VLSI chip should be marked by a slight change in geometry; In Figure 1 the block pointed to by the arrow needs to be drawn in a different color. We call the operation of associating private data to an instance $I$ a *singling* of $I$.

Private instance data cannot be attached to any specific node or arc in the graph since it is associated with a whole path in the graph. A scheme for representing single instances is needed. Note that private instance data should not be lost when the object is instantiated in another object; this is the whole point in hierarchical design. Hence, a requirement from such a scheme is that it be capable of representing sub-instances (corresponding to partial paths), not only strict instances (corresponding to paths from the root).

On the other hand, there is no reason why a single instance representation scheme should be required to represent two overlapping or containing instances. From the point of view of the design process, it is meaningless to associate different private data with two such instances; it only matters which object is instantiated and through which path from the root.

An interesting issue is the way in which instances are specified. In an interactive system, the user is obviously not expected to type in whole paths in the graph. Instead, he/she can graphically select one strict instance and be given the power to step up and down its path to narrow or widen it. An instance may also be the result of querying. For example, instances located in a specified area of space, touching a specified object, or visible from a certain location. Instance specification is an orthogonal issue to the instance representation issue discussed in this paper.

## 2.4 Possible Solutions: List, Hash Table, Partial Tree and Graph

A very simple scheme for single instance representation is to expand the graph into a tree, in which case private data can be attached to any node because there is only one way of reaching a node. However, this method loses the two main advantages of the graph. Storage efficiency is lost because common instance information will be duplicated. For large models this becomes prohibitive. Update efficiency is damaged because a modification of a generic object is no longer automatically reflected in all of its instances, and requires traversing the tree and performing the modification on every duplicated node.

A second scheme for single instance representation is to store the instances in a separate, external structure, in which a single entity corresponds to a path in the assembly graph. This scheme has the appealing interpretation that the graph stores the common instance information and the other structure stores the differing information.

The external structure can be a simple list whose nodes correspond to paths in the graph. This scheme requires a search in the list each time an instance is visited in the graph, in order to determine whether it has an associated private data. A hash table can be used instead of a list to make the search more efficient, but there are problems in designing efficient hash functions, especially that a key here is of varying length. Another alternative, an array indexed by an instance's serial number, consumes too much space and creates consistency problems when the numbers change as a result of a change in the HAG.

It is possible to combine both schemes by using a *partial tree*. A partial tree is an expansion of the graph into a tree having only the paths leading to instances with private data. Figure 3 shows a HAG (a) and a partial tree singling the sub-instance (5) (b). Arcs 6 and 7, which are not contained in any path leading to sub-instance (5), do not appear in the partial tree. A traversal of the graph is accompanied by a coordinated, synchronized traversal of the tree to identify the existence of private data.

The partial tree scheme is indeed attractive for representing strict instances, but not for sub-instances. Suppose that an object $B$ with private instance data is used a large number of times in an object $A$, i.e., it appears in many paths in the graph of $A$. The partial tree will duplicate all of the instances of object $B$, but all the duplicates will be identical (Figure 3(b)). We see that the partial tree has the same disadvantages as the fully expanded tree in terms of storage and update efficiency.

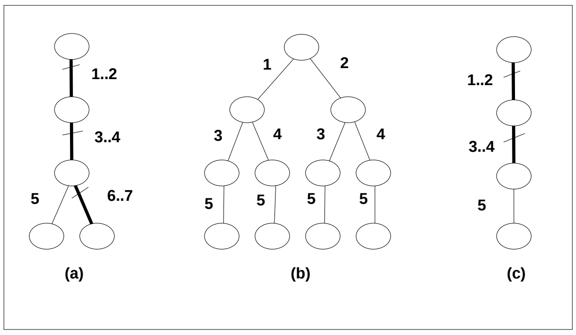

Figure 3: (a) A HAG, (b) a partial tree singling sub-instance $(5)$, (c) a partial graph singling the same instances.

A solution may be to store a *partial graph* instead of a partial tree. In a partial graph, instances are represented by a partial tree rooted at the first node on the instance's path, and all arcs which do not lead to the first node are removed from the graph. Figure 3(c) shows a partial graph singling instance (5); arcs 6 and 7 do not appear, while parts leading to the first node of instance (5), the third node, are stored as a graph. This scheme presents a problem when an instance whose path contains the path of a singled instance is to be singled, with different data (for example, instance $(1, 3, 5)$ in Figure 3(c)). An algorithm is required for singling instances in partial graphs.

The singling algorithm presented in the next section singles instances in general directed acyclic graphs. As such, it can be used on the partial graph too. However, it can be used directly on the original graph, obviating the need for the partial graph.

## 3 A Scheme for Instance Singling

In this section we present a scheme for singling instances in directed acyclic graphs. The scheme stores instances as equal-status nodes in the graph. We give an algorithm for instance singling and show how to undo its effects.

### 3.1 General Idea

The main observation on which the scheme is based is that a *path* (hence an instance) can be *identified with its last node*, if and only if the only way of reaching the last node of the path is through a unique path starting from the first node. Another way of phrasing this condition is that there is no *upward ambiguity* when going from the last node up to the first node of the path. Note that the condition has two essential parts: that the only way of reaching the last node of the path is from the first node, and that there is only one such way. When this condition is fulfilled, the instance's private data can be associated with this node or with the arc directly leading to it (Figure 4).

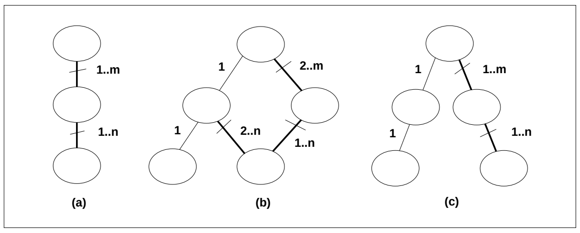

Figure 4: (a) An example of a HAG having three nodes, $m + n$ arcs, $n \times m$ strict instances, and $n \times m + m + n$ instances. No single node or arc can be identified with an instance. (b) The output of the singling algorithm when singling the path $(1, 1)$. (c) An erroneous answer in which the path is duplicated.

The graph is transformed to achieve this situation. The transformation process should be careful to preserve all original paths in the graphs, not to duplicate paths, and not to add new paths. The graph in Figure 4(a) can be transformed as in (c) to single the path $(1, 1)$. However, the old path still exists in the graph. A correct solution is shown in (b), where all and only original paths are present and no path is duplicated.

Our scheme has the advantage that it is completely transparent to graph traversal algorithms; they operate as they ordinarily would, not knowing or caring whether the data they find associated with a node is private or not. There is no need to search for instances in external structures or to coordinate the traversal with one on a partial tree or a partial graph. For example, in Figure 2, a node's color is used to update the current display color. The procedure is used with no change when the color belongs to a single instance.

Every operation that could be performed on the original graph can also be performed on the transformed graph. In particular, a different instance can now be singled, so that a singled graph represents the whole assembly, including private data of many single instances.

## 3.2 A Singling Algorithm

Denote the path of the instance $I$ to be singled by $I = (e_j, ..., e_k), 0 < j \leq k$ and the nodes it passes through by $(N_{j-1}, ..., N_k)$. $InArcs(N), OutArcs(N)$ denote the sets of in-coming and out-going arcs of node $N$, respectively. $P_i$ (for 'parents') denotes the arcs in $InArcs(N_i)$ which arrive from nodes other than $N_{i-1}$. $E_i$ denotes the set of arcs in $OutArcs(N_i)$ connecting $N_{i-1}$ to $N_i$, other than $e_i$. Pseudo-C code of the algorithm is shown in Figure 5, and Figure 6 shows the main transformation performed.

The algorithm performs $k - j + 1$ stages, such that stage $i$ deals with $N_i, i = j..k$. Note that the order is top-down. At stage $i$, if $e_i$ is the only in-coming arc to node $N_i$ then $N_i$ is marked as singled. If $e_i$ is not the only in-coming arc to $N_i$ we invoke `SingleArcNode`, whose function is to ensure upward disambiguity of its input node $N_i$ with respect to its input arc $e_i$.

First, `SingleArcNode` checks whether $N_i$ already has an unsingled brother $N_i^U$. If it exists, all the offending in-coming arcs of $N_i$ are moved to this brother and $N_i$ is marked as singled.

If an unsingled brother does not exist, $N_i$ is marked as unsingled renamed $N_i^U$, and a new brother node $N_i^I$ is created ($I$ stands for 'instance' and $U$ for 'unsingled'.) All the offending in-coming arcs of $N_i^I$ are moved to $N_i^U$, as done in the previous case.

$N_i^I$ receives a copy of the out-going arcs of $N_i^U$. This may violate upward disambiguity of child nodes of $N_i^I$, since each in-coming arc of such a node becomes two arcs (recall that upward disambiguity is the condition necessary for identifying the final node $N_k^I$ with the singled instance.) The function `SingleArcNode` is now called recursively for each child node for which upward disambiguity is required.

Note that only the arcs $OutArcs(N_i)$ are duplicated, not the whole sub-graph descending from them; both sets of arcs denoted by $OutArcs(N_i)$ in Figure 6(b) lead to the same nodes.

To prove that the algorithm is correct, we have to prove (1) instance $I$ can be identified with node $N_k^I$; (2) the new graph is equivalent to the original one in terms of their instances.

Define two paths to be *equivalent* if their respective arcs are either identical or copies of each other and their respective nodes are either identical or brothers. Define

```
Node  SinglePath (Hag $H$, Path $I$) {
   denote the arcs on $I$ by $e_j, ..., e_k$
   denote the nodes on $I$ by $N_{j-1}, ..., N_k$

   for $i = j$ to $k$
      $answer$ = SingleArcNode ($I$, $e_i$, $N_i$)
   return $answer$
}

Node  SingleArcNode (Path $I$, Arc $e_i$, Node $N_i$) {
   if there is only one arc in $InArcs(N_i)$
      mark $N_i$ as singled
      return $N_i$
   if $N_i$ has an unsingled brother $N_I^U$
      for each arc $e \in InArcs(N_i)$
         if $e \neq e_i$
            remove $e$ from $InArcs(N_i)$ and append it to $InArcs(N_I^U)$
      mark $N_i$ as singled
      return $N_i$

   mark $N_i$ as unsingled $N_i^U$
   create a new node $N_i^I$, mark it as singled,
      and append it as a brother to $N_i^U$
   remove $e_i$ from $InArcs(N_i^U)$ and append it to $InArcs(N_I^I)$
   $OutArcs(N_i^I)$ = $OutArcs(N_i^U)$
   for every $e \in OutArcs(N_i^I)$
      if $Child(e)$ is marked as singled
         SingleArcNode ($I$, $e$, $Child(e)$)
   return $N_i^I$
}
```

Figure 5: Pseudo-C code for the singling algorithm.

two graphs to be equivalent if there is a bijection between their paths such that every corresponding pair of paths are equivalent. Define two nodes to be equivalent if the sub-graphs rooted at them are equivalent.

It is easy to see (e.g. by exhaustive enumeration of the paths that pass through $N_{i-1}^I$ and $N_i$) that $N_{i-1}^I$ of Figure 6(a) is equivalent to this of Figure 6(b), and that $N_i^I$ and $N_i^U$ are equivalent. Hence the graph transformation performed by `SingleArcNode` yields an equivalent graph, which proves (2). To prove (1), note the following invariant:

- After stage $i$, for every $j \leq h \leq i$ there is a singled node $N_h$ equivalent to $N_i$ with no upward ambiguity.

Consequently, after stage $i$ there exist two nodes $N_i^I$ and $N_{j-1}$ connected by a unique singled path $N_{j-1}, ..., N_i^I$. This path can be denoted by the names of its nodes because there is no ambiguity – only a single arc connects each pair of nodes. After the last stage, in which $i = k$, we can safely identify the original path $(e_j, ..., e_k)$ with node

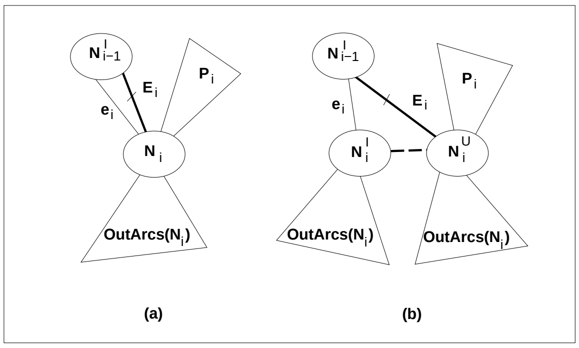

Figure 6: The situation in (a) is replaced by that in (b). Node $N_i$ is split into two nodes, $N_i^I$, which lies on the path to the singled node, and $N_i^U$, for all the other paths.

$N_k^I$.

The algorithm is optimal in terms of the number of nodes in the transformed graph, since a new node is created if and only if a node with upward ambiguity must be singled.

Finally, note that consecutive singling of each and every strict instance in the graph will result in an expansion of the graph into a tree, since at the end no node will have more than a single in-coming arc.

## 3.3 Deleting Singled Instances

In a dynamic or interactive environment it is necessary to provide a way of deleting singled instances once their private data is no longer needed, in order to optimize the graph's storage and the efficiency of graph traversal algorithms. Deleting singled instances is done by reversing the process shown in Figure 6; it requires a small modification to the singling algorithm, arc counting.

Arc counting means keeping a counter on every arc $e_i$ participating in a singled path. The counter counts the number of singled paths using this arc. Arc counting is necessary since singling of some instance $J$ may utilize arcs created for singling of a previous instance $I$. If those arcs were to be joined automatically when deleting instance $I$, instance $J$ may not be singled anymore.

In Figure 7, singling instance $(1)$ in the simple graph (a) results in the graph (b). Singling instance $(2)$ does not change the graph. Suppose it is now desired to cancel the singling of instance $(1)$. If this is done by joining the two nodes $B_1^I$ and $B_2^I$, instance $(2)$ is not singled anymore. A counter on arc 2 will show that it is used for singling some instance hence the node it leads to ($B_2^I$) cannot be joined to another.

When deleting singled instances, the process of joining two brother nodes together back into one node is only done when all counters in all in-coming arcs have a value of 1. Otherwise, no joining is done and the only operation done in stage $i$ is decrementing the counter on arc $e_i$.

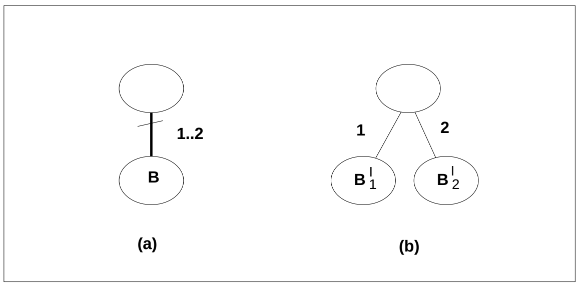

Figure 7: An example for the necessity of arc counters.

## 4 Discussion

We have motivated and defined the problem of associating private data with single instances in hierarchical assembly graphs (HAGs). An algorithm for singling instances was presented. The algorithm is suited for singling instances corresponding to arbitrary paths in the graph, not necessarily paths starting at the root (strict instances) or ending at a leaf.

The singling algorithm can be used in two ways to solve the above problem. First, it can be used on the original HAG itself. Second, it can be used on a partial graph, as defined in Section 2. The latter option is appealing since the common and private data of instances are clearly separated into two structures. The former is more elegant since the whole singling process is completely transparent to algorithms manipulating the HAG. These operate with no modification since singled nodes are similar in structure and functionality to the other nodes in the graph.

In this paper we have not dealt with updating singled instances after modification of the graph itself (i.e., when adding or deleting nodes or arcs). The arc counters (Section 3) can be easily used to notify the user that an editing operation invalidates an instance's private data; meaningful automatic treatment of this case is left to future reports.

Another topic for future work is how to organize the private data when imposed on it there is a structure different from the structure of the assembly graph (for example, if color inheritance of instances depends upon their location and not upon their sub-part hierarchy). It seems that in this situation an external structure cannot be avoided.

## Acknowledgements

I thank Maarten van Emmerik for an enjoyable and productive collaboration, in which the issue of single instance manipulation first arose. I thank Jarek Rossignac, Eyal Piassetsky and Martti Mäntylä for discussions and correspondence improving my understanding of the issue.

## References

[Braid78] Braid, I.C., On storing and changing shape information, *Computer Graphics*, 12(3):252-256, 1978 (Siggraph '78).

[Emmerik93] Emmerik, M.J.G.M. van, Rappoport, A., Rossignac, J., Simplifying interactive design of solid models: a hypertext approach. *The Visual Computer*, 9:239-254, 1993.

[Floriani88] de Floriani, L., Falcidieno, B., A hierarchical boundary model for solid object representation, *ACM Transactions On Graphics* 7(1):42-60, 1988.

[Hoffmann89] Hoffmann, C., Geometric and Solid Modeling: an Introduction, Morgan Kaufmann, 1989.

[Lee85] Lee, K., Gossard, D.C., A hierarchical data structure for representing assemblies: part 1, *Computer-Aided Design* 17(1):15-19, 1985.

[Mäntylä88] Mäntylä, M., An Introduction to Solid Modeling, Computer Science Press, Maryland, 1988.

[Requicha86] Requicha, A.G., Chan, S.C., Representation of geometric features, tolerances, and attributes in solid modelers based on constructive geometry, *IEEE J. Robotics and Automation*, 2(3):156-166, 1986.

[Rossignac86] Rossignac, J.R., Constraints in constructive solid geometry, *ACM Symposium on Interactive 3D Graphics*, ACM Press, 1986, pp. 93-110.

[Rossignac89] Rossignac J.R., Borrel P., Nackman L.R., Interactive design with sequences of parameterized transformations, in: Intelligent CAD Systems 2: Implementational Issues, Springer-Verlag, pp. 93-125, 1989.

[Rossignac90] Rossignac, J.R., Borrel, P., Kim, J., Mastrogiulio, J., BIERPAC: basic interactive editing for the relative positions of assembly components, IBM Research report, RC 17339 (#76615), October 1990.