User-Oriented Visual Layout at Multiple Granularities*

Yannis Ioannidis† **Miron Livny** **Jian Bao** **Eben M. Haber**‡
Department of Computer Sciences, University of Wisconsin, Madison, WI 53706
{yannis,miron,bao,haber}@cs.wisc.edu

Abstract

Among existing tools for laying out large collections of visual objects, some perform automatic layouts, possibly following some rules prespecified by the user, e.g., graph layout tools, while others let users specify layouts manually, e.g., CAD design tools. Most of them can only deal with specific types of visualizations, e.g., graphs, and some of them allow users to view visual objects at various levels of detail, e.g., tree-structure visualization tools. In this paper, we develop techniques that strike a balance between user specification and automatic generation of layouts, work at multiple granularities, and are generally applicable. In particular, we introduce a general framework and layout algorithm that (a) deals with arbitrary types of visual objects, (b) allows objects to be viewed in any one of several different visual representations (at different levels of detail), and (c) uses a small number of user-specified layouts to guide heuristic decisions for automatically deriving many other layouts in a manner that attempts to be consistent with the user's preferences. The algorithm has been implemented within the OPOSSUM database schema manager and has been rather effective in capturing the intuition of scientists from several disciplines who have used it to design their database and experiment schemas.

1 Introduction

Visualization of large collections of objects is a very important area. Of special interest to us are environments where the layouts of these visualizations are specified by the users. For example, large graphs are laid out to display a variety of structures, including database schemas, communications networks, flow charts, and block diagrams of large systems. For another example, large collections of diagrams can be laid out to indicate trends in several properties of a system or phenomenon, like weather characteristics changing over time at several stations, shock wave characteristics after an explosion for oil exploration, and stock market prices for stocks of several types.

Quite often, there are multiple visual representations that can be used for each visualized object, each representation having a different size. This is typically due to a desire to focus on one part of the visual information while still seeing it in the context of the whole. For example, in textual outlines and tree-structure visualizations, subsidiary information (such as outline subpoints or subtrees) may be hidden when not of interest (see Figure 1); in general graphs, regions with several nodes and edges may be replaced by a single node; in set representations, the precise contents of the set may be hidden.

PHILOSOPHY	PHILOSOPHY
TRAGEDY	Aristotele
Aeschylus	Plato
Sophocles	Socrates
Euripides	TRAGEDY
COMEDY	COMEDY
MATHEMATICS	Aristophanes
ASTRONOMY	MATHEMATICS
	ASTRONOMY

Figure 1: Example of text outlines at various levels of detail

Multiple visual representations, however, may also be desirable independent of abstraction, from a mere need for multiple views. For example, one may color an object red or may enlarge the object and put a label with a number next to it capturing the precise shade of 'red' being used (for more accurate knowledge of color as well as for display on black-and-white screens); one may present textual record structures with the details of a set of trees or show an icon for each tree whose characteristics capture the trees' properties (see Figure 2); one may show the details of a record as contained within the record or as separate nodes connected to the record node through edges.

Dealing with multiple visual representations of different sizes is a nontrivial problem due to the following two requirements on visual object layouts:

R1 Visual objects should not overlap: all objects under the screen view are individually viewable.

R2 Visual objects should not unnecessarily waste space between them: modulo any explicit object placement by the user, distances between objects should be below a certain threshold to allow one to simultaneously view as many objects as possible at any zooming level.

* Work supported in part by the National Science Foundation under Grant IRI-9224741.

† Additionally supported in part by the National Science Foundation under Grant IRI-9157368 (PYI Award) and by grants from DEC, IBM, HP, AT&T, Oracle, and Informix.

‡ Present address: Silicon Graphics, MS 8U-981, 2011 N. Shoreline, Mountain View, CA 94043-1321, eben@sgi.com.

Permission to make digital/hard copies of all or part of this material for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copyright is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires specific permission and/or fee.

AVI '96, Gubbio Italy
© 1996 ACM 0-89791-834-7/96/05..$3.50

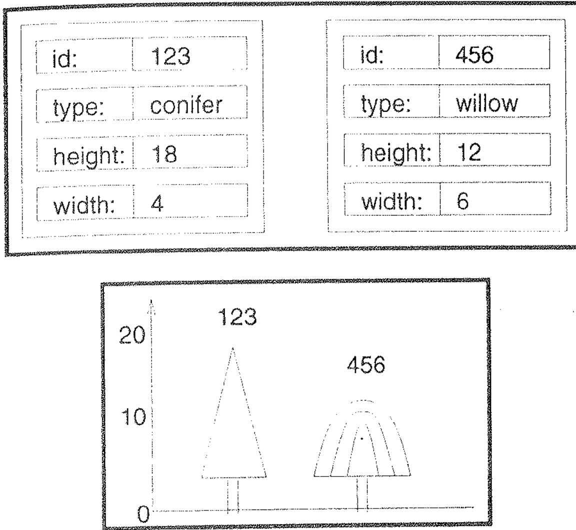

Figure 2: Example of textual and iconic representations of two trees

When moving from a small to a large visual representation of an object, requirement R1 precludes the approach of showing the larger representation in a new window that is subsequently removed. This would be appropriate to satisfy a momentary and local desire to change representation, but our goal is to deal with changes that last for an arbitrarily long time and shift the focus while continuing to maintain the global context.

In this paper, we introduce techniques that deal with layouts of large collections of objects with multiple visual representations in the context of Requirements R1 and R2. These have emerged out of our efforts to build a generic and intuitive database schema manager (OPOSSUM [6]) that can be used by scientists to lay out designs of their database and experiment schemas. We have been collaborating with scientists from various disciplines (Soil Sciences, Biochemistry, and Physics) and the need to have an effective method for user-oriented schema layout with a variety of visual representations has been universal.

We first establish a framework within which we express assumptions that must be fulfilled by the visual objects concerned and additional requirements on the style of user interaction desired, and then study the layout problem. We introduce a heuristic algorithm that maintains a small number of layouts that the user has (directly or indirectly) specified for specific combinations of representations of the visual objects. The algorithm takes these into account to automatically derive layouts for all the remaining such combinations in a way that attempts to capture the user's layout preferences. In its general form, the algorithm can deal with different types of objects associated with various visual representations. It has been implemented in OPOSSUM and has been quite successful in generating desirable and intuitive layouts.

2 Problem Formulation

2.1 Definitions and Assumptions

Without loss of generality, we assume that all objects to be visualized are of the same type, e.g., all are graph nodes. We discuss how this restriction can be removed in Section 4. Let $\mathcal{O}$ denote the (potentially countably infinite) set of objects to be visualized. Furthermore, without loss of generality, let $\mathcal{O}^{\mathcal{V}} = \{o_i | o_i \in \mathcal{O} \text{ and } 1 \leq i \leq m\}$ be the set of objects currently visualized, for some integer m. There is a (usually small) set of *visual representation kinds* that is associated with the set of objects to be visualized; if there are n such kinds, each one is identified by an integer $1 \leq j \leq n$. Every object in $\mathcal{O}$ can be visualized using any of the representation kinds; the visual object resulting from visualizing object o_i using representation kind j is denoted by v_{ij}. For example, Figure 2 shows two visual representations (one textual and one iconic) for physical trees.

A *visual configuration* (or simply configuration) consists of the visual objects v_{ij} that are used to represent the corresponding objects o_i. Clearly, given a fixed set of objects $\mathcal{O}^{\mathcal{V}}$, there are n^m possible configurations. The set of these configurations is denoted by $\mathcal{C}$. As a notational convention, we represent each configuration as a "word" of the form $a_1 a_2 \ldots a_m$, where for each object $o_i \in \mathcal{O}^{\mathcal{V}}, 1 \leq i \leq m$, the "letter" a_i is equal to j, where v_{ij} is the visual representation of o_i in the configuration. A *layout* of a configuration consists of a location on the plane for each object in $\mathcal{O}^{\mathcal{V}}$. The user is free to move among configurations; we assume that this is achieved through user actions that consist of changing the visual representation of a single object each time. The configuration that is on the screen at any point is the *current configuration*. Returning to the example of visualizing collections of trees, Figure 2 deals with two tree objects and shows two visual configurations of the collection. These would be configurations 11 and 22, if 1 was used for the iconic (smaller) representation of a tree and 2 was used for the textual representation. The remaining two configurations are indicated in Figure 3.

The following assumption holds in many situations and is important for obtaining effective and efficient layout algorithms:

A For any object $o_i \in \mathcal{O}$, its corresponding visual objects v_{ij}, $1 \leq j \leq n$, can be put in a lattice $\preceq_i$ based on their sizes so that, if $v_{ij} \preceq_i v_{ik}$, then v_{ij} can be drawn completely enclosed within v_{ik}.

Observe in Figure 2 or 3 how the textual representation is larger than the iconic representation for both trees.

Without loss of generality, and for simplicity of presentation, we further assume that $\preceq_i$ is a total order and gives rise to the same lattice on representation kinds for all i. We use $\preceq$ to denote the lattice of representation kinds and (overloading the symbol) the lattice on visual objects corresponding to an object, make the notational assumption that $v_{ij} \preceq v_{ik}$ if $j \leq k$.

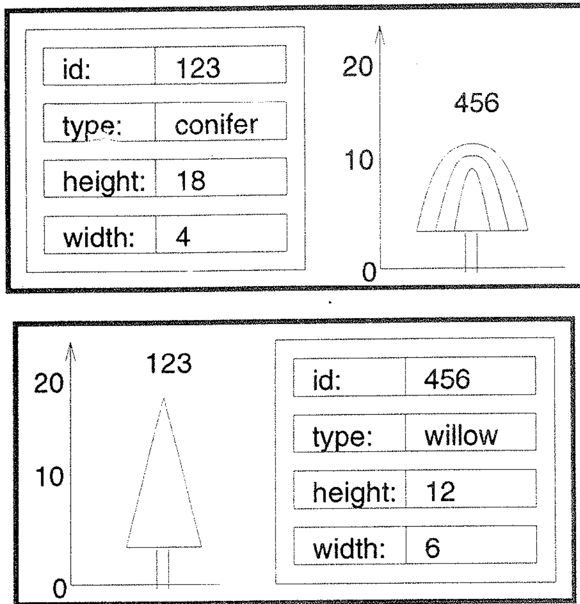

Figure 3: Two configurations with different visual representations for two trees

An important implication of Assumption A is that the visual configurations in $\mathcal{C}$ form a lattice (denoted by $\preceq_{\sqsubset}$) based on the lattices on visual objects. In particular, for two configurations $c, c' \in \mathcal{C}$, we say that $c \preceq_{\sqsubset} c'$ if for all i, $v_{ij} \in c$ and $v_{ik} \in c'$ imply that $v_{ij} \preceq v_{ik}$.

For example, consider an abstract situation where there are two objects to be visualized and three representation kinds (denoted by 1, 2, and 3, respectively). The lattice formed by all the possible configurations is shown in Figure 4, where the nodes capture configurations and the arrows capture $\preceq_{\sqsubset}$.

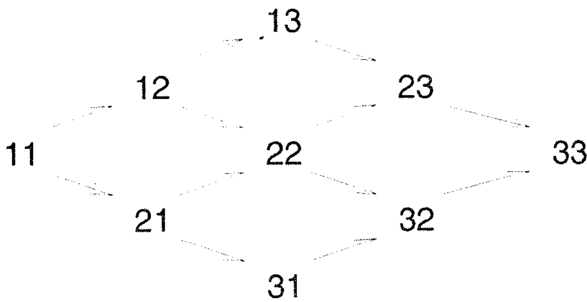

Figure 4: Size-based inclusion lattice of visual configurations

Let *valid layouts* of configurations be those that satisfy Requirement R1. Let *perfect layouts* of configurations be those that satisfy both Requirements R1 and R2. Because of Assumption A and the fact that the configuration lattice is based on visual representation sizes, it is clear that if $c \preceq_{\sqsubset} c'$, then any perfect layout of c needs no more space than any perfect layout of c'. Alternatively, if $c \preceq_{\sqsubset} c'$, then any valid layout of c' is a valid layout of c as well. These observations form the intuition for the layout algorithms given below. The algorithms only ensure the *valid*-ity of layouts resulting from user actions, while striving for *perfect*-ion of layouts resulting from internal actions.

2.2 Layout Requirements

The emphasis of this work is on design environments. That is, users interact with the system by performing the following actions:

- they create objects interactively by placing on the screen some visual representation of them;
- they delete objects interactively by removing their visual representations from the screen;
- they move visual representations of objects around to alter the current layout; and
- they view different visual configurations by changing the visual representations of objects on the screen.

As mentioned earlier, our main motivation for this work has been the needs of experimental scientists, who essentially operate within such an environment when designing experimental studies. In our collaborations with several such scientists, the following requirements have emerged with respect to the layout capabilities of a design tool (in addition to the more generic Requirements R1 and R2).

R3 A new object may be created using any visual representation for the generated visual object, while those already existing may be in any configuration.

R4 Layouts specified by the user are important and the system should be able to regenerate them.

After discussing the problems of a straightforward attempt to satisfy these requirements, the next section presents a heuristic algorithm that is effective in doing the same while avoiding the problems. This algorithm is the main contribution of our work.

3 Algorithms

Our algorithm descriptions focus on what happens during each of the action types mentioned in Section 2.2. In addition, they elaborate on what is stored in the algorithms' persistent storage structures.

3.1 Naive Algorithm

The following is a description of a naive algorithm that would accurately satisfy Requirements R1-R4.

1. *Storage:* The layout of any configuration that the system has passed through is explicitly stored.
2. *Object creation:* An object may be created only in locations that make the resulting layout satisfy Requirement R1 (valid layouts); otherwise the request is rejected. Since previously maintained configuration layouts are for a different overall set of objects, they are all removed

from the algorithm's storage structures and the layout of the current configuration becomes the only one recorded there.

3. *Object deletion:* An object may be deleted at any time. As in object creation, previously maintained configuration are removed.

4. *Object move:* An object may be moved only to locations that make the resulting layout satisfy Requirement R1 (valid layouts); otherwise the request is rejected. The layout stored for the current configuration is updated to reflect the new location of the moved object.

5. *Configuration change:* If the new configuration has been seen in the past, then its layout is already stored and is simply used to draw the visual objects of the configuration on the screen. Otherwise, the user places on the plane all existing visual objects and the resulting layout is stored for future use.

Note that the above algorithm is extremely easy to implement as most of the burden lies with the user; the algorithm simply follows the user's instructions and does not incorporate any layout techniques itself.

On the other hand, this algorithm is obviously unrealistic, particularly with respect to points 1, 2, 3, and 5. Point 1 implies that the space requirements of the algorithm may potentially be exponential in the number of visual objects ($O(n^m)$ if the user passes through all configurations, for m objects and n visual representation kinds). Points 2 and 3 imply that the mere creation or deletion of a single object results in throwing away tremendous information captured in the stored layouts and waisting all the time spent by the user to generate these layouts. Finally, point 5 implies that the effort required by the user to move to a previously unseen configuration is linear in the number of visualized objects, which for large designs would be unacceptable.

The only reason for discussing the above algorithm is that it represents a form of ideal with respect to Requirements R1-R4. Any algorithm that accurately satisfies these requirements will need the user to be involved at a similar level as in the algorithm above and will be of similar space and/or time complexity. For example, point 1 cannot be avoided if users are not to be forced to lay out again configurations through which they have already passed and are revisiting. Points 2 and 5 could be made more realistic by keeping past layouts of partial configurations (with fewer objects than currently), but that would imply even larger space overhead ($O(n^{m+1})$ in the worst case, assuming that n is constant). Finally, point 3 could be modified so that past layouts of larger sets of objects are used for the layout of smaller sets by simply ignoring the locations of deleted objects. In addition to potentially increased space overhead, however, this creates the problem of possibly having to choose among multiple inconsistent layouts for the same (smaller) configuration. This problem would arise if two stored layouts are for configurations that differ in the visual representation of a single object, which is the one deleted, and have some of the remaining objects in different locations.

3.2 Heuristic Algorithm

In this section, we present an algorithm that interprets Requirement R4 not as a strict rule but only as a desired goal, and is much more efficient and realistic in terms of space and time complexities and required user involvement. Although it does not satisfy R4 in an absolute sense, it attempts to approach it as much as possible by following heuristics that are based on a particular notion of users' preferences as expressed in the layouts specified by the user.

Understanding users' preferences and intended meaning underlying any given spatial placement of visual objects is a difficult problem, as these can be based on a wide variety of spatial characteristics [7]. In our approach, we consider relative or absolute angles of vectors connecting visual objects between them or to a fixed origin as the key features that characterize the relative placement of objects. Thus, our heuristics are based on some form of an approximate angular invariance.

The following is a description of the algorithm using the same structure as for the naive algorithm.

1. *Storage:* The layout of a very small number of carefully chosen configurations is explicitly stored. These are called *anchored configurations*. At any point, the layout of the current configuration is stored as well.

2. *Object creation:* An object may be created only in locations that make the resulting layout satisfy Requirement R1 (valid layouts); otherwise the request is rejected. A heuristic procedure is invoked (Procedure `Heur-Create` below) to insert the new object in the layout of each anchored configuration that is different from the current configuration and update the corresponding stored information. Procedure `Heur-Create` guarantees Requirements R1 and R2 (perfect layouts) with respect to the new object and attempts to satisfy Requirement R4 by taking into account the layout stored for the anchored configuration before the new object is inserted and also the location where the user placed the new object within the layout of the current configuration.

3. *Object deletion:* An object may be deleted at any time. In addition to the current configuration, the object is deleted from each anchored configuration as well. The layouts of all stored configurations remain the same with respect to all other objects, thus only guaranteeing valid but not necessary perfect layouts (Requirement R1 but possibly not R2) and also satisfying Requirement R4 by not touching any object placed by the user.

4. *Object move:* An object may be moved only to locations that make the resulting layout satisfy Requirement R1

(valid layouts); otherwise the request is rejected. The layout in the current configuration is updated to reflect the new location of the moved object. If the current configuration happens to be an anchored configuration as well, then the update is also propagated to the corresponding layout stored.

5. *Configuration change:* If the new configuration is anchored, then its layout is already stored and is simply used to draw the visual objects of the configuration on the screen. Otherwise, a heuristic procedure is invoked (Procedure `Heur-Reconfig` below) to obtain a layout for the new configuration. Procedure `Heur-Reconfig` guarantees Requirements R1 and R2 (perfect layouts) and attempts to satisfy Requirement R4 by taking into account the configuration lattice and the layout of anchored configurations, which either directly or indirectly (through Procedure `Heur-Create`) reflect user preferences. Since the old and new configurations differ only in the visual representation of a single object, they are guaranteed to be related via $\preceq_{\sqsubset}$. Let c and c' be the old and new configurations, respectively, and assume that $c \preceq_{\sqsubset} c'$. Procedure `Heur-Reconfig` identifies the closest anchored configuration c_a for which $c \preceq_{\sqsubset} c_a$ and $c' \preceq_{\sqsubset} c_a$ and moves visual objects taking into account their locations in the stored layout of c_a until a perfect layout is derived. (The algorithm proceeds similarly when $c' \preceq_{\sqsubset} c$, in which case c_a satisfies $c_a \preceq_{\sqsubset} c$ and $c_a \preceq_{\sqsubset} c'$ and the concern is wasted space.)

Note that point 3 may have the problem mentioned at the end of Section 3.1 of possibly having to choose among multiple inconsistent layouts for the same anchored configuration after an object deletion. However, if at any point any two configurations differ in the visual representation of at least two objects, then the problem does not arise. We expect that the premise of the above statement is virtually always true, so point 3 does not have to deal with this issue. Also note that point 4 remains essentially the same as in the naive algorithm, since it did not introduce any major costs. One may argue that the system should not reject object placements that violate Requirement R1, but instead should employ a *plowing*-like algorithm [12] to push visual objects so that all overlap is removed. The advantage of this approach is that users do not have to open up space themselves in the area where they want to move the object, while its disadvantage is that it distorts the user-specified placement of all other objects. Both approaches are fine alternatives, however, and one could incorporate into the algorithm either one based on the users' desires. Our choice in the above description is due to its simplicity.

Also note that in order for c_a to always exist for point 5, the configurations represented by the words $111\ldots1$ and $nnn\ldots n$, which by definition are the lowest and highest elements in the lattice (recall that n is the number of representation kinds), must be anchored. Beyond those, any other configuration may be specified as anchored as well, either dynamically by the user (so that the most important configurations have exactly the desired layout) or statically by the system implementors (so that there are enough anchored configurations spread around the lattice for system stability). In the latter case, a reasonable approach (which we have adopted as well) is to make anchored the configurations where all objects are in the same visual representation, i.e., those represented by the words $iii\ldots i$, $1 \leq i \leq n$.

We now present Procedure `Heur-Create` in a little more detail. In what follows, c is the current configuration and v is the visual representation of the newly created object. Recall that perfect layouts satisfy both Requirements R1 and R2, which call for no overlap and no wasted space between any two visual objects.

procedure `Heur-Create`(c, v)
    begin
    $\underline{l}$ = vector from origin to location of v in c
    for each anchored configuration c_a s.t. $c_a \neq c$ **do**
        move v along the vector $\underline{l}$
                until a perfect layout is obtained
        update location of v in c_a
    od
    end

Procedure `Heur-Create` is illustrated in Figure 5. The visual objects A, B, C, and D in the current configuration, c, are shown in Figure 5a with thin dashed borders and the newly inserted object E is shown with a thick dashed border. Object E is placed in a location that results in a valid layout for c. Consider now an anchored configuration c_a whose layout before the creation of the new object is shown in Figure 5b with the visual objects having thin solid borders. The visual representation of E in c_a is shown with a dashed border placed in the location specified by the user in c. The resulting layout is not valid since, in that location, E overlaps with other visual objects in c_a. Procedure `Heur-Create` pushes E along the vector $\underline{l}$ (between the origin and the location of E) until all overlap disappears. In its final location, E is shown with thick solid border.

Note how Procedure `Heur-Create` tries to satisfy Requirement R4. The new object E is initially placed in the same location where the user placed it, and is then moved along the line between that location and the origin. If the user has been consistent with respect to laying out anchored configurations and the current one, then the resulting location of the new object is likely to have spatial similarities to that chosen by the user in the current configuration. In the above, consistency may be precisely defined in various ways, e.g., it may be defined as, in all configurations, placing objects at the same angle (or within a bounded angle difference) in radial coordinates with respect to the origin. We avoid giving a specific definition as Procedure `Heur-Create` is indepen-

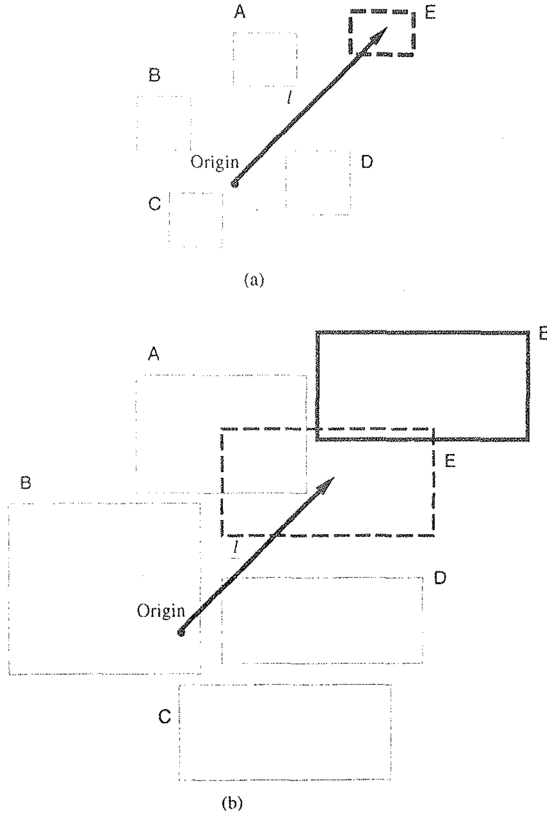

Figure 5: Example of Procedure `Heur-Create`: (a) current configuration; (b) anchored configuration

dent of that, but want to emphasize that the essence of any such definition is that there should be no dramatic swappings of relative positions of elements. Even more important for Requirement R4 than the direction of the object move is the fact that procedure `Heur-Create` does not alter the location of any existing element in the layout of the anchored configuration. If the user has spent a long time to place things 'just right', a heuristic algorithm will not destroy that. Even if the location resulting for E is not optimal, the user can change that location when visiting that anchored configuration in the future.

We also present Procedure `Heur-Reconfig` in a little more detail below. In what follows, c is the old configuration, c' is the new configuration, and v is the new visual representation of the object whose representation change causes the configuration change.

procedure `Heur-Reconfig`(c, c', v)
  begin
  $\underline{l}$ = vector from origin to location of v in c (and c')
  if $c \preceq_{\sqsubset} c'$
    then c_a = anchored configuration
        closest to c' s.t. $c' \preceq_{\sqsubset} c_a$
    else c_a = anchored configuration
        closest to c' s.t. $c_a \preceq_{\sqsubset} c'$
  $\underline{l_a}$ = vector from origin to location of v in c_a
  $\underline{l_\delta} = \underline{l_a} - \underline{l}$
  $V = \emptyset$
  for each v' violating Requirements R1-R2
        with respect to v in c' **do**
    $V = V \cup \{v\}$
  od
  while $V \neq \emptyset$ **do**
    choose v' an element in V
    $\underline{l'}$ = vector from origin to location of v' in c
    $\underline{l'_a}$ = vector from origin to location of v' in c_a
    $\underline{l_{move}} = \underline{l'_a} - \underline{l_\delta} - \underline{l'}$
    move v' along the vector $\underline{l_{move}}$
        until a perfect layout is obtained
        with respect to v' and v
    update new location of v' in c'
    for each v'' violating Requirements R1-R2
        with respect to v' in c' **do**
      $V = V \cup \{v''\}$
    od
    $V = V - \{v'\}$
  od
  end

Procedure `Heur-Reconfig` is illustrated in Figure 6. The visual objects A and B in the current configuration, c, are shown with thin dashed borders. Object A has just changed visual representation in c, which has resulted in an invalid layout due to overlap with B. Consider the closest anchored configuration c_a that satisfies the condition expressed in the pseudocode above. Its layout is shown in the figure with the visual objects having thin solid borders. Vectors $\underline{l}$, $\underline{l_a}$, $\underline{l_\delta}$, $\underline{l'}$, and $\underline{l'_a}$ as defined in Procedure `Heur-Reconfig` are shown in the figure as thin arrows, while vector $\underline{l_{move}}$ is shown as a thick arrow. Procedure `Heur-Reconfig` pushes B along the vector $\underline{l_{move}}$ until its overlap with A disappears. In its final location, B is shown with thick dashed border.

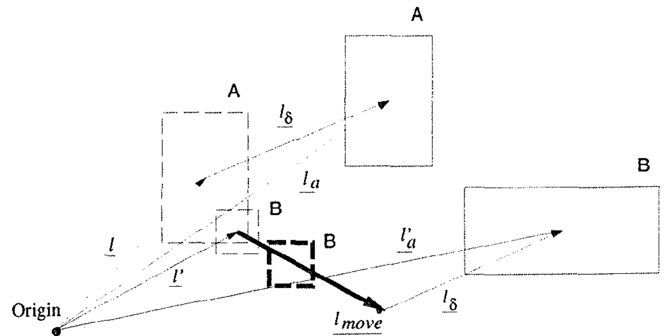

Figure 6: Example of Procedure `Heur-Reconfig`

What happens essentially is that the entire layout of configuration c_a is "virtually" moved so that the location of object A is the same in both c and c_a. This "move" is equivalent to subtracting $\underline{l_\delta}$ from the locations of all objects in c_a, so that one essentially deals with locations in c_a that are "relative" to the location of object A. Thus, object A is considered to be in c in its correct location in c_a and does not move. Any objects

causing overlaps in c are then recursively pushed towards their "relative" locations in c_a so that all overlap may disappear. This will certainly happen eventually, because even if in the worst case all objects need to reach their "relative" locations in c_a, at that point there will be no overlap since c_a is an anchored configuration (directly or indirectly) specified by the user and is therefore guaranteed to satisfy Requirement R1.

Again note how Procedure `Heur-Reconfig` tries to satisfy Requirement R4. Relative to object A, the longest that object B can be pushed is up to its "relative" location in the stored layout of the anchored configuration c_a, which is a layout of the two objects that captures the user's desires. Thus, any partial movement in that direction is likely to result in spatially similar layouts.

4 Generalizations

In this section, we briefly discuss how our approach can be generalized in some useful directions.

4.1 Multiple Kinds of Objects

In Section 2.1, we made the assumption that all objects to be visualized are of the same type, and that was the basis for the description of all our algorithms above. Removing this restriction is straightforward. Consider designs that involve k object types, $k \geq 1$, and must satisfy Requirements R1-R2 with respect to objects of all types. Let m_i be the number of objects of type i currently visualized and n_i the number of visual representation kinds for objects of type i. Then, there are $\prod_{i=1}^{k} n_i^{m_i}$ configurations in $\mathcal{C}$ and $\sum_{i=i}^{k} m_i$ "letters" in the "word" representing a configuration (total number of objects). The rest of the preceding discussion, however, as well as the developed algorithms remain valid for this case as well.

For example, consider an abstract situation where there are two objects to be visualized, one of a type with two representation kinds (denoted by a and b, respectively, with a$\preceq$b) and one of a type with three representation kinds (denoted by 1, 2, and 3, respectively). The lattice formed by all the possible configurations is shown in Figure 7, where the nodes capture configurations and the arrows capture $\preceq_{\mathcal{C}}$.

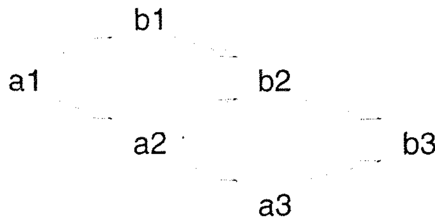

Figure 7: Size-based inclusion lattice of visual configurations with different types of objects

We should also mention that there are often object types that do not need to satisfy Requirements R1-R2. For example, in graphs with nonoverlapping nodes, it may be acceptable for edges to overlap nodes or (cross) other edges. In these cases, objects of the unrestricted types simply do not participate in the layout algorithms discussed above.

4.2 Object Groups

Implicit in the entire discussion of earlier sections has been the assumption that there is a single, given, origin from which the locations of all objects in the layout are measured, e.g., see Figures 5 and 6. There are applications, however, where the set of visualized objects is not flat as above, but is embedded in a hierarchical structure with objects organized in groups, groups organized in higher-level groups, and so on. Each group has its own origin from which the locations of all objects in the group are measured, and behaves as a unit within the higher-level group in which it belongs, i.e., it is considered as a single visual object. As an example, consider the visualization of a tree structure with every subtree defining a group (Figure 8) and the location of its root considered as the origin of all locations for that group. In this case, the hierarchical structure of groups coincides with the hierarchy defined by the tree in a natural way.

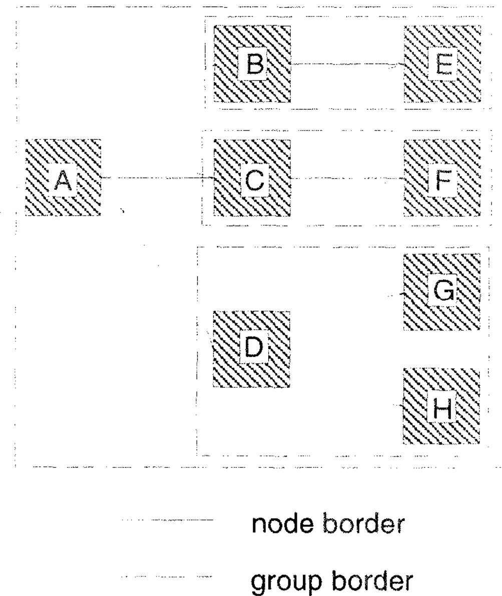

Figure 8: Grouping of subtrees

As a more complex example, consider the visualization of a general graph (i.e., not necessarily a tree) with groups defined as user-specified subgraphs occupying rectangular regions (Figure 9) whose center is considered as the origin of all locations for the corresponding groups. In this case, there is no natural hierarchy defined by the visualized graph, so the group hierarchy is orthogonally defined by the user. This is illustrated in Figure 10, where the graph has striped nodes and solid arcs while the group hierarchy has 3-D boxes and dashed arcs. This form of grouping on graphs is very important in many applications, so we will use it as an example in the remaining discussion.

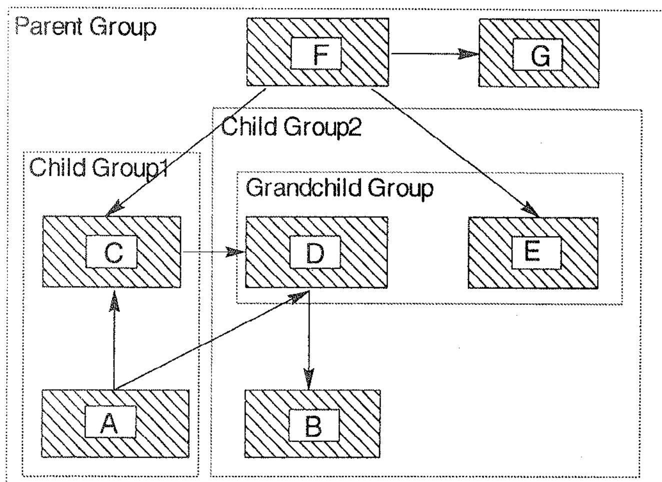

Figure 9: Groups visualized as nested regions

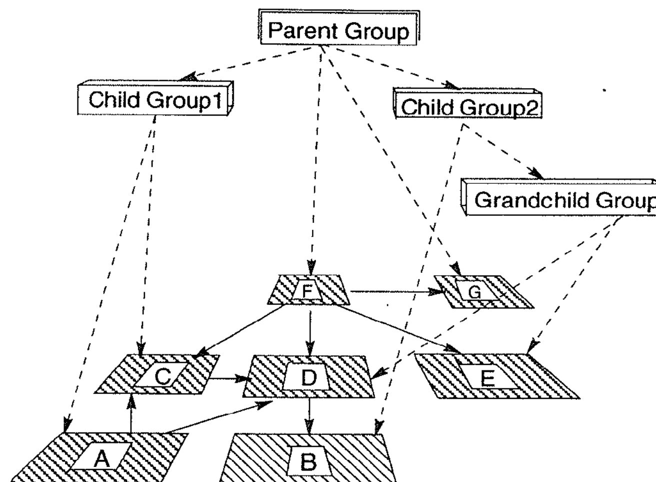

Figure 10: The grouping hierarchy, orthogonal to graph structure

Dealing with multiple visual representations for grouped objects requires only a small extension to our approach of Sections 2 and 3. In particular, each individual group has its own current and anchored configurations of its (immediate) members. At any point, a single group is the focus, and actions on objects *within* the group (object creation, deletion, movement, or visual representation change) cause local layout decisions by following the exact same algorithm described earlier. Of course, many layout changes in a group imply a visual representation change for it within its parent group and, therefore, recursively trigger the configuration-change part of the algorithm for the parent.

A major reason for grouping objects is abstraction, so that users may be able to view the details (members) of a group or simply view a single object representing the group and ignoring its details. This is naturally achieved within our framework by treating groups as first-class objects that must satisfy Requirements R1-R4, and associating with them two visual representations, one "collapsed" and one "expanded", with the obvious meaning. Any change of a group from one representation to the other is simply a configuration change for the parent group and the earlier layout algorithm is directly applicable. Continuing on with the example of grouping of subgraphs of a graph, Figures 11 and 12 show two consecutive collapses of a group and its parent and the resulting layouts.

One complication that arises in the presence of groups is that the user may change group membership or create groups at any level of an existing grouping hierarchy. Both types of actions, especially group creation, may result in extensive changes in group membership and, therefore, configuration structures and location origins. Nevertheless, each such action may be analyzed into individual object creations and deletions, which can then be dealt with directly by the general algorithm.

5 Implementation and Experience

5.1 Implementation

The entire framework and algorithms described above, including the generalizations of Section 4, have been implemented in the OPOSSUM visualization tool [6], which is part of the user interface of the ZOO Experiment Management System [8]. ZOO is a system under development whose goal is to allow scientists from arbitrary disciplines manage all phases of their experimental studies using a single tool. One of the key issues in ZOO is providing a user interface that is primarily visual (so that it is intuitive to users who are not experts in computer science), relatively generic (so that it can be used for different styles of visualization and serve the needs of many disciplines), and able to deal with large numbers of visual objects (which is common in these applications).

OPOSSUM is part of the ZOO interface that deals with visualization of database schemas and also forms the basis for other parts of the interface. It is a generic tool that accepts as input descriptions of a data model, a visual model (capturing the structure of some class of visualizations, e.g., graphs, tables, rectangles contained into each other), and a visual metaphor that maps some of the elements of the visual model to elements of the data model. Thus, given a visualization on the screen the metaphor assigns meaning to its individual visual objects with respect to some underlying schema.

One may define mixed metaphors, where a single concept in the data model may be visualized in different ways, using different concepts in the visual model. The input file describing the metaphor includes information about which types of objects should satisfy Requirements R1-R2 and also a size-based total order $\preceq$ on each one of these object types. The heuristic algorithm of Section 3.2 has been implemented in OPOSSUM in a generic fashion, without any reference to specific types of visual objects or kinds of visual representation. During any OPOSSUM session, all these are instantiated based on the contents of the input model and metaphor

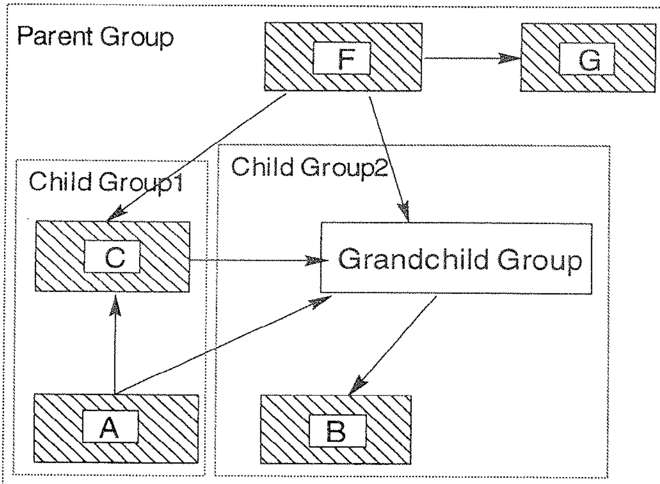

Figure 11: The same graph with "Grandchild Group" collapsed

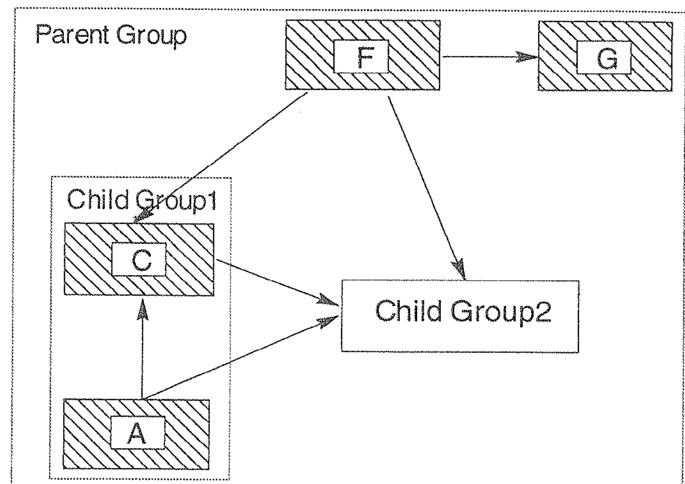

Figure 12: The same graph with "Child Group2" collapsed

files, and the algorithm proceeds accordingly.

We should emphasize that visualizations of graphs with groups defined based on subgraphs in rectangular regions, as in Section 4.2, can be described as a visual model to be given as input to OPOSSUM. Screendumps of the OPOSSUM display showing a large object-oriented schema in graph form, at two configurations with respect to groups being expanded or collapsed, are shown below [6]. Figure 13 presents the schema graph with all groups expanded, while Figure 14 shows the internal structure of two top-level groups allowing easier study of their details.

5.2 Experience

OPOSSUM has been used by scientists in Soil Sciences, Biochemistry, and Physics to design relational and object-oriented schemas for their databases and experiments using a variety of visual models (mostly graph-based). (The schema graphs shown in the previous subsection are from a large simulation experiment on plant grown by the soil scientists.) The feedback in all cases has been very positive for all aspects of the system, including the layouts resulting from the algorithm of Section 3.2. With few exceptions, these layouts have captured what the users expect. In addition, the notion of groups in the input visual models and their collapsed/expanded representation has proved to be extremely valuable in dealing with the very large schemas that these scientists generate.

6 Related Work

To the best of our knowledge, this is the first work that attempts to deal in a generic fashion with multiple visual representations of objects and their layouts so that Requirements R1-R4 are satisfied, especially R4 which calls for taking into account user placement.

On the other hand, there has been considerable work on automated layout of graphs and trees [5, 9, 10]. Only [10] considers user-specified location information, and that is in the context of trees (maintaining user-specified ordering of siblings). The EDGE system [3] uses user-provided information for graph layout, but this is in the form of explicit constraints to restrict or supplement decisions made by the automatic layout algorithm. It requires these constraints for each spatial relationship, whereas we use user-provided information to infer layout guidelines.

Also related is work on showing different amounts of detail in visualizations. Pad++, demonstrates improved approaches to zooming [2]. Fisheye views of graphs [11, 13] allow zooming of one part of a graph more than others. Neither approach offers abstractions when detail is removed. Magic lenses [14] offer different amounts of detail, but without layout management to prevent sparse or overlapping information. Hy++ [4] uses limited abstraction to better visualize graphs describing relationships between instances in a logical database. Groups of instances can be represented by their type class when there is insufficient space to display all the instances. Finally, work has been done on applying multiple-focus fisheye views to allow different parts of a hierarchically organized visualization to be seen with certain groups collapsed and others expanded [1]. The main limitation of this effort is that it only considers visualizations where everything fits on the screen at once, and whenever something gets larger, everything else must get smaller (or collapse). Also, the multiple fisheye lens approach requires huge amounts of processing time (needs two separate workstations to run).

7 Conclusions

We have addressed the problem of user-oriented layout for large collections of objects that can be viewed through multiple visual representations. We have devised a general algorithm that uses a small number of layouts that the user has (directly or indirectly) specified explicitly to derive other layouts that attempt to follow the user preferences thus expressed. The algorithm has been implemented in a system and has been used by real users who have found it quite effective. The future calls for implementation enhancements

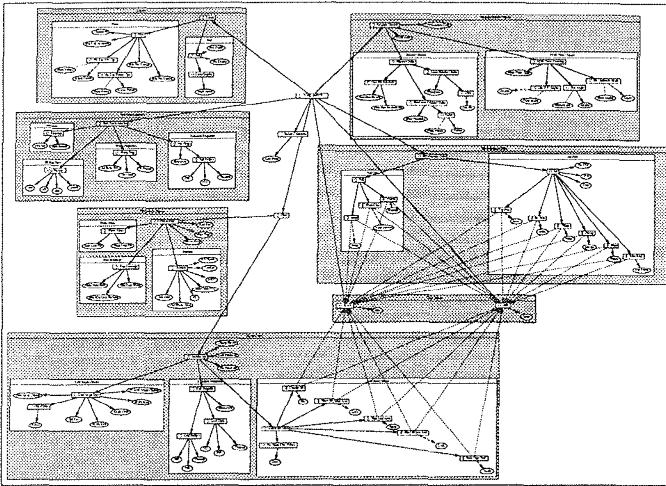

Figure 13: The CUPID input schema, partitioned into Groups

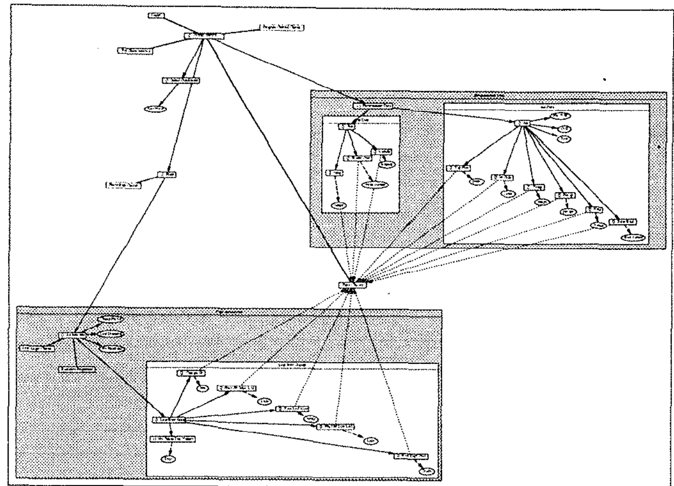

Figure 14: The same schema, partially abstracted

for optimal performance and a more comprehensive usability study of the algorithm.

References

[1] L. Bartram, R. Ovans, J. Dill, M. Dyck, A. Ho, and W. S. Havens. Contextual assistance in user interfaces to complex, time-critical systems: The intelligent zoom. In *Proc. Conference on Graphics Interfaces*, 1994.

[2] B. B. Bederson, L. Stead, and J. D. Hollan. Pad++: Advances in multiscale interfaces. In *Proc. CHI94 Companion to the Conference on Human Factors in Computing Systems*, pages 315–316, Boston, MA, April 1994.

[3] K. Bohringer and F. N. Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proc. CHI90 Conference on Human Factors in Computing Systems*, pages 43–51, April 1990.

[4] M. P. Consens and A. O. Mendelzon. Hy+: A hygraph-based query and visualization system. Technical Report CSRI-285, Computer Systems Research Institute, University of Toronto, June 1993.

[5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, 1994.

[6] E. Haber, Y. Ioannidis, and M. Livny. Opossum: Desktop schema management through customizable visualization. In *Proc. 21st International VLDB Conference*, pages 527–538, Zurich, Switzerland, September 1995.

[7] T. Igarashi, S. Matsuoka, and T. Masui. Adaptive recognition of implicit structures in human-organized layouts. In *Proc. of the 11th Symposium on Visual Languages*, pages 258–266, Darmstadt, Germany, September 1995.

[8] Y. Ioannidis, M. Livny, E. Haber, R. Miller, O. Tsatalos, and J. Wiener. Desktop experiment management. *IEEE Data Engineering Bulletin*, 16(1):19–23, March 1993.

[9] E. B. Messinger, L. A. Rowe, and R. H. Henry. A divide-and-conquer algorithm for the automatic layout of large directed graphs. *IEEE Trans. on Systems, Man, and Cybernetics*, 21(1):1–12, 1991.

[10] S. Moen. Drawing dynamic trees. *IEEE Software*, 7(4):21–28, July 1990.

[11] E. Noik. Exploring large hyperdocuments: Fisheye views of netsted networks. Technical Report CSRI-28, Computer Systems Research Institute, University of Toronto, June 1993.

[12] J. Ousterhout. Corner stitching: A data structuring technique for vlsi layout tools. *IEEE Transactions on Computer-Aided Design*, 3(1):87–100, January 1984.

[13] M. Sarkar and M. H. Brown. Graphical fisheye views of graphs. In *Proc. CHI92 Conference on Human Factors in Computing Systems*, pages 83–91, April 1992.

[14] M. C. Stone, K. Fishkin, and E. A. Bier. The movable filter as a user interface tool. In *Proc. CHI94 Conference on Human Factors in Computing Systems*, pages 306–312, Boston, MA, April 1994.