Smooth Structural Zooming of h-v Inclusion Tree Layouts

Kevin Pulo, Peter Eades, Masahiro Takatsuka
School of Information Technologies
University of Sydney
NSW, 2006, Australia
{kev,peter,masa}@it.usyd.edu.au

Abstract

We present a new paradigm for achieving Focus + Context visualizations called smooth structural zooming, which varies the level of detail of the data in different areas of the visualization, as opposed to geometrically distorting the visualization or employing rapid zooming techniques. A smooth structural zooming technique for horizontal-vertical (h-v) inclusion tree layouts is described and applied to the domain of the software design process, specifically, Design Behaviour Trees (DBTs). This system has the ability to navigate and explore data too large to be fully displayed, whilst maintaining an approximately constant level of visual complexity, good visualization aesthetics and preservation of the user's mental map through animation. The technique may be readily extended to arbitrary layout styles and algorithms, and to other hierarchical data structures and relational information, such as clustered graphs.

Keywords—**Focus + context, smooth structural zooming, inclusion tree layout, visual complexity, level of detail, mental map, animation, software design process**

1 Introduction

The ability to effectively visualize very large amounts of relational information is becoming increasingly important. The size of datasets is increasing rapidly, whilst the majority of computer displays are around the single megapixel mark, with multi-megapixel large-scale displays being expensive and cumbersome, despite becoming more prevalent. In addition, the amount of bandwidth available to the human perceptual system is limited. Both of these reasons mean that the amount of data which can be effectively visualized at any given time is limited. This is a fundamental problem in information visualization known as the detail-context tradeoff — in any fixed size display only small amounts of information can be displayed at high detail, resulting in a lack of context (and vice-versa).

This problem is generally resolved by using *geometric zooming* techniques, which fall into two broad categories, distortion and rapid zooming. Distortion techniques include Focus + Context techniques such as the fisheye lens, hyperbolic browser and perspective wall [3]. These involve displaying a central *focus region* at full magnification, so that details in the data may be easily seen, and a surrounding *context region* at lower magnification, so that only a general, high-level structure of the data is seen. The user investigates and explores the data by moving the focus region, and retains their overall location in the visualization by using the context region. Rapid zooming techniques include "Zooming User Interfaces", such as Jazz [1] (and its predecessor, Pad++) and GeoZUI3D [15] [14]. Rapid zooming attempts to utilise the user's mental map and short term memory to provide context, by only showing a high detail focus region but allowing users to very quickly and easily zoom out to a low-detail context view and then zoom back in to the high-detail region of interest.

We present an alternate method of performing focus + context called *smooth structural zooming*, where context information is summarised or abstracted rather than being distorted or rapidly accessible. Smooth structural zooming has the ability to navigate and explore data too large to be fully displayed, whilst maintaining an approximately constant level of visual complexity, good visualization aesthetics and animation to preserve the user's mental map.

We illustrate the use of smooth structural zooming by applying it to the particular case of h-v inclusion tree layout visualizations. Section 1.1 gives a brief introduction to inclusion tree layouts. Section 2 explains the concepts of smooth structural zooming at a general level, followed by the specific application to h-v inclusion tree layouts in Section 3. Section 4 contains anticipated future enhancements and improvements to smooth structural zooming.

1.1 Inclusion tree layouts

We are interested in applying this technique to relational information which can be modelled by graphs, and in par-

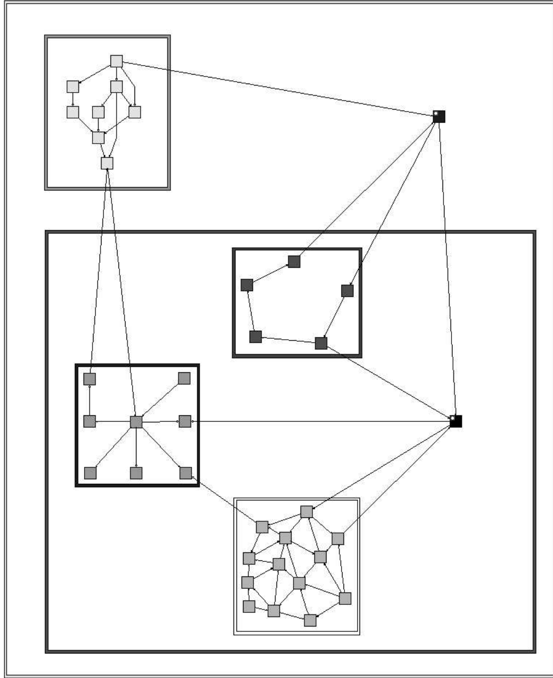
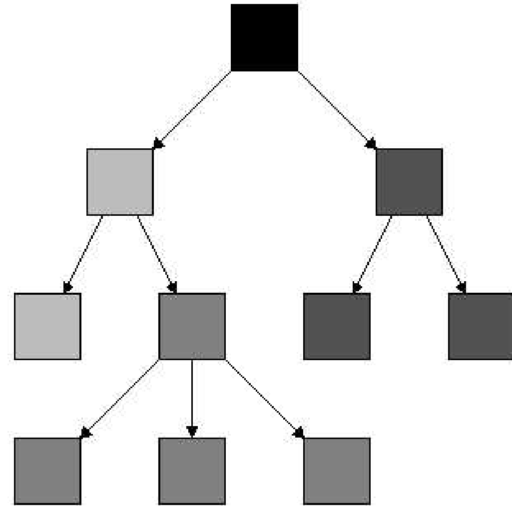
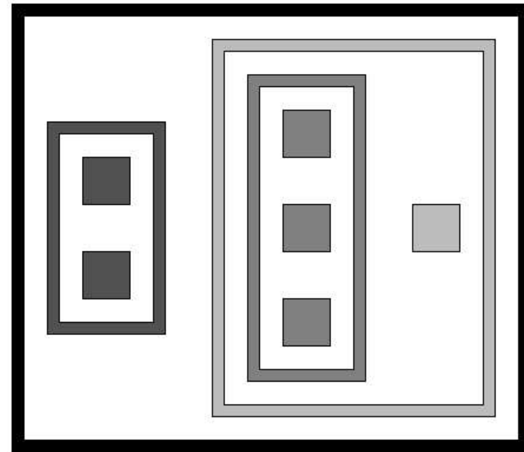


Figure 1. An example of a clustered graph.



(a) Classical Layout Convention



(b) Inclusion Layout Convention

Figure 2. An example tree in classical and inclusion layout conventions.

ticular *clustered graphs*, which support varying levels of detail by defining a recursive clustering of related nodes. Clustered graphs are most often visualized by drawing the contents of each cluster inside a rectangle representing that cluster, as shown in Figure 1. This allows clusters to be summarised by simply drawing the cluster rectangle without its contents. Although we would like to apply smooth structural zooming to clustered graphs, at this initial stage of our investigation we instead consider the simpler case of smooth structural zooming of inclusion tree layouts, which are effectively clustered graphs with no edges.

The *inclusion tree layout convention* [5] is an alternate method of drawing trees where the parent–child relationship is visually represented by the child node being completely contained within the parent node. For simplicity, nodes are usually represented as rectangles. The familiar *classical tree layout convention* draws the tree in a "level" fashion, where the y coordinate of a node is proportional to its depth k from the root, with lines drawn between the child and parent nodes. Figure 2 illustrates an example tree in both the classical and inclusion conventions.

In addition, the inclusion tree layout convention is similar to *treemaps* [8], a space-filling technique for drawing trees in the plane. Figure 3 shows an example treemap of the tree shown in Figure 2. Treemaps tend to be used more commonly where some statistical data is associated with the nodes, and treemap algorithms are geared towards using this

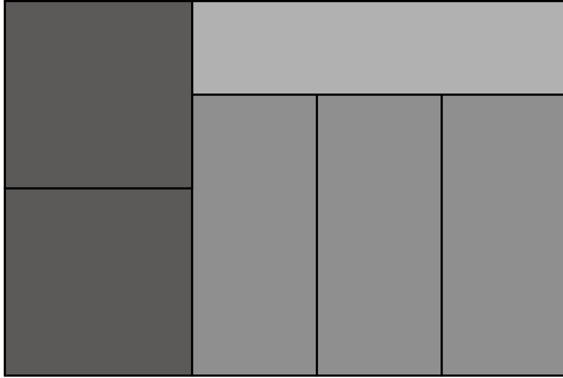


Figure 3. Treemap for the tree shown in Figure 2.

data when computing the layout. We use inclusion layouts as we are more concerned with the structure of the nodes, however, since treemaps can be considered to be inclusion trees with no margins around the internal nodes, the ideas presented in this paper are also applicable to treemaps.

One disadvantage of the inclusion tree layout is that it does not scale well to very deep trees. It can require exponential area (or exponentially small resolution) in terms of the number of nodes, which means that in a practical sense it is not very useful for trees with depth greater than about 4 or 5. Thus it is very amenable to a dynamic navigation system such as that provided by smooth structural zooming. In addition, the hierarchy of the tree allows for natural summarising of context information by displaying the appropriate non-leaf node, rather than the full structure of the sub-tree below the non-leaf node.

Our smooth structural zooming method for inclusion trees involves allowing the user to expand nodes to reveal their children nodes, with the system compensating by collapsing the least recently used expanded node. The system can also zoom in to deep trees when necessary, and uses animated transitions to preserve the user's mental map when adjusting the layout of nodes in the display.

This technique is similar to that employed by Spacetrees [11] and Degree of Interest Trees (DOI Trees) [2], except we are concerned with inclusion trees rather than classical node-link trees. We do not consider the performance of our technique compared to classical tree navigation systems such as Spacetrees or DOI Trees. This is because inclusion trees are required in order to support clustered graphs, but classical tree systems are generally not appropriate for the visualization of the hierarchy of clustered graphs. Although DOI Trees allow other edges to be shown in addition to the main hierarchy edges, the relevant edges are only visible when the user points at a node, and the edges aren't used at all in the layout of the tree. Nevertheless, the experimental evaluation of Spacetrees [11] is encouraging, as it supports the ideas of summarising the context information and animating view transitions. A thorough evaluation of smooth structural zooming will be carried out at a more appropriate stage of the project, such as when it has been applied to clustered graphs, rather than at this early stage.

2 Smooth structural zooming

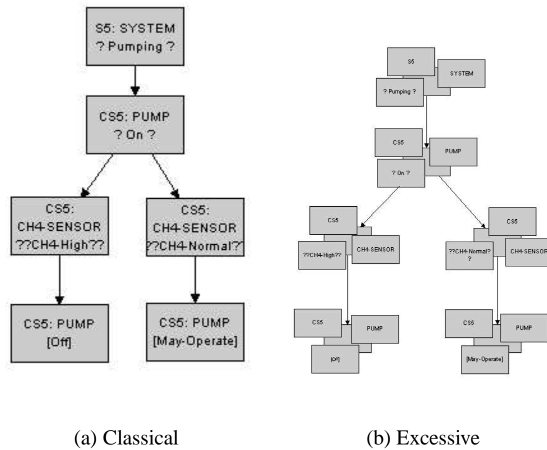
Smooth structural zooming aims to facilitate the user's exploration of data by providing interactive 'structural zooming'. Structural zooming differs from the more common 'geometric zooming' techniques by showing different parts of the data at different levels of detail, rather than geometrically distorting the visualization. Smooth structural zooming is concerned with performing structural zooming in a fashion which preserves the user's mental map whilst navigating through the data. In particular, the specific requirements of smooth structural zooming are:

- changing the level of detail, that is, which parts of the overall data are to be displayed,
- display the data without distortion, while still allowing the user to 'zoom' or concentrate on specific areas,
- preservation of the user's mental map between visualizations of different levels of detail,
- constant level of visual complexity, and
- consistently good layout and presentation of the data.

2.1 Detail and visual complexity

We say that a visualization has an intrinsic *level of detail* and a *level of visual complexity*. The level of detail (or simply 'detail') indicates the amount of data which is present in the given visualization, while the level of visual complexity (or simply 'visual complexity') indicates how many visual elements or attributes are being used to present this data. The greater the amount of data displayed, the greater the detail, and similarly for visual complexity. Detail is a double-edged sword — one needs detail in order to be able to gain insight into the data, yet too much detail results in a lack of available screen space and poor resolution (in terms of the space allocated to each data element). This is the crux of the classic detail–context tradeoff.

Detail may be quantified by a measure, for example, the number of nodes or leaves (for tree data), and similarly visual complexity may be quantified, for example, by the number of graphics primitives used. Different visualizations of the same data at the same level of detail may



(a) Classical (b) Excessive

Figure 4. A small tree (a section of the DBT from Figure 5), shown using classical notation and an excessive notation. The same data is present in both, but the excessive notation has a higher visual complexity.

have different visual complexities. For example, Figure 4 shows a small tree using a classical notation and a notation which is deliberately excessive. The same data is present in both visualizations, but the excessive notation clearly has a higher visual complexity. Nevertheless, for given data there are bounds on the possible complexities. In particular, we consider the visual complexity of a higher detail visualization to always be greater than the visual complexity of any lower detail visualization. More formally, if $v(i, j)$ is the visual complexity at detail i with visualization technique j, then we have $v(i, a) < v(j, b) < v(k, c)$, for varying levels of detail $i < j < k$ and all arbitrary visualization techniques a, b, c.

2.2 Navigation technique

Our goal is a system which maintains an approximately constant level of visual complexity, while allowing the user to visualize and navigate relational data that would ordinarily require a much higher visual complexity. Note that distortion techniques, such as the fish-eye lens, have higher visual complexity — rather than drawing all of the context data in a distorted fashion, it is better to summarise that context data, giving a lower visual complexity.

Since the user is the driving force behind the investigation of the data, the system must provide operations will support the user in directing the navigation through the data. These *user operations* allow the user to choose the data that is to be included in the visualization, but currently isn't. That is, they are increasing the detail of the visualization (*detail-increasing operations*), and therefore by necessity also increase the visual complexity. It is the role of the system to maintain the approximately constant level of visual complexity. This means that the system must reduce the detail and visual complexity in response to the user's increases. In particular, for every type of detail-increasing operation available to the user, a corresponding inverse *detail-reducing operation* must be available to the system.

When the user increases the detail by performing an operation (the *stimulus*), the system must determine the:

- *Detail reducing condition*: If the detail is now too great as a result of the stimulus. The simplest method for determining this is if the detail measure has risen above some pre-determined threshold value.

- *Response*: If the detail reducing condition is true, how the system should respond to bring the detail back down to acceptable levels — without disturbing the user's navigation or investigation process. Notably, the response cannot include the inverse operation of the stimulus.

It is also possible to allow the user access to perform the detail-decreasing operations. The user will arguably always have a better idea of their overall goal than the system can, and thus may choose to anticipate a better "response" to a stimulus they are yet to perform. For example, prior to expanding a node the user may choose another large expanded node to manually collapse, in order to make space for the node to be expanded, and to ensure that the particular node they have selected is collapsed (rather than whichever node is selected by the system). The system doesn't respond to any detail-decreasing operations performed by the user (even though the symmetric thing to do is increase the detail somehow).

The layout of the visualization must be updated as the user changes their view of it, usually by using some sort of layout algorithm for the data involved (for example, a graph drawing algorithm). This is because as with any visualization, the quality must be maintained — that is, the visualization must have good aesthetic properties, be understandable, facilitate insight, and so on. As the data being visualized is changing, so too must the layout change in order to accomodate this.

As these changes in visualized data and layout are (generally) discrete operations, care must be taken to ensure that the user always experiences smoothly animated transitions between the different views. In addition, simple animation techniques such as linear interpolation may not be sufficient [6], and so layout specific animations are required for each type of operation which may be performed. It is preferable

for the animation of an inverse operation to be the time-reversed animation of the original operation, but this is not essential.

Sometimes the system may need to perform more than one operation in order to bring the visualization into an acceptable state. For example, both a detail-increasing and a detail-reducing operation may need to be performed. In this case, there is a choice between animating these operations *consecutively* or animating them *concurrently*. When animating consecutively, there is the additional issue of the order in which the animations should be performed. Animating consecutively can be confusing because when an animation ends, the user is not sure if another will be starting or not, and may attempt to continue navigating only to find that another animation has begun. This may be alleviated by increasing the speed of the animations such that the total animation always takes some fixed amount of time, however this isn't feasible if there are many animations to be performed. Animating concurrently tends to be more visually appealing, as users can still follow the movement of the nodes when several animations are occuring. However it can get confusing if there are more than approximately 3 animations being performed concurrently, and so in these cases a combined or hybrid approach is expected to be best, where a strategy is used for choosing groups of animations to be performed concurrently, and these groups are then animated consecutively. Currently our system performs all animations concurrently, as there is a maximum of 3 possible animations at any given time.

3 Application to inclusion tree layout

We now apply the smooth structural zooming concepts from Section 2 to the case of tree visualization and navigation. In particular, we examine the inclusion tree layout convention, due to its pivotal role in the visualization of clustered graphs. In Section 3.1 we introduce the sample data used, that of design behaviour trees (DBTs). This is followed in Section 3.2 by details of the inclusion tree layout algorithm used. Finally, Section 3.3 describes the application of smooth structural zooming to inclusion tree layouts.

3.1 Design behaviour trees

In the field of software engineering, one of the challenges presented by the ever-increasing size and complexity of modern software systems is that of designing such software systems from the ground up in an efficient and error-free way. Visualization has often played an important role in the software design process, for example, data-flow diagrams and Unified Modelling Language (UML) [13], and at a somewhat lower level, flowcharts and Nassi-Shneiderman

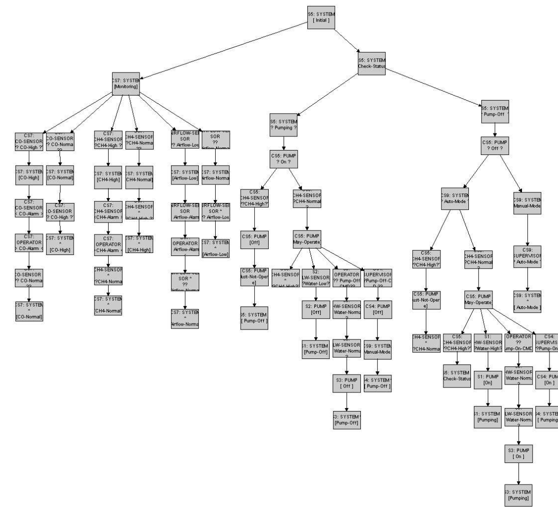


Figure 5. The Mine Pump DBT using a classical layout convention.

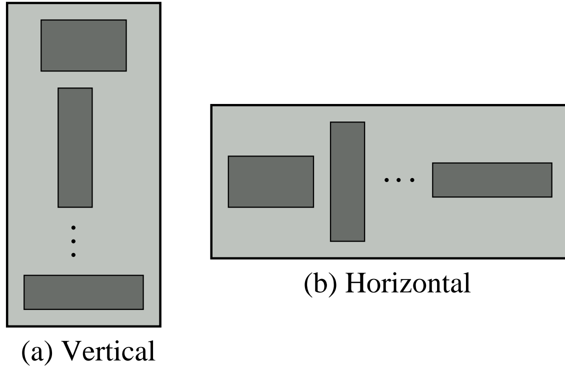
diagrams [10]. However, these visualization techniques have very rarely explicitly considered the non-trivial task of scaling up to very large software systems.

One technique which does address this concern is that of *design behaviour trees*, or DBTs [4]. In this paradigm, the system designer creates behavioural models of small, individual parts of the system. A process called *genetic software engineering* is used to merge these individual DBTs into a large overall DBT for the entire system. This is then used as a basis of the software architecture, allowing the system to be built directly from its functional requirements, rather than the more traditional activity of building a system which satisfies those requirements. This helps to support the design of large software systems, but doesn't address the problem of visualizing large software architectures. In fact, the overall software design DBT can easily be too big to completely visualize on-screen, even if a large-scale multi-megapixel display device is used, and this is our motivation in visualizing them using smooth structural zooming for inclusion trees.

The size of a DBT depends on the size of the software system it describes, and current examples range from around 20 nodes to several thousand. A typical DBT is shown in Figure 5. This DBT describes the operation of a software system controlling a water pump in a mine.

3.2 Inclusion tree layouts

The formal definition of an inclusion layout for a tree T is a rectangle R_u in the plane $\Re^2$ for each node u of T, such that



(a) Vertical

(b) Horizontal

Figure 6. The two different types of node arrangements considered.

- if u has a child w then R_w is within R_u, and
- if u has children v and w then the rectangles R_v and R_w do not overlap and are separated by a distance of at least δ.

We are interested in inclusion layouts that have a small overall size given sizes of the leaf nodes. This is because, in practice, nodes must contain text and the available screen space is limited.

When evaluating the size of a rectangle we use the *minimum enclosing square* size measure, which gives a size of $\psi(x, y) = \max(x, y)$ for a rectangle of width x and height y. Empirical results suggest that in practice this size measure gives good results for inclusion layouts [12].

The fundamental problem for inclusion layout is as follows:

> **Minimum Inclusion Layout Problem** (MILP): Given a tree T and a width X_v and height Y_v for each leaf v of T, find a minimum size inclusion layout for T such that for each leaf v, the dimensions of R_v are $X_v \times Y_v$.

If we consider the tree in which every non-root node is a leaf, we can see that MILP is equivalent to a 2 dimensional bin packing problem, and is thus NP-hard [9]. However, this can be avoided by allowing only two possible ways of arranging the children of a node, horizontal and vertical, as shown in Figure 6, called *h-v arrangements*. In this restricted case of h-v arrangements (and integer node dimensions), we use a dynamic programming approach which solves MILP in polynomial time [5]. Figure 7 shows the result of applying this inclusion tree layout algorithm to the Mine Pump DBT from Figure 5.

We observe that inclusion trees tend to be poor at visualizing chains of nodes with (out) degree of 1, as the nested

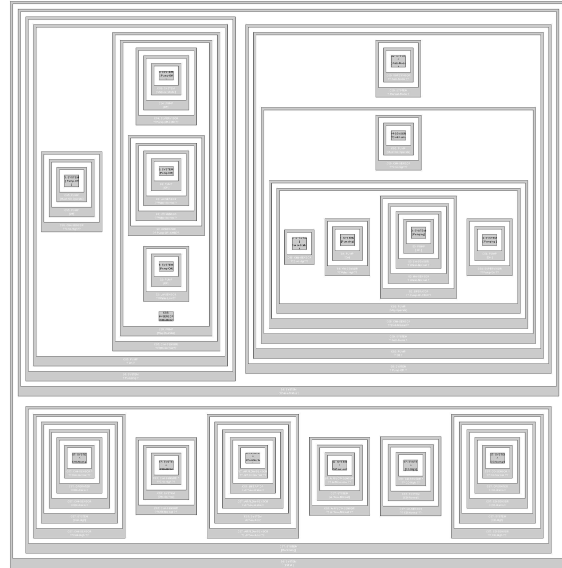


Figure 7. The Mine Pump DBT using an inclusion layout convention.

rectangles and margins waste screen space and add unnecessary complexity to the visualization. A better solution for the inclusion layout convention in this case is to 'compress' each of these chains of nodes into a single representative node. A visual cue such as a gradient may be applied to the representative node, informing the user that some information has been compressed in order to improve the visualization. In addition, the representative node may contain a summary of the text from the compressed nodes. Figure 8 shows the results of applying chain compression to the Mine Pump DBT from Figure 7. From Figure 8 we can see that the inclusion layout is easier to understand with chains compressed, although a visual cue would be useful to regather some of the lost information and no text summaries have been generated for the representative nodes.

3.3 Smooth structural zooming technique

This section describes the application of our navigation technique to the specific case of inclusion layout trees.

The detail measure used is the number of leaf and collapsed nodes visible, although a different measure could be used, most notably the number of nodes (including internal non-leaf nodes). However when exploring deep trees while using the number of nodes as the detail measure, much of the screen space is used by non-leaf nodes, which tend to increase the visual complexity and clutter the display. For this reason we use the number of leaf and collapsed nodes, despite the two measures being very similar.

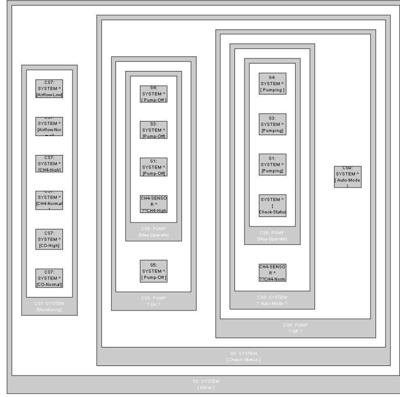


Figure 8. Results of applying chain compression to the Mine Pump DBT shown in Figure 7.

The detail-increasing operations available to the user are:

- expanding a node, revealing its child nodes and increasing its size to accomodate these children, and
- zooming out a level, allowing the user to see parts of the tree previously obscured by the system zooming in.

The detail-decreasing operations available to the system are:

- collapsing a node, hiding its children and returning to its original size, and
- zooming in a level, obscuring outer regions of the tree (generally containing collapsed nodes), allowing the user to concentrate on the central region of expanded nodes.

Zooming is handled by keeping a current *pseudo-root* node. The system ensures that the pseudo-root node (and thus everything inside it) is visible on the screen by scaling the visualization such that the pseudo-root node fills the screen (preserving aspect ratio). Zooming out by a level involves changing the pseudo-root to be the parent node of the current pseudo-root. Zooming in by a level moves the pseudo-root to its child which is an ancestor of the most recently expanded node. Thus the pseudo-root is always an ancestor of the most recently expanded node.

The response of the system to zooming out:

- cannot include zooming in (as this would be idempotent), but
- may include the collapsing of any nodes.

The response of the system to the expanding of a node:

- cannot include the collapsing of that node or any of its ancestors, but
- may include collapsing of any other node, and
- may include zooming in one or more levels.

There is an important asymmetry between the two detail-decreasing operations available to the system. Collapsing a node does not lower the *data availability*, since after performing the operation, all of the other nodes are still available to be expanded, as is the just-collapsed node (and zooming out is still possible). However, zooming in does lower the data availability, because now some nodes may be outside the visualization, precluding their being expanded. Thus the user-controlled zoom out operation is absolutely essential to allow the user to return to those hidden nodes at some later point.

In response to expanding a node, the system must decide which nodes to collapse (if any), and how many levels to zoom in (if at all). If there are expanded nodes available to be collapsed, the system will consult a queue of nodes in order to find the least recently used (LRU) nodes. As many nodes are removed from the queue and collapsed as is necessary to reduce the detail to an acceptable level. After a node is expanded, it is added to the end of the queue, followed by its ancestors (in order). This means that when collapsing nodes, the deepest possible node (respecting the LRU queue) which reduces the detail sufficiently will be used. However, if there are no nodes available for collapsing (that is, all expanded nodes are ancestors of the currently expanding node) then zooming in is the only recourse, and so the system should zoom in as many levels are necessary to sufficiently reduce the detail. However, if this is the only situation in which the system zooms in, the user is unable to zoom in on (for example) two expanded siblings which may be of interest. This can be solved by:

- Allowing the user to manually zoom-in. This solution is somewhat inelegant, but adequate.
- Having the system zoom in earlier, while some nodes are still expanded. However, deciding when to collapse and when to zoom in, while both are possibilities, is a non-trivial task. The most promising possibility appears to be limiting the size of the LRU queue to some minimum size, which would allow some nodes to remain expanded whilst the view is zoomed in. However, it is complicated by the ancestors stored in the LRU queue and the possibility of zooming in past one or more of the expanded nodes.

When zoomed in, the layout outside the pseudo-root node is fixed, while the layout inside it is updated and adjusted as usual. This helps to preserve the user's mental map, as

hidden parts of the visualization don't change appearance whilst out of view. There are two obvious methods for updating the layout inside the pseudo-root node:

1. Recompute the layout for the entire tree, but only update the positions of descendants of the pseudo-root.

2. Recompute the layout as though the pseudo-root is the overall root of the tree.

Put another way, although only the descendants of the pseudo-root are updated, recompute the layout using either the actual root or the pseudo-root of the tree as the root of the layout.

The first alternative has the major disadvantage that the quality of the layout of the pseudo-root may be poor when taken on its own, as the layout algorithm optimises the layout of the overall tree. For example, the pseudo-root may have an extreme aspect ratio (compared to the desired aspect ratio), because the inclusion layout algorithm has optimised the layout so that the overall tree has a good aspect ratio.

The second alternative has the disadvantage that when the pseudo-root is used as the root the resulting layout may be quite different from previous layouts, resulting in large layout changes when zooming in. However, the animation used helps to alleviate this problem somewhat, and the advantage of having an optimal layout far outweighs this disadvantage. It also makes sense to use the pseudo-root as the root of the layout computation as it is the root of what the user can see. For these reasons our system uses the pseudo-root as the root of the layout computations.

3.4 Animation

An animation exists for each type of operation that can be performed, as well as for layout updates. Expanding is performed by linearly interpolating the size of the collapsed node between its original collapsed size and its expanded size, followed by drawing the child nodes. Collapsing is similar, the expanded node's children are removed and the size of the node is then linearly interpolated to its original collapsed size. In both cases, changes in node sizes affects other nodes such that no occlusions occur, for example, as a node is expanded its ancestor nodes are also expanded as necessary. A useful improvement would be to scale the size of the node whilst it was expanded and its children visible, however technical limitations prevented such an animation in this early work. Zooming in and out is simply achieved by linearly interpolating the clipping rectangle of the display between the old pseudo-root and the new one.

Adjusting the layout requires changing the arrangement of one or more nodes from horizontal to vertical, or vice-versa. This is achieved by "rotating" the children nodes

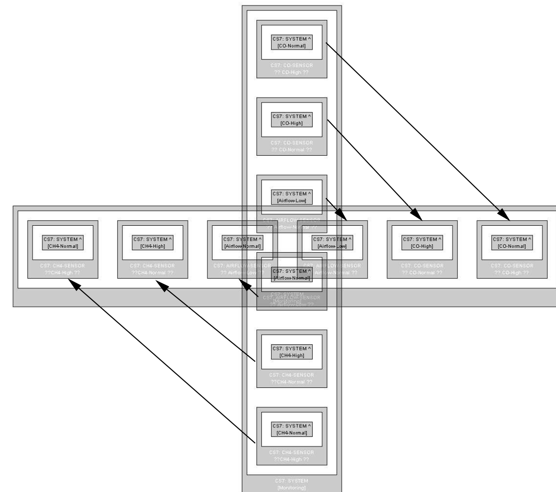


Figure 9. Rotating the child nodes to change a node arrangement from vertical to horizontal.

about the center of the node in question, as illustrated in Figure 9. Figure 10 illustrates the three different types of rotation our system uses:

1. *linear*, which simply linearly interpolates the positions of the children from their initial location to their final location,

2. *circular*, which interpolates the positions of the children along an elliptical arc, and

3. *orthogonal*, which interpolates the positions of the children along a "Manhattan path".

Linear and circular are subject to occlusions between moving siblings, although circular is not as prone to it. Orthogonal requires the nodes to traverse a longer distance, and can cause the size of the node to increase considerably during the animation, but it has the advantage of avoiding occlusions altogether. Somewhat surprisingly, when occlusions do occur, they don't appear to be a major hindrance to following the action of the nodes.

The video accompanying this paper [16] shows the animations used by the system on the Mine Pump DBT from Figure 7. It shows a navigation through the DBT, illustrating the main features of the smooth structural zooming system. The compressed version of the Mine Pump DBT from Figure 8 is not used, as it is not deep enough to exhibit zooming in. The detail measure used is the number of nodes present, again, this is to show the zooming in operation. This detail measure has the side-effect that some nodes are

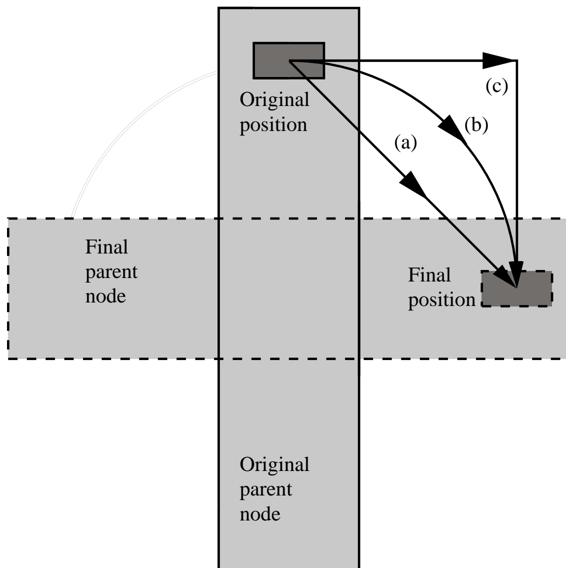


Figure 10. The different types of rotation available for updating the layout of a node, (a) linear, (b) circular and (c) orthogonal. The node is moved from its original position to its final position along one of the indicated paths.

closed immediately after being opened in the video, however this occurs much less frequently when the more usual number of leaf and collapsed nodes detail measure is used.

3.5 Image gallery

We now present an image gallery showing the effects and results of the main operations of the system. In addition to showing the actions of the system, these images also reinforce how valuable the animation is in smooth structural zooming. When "jumping" directly from one image to the next (as would be the case if animation were not present), considerable thought and explanation is required to determine the changes that have taken place. However when viewing the accompanying video [16] it is much easier to follow the changes being performed on the tree.

Figure 11 shows the effect of expanding the right of the two bottom-most collapsed nodes. We first observe that the layout has changed such that the large expanded section is now laid out vertically approximately in the centre of the visualization, and its sibling is also now laid out vertically on its right. In addition, we notice that the large expanded section in the upper right area previously had no collapsed nodes, but now has 4 collapsed nodes. Both displays have a similar number of nodes, and the layout has been updated in order to keep the new display from becoming larger.

Figure 12 shows the effect of expanding the left-most collapsed node. In this case, the node expands to reveal 6 children, laid out vertically. However, the expanded nodes on the right hand side have had to collapse considerably in order to make sufficient space for this. The right expanded node has collapsed completely, whilst the children of the left one have collapsed, and its arrangement has changed from horizontal to vertical. Again, the displays are similar in detail and size.

Figure 13 shows the effect of zooming into the visualization. Figure 13(a) shows what the visualization would look like if zooming in was not used as a response, while Figure 13(b) shows the actual visualization obtained with zooming in enabled. We observe that there are less on-screen nodes, and those present are larger, more easily readable and closer to the central node which was expanded. In addition, the layout of the node containing 4 children is horizontal rather than vertical, which is due to the layout being computed only inside the pseudo-root node, rather than over the entire tree.

4 Future work

This paper describes initial work into smooth structural zooming, a new paradigm for Focus + Context display of relational data. There are many enhancements and and continuations of this work which we are currently investigating; some of the more interesting and promising are presented in this section.

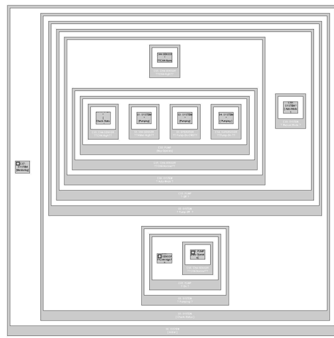
4.1 Layout style and algorithm independance

The current system is heavily based on the MILP dynamic programming algorithm, in particular the layout rearranging animations only work for h-v layouts. It would be very useful if an animation system could be developed which was independant of the particular style of layout and algorithm being used. This would allow much more flexible algorithms, such as [7], to be used. Smoothly animating between arbitrary layouts is non-trivial, although the graph animation techniques presented in [6] are expected to be useful.

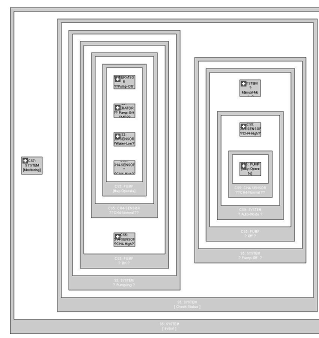
The current animation techniques also don't allow for the possibility of changing the order of the nodes within the horizontal or vertical layout. It is expected that results from the area of sorting algorithm animation will be useful for this.

4.2 Clustered graphs

Clustered graphs are a particular type of graph which additionally have a cluster hierarchy tree imposed on the

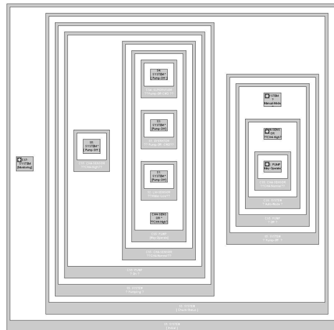


(a) Before

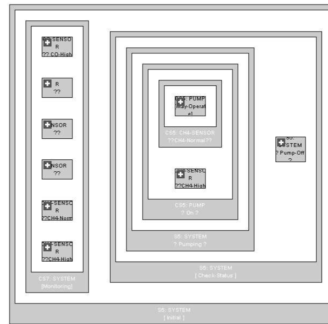


(b) After

Figure 11. Expanding the right of the two bottom-most collapsed nodes.

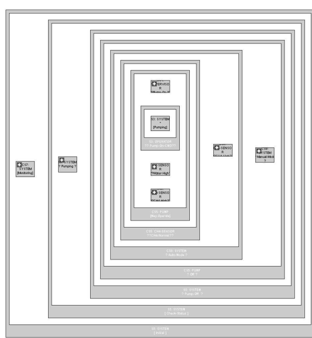


(a) Before

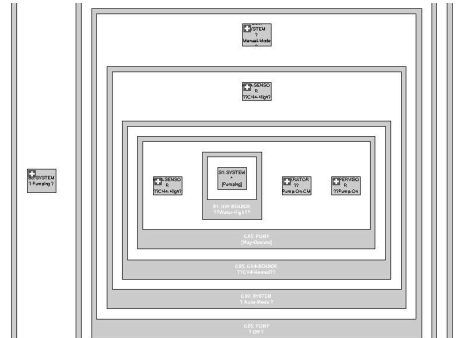


(b) After

Figure 12. Expanding the left-most collapsed node.



(a) No zoom in



(b) Zoom in

Figure 13. The effect of zooming in.

nodes. They are generally drawn with clusters of nodes inside their parent node, similarly to the inclusion tree layout, with edges between the nodes routed by means of a clustered graph drawing algorithm. At a simplistic level, they are the same as inclusion tree layouts, with the addition of edges between nodes. As a result, extending this smooth structural zooming technique to clustered graphs would allow many more types of diverse data to be used.

4.3 Layout stability

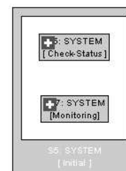
Sometimes when expanding nodes (such as a set of siblings of similar size), the layout algorithm changes the arrangement of a fairly high level node back and forth between horizontal and vertical, which can be very distracting and disruptive to the user's mental map construction. Possible solutions include allowing the layout to only change on screen once the quality of the layout is substantially improved, rather than marginally improved; or using a layout size measure which incorporates the size of the current layout, allowing the notion of the "optimal" layout to be modulated by candidate layout's similarity to the current layout (either in terms of the h-v arrangements of nodes or dimensions of the nodes).

4.4 Initial expansion

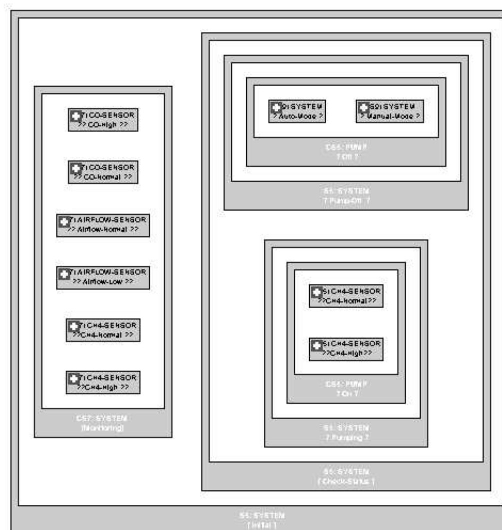
Currently, the initial display presented to the user is simply the root node in expanded form. This is because clearly the user will always want to navigate into the root node, and so presenting the root node in collapsed form is unnecessary. However, a better system would be to initially expand the tree as much as possible. That is, perform a breadth first search (BFS) on the tree, expanding nodes while the detail is not too large. The user should not see this expansion occuring, giving the user a better idea of what the tree is like when they initially load it, rather than having to manually explore the nodes near the root. The BFS may be such that each layer is expanded simultaneously (that is, layers of the tree are never partially expanded), and the BFS stops when the expansion of a layer would cause the detail to be too high; or else it may be such that nodes are expanded individually, stopping only when expanding any node would cause the detail to be too high. Figure 14 shows an example of the difference between only expanding the root node, which gives poor initial user orientation, and expanding several levels (in this case, aiming to have 10 leaves present), giving the user a better idea of the structure of the tree in the levels close to the root.

4.5 Mouse cursor warping

When the user is navigating through the tree, they must often "chase" nodes with the mouse as the nodes move with



(a) Only the root expanded



(b) Several levels expanded

Figure 14. The difference between initially expanding only the root node and expanding several nodes.

layout changes. For example, when the user clicks on a node to expand it, if the node moves away from its initial position then the user may no longer be pointing at that node, even though they have not moved the mouse and are probably still interested in that node (as it was just expanded). It would be good if the system could "warp" the position of the mouse cursor so that it followed the movement of the node to its final position. The transformations applied to each node in the display could also be applied to the mouse cursor, causing the mouse to move along with the nodes, keeping the same relative position to the nodes. The user should still be able to move the mouse while this is occuring, although this may be problematic if the user is required to "fight" the system's movement of the mouse.

5 Conclusion

We have presented a new paradigm for Focus + Context called smooth structural zooming, and an application of it to horizontal-vertical (h-v) inclusion tree layouts (in the context of Design Behaviour Trees). This paradigm allows the level of detail shown in different regions of the visualization to be varied by summarising or abstracting the data, rather than geometrically distorting the visualization or using rapid zooming to make obscured regions quickly accessible. It has the ability to navigate and explore data too large to be fully displayed, whilst maintaining an approximately constant level of visual complexity, good visualization aesthetics and preservation of the user's mental map through animation. Although our inclusion tree layout application is currently specific to h-v layouts, it can be generalised to arbitrary layout styles and algorithms, and to other hierarchical data structures and relational information, most notably clustered graphs.

References

[1] B. Bederson, J. Meyer, and L. Good. Jazz: an extensible zoomable user interface graphics toolkit in java. In *Proceedings of User Interface and Software Technology (UIST 2000)*, pages 171–180. ACM Press, 2000.

[2] S. Card and D. Nation. Degree-of-interest trees: A component of an attention-reactive user interface. In *Proceedings of Advanced Visual Interfaces*. Trento, Italy, May, 2002.

[3] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization (Chapter 4)*. Morgan Kaufmann, 1999.

[4] R. G. Dromey. Genetic software engineering - simplifying design using requirements integration. In *Proceedings of IEEE Working Conference on Complex and Dynamic Systems Architecture, Brisbane*. IEEE, December 2001.

[5] P. Eades, T. Lin, and X. Lin. Two tree drawing conventions. *International Journal of Computational Geometry and Applications*, 3(2):133–153, 1993.

[6] C. Friedrich. *Animation in Relational Information Visualization*. PhD thesis, University of Sydney, 2002.

[7] T. Itoh, Y. Kajinaga, Y. Ikehata, and Y. Yamaguci. Data jewelry-box: A graphics showcase for large-scale hierarchical data visualization. *IBM Research, TRL Research Report, RT0427*, 2002.

[8] B. Johnson and B. Shneiderman. Tree-maps: A space-fi lling approach to the visualization of hierarchical information structures. In *Proc. IEEE Visualization '91*, pages 284–291. IEEE, 1991.

[9] S. Martello and D. Vigo. Exact solution of the two-dimensional fi nite bin packing problem. *Management Science*, 44(3):388–399, 1998.

[10] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *ACM SIGPLAN Notices*, 8(8):12–26, 1973.

[11] C. Plaisant, J. Grosjean, and B. Bederson. Spacetree: supporting exploration in large node link tree, design evolution and empirical evaluation. In *Proceedings of IEEE Symposium on Information Visualization 2002*, pages 57–64. IEEE, Boston, October, 2002.

[12] K. Pulo and M. Takatsuka. Inclusion tree layout convention: An empirical investigation. In *Proceedings of the Australian Symposium on Information Visualisation*, pages 27–35. CRPIT Vol 24, Tim Pattison and Bruce Thomas, eds, 2003.

[13] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.

[14] C. Ware and M. Plumlee. Modeling performance for zooming vs multi-window interfaces based on visual working memory. In *Proceedings of Advanced Visual Interfaces*. Trento, Italy, May, 2002.

[15] C. Ware, M. Plumlee, R. Arsenault, L. Mayer, S. Smith, and D. House. GeoZui3D: Data fusion for interpreting oceanographic data. In *Proceedings of Oceans 2001*. Hawaii, CD ROM Proceedings, 2001.

[16] http://www.it.usyd.edu.au/˜kev/cmv03-kpulo-ssz.wmv.