Multiple Object Selection in Pattern Hierarchies

Justin Jang and Jarek Rossignac
School of Interactive Computing, Georgia Institute of Technology
justin.jang@gatech.edu, jarek@cc.gatech.edu

Abstract

Hierarchies of patterns of features, of sub-assemblies, or of CSG sub-expressions are used in architectural and mechanical CAD to eliminate laborious repetitions from the design process. Yet, often the placement, shape, or even existence of a selection of the repeated occurrences in the pattern must be adjusted. The specification of a desired selection of occurrences in a hierarchy of patterns is often tedious (involving repetitive steps) or difficult (requiring interaction with an abstract representation of the hierarchy graph). The OCTOR system introduced here addresses these two drawbacks simultaneously, offering an effective and intuitive solution, which requires only two mouse-clicks to specify any one of a wide range of possible selections. It does not require expanding the graph or storing an explicit list of the selected occurrences and is simple to compute. It is hence well suited for a variety of CAD applications, including CSG, feature-based design, assembly mock-up, and animation. We discuss a novel representation of a selection, a technology that makes it possible to use only two mouse-clicks for each selection, and the persistence of these selections when the hierarchy of patterns is edited.

Keywords: CAD, Patterns, Features, Hierarchy, Naming, Persistence, Selection

1. Introduction

Hierarchies of patterns of features, of sub-assemblies, or of CSG sub-expressions are used in architectural and mechanical CAD to eliminate laborious repetitions from the design process. For example, a single CAD model is used for all of the occurrences of a seat in the digital mock-up of an airplane. The corresponding hierarchy of patterns may for instance define a pattern of 42 rows, each being a pattern of 8 seats. A simple translation is used to specify the relative position of each occurrence of seat or row in the parent pattern.

Often the placement, shape, or even existence of a selection of the occurrences in the pattern must be adjusted. For example, the last two seats of each row must be displaced to leave room for the aisle and the last seat of row 12 should be deleted to clear the access to an emergency exit. The specification of the desired selection of occurrences in a hierarchy of patterns is often tedious (involving repetitive steps) or difficult (requiring interaction with an abstract representation of the hierarchy graph).

The most common approaches for multiple object selection (MOS) include serial selection techniques that require the user to select objects one at a time, e.g. the ubiquitous ctrl+click (or shift+click) approach, and parallel selection techniques such as brushes, lassos, and selection shapes. However as Lucas et al. [2005] point out, each has certain limitations, especially in 3D. For instance, multiple objects may be difficult to distinguish, isolate, or even see due to occlusion, rendering size, environment clutter, and other display factors. Requiring the user to adjust the view can be tedious, cumbersome, and even burdensome, especially when the number of objects to select is high, and may still fail to make certain objects accessible. Systems commonly address this issue with an indirect selection technique, that is, by allowing the user to select using an alternate representation such as a model tree or component list. Some systems allow selection by common attribute (e.g. [MSWord] for text) or provide a more general selection query or search (e.g. [AutoCAD] and [Pro/ENGINEER] for geometry and [Miller and Myers 2002] for text). Such indirect selection techniques are useful, but are generally abstract and less intuitive than direct manipulation techniques.

Oh et al. [2006] describe an approach for selecting objects in groups. Their approach relies on dynamically computing a group hierarchy based on the notion of gravitational proximity using heuristics such as contact or intersection and factors such as speed and direction of mouse drag. Their approach does not rely on semantic or user specified information for structure and is appropriate for dynamic environments or situations where flexibility is required. It is less appropriate for rigid and exact specification of selections, particularly when objects or components are frequently or always in contact with or intersecting each other, e.g. when modeling parts, assemblies, and structures, or with csg and feature-based modeling.

The OCTOR system introduced here takes a direct selection approach, yet does not require direct access to more than just a small subset of the objects/occurrences to be selected. The general idea is to have the user directly select a small number of occurrences and let the system guess the rest in a way that is predictable and repeatable. At the same time, it is flexible and interactive, allowing for iterative refinement which can be guided or scaffolded. The aim is not to replace other selection techniques but to give users another option which is more intuitive, efficient, and accurate in certain cases.

On the developer's side, the octor representation provides a compact encoding for multiple occurrence selections and is easy to compute. It does not require expanding the graph or storing an explicit list of the selected occurrences and is simple enough to be extensible and combinable. For instance, multiple octor selections can be combined in a list or with Boolean operations.

The rest of the paper is organized as follows. Section 2 defines our terminology and reviews a simple pattern hierarchy approach. Section 3 defines exceptions, and describes three approaches for obtaining and representing sets of selections. Section 4 introduces the octor representation for selections and section 5 shows how the user can specify them. Section 6 describes an approach to handling transformation exceptions. In section 7 we discuss the persistence of octor selections when the hierarchy is edited. Finally we suggest some applications and extensions in section 8.

2. Hierarchy of patterns

In this section, we introduce our terminology and review a simple approach for designing, representing, and processing hierarchies of patterns.

We will use the term **component** to denote a solid, a CSG primitive, a feature, a sub-assembly, or a CSG expression. For simplicity, one may think of a component as being a shape, such as a chair or table. Each component is defined in a local coordinate system. The size, position, and orientation of such a local coordinate system in a given model is called its **pose**. The pose of one component may be derived from another pose by a **transformation**, which typically is the result of a series of scalings, rotations, and translations and can be represented as a 3×4 matrix. A given component may appear in several places in a model. Each appearance is called an **occurrence** of the component. A **pattern** (Figure 1) is a series of occurrences of the same component, such that the pose of each subsequent occurrence is obtained from the pose of the previous occurrence by the same transformation. The unique component repeated by a pattern is called the **pattern-leader**, the number of occurrences is called the **pattern-count**, and the transformation between successive occurrences is called the **pattern-transform**.

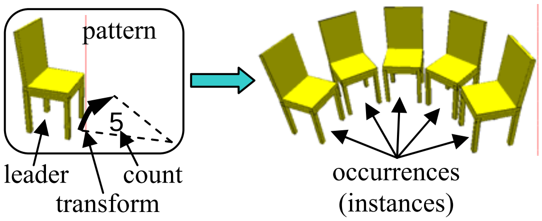

Figure 1. Definition of a pattern, pattern-leader, pattern-count, pattern-transform, and occurrences.

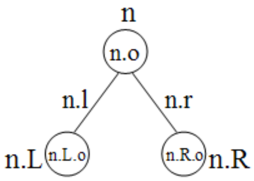

Figure 2. Hierarchy node.

Consider the small assembly (Figure 3 top left) comprising a table T and a chair C transformed by a transformation s_2. We may create a dining set S comprising a pattern of 5 chairs (Figure 3 bottom left) around the table by specifying the pattern-count (5) and the pattern-transform (s_3), which is a rotation around the center of the table. To do so, the designer would for instance type S=T+5C. The resulting model is stored as a graph (Figure 3 right), where each circled node n (Figure 2) indicates a pattern-count, n.o. Its left link n.L leads to the component to be used in the pattern. The associated pattern-transform is accessible as n.l. When n.o=1, the pattern-transform n.l is not used. (For example, the root node in Figure 3 has pattern-count 1, hence s_1 is not used.) The right link n.R may be used to reference additional single-instance or multiple-instance patterns. The associated transform, n.r specifies the pose of the relative transformation of these additional patterns with respect to the previous ones. Then, as proposed in [vanEmmerick 1993], the designer may click on the "=" symbol to select s_1, on the "+" symbol to select s_2, or on the pattern-count ("5") to select s_2, and then proceed to edit the corresponding transform either through direct manipulation of graphically selected occurrences or by providing precise scaling factors, rotation angles, or displacement coordinates.

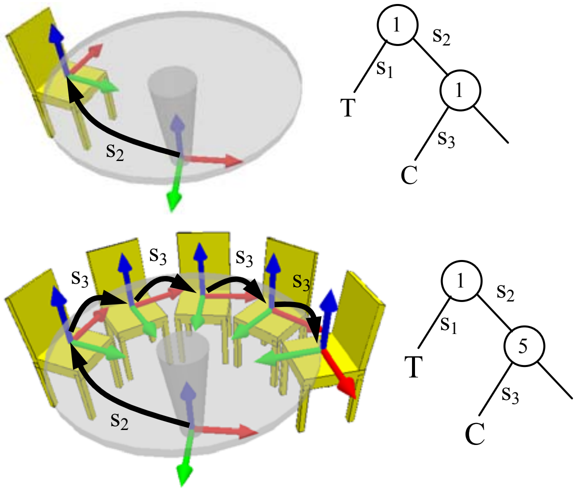

Figure 3. Table T and chair C (top left) with graph representation (top right). T combined with a pattern of chairs (bottom left). A graph representation of the corresponding set, specified as S=T+5C (bottom right). Transformation s_1 is not applied since there is only one occurrence of T. Transformation s_2 is applied to the first occurrence of C. Transformation s_3 is combined repeatedly with s_2 to obtain the transformation of each subsequent occurrence of C. Hence, the five successive occurrences of chair C would be transformed respectively by s_2, $s_3 \bullet s_2$, $s_3 \bullet s_3 \bullet s_2$, $s_3 \bullet s_3 \bullet s_3 \bullet s_2$, and $s_3 \bullet s_3 \bullet s_3 \bullet s_3 \bullet s_2$. The mini axes represent the poses for each occurrence resulting from the corresponding sequence of transformations.

A more complex component (Figure 4 left) could for example be defined by D=T+5C+6C, where d_4 is a translation which defines a second chair behind the first one and where d_5 is a rotation around the center of T, but by a slightly smaller angle. The corresponding graph (Figure 4 right) is not a tree, since two components reference C. Note that the definition of this new model did not require the use of a hierarchy of patterns, but is simply an assembly of two different patterns.

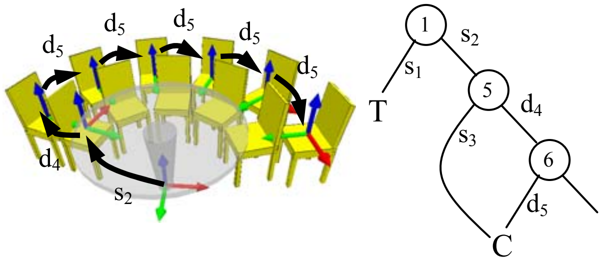

Figure 4. Table T and two patterns of chairs C (left). The poses for some of the occurrences are shown as mini axes. A graph representation of the corresponding set, specified as D=T+5C+6C is shown right.

We will use a hierarchy of patterns to make a row R=4S of 4 dinning sets S and then make a floor F=3R of 3 such rows R, and finally a bar B=2F with two identical floors F. The graph representation of the bar is shown in Figure 5.

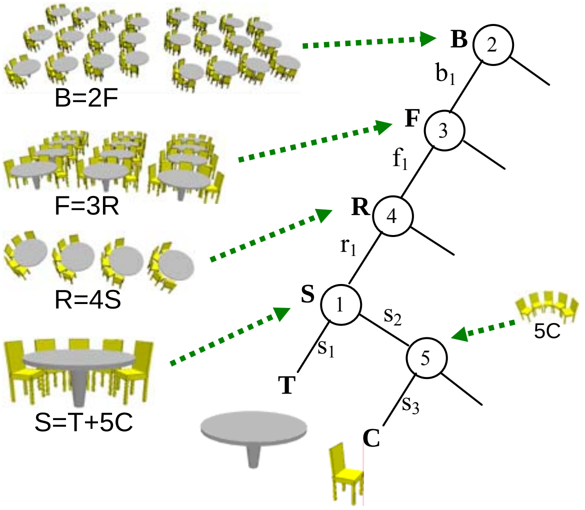

Figure 5. Graph representation of the bar defined as B=2F, F=3R, R=4S, S=T+5C. We show the names of the components and use the corresponding lower-case letters with consecutive subscripts to denote the successive transformations appearing in their definitions.

The designer could edit the hierarchy by altering its text definition (for example to change the number of occurrences in a pattern or to add a second row of chairs behind each table) or any selected transformation.

To render such a model, or to simply access each occurrence and its final pose, we propose to use a simple recursive traversal of the graph. It is illustrated by the procedure eval(n), where n is a node in the graph. (Note that not all nodes represent named components.

For example, S is defined in terms of a node that represents a pattern of 5 chairs but does not have a user-given name.) We start the traversal at the root of the graph, by invoking eval(B).

```
eval(n) {
  if (isPrimitive(n)) process(n);
  else {
    pushMatrix();
    for (int i=1; i<=n.o; i++) {
      eval(n.L);
      applyMatrix(n.l);
    }
    popMatrix();
    pushMatrix();
    applyMatrix(n.r);
    eval(n.R);
    popMatrix();
  }
}
```

The definitions of n.o, n.L, n.l, n.R, and n.r were provided above.

Such hierarchies of patterns may be used to considerably simplify the creation and editing of models of assemblies or of solids constructed through CSG operations. Figure 6 provides examples.

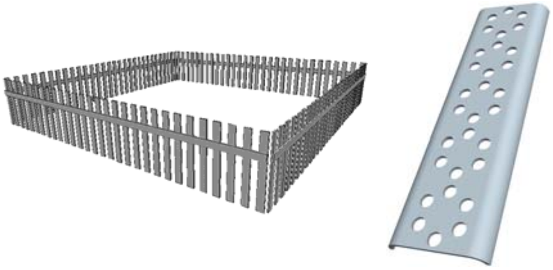

Figure 6. Examples designed using a hierarchy of patterns: A fence (left) defined as a pattern F=4R of 4 rows, each defined as a combination R=30V+H of a pattern of 30 vertical beams and one horizontal and a CSG model of a fuselage plate (right) defined as F=P–5C, a plate from which one has subtracted a pattern of five arrangements C, each defined as a pattern C=6H of 6 holes.

3. Exceptions

Often, the poses of some of the occurrences in a pattern or in a hierarchy of patterns must be adjusted. For example, a pillar may require that we remove the same chair on each floor (Figure 7 left). Or one may wish to move the last chair of each dining set so that it faces the others (Figure 7 right).

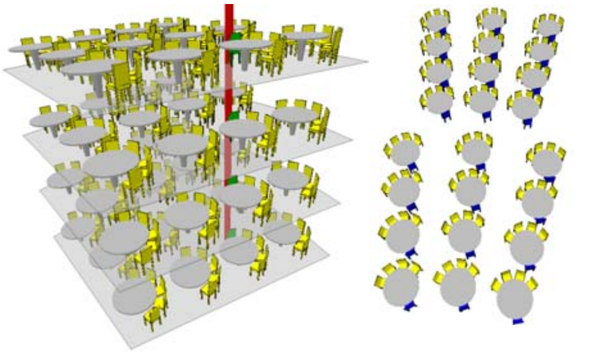

Figure 7. The same chair on each floor (in green) needs to be removed due to a column (in red) (left). The fifth chair (in blue) at each table has been rotated to face the others and tucked under the table (right).

To specify an exception one must indicate which components are to be adjusted and how to adjust them. Thus we define an exception E=(S, T) to be a selection plus an exception treatment. An exception **selection** is the set of components to be modified and an exception **treatment** is the modification information (e.g. displacement, disappearance, color, etc.) to be applied to that set.

3.1. Selections

In this section, we discuss the benefits and drawbacks of three simple techniques for specifying and representing exception selections in patterns of patterns.

Approach 1: The first approach is to expand the graph into a non-binary tree (Figure 8). The child-nodes of nodes with pattern-count larger than one are replicated, replacing the n.L link with n.o such links. This expansion is performed recursively. In our example of a bar, such an expansion would produce a tree with 24 table leaves and 120 chair leaves. The designer would then be able to select individual leaves one by one and adjust their poses or attributes. This approach has the drawback of increasing storage and of not preserving the structure of the pattern hierarchy, which represents the designer's intent [Rossignac et al. 1988] and should be preserved to facilitate further editing. For example, the designer may later decide to add a third floor or to squeeze in more chairs at each table. Even with an approach based on partial graph expansions, managing change can be challenging and maintaining certain selections may still require an external structure [Rappoport 1993]. Hence the remainder of the paper is focused on approaches that do not require such a graph expansion.

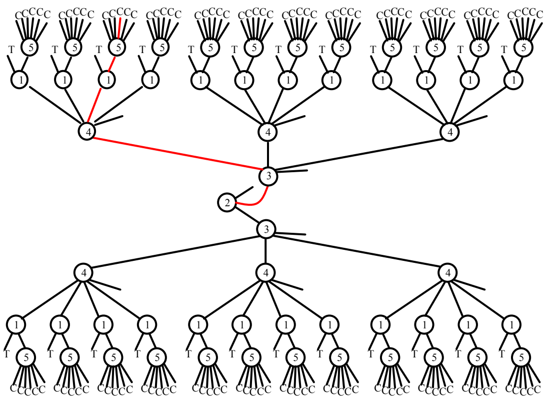

Figure 8. Expanded graph of the bar scene with path "21304" highlighted in red.

Note that each leaf in the expanded graph is an occurrence of a component. Each leaf may be represented by a **path**. The path is the concatenation of integers, each specifying which link is followed from one node to its child. The order of these integers corresponds to the traversal of the expanded graph from the root to the desired leaf. When the path follows link K from a node n, we append "0" to the path when K=(n, n.R) (corresponding to link number k=n.o+1) and "k" otherwise, i.e. when K=(n, n.L) (corresponding to link number k=1…n.o). For instance, the path "21304" corresponds to following links 2, 1, 3, 2, and 4 in the expanded graph of the bar scene (Figure 8).

Approach 2: The notion of a path suggests an alternative approach where one represents each occurrence by its path in the non-expanded graph. For example, the red chair in Figure 9 corresponds to path "21304".

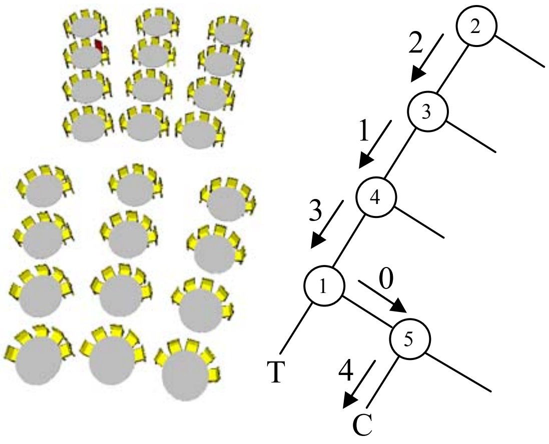

Figure 9. The red chair is identified with path "21304" (left). The path is illustrated on the unexpanded tree (right).

The designer would manually select each chair that should be treated as an exception and specify the associated exceptional treatment. A list of exceptions (selections plus treatments) is maintained separate from the graph.

This approach avoids graph expansion while still allowing arbitrary selections; however, it still has the drawback of requiring a manual selection and explicit storage of each occurrence in the exception selection set. Hence, to further reduce the designer's labor and associated storage, we will develop an implicit approach where the designer will not, in general, need to select each exception instance.

Approach 3: A third approach would be to ask the designer to associate each exception with a node n in the graph and to represent the set of target occurrences by a partial path in n. For example, every occurrence of the fourth chair of each set would be identified by select(S,"04") (Figure 10) and the every occurrence of the fourth chair in the third sets of each row would be specified by select(R,"304") (Figure 11).

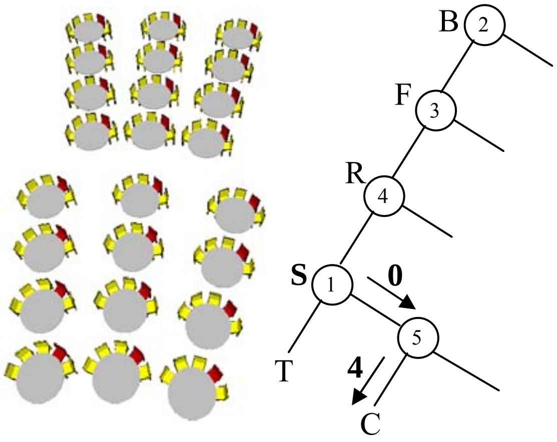

Figure 10. The set of red chairs is selected with the expression select(S,"04") (left). The partial path is shown on the graph (right).

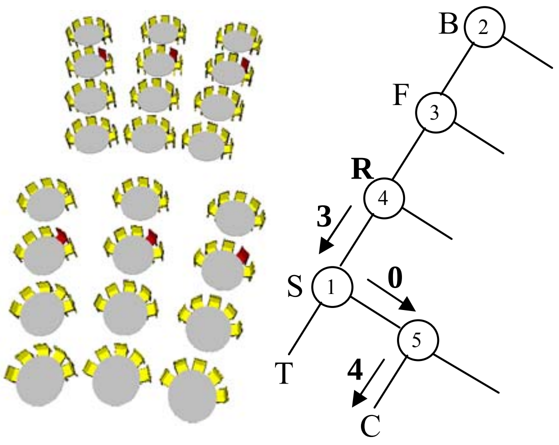

Figure 11. The set of red chairs is selected with the expression select(R,"304") (left). The partial path is shown on the graph (right).

Even though this approach was successfully used by Rossignac [Rossignac 1986] to specify a set of constraint-satisfying adjustments to features in CSG models, its limitations may require unnecessary replication of the designer's effort. For example, this approach would not allow us to select the fourth chair of the third set on each row of the first floor (Figure 12 left), because select(R,"304") does not let us differentiate floors (Figure 11) and because select(B,"11304"), select(B,"12304"), and select(B,"13304") would only specify a single chair each. To obtain the desired selection would require the union of the three individual selections (Figure 12 right).

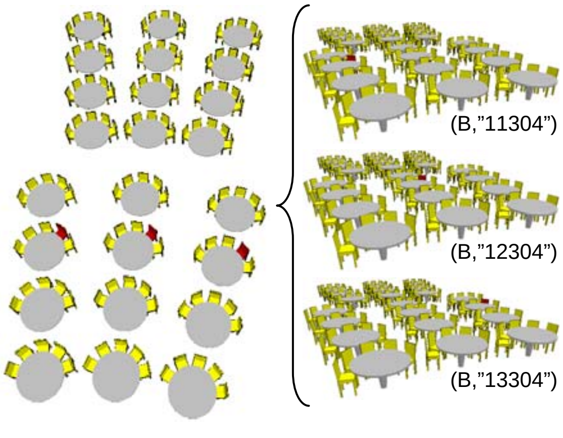

Figure 12. The set of red chairs (left) cannot be specified using a single partial path but requires the union of three partial path selections (right).

4. OCTOR selections

Our Approach: Now we describe our solution, which does not suffer from the limitations discussed above of these three approaches, and which offers several advantages: conciseness of representation, elegance of the user interface and reduction of the required user actions and cognitive burden, and increased generality. We begin by describing a concise representation for selections based on wildcards and show that it supports an elementary and essential set of selections. Later we show how the approach simplifies the user interaction required to make a selection.

4.1. Wildcards

As discussed earlier, we do not want to represent the group of selected occurrences by a list of paths and may not be able to represent important sub-patterns by a single node name and partial path. Instead we propose to represent a selection by a path to any one of the selected occurrences (the one clicked by the designer) and by a mask string of bits, one for each link on the path. A '0' in the mask corresponding to link (n, n.L) indicates that the subsequent selection should be applied to all occurrences of n.L. A '1' indicates that it should be restricted to the occurrence of n.L specified by the path. For example, a '0' bit would let us interpret the second field in the paths (B, "11304"), (B, "12304"), and (B, "13304") as a wildcard and let us interpret this path as "1*304" (using path "1i304" with mask "10111"), hence producing the selection in Figure 12. Note that only mask fields that correspond to left links (pattern links) are allowed to contain a wildcard '0'. Mask fields corresponding to right links (group links) should always be a constraint '1' to respect the unique identity of all the occurrences.

4.2. Exception Culling

We may modify eval() to incorporate this additional flexibility. We use a depth-first traversal where the left node is visited first, though in general the right node could be visited first. As we traverse, we follow the path with respect to the exception selection. If we walk off the selection path, mark that exception as inactive, visit the rest of the path recursively, and then mark it back active. When we reach a primitive (leaf), if the exception is active then its treatment is applicable to that occurrence. For a single selection, there is no need to continue traversing a path for which the selection is already marked inactive; however, the idea of *exception culling* can be applied to a whole list of exceptions, not just one as is listed in eval2(). Furthermore, exception culling also supports exceptions that are not applied at leaf nodes. All active exceptions are potentially valid at any given node and a simple node id check is needed to confirm that it is applicable to the node.

```
eval2(n, r, selected) {
  boolean deactivated = false;
  if (isPrimitive(n)) process(n, selected);
  else {
    pushMatrix();
    for (int i=1; i<=n.o; i++) {
      cullX(i, &selected, &deactivated);
      eval2(n.L, r+1, selected);
      restoreX(&selected, deactivated);
      applyMatrix(n.l);
    }
    popMatrix();
    pushMatrix();
    applyMatrix(n.r);
    cullX(0, &selected, &deactivated);
    eval2(n.R, r+1, selected);
    restoreX(&selected, deactivated);
    popMatrix();
  }
}
cullX(i, *selected, *deactivated) {
  if (*selected && (mask[r] == 1) &&
     (path[r] != i)) *deactivated = true;
  if (*deactivated) selected = false;
}
restoreX(*selected, deactivated) {
  if (deactivated) selected = true;
}
```

A selected group of k occurrences may be represented in octor in k different ways, each one comprising a path to a different occurrence and the associated mask. Since the mask only requires one bit per link, we can incorporate it into the path string by using the sign bit. Practically, we can adopt an even simpler encoding such that path[j]=-1 when mask==0, thus the path value is -1 for wildcards, 0 for going right, and 1…n.o for going left. We trivially modify cullX() to incorporate this simplification.

```
cullX(i, *selected, *deactivated) {
  if (*selected && (path[r] >= 0) &&
     (path[r] != i)) *deactivated = true;
  if (deactivated) selected = false;
}
```

4.3. Equivalence

One may think of the octor selections as representing all the axially-aligned slices of 0…p dimensions through discrete p-dimensional space, where p is the number of pattern links (i.e. left links) and thus represents the pattern nesting depth. Because each possible bit mask represents an equivalence class of slices (Figure 13), it may be convenient to keep the bit mask and the path string separate for certain applications. For example, the user can make complex selections on components that are obscured or difficult to visualize by specifying the selection pattern (slice) on more accessible occurrences, on an alternate representation, or on another model and then applying the slice somewhere else using one click (a single pick) or in an exploratory fashion.

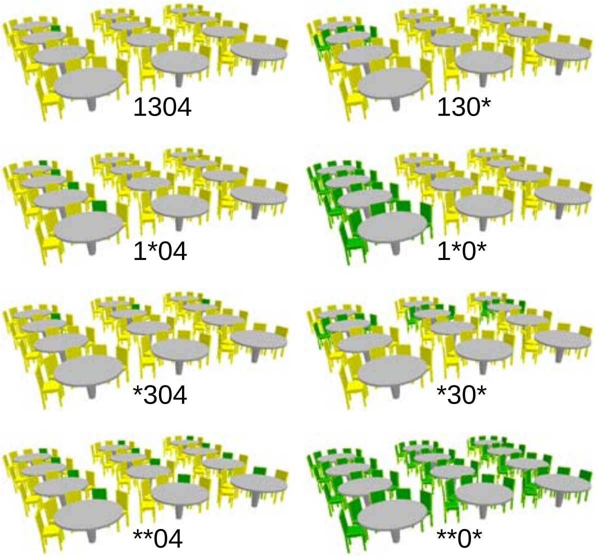

Figure 13. The 8 equivalence classes of selections for one floor of the bar are shown. The green chairs are selected using path "1304" with 8 masks "bb1b" where b is a binary digit '1' or '0' starting with "1111" (first row left) and ending with "0010" (bottom row right).

Only a small subset of the 2^r (where r is the total number of occurrences) possible selections can be represented with octor. For example, the selection shown in Figure 14 cannot be specified by a single octor path string.

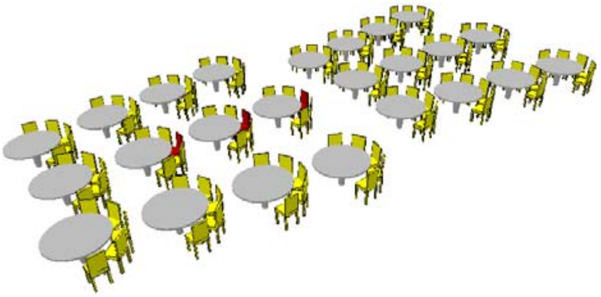

Figure 14. The selection cannot be specified by a single octor path string. Using Boolean operations, it can be specified in 2 strings with the expression "12*03" minus "12103" as opposed to 3 strings using a list of paths.

Nevertheless, we feel that the selections directly accessible through octor provide a valuable extension to other mechanisms discussed above. It allows us to make generalizations (using wildcards) and constraints (path position with no wildcard) directly corresponding to the pattern hierarchy which represents the designer intent. Of course several octor selections may be combined (union) to produce more elaborate sets. Furthermore, one may envision more a general scheme offering Boolean operations on selected sets, where the selections directly accessible through octor provide an elementary set of selections which can be combined to form all others.

5. GUI for specifying a selection

A basic approach for specifying a selection is to obtain the path from the first click (the first pick) and to obtain the mask from subsequent interaction such as additional clicks (picks).

To support a graphical user interface for octor, we have four challenges.

1. Allow the user to **pick** individual occurrences.
2. Compute the **path** of a selected occurrence from a user click.
3. Produce a candidate set for further **refinement**.
4. Use these three tools to let the user interactively build a selection **mask**.

5.1. Computing the path

To compute a path, we simply use eval(n) while tracking the path as follows:

```
eval(n, path, r) {
  if (isPrimitive(n)) process(n);
  else {
    pushMatrix();
    for (int i=1; i<=n.o; i++) {
      path[r]=i;
      eval(n.L, path, r+1);
      applyMatrix(n.l);
    }
    popMatrix();
    pushMatrix();
    path[r]=0;
    applyMatrix(n.r);
    eval(n.R, path, r+1);
    popMatrix();
  }
}
```

5.2. Picking components from a user click

In process() we need to decide if the user click position corresponds to the current occurrence. There are various solutions to this basic problem [Lucas et al. 2005] including ray-casting which works even for occluded objects. However in this case, the intended target needs to be resolved. Though reasonable solutions exist for determining or specifying the intended target, the fact that the target needs any disambiguation at all makes it undesirable for MOS.

Our MOS approach has the advantage that it only requires a single pick to be disambiguated since subsequent picks are on occurrences of the same pattern component. After resolving a single pick, these related occurrences can be isolated, for instance, by hiding all other objects. If none of the desired occurrences are visible, the designer may select the primitive in the text representation (or a component list, tree, or graph) to temporarily hide all others. For example, a subtracted CSG component located inside of another may require indirect selection of the first click. After that, the other instances are made visible and other components made invisible or diminished. Subsequent clicking can occur directly on the scene.

5.3. Building a selection mask

When the user makes the first pick, this defines a path of length d. The user now needs a way to specify a wildcard mask of length d.

A naïve approach is to specify the d bits using d clicks. Each time we ask 1 question by proposing a candidate set. The user would choose Y or N to decide if they wish to toggle that bit resulting in the selection of the candidate set.

A more direct approach is to allow the user to directly click on additional occurrences, i.e. a second, third, etc., and have the system guess or infer the selection from the cumulative set of picks. For example, when the designer selects chair “11304” and “12304”, the system generates mask “10111” producing the selection in Figure 12. Adding a third chair “21304” results in a mask of “00111” (Figure 11) and adding a fourth chair “11204” results in a mask of “00011” (Figure 10).

Observe that path fields that differ indicate generalizations and fields which are identical indicate constraints. Thus we see that only the latest pick along with the first is necessary for specifying a path plus mask. In fact, any selection supported by octor can be specified with only two clicks. For example, to specify “**304” (Figure 11) the user can select chairs “12304” and “23304”, which results in path “12304” with mask “00111”.

Another advantage of this approach is that it is intuitive. The user directly clicks the occurrences in the desired selection set and the system updates the highlighted set. Thus the user may interactively refine their selection by selecting alternative occurrences.

5.4. Refinement set

A direct clicking approach requires the user to find the right occurrence to click. While the user can interactively sample the selection space by trial and error, the system may be able to help the user by identifying a small set of occurrences to click. For example, after the user clicks one chair in the bar scene there are only 16 octor selections possible which use the path of the picked chair. (These correspond to the 16 possible masks which correspond to the 16 different equivalence classes of selections.) Yet the user can click any of the 120 chairs to make one of these 16 selections. The system can help lessen the user burden of identifying occurrences to click by presenting just one option for each of the 16 equivalence classes. Fortunately, this is straightforward to compute. For each possible bit mask, construct a path string which is the same as the path of the first pick in all fields except wildcard fields. Any variation of the field value within the pattern-count range corresponding to that field is acceptable. For instance, on may use the next or previous value as a simple heuristic. This has the benefit of near access to far selections. That is, the user can select occurrences located far apart by picking occurrences close together (i.e. close to the first pick). For example, on the airplane seating example, the user can make any octor selection (i.e. single, row, column, all) by clicking on 2 of 4 seats (Figure 15).

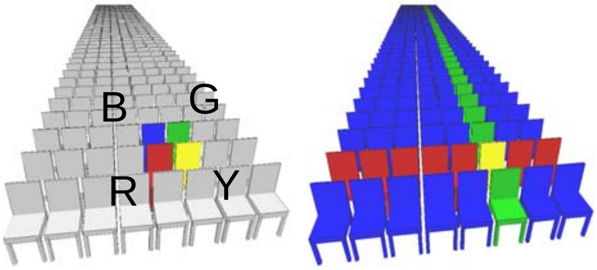

Figure 15. Clicks on just 2 of only 4 seats (left) are required to select a single seat (Y, Y), a row (Y, R), a column (Y, G), or all seats (Y, B). In fact, after clicking seat Y first, a second click on any other seat in the same row selects the row, any other seat in the column selects the column, and the rest of the seats select all (right). This selection principle extends to n-dimensions.

An alternative refinement guide is to highlight one or all occurrences that vary in only one field in the path. This visual guide helps the user choose and maintain generalizations or constraints when endeavoring to expand or refine the selection. For example, to specify selection “***04” (Figure 10), the user first selects chair “12304” and then needs to find a chair that is on a different floor, row, and set. This guide makes it clear which chairs are on the same floor, row, and set and thus helps the user find one that differs.

5.5. Consistent interaction

One notion of consistency is that octor selections are repeatable. The computed selection corresponding to any two picks is completely deterministic and unambiguous.

Now assume that we are given the bar scene as a model without the scene graph and assume that we are able to extract and determine a hierarchical structure to describe it [Thompson et al. 1999, Langbein et al. 2001]. There likely exist alternate hierarchies which generate equivalent scenes. For example, the tables could be separated from the chairs at a high level instead of the dining set. Perhaps the first chairs at each table become a row by themselves, and so on. In the alternate hierarchies, octor will still behave the same for making selections of components at the leaf if and only if the alternate hierarchy is a re-sequencing of the same pattern dimensions. In these cases, the fact that the paths will be different is transparent to the user. However, selecting non-leaf components and selecting leaf components on a hierarchy with different dimensional structure results in different behavior since different non-leaf components exist in alternate representations and the dimensional structure affects the extent of the generalizations. Here the unexpected selections are a reflection of the ability of the reverse engineering process to extract designer intent and not of the consistency of the selection mechanism of octor. In fact, making octor selections can serve to reveal the hidden scene structure and aid the process of fixing it.

6. Exception treatments

Given an exception selection, the exception treatment determines the effect of the exception on the selected occurrence set (SoS). An exception treatment can be simply defined as a treatment type along with parameters: T=(type, params).

In our approach, treatments may be applied to components at a leaf before rendering, e.g. in procedure process(), or to components not at a leaf before evaluating the subtree, e.g. calling eval(). The basic procedure for rendering is to: 1) save state (e.g. push matrix), 2) apply active exceptions, 3) render or evaluate, and 4) restore state (e.g. pop matrix).

The eval2() recursive procedure may be trivially adapted to change the rendering attributes of the SoS. For example:

```
process(n, selected) {
  if (selected) color(red) else color(n.col);
  render(n);
}
```

Now let us consider applying a transformation M to the selected occurrences. There are two issues: 1) how to apply M to each occurrence and 2) in which reference frame to define M for each occurrence.

6.1. Applying transformations

Transformation exceptions are applied at the leaf immediately before processing (e.g. rendering) the occurrence. For example, the pose for chair "22202" is defined by $p_0=s_3s_2r_1f_1b_1$. Applying an exception transform e_1 results in pose $p_{e1}=e_1s_3s_2r_1f_1b_1=e_1p_0$. Like the rendering attributes, this functionality is easily adapted into the procedure. For example:

```
process(n, selected) {
  pushMatrix();
  if (selected) applyMatrix(e1);
  render(n);
  popMatrix();
}
```

Transformation exceptions applied at non-leaf nodes use the same procedure: push the matrix stack and apply M, process the node (recursive call), and pop the matrix stack. This has the effect of inserting the new transformation in the stack of the relevant components. For example, applying e_2 to the second row on the second floor, i.e. row "22", causes the pose for chair "22202" to become $p_{e2}=s_3s_2r_1e_2f_1b_1$. In fact, since all sub-components in that row are transformed by e2, we can design our selection approach as if it applies to just leaf nodes without loss of generality. For components not at an actual leaf, e.g. a dining set or row, first select the component as part of selecting the first pick. Then we may treat the chosen component node as a leaf both algorithmically and in what is presented to the user. For instance, a dining set becomes the smallest selectable unit.

6.2. Defining transformations

In the hierarchy, each successive transformation on the links in the path is effectively defined with respect to the cumulative transformation at its parent or previous node in the graph. However, suppose we define a second row of chairs around the table as in Figure 2 to be the regular form for the dining set. Or suppose we want to move some of the chairs closer to the table or slide them around the table. Or instead, suppose we want to move them up the row or with respect to the balcony. This indicates that for modeling the basic scene or for specifying exceptions, we may want to define transformations with respect to a reference frame other than the global frame or the one defined at the parent.

The traditional way to define M is to build the matrix from user-specified or default rotation (including center of rotation and rotation axes), translation, and scaling values [vanEmmerik et al. 1993]. We take a similar approach but define M with respect to a local frame of reference R which consists of a fixed point and axes directions. The fixed point is the center of rotation and the axes directions give the axes of rotations as well as the directions of the translations. Rossignac [Rossignac et al. 1991] proposes to define the reference frame R by combining the translation elements from a center frame C with the rotation elements from an axis frame A. That is, for 3x3 rotation matrices R_C and R_A and 3x1 translation vectors T_C and T_A, define 4x4 matrices C, A, and R:

$$C=\begin{bmatrix} R_C & T_C \\ 0 & 1 \end{bmatrix},\ A=\begin{bmatrix} R_A & T_A \\ 0 & 1 \end{bmatrix},$$

$$R=\begin{bmatrix} R_A & T_C \\ 0 & 1 \end{bmatrix}$$

We allow both C and A to be chosen independently from among any of the existing frames in the scene or newly defined by the designer (i.e. the traditional way). Thus we may rotate the chairs around the rotation frame of the table while still translating them in their own frames to, for example, push them under the table. While both methods can be used to specify equivalent transformations, the expectation is that the alternative method can provide a more natural coordinate system for the designer in certain cases and that it can make the specification more efficient. Here we may point out that in order to use existing poses as frames for specifying transformations, we need to specify which components to get the poses from. Thus, not only can we use octor to select the occurrences to transform, but we can also use octor to specify the occurrences or the components from which we obtain C and A, or alternatively R directly by having C=A. This idea also makes it possible to optimize other potentially tedious tasks such as specifying reference components for modeling constraints and any other task requiring the specification of multiple selection sets.

7. Persistence

7.1. Persistent naming

After specifying one or more exceptions, the user may continue to make modifications to the model. For example, the entire bar could be patterned (e.g. mall M=7B), more items could be placed on the table as part of the standard dining set (e.g. a lamp with S=T+5C+L), or a design pattern could be placed on the standard chair (e.g. H=C+10P, S=T+5H). Even the pattern counts may be adjusted (e.g. S=T+8C) or patterns removed completely (e.g. flatten B=2F and F=3R into B=3R). The challenge is deciding how to update octor selections when the model is edited such that they continue to identify or *name* the same components.

The persistent naming problem for parametric, feature, and history based modeling has been well studied [Marcheix and Pierra 2002]. Fortunately, our version of the problem is less complicated since we directly name the occurrences and their existence or location in the hierarchy is explicit, whereas the structure of topological features may be implicit depending on where and how they are combined.

Now we list some basic modification scenarios and the effect of each on a path. (Targets of the paths are underlined.) Note that this is only a subset of the possible scenarios and that every possible graph modification does not necessarily correspond to actual modeling operations.

1. Insert/delete branch not in path – no change.
   $S=3A+4\underline{B} \rightarrow S=3A+4\underline{B}+5C$
2. Insert/delete right branch in the path – insert '0' (go right) or delete field in all affected path strings.
   $S=3A+5\underline{C} \rightarrow S=3A+4B+5\underline{C}$
3. Insert/delete left branch in path – insert '-1' (wildcard) or delete field in all affected path strings.
   $R=4\underline{S} \rightarrow R=4G, G=3\underline{S}$
4. Delete component – a deleted leaf means all associated exceptions can be deleted.
5. Change occurrence count – no change.

The different scenarios are implicitly identified by running a path-following traversal with modified exception culling. Instead of maintaining the active/inactive status of each exception we only need to keep track of whether we are still on the path. When the node location for the modification is reached, we can determine what to do to each octor path based on the modification information and the on/off state. If the component at the path destination target is being deleted, we can delete the exception. If we are off the path or changing the occurrence count, the path does not need to be changed. If we are deleting a node, the path field corresponding to this node (actually, its link) is deleted. If we are inserting a node, we insert '0' when the octor path will be nested to the right and '-1' (wildcard) when the path will be nested to the left.

Inserting a wildcard for left branch insertions is reasonable because for the user to insert a left branch is to take a sub-expression in the hierarchy and make it the leader of a new pattern to be nested at the same location in the hierarchy. Thus, inserting a wildcard preserves this intent as opposed to the less likely intent of making it an exception and taking the original structure, which is not currently manifested in the design, as the leader.

7.2. Updating existing exceptions explicitly

Another notion of persistence is updating the descriptions of existing exceptions. For instance, the user specifies an exception E_1=(A, "transform", M_1), i.e. a selection A to be transformed by M_1. Now the user wants to transform the occurrences in selection A by M_2 and so specifies E_2=(A, "transform", M_2). Now there is the option of creating a new exception E_2, or alternatively the system recognizes that the selection is the same and the treatment type is the same (in this case a transformation) and updates the existing exception E_1=(A, "transform", M_2M_1).

The user may also want to modify the selection set of an existing exception. For instance, in the previous example the user modifies E_1 resulting in E_1=(B, "transform", M_1). The challenge here is identifying which exception to update since multiple exceptions may have the same treatment. Thus the problem can be more generally stated: modify the selection set of one or more exceptions given a particular treatment. This is a complex discussion which involves many possible approaches and is left to future work.

8. Applications and extensions

The octor representation and approach described has been illustrated using scene graphs, a general description for geometric compositions with widespread application including geometric modeling approaches such as CSG, BReps, and parametric models. For instance, it can be used to support modeling (e.g. representation, identification, and selection) of CSG primitives (e.g. a pattern of holes) and features in BRep models (e.g. features in features) and parametric models. It can also be used for specifying and handling exceptions in animation design (e.g. choreography) and specifying reference sets for constraint satisfying transforms.

We identify several opportunities for future work. (1) The approach can be extended to handle recursive structures. For instance, a recursive pattern refers to itself as the leader (i.e. n.L points to n). A straightforward approach would be to maintain a node access count within a recursion limit. (2) Ranges can be supported by using two octor strings. Explore approaches for specifying ranges. (3) A variant of octor can be used to select arbitrary levels in object group hierarchies. For example, compute the selection as the smallest group containing both picks. (4) Develop a way to specify, represent, and compute Boolean combinations of octor selections.

9. Contributions and conclusions

We have presented an approach for modeling exceptions in regular pattern hierarchies. In particular, octor provides a representation for a subset of occurrences of modeling components and support for a user to graphically specify such sets. The representation based on a path and wildcards is more compact than a list of paths and is more general than a node with partial path. The GUI support is flexible and offers several facilities for specifying selection sets. A 2-click method for making any octor selection is described, but other methods including 1-click plus guided set of Y/N queries could be used. Refinement sets or other highlighting guides were also proposed. While the approach is well suited and intuitive for direct manipulation (direct picking), it also supports indirect picking of components. Octor provides a selection method that is less laborious than clicking each occurrence and less demanding than node and path.

10. References

[AutoCAD] AutoCAD 2007. Autodesk, Inc. http://www.autodesk.com/autocad.

[Langbein et al. 2001] F. C. Langbein, B. I. Mills, A. D. Marshall, R. R. Martin. Recognizing Geometric Patterns for Beautification of Reconstructed Solid Models. In Proc. International Conference on Shape Modelling and Applications, pp. 10-19, 2001.

[Lucas et al. 2005] Lucas, J.F., Bowman, D.A., Chen, J., and Wingrave, C.A., "Design and Evaluation of 3D Multiple Object Selection Techniques" Proc. of the ACM I3D, March 2005.

[Marcheix and Pierra 2002] Marcheix, D. and Pierra, G. A survey of the persistent naming problem. In Proceedings of the Seventh ACM Symposium on Solid Modeling and Applications, Saarbrücken, Germany, June 17-21, 2002.

[MSWord] Microsoft Office Word 2007. Microsoft Corporation. http://office.microsoft.com/word/.

[Miller and Myers 2002] Robert C. Miller and Brad A. Myers. "Multiple Selections in Smart Text Editing." Proceedings of the 6th International Conference on Intelligent User Interfaces (IUI 2002), San Francisco, CA, January 2002, pp 103-110.

[Oh et al. 2006] Oh, Ji-Young, Stuerzlinger, W., Dadgari, D. Group Selection Techniques for Efficient 3D Modeling. IEEE Symposium on 3D User Interfaces (3DUI 2006), University of Arizona, Tucson, March 25-29, pp 95-102, 2006.

[Pro/ENGINEER] Pro/ENGINEER Wildfire 3.0. Parametric Technology Corporation. http://www.ptc.com/.

[Rappoport 1993] Rappoport Ari. A scheme for single instance representation in hierarchical assembly graphs. Falcidieno, B., Kunii T.L. (eds), Geometric Modeling in Computer Graphics, pp. 213-224, 1993.

[Rossignac 1986] Jarek Rossignac. Constraints in Constructive Solid Geometry. Proc. ACM Workshop on Interactive 3D Graphics, ACM Press, pp. 93-110, Chapel Hill, 1986.

[Rossignac et al. 1991] Jarek Rossignac, Paul Borrel, J. Mastrogiulio, Jai Kim. BIERPAC: Basic Interactive Editing for the Relative Positions of Assembly Components. IBM Research Report RC 17339, 1991.

[Rossignac et al. 1988] Jarek Rossignac, Paul Borrel, and Lee Nackman. Interactive Design with Sequences of Parameterized Transformations. Proc. 2nd Eurographics Workshop on Intelligent CAD Systems: Implementation Issues, April 11-15, Veldhoven, The Netherlands, pp. 95-127, 1988.

[Thompson et al. 1999] Thompson, W.B.; Owen, J.C.; de St. Germain, H.J.; Stark, S.R., Jr.; Henderson, T.C. Feature-based reverse engineering of mechanical parts. IEEE Transactions on Robotics and Automation, 15(1), pp. 57-66. 1999.

[vanEmmerik et al. 1993] Martin van Emmerik, Ari Rappoport, and Jarek Rossignac. Simplifying interactive design of solid models: A hypertext approach. The Visual Computer, vol. 9, No. 5, pp. 239-254, March 1993.

11. Appendix

Here we give expressions for the number of selections and equivalence classes of selections that the discussed approaches support. While the list of paths approach appears to have the advantage by supporting every possible selection, this is achieved as the union of individual selections. Its base selection unit is a single occurrence whereas the other approaches are able to select a single occurrence along with larger selections as a base selection unit. In general, all of these approaches are able to specify any selection by combining one or more base selection unit.

Approach 1 (expanded tree) and Approach 2 (list of paths): These approaches can be used to specify completely arbitrary selection sets numbering $\mathbf{q_0=2^r}$ where r is the total number of occurrences of the component. For a tree or subtree rooted at n_a and given a component represented by node n_k:

$$r(n_a) = \prod_{i=1}^{d} n_i.o$$

where d is the depth of the path from root node n_a to node n_k and n_i is the i-th node on this path. In the bar scene, r=2*3*4*5=120 chairs, so $q_0=2^{120}$ combinations of chair selections in the bar scene. There is only c=1 equivalence class for kinds of selections (arbitrary) and it contains q_0 selections.

Approach 3 (partial path): In the partial path approach, the number of selections is:

$$q_3 = \sum_{j=1}^{p} r(n_j)$$

where p is the number of pattern links (left links) on the path to the leaf, n_j is the parent node associated with link j, and r(n) gives the total number of occurrences for the subgraph rooted at n. For the chairs in the bar scene, q_3=2*3*4*5+3*4*5+4*5+5=205 in c=p=4 equivalence classes: the same chair in each dining set (5), row (4*5), floor (3*4*5), or bar (2*3*4*5).

Octor (path with wildcards): Out of the 2^{120} possible selections of a subset of chairs in our bar, octor can select:

$$q_4 = \prod_{i=1}^{p} (n_i.o + 1)$$

of them in $c=2^p$ different equivalence classes. For example, in the bar scene example we have r=2*3*4*5=120 paths and $c=2^4=16$ different masks. Since different path-mask combinations may define the same selection due to the wildcards, the number of unique selections is q_4=3*4*5*6=360 out of the possible 2^{120}.

Note that $q_4 \geq q_3$. Proof. Rewrite $q_3=\{t_p: t_0=0, t_i=o_i(t_{i-1}+1), 1\leq i\leq p\}$ and $q_4=\{f_p: f_0=1, f_i=(o_i+1)f_{i-1}, 1\leq i\leq p\}$, where $o_i=n_i.o$ and $o_i\geq 1$. Expand the expressions for f and t, then divide by o_p and subtract 1, divide by o_{p-1} and subtract 1, etc. Since $o_i\geq 1$, we can reduce these expressions such that $f_p\geq t_p$ iff $1+1/(o_1 o_2 \ldots o_p) \geq 1$, which is true by the same assumption $o_i\geq 1$. Therefore $q_4\geq q_3$. This makes sense since the set of masks for partial paths is the subset of masks of length p with only leading zeros whereas the set of masks for octor paths is the set of all binary masks of length p.