An Algorithmic Framework for Visualizing Statecharts*

R. Castelló, R. Mili, and I. G. Tollis

Department of Computer Science
The University of Texas at Dallas
Box 830688, Richardson, TX 75083-0688, USA
email: {castello, rmili, tollis}@utdallas.edu

Abstract. Statecharts [9] are widely used for the requirements specification of reactive systems. In this paper, we present a framework for the automatic generation of layouts of statechart diagrams. Our framework is based on several techniques that include hierarchical drawing, labeling, and floorplanning, designed to work in a cooperative environment. Therefore, the resulting drawings enjoy several important properties: they emphasize the natural hierarchical decomposition of states into substates; they have a low number of edge crossings; they have good aspect ratio; and require a small area. We have implemented our framework and obtained drawings for several statechart examples. The preliminary drawings are very encouraging.

1 Introduction

Statecharts [9] is a graphical notation widely used for the requirements specification of reactive systems. Because of their hierarchical property, statecharts are prime candidates for visualization. Nice and intuitive drawings of statecharts would be invaluable aids to software engineers who would like to check the correctness of their design visually. In this paper, we study the problem of visualizing statecharts and present an algorithmic framework for producing clear and intuitive drawings.

Several visualization tools for reactive system specification and design are available in the market [10,19,18,26]. Eventhough these tools are helpful in organizing a designer's thoughts, they are mostly sophisticated graphical editors, and therefore are severely inadequate for the modeling of complex reactive systems. For example, the Rational Rose tool [21] provides a feature to layout UML [3] statechart diagrams. Figure 1 shows an example of a statechart after the Rational Rose layout feature is applied. We notice that transition labels overlap; transition edges overlap with state boxes; and there is a large number of unnecessary edge bends and edge crossings. Figure 5 in Section 5 shows a drawing of the same diagram using our algorithmic framework.

* Research supported in part by Sandia National Labs and by the Texas Advanced Research Program under grant number 009741-040.

J. Marks (Ed.): GD 2000, LNCS 1984, pp. 139–149, 2001.

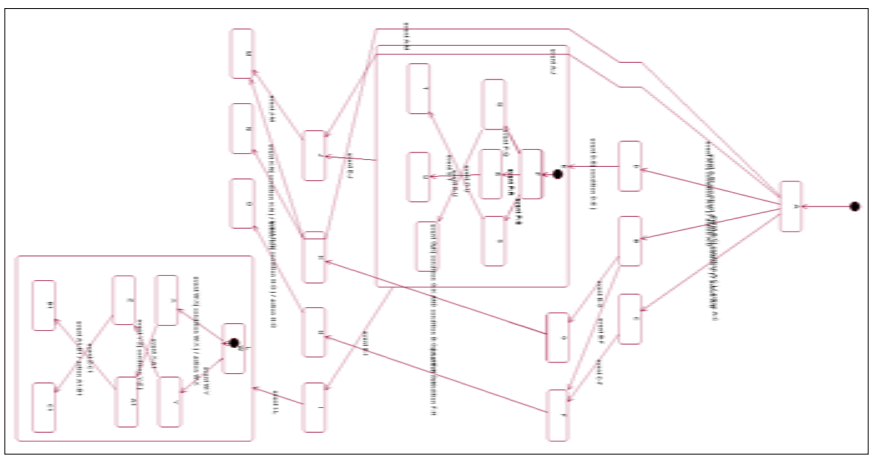

Fig. 1. An example of a drawing of a Statechart generated by Rational Rose (drawing rotated by 90 degrees due to space limitations).

A comprehensive approach to hierarchical drawings of directed graphs is described in Sugiyama et al. [25]. Several extensions and variations of this approach have been introduced in the literature. A comprehensive survey is given in [1]. A first extension that takes into consideration cycles and dummy nodes for large edges (i.e., edges that span more that one level) was introduced by Rowe et al. [22]. Gansner et al. [8,7] provide a technique to draw directed graphs using a simplex-based algorithm that assigns vertices to layers; at the same time, they provide an extension to the basic algorithm of Sugiyama et al. by drawing edge-bends as curves. A divide-and conquer approach is described by Messinger et al. [17] to improve the layout-time performance for large graphs consisting of several hundreds of vertices. More recently, a combination of the algorithm of [25] with incremental-orthogonal drawing techniques was proposed by Seemann [23] to automatically generate a layout of UML class diagrams. In [11], Harel and Yashchin discuss an algorithm for drawing edgeless highgraph-like structures. The problem of drawing clustered graphs without crossings was studied in [5,6]. Most of the research on the *Edge Labeling Problem* (ELP) has been done on labeling graphs with fixed geometry, such as geographical and technical maps [14]. Kakoulis and Tollis [13] present an algorithm for the ELP problem that can be applied to hierarchical drawings with fixed geometry. Gansner et al. [7] use a simple approach to solve the ELP problem for hierarchical drawings: they assign labels to the middle position of edge lines. However, they assume that edge labels are small and do not consider the possibility of overlap with other drawing components.

In this paper, we present a framework for the automatic generation of layouts of statechart diagrams. Our framework is based on several techniques that include hierarchical drawing, labeling, and floorplanning. Our algorithm for hier-

archical drawings is a variant of the algorithm by Sugiyama et al. [25] that is tailored to statecharts. Since edge labels are crucial in describing transitions in statecharts, we have developed edge labeling techniques. Previously, edge labeling techniques were described for graph drawings, and geographical and technical maps with fixed geometry [14,13]. In our work, we address the problem of graph drawings with flexible geometry. Finally, in order to reduce the area and improve the aspect ratio of the statechart drawings we apply floorplanning techniques inspired by the ones used for the area minimization of VLSI layouts [24, 16]. In our approach, the hierarchical, labeling, and floorplanning techniques are designed to work in a cooperative environment. Therefore, the resulting drawings enjoy several properties: they emphasize the natural hierarchical decomposition of states into substates; they have a low number of edge crossings; and require a small area. We have implemented our framework and have obtained drawings for several statechart examples. The preliminary drawings are very encouraging.

2 Statecharts

Statecharts [9] are extended finite state machines used to describe control aspects of reactive systems. They provide mechanisms to describe synchronization and concurrency, and manage exponential explosion of states by using state decomposition. In the statechart notation, a state is denoted by a box labeled in the upper left corner. Directed arcs are used to denote transitions between states. A transition label has the form $E[C]/A$, where E is a boolean combination of external stimuli; C is a boolean combination of conditions; and A is an action that is executed when the transition is active, E occurs, and C is true. A *superstate* is a state that can be used to aggregate sets of states with the same transitions. A state can be repeatedly decomposed into substates in two ways, through the *OR* or the *AND* decomposition. The *OR* decomposition reflects the hierarchical structure of a state machine and is represented by encapsulation. The *AND* decomposition reflects concurrency of independent state machines and is represented by splitting a box with lines.

In our approach, a statechart is treated as a graph. Nodes [1] in the graph correspond to states, and arcs correspond to transitions between states. A node includes the following information: its name; its width and height; the coordinates of its origin point; a pointer to its parent; the list of its children; its decomposition type (e.g., *AND*, *OR* or *leaf*); the list of incoming arcs; the list of outgoing arcs; a list of attributes; and finally its aliases.

The underlying structure of a statechart is an *AND*/*OR* tree where the leaves are called *basic* states. We call this structure a *decomposition tree*. The root of a decomposition tree corresponds to the system state; leaves correspond to atomic states. Each object in the tree can be decomposed through the *AND* or *OR* decomposition. In the remainder of the paper, we assume that relevant information is extracted from a textual description of requirements, and stored in a decomposition tree.

[1] In the remainder of this paper we will use the words *node* and *object* interchangeably.

3 Automatic Layout of Statecharts

In this section, we describe our statechart drawing algorithm. Our algorithm proceeds as follows: first, the decomposition tree is traversed in order to determine the dimensions (and origin point) of every node in a recursive manner. If a node v is a leaf then a drawing procedure is called. This procedure produces a labeled rectangle and returns the dimensions of the rectangle. If v is an *AND* node then a recursive algorithm constructs the drawings of each child of v and places the drawings next to each other. If v is an *OR* node then a recursive algorithm constructs the drawings of v's children, then assigns each child to a specific layer. For the sake of simplicity, we generate our drawings horizontally, from left to right. A similar approach can be used to generate vertical drawings.

An *AND* node reflects concurrency of independent state machines. The children of an *AND* node are drawn as adjacent rectangles. The height of an *AND* node is equal to the maximum height of its children's rectangles; its width is equal to the sum of the widths of its children's rectangles. This algorithm is very simple and thus not very efficient in terms of area. As the size of each node depends on the recursive drawings of the substate nodes that are nested in it, and these drawings depend also on the size of the edge labels, it becomes clear that drawing an *AND* node should be done more carefully. More area-efficient drawings can be obtained by applying techniques similar to floorplanning as used in VLSI layout [15,24,27]. We will revisit this topic in Section 5.

An *OR* node reflects the decomposition of states into substates. The substates of an *OR* node are drawn as rectangles. The drawing (and hence the dimensions of the enclosing rectangle) of an OR node is obtained by recursively performing a hierarchical drawing algorithm [2] on the node and each of its substates. The algorithm that constructs the drawing of an OR node has the following characteristics: (i) substates are drawn recursively; (ii) substates are assigned to layers by using a modified version of Sugiyama's algorithm [25] (procedure *realDimensionHierarchyDrawing*).

Procedure *realDimensionHierarchyDrawing* (see Figure 2) consists of two steps:

1. We construct a hierarchy of substates by treating each substate as a point by calling procedure *hierarchyDrawing*, which proceeds as follows:
   a) We assign the node that corresponds to the initial state to the first layer.
   b) We apply a *depth-first* search to identify those edges that form graph-cycles; then we temporarily remove them.
   c) Once the cycles are removed, we assign every node v to a specific layer which is determined by the length of a longest path from the start node to v. At this stage, every node is assigned an x coordinate.
   d) We add dummy vertices to deal with edges whose initial and final states are not in adjacent layers.
   e) Finally, we apply a node ordering procedure whose purpose is to minimize edge crossings within each layer. This ordering provides the y coordinate for each node.

realDimensionHierarchyDrawing(ObjectList o.children)
Begin
hierarchyDrawing(o.children);
hierarchy.height = 0;
hierarchy.width = 0;
for i = 1 to **depth**(hierarchyDrawing of o.children) **do**
begin_do

1. layer[i].largestWidth = largest width among the objects in layer[i];
2. **if** (layer[i+1] $\leq$ **depth**(hierarchyDrawing of o.children)) **then** add layer[i].largestWidth as an offset to the origin_x of every object in layer[i+1];
3. layer[i].height = summation of each object's height at layer[i];
4. **if** (hierarchy.height $<$ layer[i].height) **then** hierarchy.height = layer[i].height;
5. hierarchy.width = hierarchy.width + layer[i].largestWidth;
6. Increase the origin_y of each object in layer[i] in order to deal with the height of each object and avoid overlapping;

end_do;
End

Fig. 2. Procedure that generates the final hierarchy of an OR node.

2. We incorporate into the hierarchy the dimensions (i.e., height and width) of each node in the drawing, as described in Figure 2. The resulting hierarchy is used to determine the height and width of the parent object/state, as well as the coordinates of the origin of the object's rectangle.

Most of the steps of the algorithm have linear time-complexity with respect to the number of edges of the graph. The last step of procedure *hierarchyDrawing* attempts to beautify the obtained drawing by reducing the number of edge crossings. Our approach is based on the general *layer by layer sweep* paradigm [2]. The time-complexity of this step of the algorithm depends on the number of vertices that exist on each layer. If layer L contains $|L|$ nodes, then the time required the algorithm is $O(|L|^2)$. Clearly, the total time for this step depends upon the distribution of nodes into layers. Any step of the above framework can be replaced by any algorithm that achieves results that are acceptable for the next step. Due to space limitations, we cannot provide more details in this paper. For more details please see [4].

4 Labeling

In the labeling literature, it is common to distinguish between *node label placement* (NLP) and *edge label placement* (ELP). In the Statecharts [9] notation, NLP depends primarily on the node type. Hence, the label placement for nodes in statecharts is rather simple: if a node is a *leaf*, then the label size will deter-

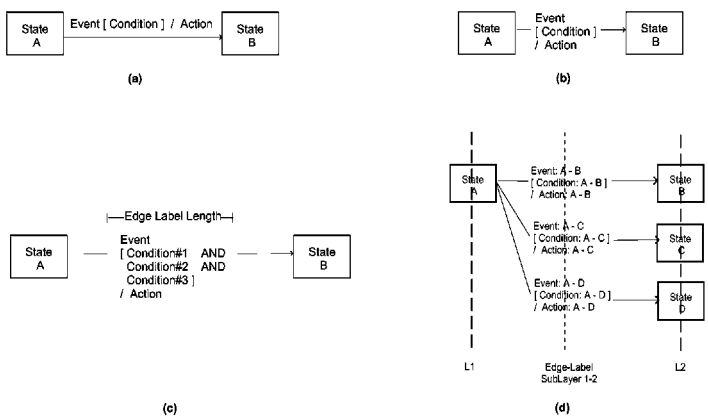

Fig. 3. Edge label placement in statecharts: (a) label on a single line, (b) one label component per line, (c) label with fixed length, (d) edge label placement.

mine the node size. If a node is an AND or an OR, then the label is placed in the top left corner of the enclosing rectangle.

Now we discuss our solution to the ELP problem for statecharts. In cartography, the placement of an edge label must satisfy the following criteria [12, 28,13]:

1. A label cannot overlap with any other graphical component except with its associated edge.
2. The placement of a label has to ensure that it is identified with just one edge in the drawing. Therefore it must be very close to its associated edge.
3. Each label must be placed in the best possible position among all acceptable positions.

In the statecharts notation, an edge label consists of three components: *event*, *condition* and *action* (see Figure 3(a)). In order to satisfy the labeling criteria discussed above we have defined the following steps:

1. We fix the maximum length of the label to a constant, and we write the transition's three components (i.e., *events*, *conditions* and *actions*) on three separate lines (see Figure 3(b)). If the size of a component is greater than the maximum length of the label, then we write it on several lines (see Figure 3(c)).
2. At the beginning of the execution of the drawing algorithm (see Section 3), we assign labels to sublayers (see Figure 3(d)).
3. We traverse the hierarchy from left to right, considering two adjacent layers L_1 and L_2 at a time (see Figure 3(d)). For each vertex a in L_1, we identify the set of edges E_a between a and the vertices in L_2. We order E_a in such a way that potential label crossings are removed.

The time complexity of this step is linear with respect to the number of edges in the graph.

5 Floorplanning Heuristics

Because of the representation of statecharts, it is possible that certain AND nodes of the decomposition tree are very large in one dimension or the other. Recall that our algorithm places all the subnodes (vertically) next to each other. The height of the resulting drawing of an AND node is equal to the maximum of the heights of the subnodes and the width is equal to the sum of the widths of the subnodes. This implies that a bad combination of two subnode rectangles (one with large height and one with large width) will result in a drawing of the AND node that occupies a very large area. This is clearly undesirable. Additionally, the aspect ratio of the drawing, another important aesthetic criterion, is not controllable. We tackle this problem by applying a technique similar to the one used for the minimization of VLSI chip areas [15,24,27], namely *floorplanning*. Floorplanning partitions a floor rectangle into *floorplans* using line segments called *slices*. Floorplans are combined in such a way that the enclosing rectangle covers a minimum area. A floorplan is *slicing* whenever the floorplan is an atomic rectangle or there exists slice that divides the rectangle into two. The floorplanning problem has an efficient solution when the floorplan is slicing [24, 16].

Although one could apply the slicing floorplanning technique for drawing the AND nodes [24], due to the special representation of statecharts, we have simplified this technique. We apply the slicing floorplanning concept to derive a set of heuristics that can be applied to statecharts. To this effect, we define the following drawing criteria for statecharts:

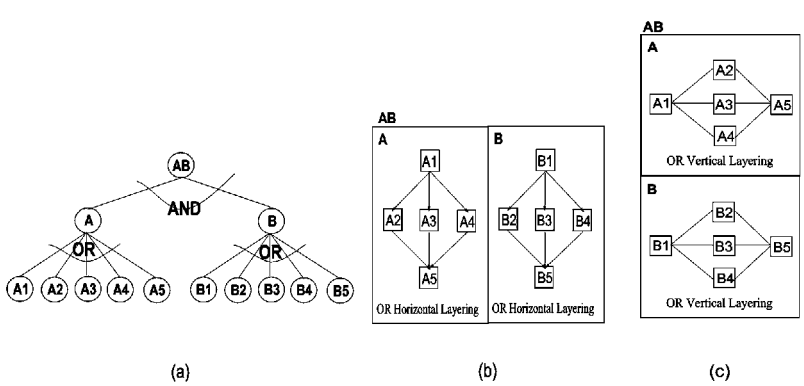

Fig. 4. AND-OR combination: (a) AND/OR decomposition tree, (b) AND vertical slicing with OR horizontal layering, (c) AND horizontal slicing with OR vertical layering.

– Leaves are used to represent atomic states whose size depends solely on their labels. Since labels are usually written horizontally (for readability purposes), we will draw leaves horizontally.

- The AND decomposition reflects concurrency, and is represented by splitting an AND-state box into a number of concurrent substates. Since one of the most important aesthetic criteria in graph drawing is *symmetry* [20], we choose to slice AND-state boxes either horizontally or vertically.
- OR states can be drawn in a hierarchical fashion using either a horizontal or a vertical layering depending on the slicing type of the parent node (i.e., horizontal / vertical slicing).

Our goal is to generate drawings that use the horizontal and vertical dimensions in a uniform way, in order to optimize the drawing area. To this effect we define several heuristics: The AND/OR heuristic applies to the case where the parent is an AND node and the children are OR nodes (see Figure 4(a)). There are two cases:

1. The parent node (AND) is sliced vertically. Then the children nodes (OR) are drawn on horizontal layers (see Figure 4(b)). In this case, the height of the parent object is the height of the highest child node; and the total width of the parent is the sum of the children's widths.
2. The parent node (AND) is sliced horizontally. Then the children nodes (OR) are drawn on vertical layers (see Figure 4(c)). In this case the height of the parent node is the sum of children's heights; and the width of the parent node is the width of the widest child.

Heuristics that handle the other cases (OR/AND, AND/AND, and OR/OR) are defined similarly, and are omitted due to space limitation. For more details please see [4].

Figure 5 shows the statecharts diagram after we applied our improvement drawing techniques (i.e., edge-crossing and edge-bend reduction, edge labeling, and floorplaning) to the diagram of Figure 1. We observe that both, the horizontal and vertical dimensions, grow in a uniform manner; edges do not overlap with any other drawing component; every edge crossing has been removed; and the number of edge bends has been reduced considerably.

6 Conclusions and Experimental Results

In this paper we presented an algorithmic framework for the automatic generation of layouts of statechart diagrams. Our framework is based on hierarchical drawing, labeling, and floorplanning techniques. Clearly any algorithm used for any step can be replaced with an improved algorithm thus resulting in an improved tool. We implemented a tool using the algorithms described in this paper, and ran the tool on four statechart examples. We generated drawings using first, the basic version (without the optimized algorithms), then the optimized version. Due to space limitations, the drawings produced by our tool are available at http://www.utdallas.edu/∼rmili/GD2000/. Our results are described in Table 1. We notice that, after the application of the optimized algorithms, (1)

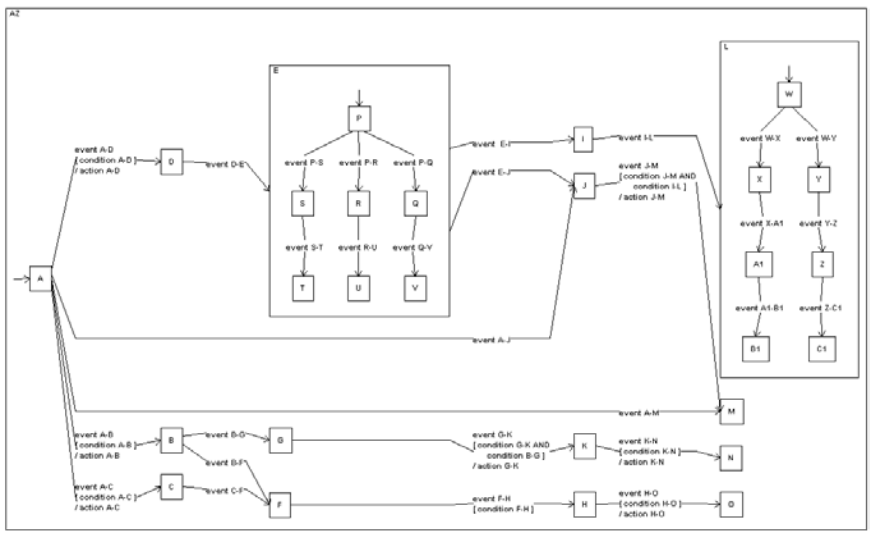

Fig. 5. Same statechart as in Figure 1, generated by our drawing algorithm with optimization techniques.

edge-crossings are completely eliminated; (2) the number of edge-bends is considerably reduced; (3) the drawings enjoy a good aspect ratio. This optimization improves considerably the readability of the diagrams. Therefore, it constitutes an invaluable tool to the specifier who will shift his/her focus from organizing the mental or physical structure of the requirements to its analysis.

Table 1. Comparison of four examples of statecharts drawn by our algorithms

Aesthetic Criteria	Drawing 1		Drawing 2		Drawing 3		Drawing 4	
	Without Improve	With Improve	Without Improve	With Improve	Without Improve	With Improve	Without Improve	With Improve
Edges Crossings	33	0	23	0	22	0	34	0
Edge Bends	53	19	39	18	24	18	70	32
Width	1,953	875	2,733	1,029	1,794	861	2,820	1,632
Height	548	1,259	735	1,722	736	1,593	651	1,424
W/H Ratio	3.5227	0.695	3.718	0.5975	2.4375	0.54	4.33	1.44
Area	1,059,284	1,102,884	2,008,755	1,771,938	1,320,384	1,371,573	1,835,820	2,323,968

References

1. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. Algorithms for drawing graphs: an annotated bibliography. *Comput. Geom. Theory Appl.*, (4):235–282, 1994.
2. G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

3. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.
4. R. Castelló, R. Mili, and I. G. Tollis. Automatic layout of statecharts. Technical Report UTD-04-00, University of Texas at Dallas, 2000.
5. P. Eades and Q. Feng. Drawing clustered graphs on an orthogonal grid. In G. Di Battista, editor, *Graph Drawing (Proceedings GD'97)*, pages 146–157. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.
6. P. Eades, Q. Feng, and X. Lin. Straight-line drawing algorithms for hierarchical graphs and clustered graphs. In S. North, editor, *Graph Drawing (Proceedings GD'96)*, pages 113–128. Springer-Verlag, 1997. Lecture Notes in Computer Science 1190.
7. E. R. G., E. Koutsofios, S. C. North, and K. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(32):214–230, March 1993.
8. E. R. Gansner, S. C. North, and K. P. Vo. Dag—a program that draws directed graphs. *Software Practice and Experience*, 18(11):1047–1062, November 1988.
9. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
10. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, May 1990.
11. D. Harel and G. Yashchin. An algorithm for blob hierarchy layout. In *Proceedings of International Conference on Advanced Visual Interfaces*, AVI'2000, Palermo, Italy, May 1990.
12. E. Imhof. Positioning names on maps. *The American Cartographer*, 2(2):128–144, 1975.
13. K. G. Kakoulis and I. G. Tollis. An algorithm for labeling edges of hierarchical drawings. In G. Di Battista, editor, *Graph Drawing (Proceedings GD'97)*, pages 169–180. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.
14. K. G. Kakoulis and I. G. Tollis. On the edge label placement problem. In S. North, editor, *Graph Drawing (Proceedings GD'96)*, pages 241–256. Springer-Verlag, 1997. Lecture Notes in Computer Science 1190.
15. E. S. Kuh and T. Ohtsuki. Recent advances in VLSI layout. *Proceedings of the IEEE*, 78(2):237–263, 1990.
16. T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, 1990.
17. E. B. Messinger, L. A. Rowe, and R. R. Henry. A divide-an-conquer algorithm for the automatic layout of large graphs. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(1):1–11, February 1991.
18. R. O'Donnel, B. Waldt, and J. Bergstrand. Automatic code for embedded systems based on formal methods. Available from Telelogic over the Internet. http://www.Telelogic.se/solution/techpap.asp. Accessed on April 1999.
19. J. Peterson. Overcoming the crisis in real-time software development. Available from Objectime over the Internet. http://www.Objectime.on.ca/otl/technical/crisis.pdf. Accessed on April 1999.
20. H. Purchase. Which aesthetic has the greatest effect on human understanding. In G. Di Battista, editor, *Graph Drawing (Proceedings GD'97)*, pages 248–261. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.
21. Rational. Rose java. Downloaded from Rational over the Internet. http://www.rational.com. Accessed on November 1999.

22. L. A. Rowe, M. Davis, E. Messinger, and C. Meyer. A bowser for directed graphs. *Software Practice and Experience*, 17(1):61–76, January 1987.
23. J. Seeman. Extending the sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In G. Di Battista, editor, *Graph Drawing (Proceedings GD'97)*, pages 415–424. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.
24. L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Information and Control*, (57):91–101, 1983.
25. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(2):109–125, February 1981.
26. Artisan Software Tools. Real-time studio: The rational alternative. Available from Artisan Software Tools over the Internet. http://www.artisansw.com/rtdialogue/pdfs/rational.pdf. Accessed on April 1999.
27. S. Wimer, I. Koren, and I. Cederbaum. Floorplans, planar graphs and layout. *IEEE Transactions on Circuits and Systems*, pages 267–278, 1988.
28. P. Yoeli. The logic of automated map lettering. *The Cartographic Journal*, 9(2):99–108, 1972.