Journal of Graph Algorithms and Applications
http://jgaa.info/
vol. 6, no. 3, pp. 313–351 (2002)

A Framework for the Static and Interactive Visualization of Statecharts

Rodolfo Castelló
School of Engineering and Computer Science
ITESM Campus Chihuahua
http://www.chi.itesm.mx/
rodolfo.castello@itesm.mx

Rym Mili *Ioannis G. Tollis*
Department of Computer Science
The University of Texas at Dallas
http://www.utdallas.edu/
rmili@utdallas.edu tollis@utdallas.edu

Abstract

We present a framework for the automatic generation of layouts of statechart diagrams. Statecharts [16] are widely used for the requirements specification of reactive systems. 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 also present techniques for interactive operations. We have implemented our framework and obtained drawings for several statechart examples.

Communicated by Michael Kaufmann: submitted April 2001; revised January 2002.

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

1 Introduction

The representation and visualization of software requirements specification is very important in software engineering. Statecharts [16] 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. Our framework is based on several techniques that includes 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.

There are several visualization tools for the specification and design of reactive systems available in the market [17, 27, 29, 36]. These tools are helpful in organizing a designer's thoughts. However, they are mostly sophisticated graphical editors, and therefore, are severely inadequate for the modeling of complex reactive systems. For example, the Rational Rose (year 1999) tool [30] provides a feature to layout UML [5] 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 9 in Section 4 shows a drawing of the same diagram using our algorithmic framework.

A comprehensive approach to hierarchical drawings of directed graphs is described in Sugiyama et al. [35]. 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. [31]. Gansner et al. [14, 15] 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. [24] to improve the layout-time performance for large graphs consisting of several hundreds of vertices. More recently, a combination of the algorithm of [35] with incremental-orthogonal drawing techniques was proposed by Seemann [33] to automatically generate a layout of UML class diagrams. In [18], Harel and Yashchin discuss an algorithm for drawing edgeless hi-graph like structures. The problem of drawing clustered graphs without crossings was studied in [10, 11]. Most of the research addressing the labeling problem has focused on labeling features of geographical and technical maps, which have fixed geometry [20, 21]. Labeling is typically partitioned into smaller tasks: (a) labeling points, also known as the *Node Label Placement* (NLP) problem, and (b) labeling lines, also known as the *Edge Label Placement* (ELP) problem.

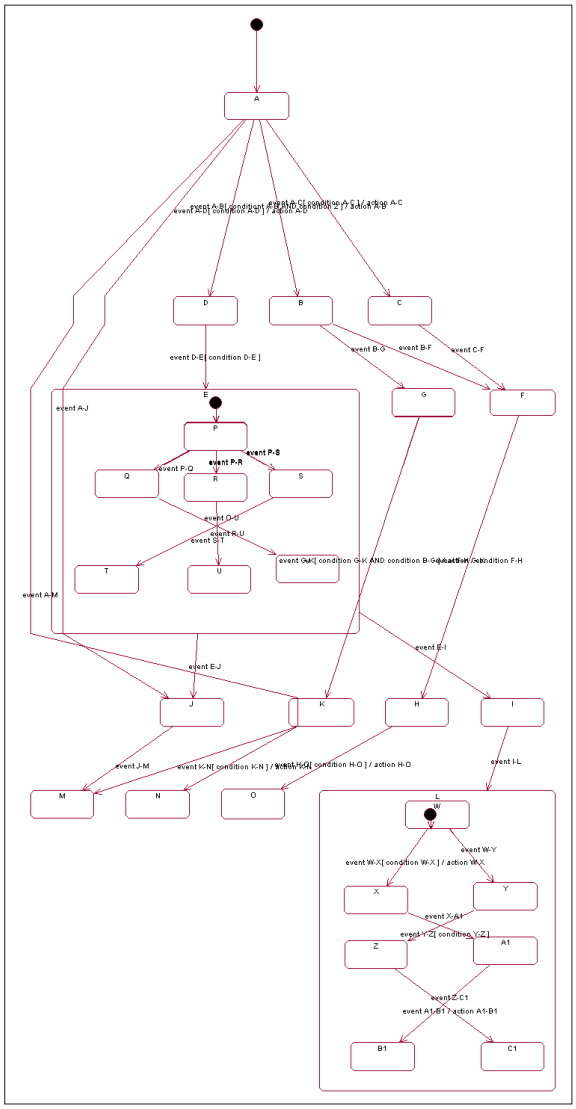


Figure 1: A statechart example generated by Rational Rose (drawing rotated 90 degrees due space limitations).

A survey of algorithms for the labeling problem including simulated annealing techniques are presented in [8]. Approximation algorithms for restricted versions of the NLP problem are presented in [9, 13, 37]. Most of the research on the ELP Problem has been done on labeling graphs with fixed geometry, such as geographical and technical maps [21]. Kakoulis and Tollis [20] present an algorithm for the ELP problem that can be applied to hierarchical drawings with fixed geometry. Gansner et al. [14] utilize 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.

Here, we present a framework for the static and interactive visualization of statechart diagrams. The framework uses techniques for hierarchical drawing, labeling, and floorplanning. We also present algorithms for interactive operations (such as insertions and deletions) that preserve the mental map of the drawings. Our algorithm for hierarchical drawings is a variant of the algorithm by Sugiyama et al. [35] that is adapted to statecharts. In our work, we have developed edge labeling techniques to address the problem of graph drawings with flexible geometry. Finally, we apply floorplanning techniques in order to reduce the area and improve the aspect ratio of the statechart drawings. Our floorplanning techniques are inspired by the ones used for the area minimization of (a) VLSI layouts [23, 34] and (b) inclusion drawings of trees [12].

We have also incorporated our implementation into a complete system for visualizing software requirements [7]. Several drawings of statechart examples are included in the Appendix.

2 Statecharts

This graphical notation was developed by Harel [16] to specify the behavior of reactive systems. Statecharts are extended finite state machines. They provide mechanisms to represent *hierarchical decomposition*, *concurrency* and *synchronization*. In the statechart notation transitions (i.e., edges in a finite state machine) can be defined as function of a stimulus, and the truth-value of a condition. Output events can be attached optionally to the triggering event. Transition labels have the form $E[C]/A$, where E is the event that triggers the transition, C is a condition that guards the transition from being taken unless it is true, and A is an action that is executed when the transition is taken. Statecharts use the concept of *superstate* to overcome the exponential blow-up of states encountered in traditional finite state machines. They support the repeated decomposition of states into substates through the OR or the AND decompositions. The OR decomposition reflects the hierarchical structure of state machines and is represented by encapsulation (see Figure 2). 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

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

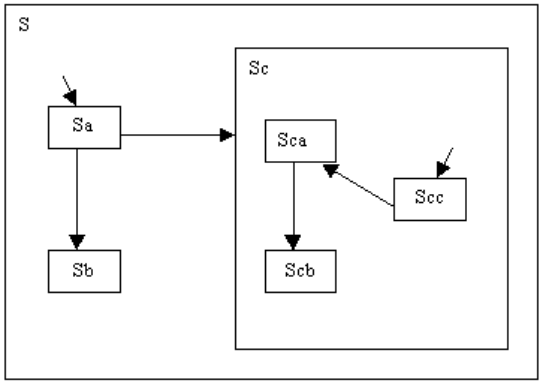


Figure 2: State encapsulation.

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 point of origin; 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 that we call *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.

3 Automatic Layout of Statecharts

In this section we describe our framework for the visualization of statecharts. The main algorithm that is the foundation of the framework proceeds as follows:

1. The decomposition tree T is traversed in order to determine the dimensions and point of origin of the drawing of every node, in a recursive manner.

2. If a node v is a leaf then a simple drawing procedure is called that produces a labeled rectangle. It returns the point of origin and the dimensions of the rectangle.

3. If v is an *AND* node then a recursive algorithm constructs the drawings of every child node of v and places the drawings next to each other.

4. If v is an *OR* node then a recursive algorithm:

- first constructs the drawings of every child of v;
- then, it 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.

The procedure that draws leaves is trivial: Each leaf is drawn as a rectangle that is wide enough to accommodate its label. The algorithms for drawing the *AND* and *OR* nodes are more complicated and one can choose among many choices. We have chosen to draw (a) *AND* nodes using techniques similar to floorplans [22, 34, 38], or inclusion drawings [12], and (b) *OR* nodes using a modified version of Sugiyama's algorithm [35] for producing a hierarchical drawing. Our framework will work even if other implementors of a similar system decide to substitute their favorite technique/algorithm for any of the steps. In the next subsections, we will describe the algorithms that we chose for our implementation.

3.1 Drawing *AND* Nodes

An *AND* node reflects concurrency of independent state machines. The simplest approach to draw an *AND* node is to place the drawings of its children vertically next to each other. Then the height of the *AND* node is equal to the maximum height of its children's rectangles and its width is equal to the sum of the widths of its children's rectangles. This approach is very simple and thus very desirable. However, it is not very efficient in terms of area. The size of each node depends on the recursive drawings of the substate nodes that are nested in it. Hence, it is possible that certain *AND* nodes of the decomposition tree are very large in one dimension or the other. This implies that an unfortunate 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 (which is mostly empty). 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 (a) *floorplanning*, a technique used for the minimization of area of VLSI chip [22, 34, 38], and (b) the technique for minimizing the area of drawing trees in the *inclusion* convention [12].

Floorplanning partitions a floor rectangle into *floorplans* using line segments called *slices*. Each slice is given as either a vertical or a horizontal slice. Floorplans are combined in such a way that the enclosing rectangle covers a minimum area (see Figure 3). A floorplan is *slicing* whenever the floorplan is an atomic rectangle [2] or there exists a slice that divides the rectangle into two. The floorplanning problem has an efficient solution when the floorplan is slicing [23, 34]. However, the slices in our problem are not fixed as either vertical or horizontal. This has to be decided by some algorithm. From this point of view, our problem is also similar to the tree drawing problem mentioned above. For the

[2] one that can not be further decomposed.

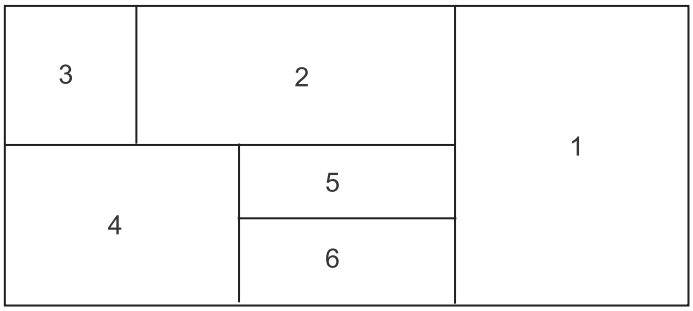


Figure 3: A slicing floorplan.

inclusion convention each non-leaf node is a rectangle that contains the rectangles of its subtrees. The problem of optimizing the area by choosing vertical and horizontal slices is NP-hard in general, see for example [12]. If the tree is binary and the sizes of the leaf nodes are restricted, then a dynamic programming type algorithm can be used to minimize the area of an inclusion drawing of the tree in about quadratic time [12]. Unfortunately, our trees are general and the size of the rectangles representing the subtrees depends heavily on the graph represented by the statechart. This is clearly the case for *OR* nodes.

Although one could apply either (a) the general slicing floorplanning technique for drawing the *AND* nodes [34], or (b) a modification of an inclusion drawing algorithm, none of them could give us any performance guarantees. Also, due to the special representation of statecharts, and the fact that the size of any *OR* node can be arbitrarily large we have decided to use some simple heuristics that can be applied to statecharts. To this effect, we define the following drawing criteria for statecharts:

- 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 rectangle into a number of concurrent substates. As discussed before, due to the recursive nature of each (sub)state, the dimensions of the various substates could be incompatible if placed next to each other. Thus we choose to slice AND-state rectangles either horizontally or vertically, in order to reduce the total area and control the aspect ratio.
- 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). This is discussed in the next subsection.

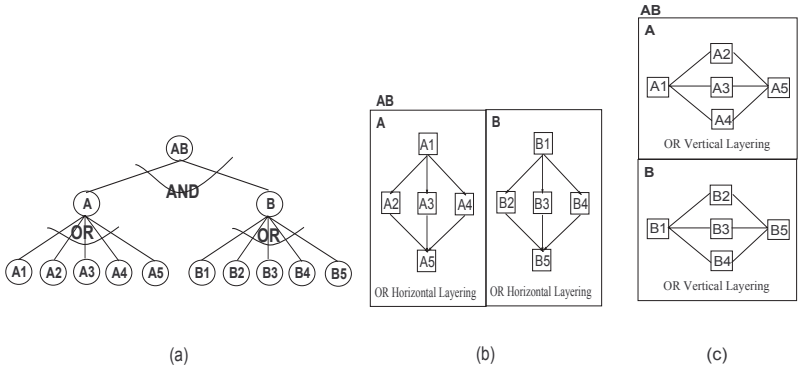


Figure 4: AND-OR combination: (a) AND/OR decomposition tree, (b) AND vertical slicing with OR horizontal layering, (c) AND horizontal slicing with OR vertical layering.

Our goal is to generate drawings that use the horizontal and vertical dimensions in a uniform way, in order to obtain drawings with small area and good aspect ratio. 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) will be 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.

Clearly, the above algorithm for drawing *AND* nodes (excluding the drawing of *OR* nodes) has linear time-complexity with respect to the number of substates in such a node. Heuristics that handle the other cases (OR/AND, AND/AND, and OR/OR) are defined similarly. An extensive discussion of all cases appears in [6].

Finally, it is possible to further improve the area and aspect ratio of the resulting parent drawing at the expense of reducing the readability of the drawing. This can be done by allowing the drawings of the children of a node to have different orientations. For example most of them can be placed vertically, while the tallest one will be placed horizontally, in order to reduce the total height and hence the area requirement, if this is deemed appropriate. Several other similar heuristics can be implemented. The end result will be a drawing which

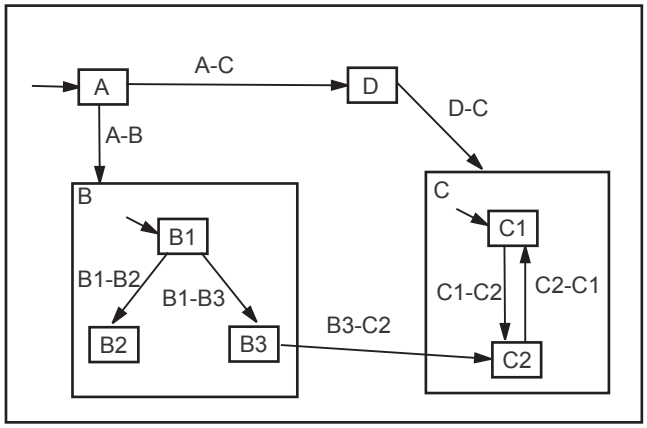


Figure 5: Inter-level transitions.

will be a general slicing floorplan, see Figure 3. However such a drawing of a statechart diagram may be unacceptable for several applications.

3.2 Drawing *OR* Nodes

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.

In the statechart notation [16], it is possible for transitions to cross superstate boundaries. We call these special edges *inter-level transitions*. For example, in Figure 5, we notice that transition *B3-C2* crosses the boundaries of parent state *B*. In our approach, inter-level transitions are treated as follows: a) we define the final state of the transition as a *GOTO* node and place it in the parent state box; b) we label the GOTO node "*GOTO final-state-name*" and we process it as a regular OR-node (see Figure 6). We believe that this approach improves the readability of the drawings.

The algorithm that constructs the drawing of an OR node has the following steps: (i) substates are drawn recursively; and (ii) substates are assigned to layers by using a modified version of Sugiyama's algorithm [35] (procedure *realDimensionHierarchyDrawing*).

Procedure *realDimensionHierarchyDrawing* (see Figure 7) 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

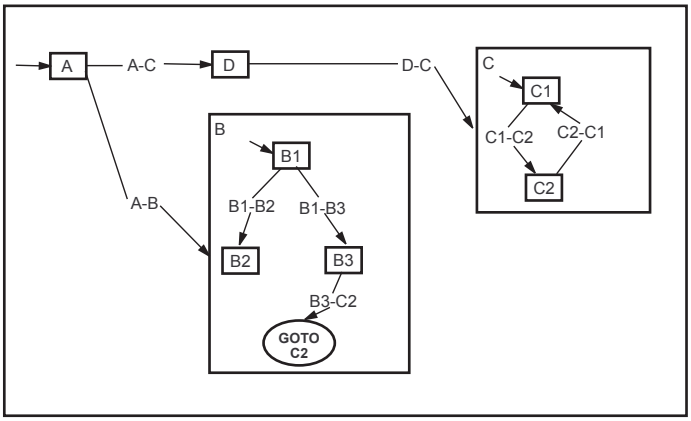


Figure 6: Goto State.

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

Figure 7: Procedure that generates the final hierarchy of an OR node.

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.

2. We incorporate into the hierarchy the dimensions (i.e., height and width) of each node in the drawing, as described in Figure 7. 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.

As described, the above algorithm produces drawings that are directed from top to bottom and their nodes are placed on horizontal layers. It is easy to modify it in order to produce drawings that are directed from left to right and their nodes are placed on vertical layers. 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 by the algorithm is $O(|L|^2)$. Clearly, the total time for this step depends upon the distribution of nodes into layers. Note that any algorithm used in any step of the above framework can be replaced by another algorithm (chosen by another designer) as long as it achieves results that are acceptable for the next step.

3.3 Final Drawing

The final drawing of the statechart represented by the decomposition tree is computed by performing the following steps:

1. Each non-leaf node of the decomposition tree is equipped with two (*width, height*) pairs as follows:

   OR node: (vertical_layering_width, vertical_layering_height), and (horizontal_layering_width, horizontal_layering_height).

   AND node: (vertical_slicing_width, vertical_slicing_height), and (horizontal_slicing_width, horizontal_slicing_height)

2. The horizontal/vertical drawing dimensions are computed in a recursive manner, by traversing the decomposition tree in a top-down fashion:
   for all $node_i \in$ decomposition_tree **do**
   begin_for
   if ($node_i.decompositionType$ ==AND) **or** ($node_i.decompositionType$ ==OR) **then**
   Determine the dimensions for both, the vertical drawing and the horizontal drawing.
   else
   Determine the dimensions as a LEAF.
   end_for

3. The root node of the decomposition tree will decide which is the best width/height pair:
   if (root.horizontal_width * root.horizontal_height < root.vertical_width * root.vertical_height)
   then
   generate drawing with root as horizontal_layering (slicing)
   else
   generate drawing with root as vertical_layering (slicing)

4. We generate the drawing by traversing the decomposition tree top down once more, assigning the correct layering/slicing option to every node.

As described, the above algorithm keeps only two possible rectangles for each internal node of the tree: a horizontal and a vertical drawing of the graph corresponding to the tree node. Hence, once the children-node drawings are obtained, it takes time proportional to the number of children to compute the two possible drawings (vertical and horizontal) generated by our algorithm for the parent node. Of course, at the expense of extra computation time one can allow a higher number of possible drawings (rectangles) to be computed for each internal node, and several more level combinations during the recursive drawing of *AND* nodes can be computed. However, the number cannot be allowed to grow arbitrarily. In other words, if we allow all possible drawings and combinations of rectangles, for the children of each node, then there will be exponentially many combinations. Recall that the problem of optimizing the area of inclusion drawings of trees (which is a special case of our problem) by choosing vertical and horizontal slices is NP-hard in general [12].

4 Labeling

In the labeling literature, it is common to distinguish between *node label placement* (NLP) and *edge label placement* (ELP). In the Statecharts [16] 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 determine 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 [19, 20, 39]:

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 8(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 8(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 8(c)).
2. At the beginning of the execution of the drawing algorithm (see Section 3), we assign labels to sublayers (see Figure 8(d)).
3. We traverse the hierarchy from left to right, considering two adjacent layers L_1 and L_2 at a time (see Figure 8(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.

Figure 9 shows the statechart diagram after we applied an implementation of our drawing framework 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.

5 Interactive Operations in Statecharts

The framework presented in this paper describes algorithms that produce drawings of static statecharts. The techniques try to optimize various aesthetics, such as bends, crossings, area, etc. This implies that even when a minor modification is performed on a drawing (e.g., addition of a node or a transition), the layout algorithms will re-order the nodes in such a way that these aesthetic criteria are met. Hence, the structure of the resulting drawing could be very

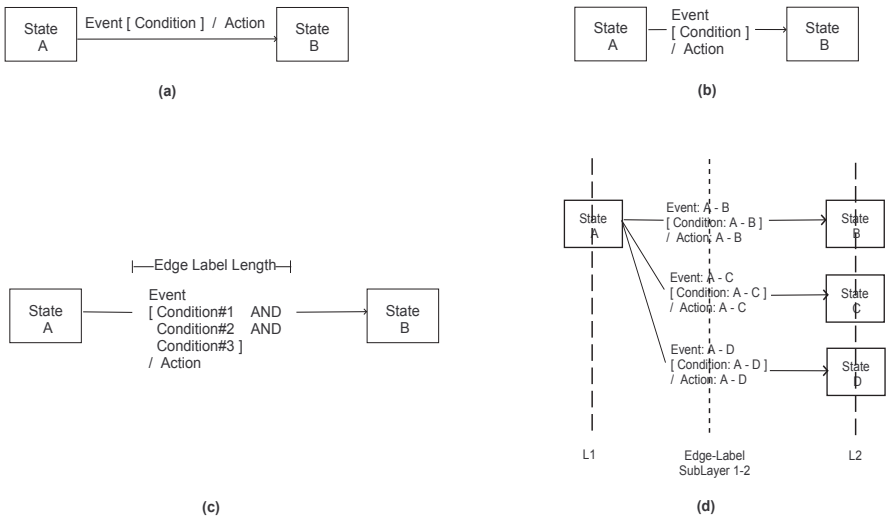


Figure 8: Edge label placement in statecharts: (a) label on a single line, (b) one label component per line, (c) label with fixed length, (d) edge label placement.

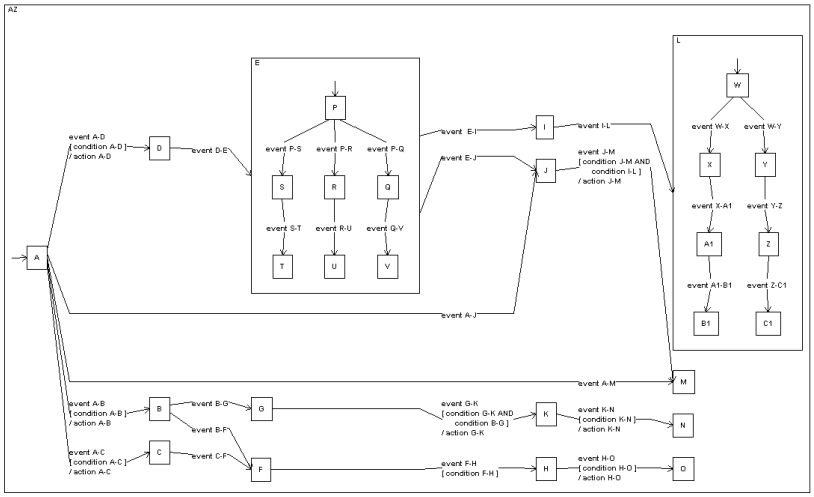


Figure 9: Same statechart diagram as in Figure 1, generated automatically by our drawing algorithm.

different from the original one. In this section, we discuss a technique that can be used to preserve the *mental map* (i.e., the structure) of statecharts. This is important since the designer/specifier may find mistakes/omissions and hence may need to delete and/or insert states and/or transitions without loosing the mental map.

5.1 Previous Work

Misue et. al. [25] present three models for mental map maintenance, based on the position of the nodes in the diagram: the orthogonal ordering; the proximity relations; and the topology. They provide two layout adjustment techniques that preserve the mental map. The first technique aims at making nodes disjoint during the modification of the original diagram. The second technique is oriented towards displaying certain parts of the diagram like a view.

North [26] presents a heuristic for hierarchical layouts of directed graphs that incorporates position and order node stability. This heuristic moves nodes between adjacent ranks based on median sort. Seemann [33] combines hierarchical with orthogonal techniques for the incremental layout of UML class diagrams. Ryall et. al. [32] present a constraint-based approach to layout small graphs. These constraints are enforced by a generalized spring algorithm. Several techniques for the preservation of the mental map in orthogonal drawings have been proposed [3, 4, 28]. Papakostas and Tollis [28] discuss a systematic approach that applies to interactive orthogonal graph drawings of vertices of degree at most 4. They describe the following scenarios:

1. *Full-control scenario.* When a new vertex is inserted in the current drawing, the user has full control over the vertex location. The edges can also be routed by the user.

2. *Draw-from-scratch scenario.* At the user's request, the new graph is redrawn using any known technique. This scenario has two main disadvantages: first, it is slow; and second, it does not preserve the mental map.

3. *Relative-coordinates scenario.* This approach preserves the general shape of the current drawing by proportionally changing the coordinates of vertices and/or bends. An insertion of a new vertex results in the insertion of new rows and/or columns.

4. *No-change scenario.* The insertion of a new vertex/edge does not modify the coordinates of existing vertices, bends, and edges in the new drawing.

5.2 Our Approach

In this section we describe techniques for interactive operations in a statechart that preserve the mental map. Our approach is based on the *relative coordinate* and *no-change* scenarios:

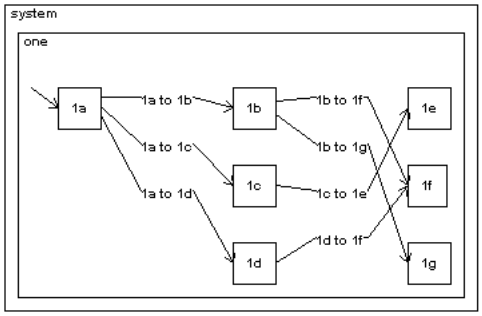


Figure 10: Statechart diagram.

- The *delete* operation allows to delete nodes and edges. The simplest way to implement the delete operation is to follow the no-change scenario. In other words, the node and/or edge are simply removed from the drawing without changing the coordinates of the rest of the nodes and edges.
- The *insert* operation allows to insert nodes and edges. Here we use the relative-coordinates scenario. We apply a layout algorithm that preserves the relative position of the selected states (nodes). The main drawing features of the algorithm that are affected by the insert operation are: (1) the placement on layers of the substates of an OR decomposition, and (2) the position of these substates inside the layers.

Let us consider the first feature, i.e., the placement of OR-substates on layers. When a new transition (edge) is added to the drawing, the insert operation works as follows:

1. Verify that the layout algorithm has been applied.
2. Select the set of states that need to preserve their mental map.
3. Add a new transition to the diagram.
4. Keep the direction of the transitions.
5. Do not identify new cycles, that is, disable the depth-first search algorithm.
6. Determine the direction of the new transition using the current layer assignments of its initial and final states.
7. If the transition is between two states in the same layer, then stop the interactive insert operation and apply the normal layout algorithm.

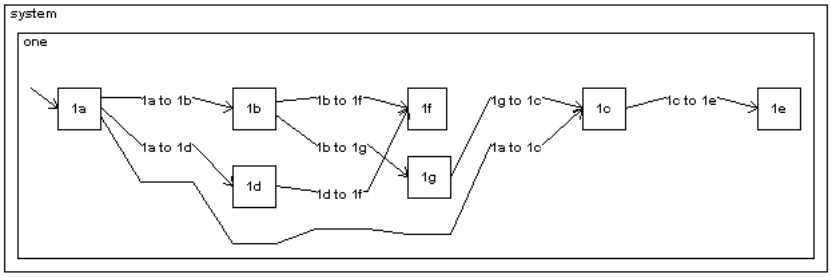


Figure 11: Modified statechart diagram without mental map preservation.

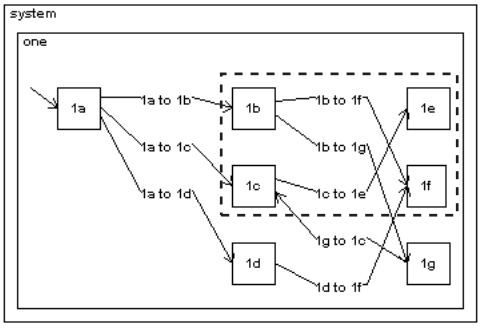


Figure 12: Modified statechart diagram with mental map preservation.

Figure 10 shows a statechart drawing generated by our layout algorithm. If we add a transition from $1g$ to $1c$ and apply again our layout algorithm, we notice (see Figure 11) that the drawing has completely changed. Now, if we select states $1b, 1c, 1e$ and $1f$ to be preserved and apply again the layout algorithm, we notice (see Figure 12) that the four states are kept in their original layers while the new transition $1g - 1c$ is drawn backwards.

When a new state (node) is added to the drawing, insert operation for the placement of substates on layers works as follows:

1. Verify that the layout algorithm has been applied.
2. Select the set of states that need to preserve their mental map.
3. Add a new state:
4. Keep the direction of the transitions.
5. Do not identify new cycles, that is, disable the depth-first search algorithm.

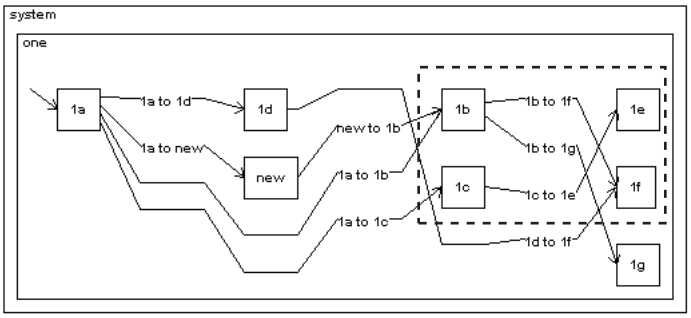


Figure 13: Adding a new state to a statechart diagram with mental map preservation: case (a).

6. Determine the new layers by applying the following criteria:

   (a) If the new state shifts the position of one selected state, then shift the position of the entire selected group. Figure 13 shows the drawing of Figure 10 after the insertion of state *new*. We notice that states $1b, 1c, 1e$ and $1d$ are shifted one layer to the right, and their relative position is unchanged.

   (b) If the new state is positioned in between selected states of the group, then keep the new state in its position, and shift the selected states positioned on subsequent layers to the right. Figure 14 shows the drawing of Figure 10 after the insertion of a *new* state. We notice that the selected group of states is divided in two parts. However, the relative position of the selected states within layers is preserved. For example, both $1b$ and $1c$ are placed on the same layer, at position 1 and 2 respectively. The same applies to nodes $1e$ and $1f$.

The second feature that is affected by the preservation of the mental map, is the placement of states on the same layers. Specifically, our *edge crossing reduction* algorithm (see [6] for details) is modified as follows: the selected group of states on a specific layer is treated as a single state; the edge crossing reduction algorithm shifts the complete group as needed, and inserts invisible nodes when necessary.

We consider three cases that we illustrate through Figures 15 and 16.

1. *Case 1.* If the edge crossing reduction algorithm determines that state x is best placed below b (Figure 15(b)):

   (a) Insert an *invisible state* in x's position.

   (b) Shift down all states below state b, e.g., states y and z.

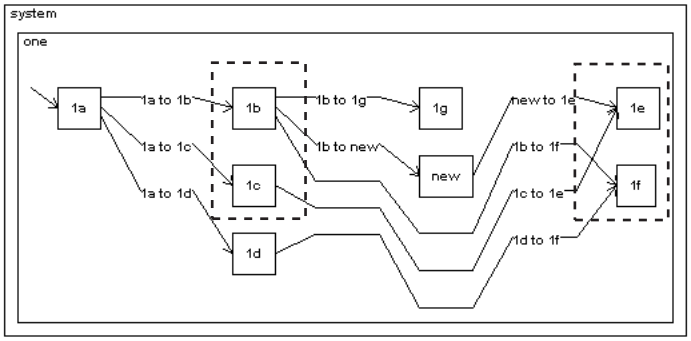


Figure 14: Adding a new state to a statechart diagram with mental map preservation: case (b).

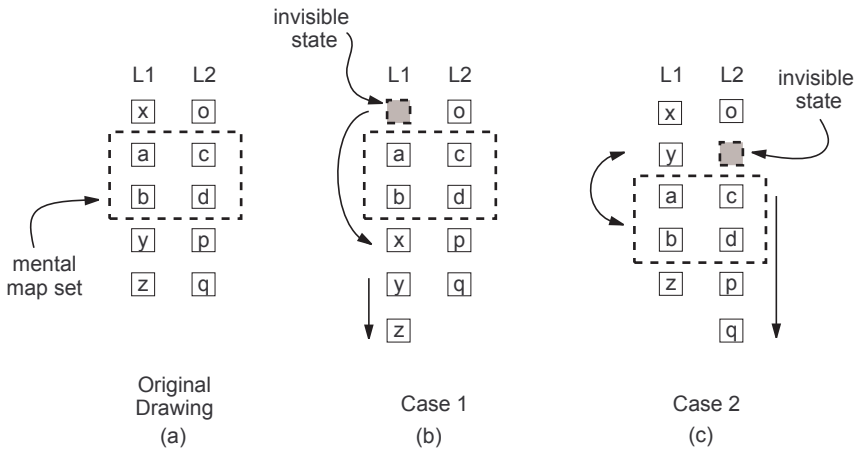


Figure 15: Edge crossing reduction for mental map preservation: cases 1 and 2.

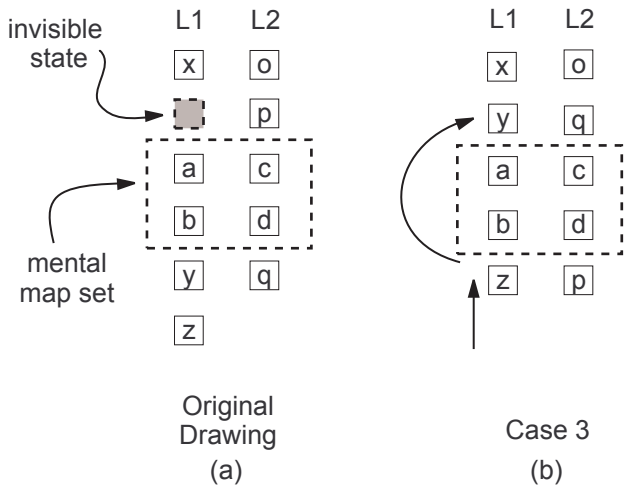


Figure 16: Edge crossing reduction for mental map preservation: case 3.

(c) Place x below b.

2. *Case 2.* If the edge crossing reduction algorithm determines that state y is best placed above a (see Figure 15(c)):

   (a) In layer $L1$ shift down the position of the selected states, e.g., a and b.

   (b) Place y above a.

   (c) For all the states in other layers that include part of the selection (e.g., layer $L2$):

      i. Shift down the position of the states in such a way that the original structure of the selection is preserved (e.g., states c, d, p, and q are shifted down).

      ii. Insert invisible states to fill the empty positions (e.g., above state c).

3. *Case 3.* In Figure 16(a), if the edge crossing reduction algorithm determines that state y is best placed above a and the position above a is occupied by an *invisible state*:

   (a) Move y to the position occupied by the invisible state.

   (b) Shift up the position of every state below the original position of y (e.g., z).

   (c) Delete the *invisible state* (see Figure 16(b)).

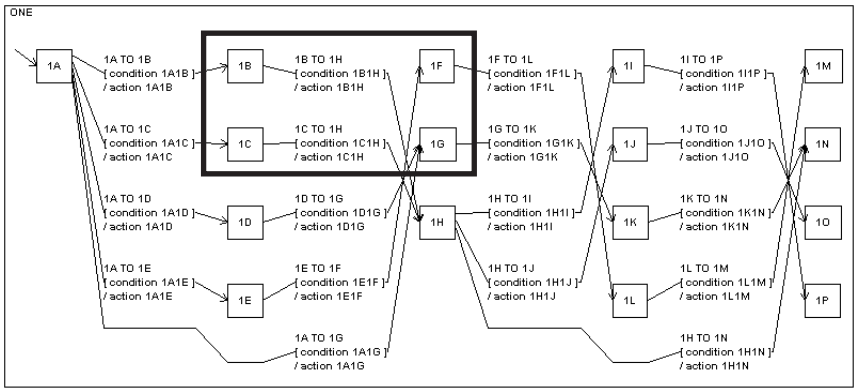


Figure 17: Statechart diagram before edge-crossing with layer mental map.

Figure 17 shows a statechart drawing generated by our layout algorithm before the reduction of the edge crossings. In this diagram, we have selected states $1B$, $1C$, $1F$, and $1G$ to preserve the mental map. Figure 18 shows the same statechart drawing of Figure 17 after the reduction of the edge crossings while preserving the mental map. This diagram shows the three *invisible* states that were added as a result of the required shifts generated by our algorithm.

The preservation of the mental map has disadvantages: some edge crossings may not be removed; improper edges may be inserted; and the insertion of invisible states may increase the total drawing area. Hence, the mental map feature is offered as an option to the user.

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. The design of the framework is modular and thus any algorithm used for any step can be replaced with an improved algorithm thus resulting in an improved tool. Furthermore, we presented techniques for performing interactive operations (insertions and deletions) in the statechart while preserving the mental map of the drawing. Since the resulting drawings improve considerably the readability of the diagrams, they constitute 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.

We implemented a tool, called Vista, using the algorithms described in this paper, and ran the tool on two different sets of statechart examples. First, we created four statechart examples which we used as input for Rational Rose 1999, and Vista. Our results are summarized in Table 1 and the respective drawings are included in the Appendix.

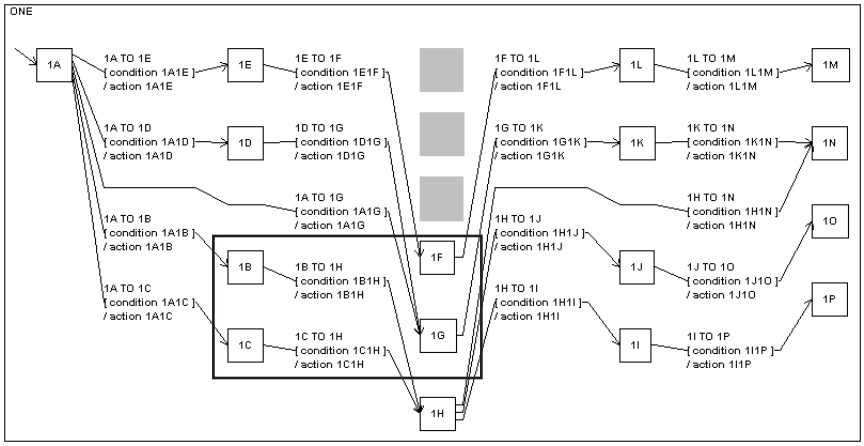


Figure 18: Statechart diagram after edge-crossing with layer mental map.

Aesthetic Criteria	Example 1		Example 2		Example 3		Example 4	
	Rational	ViSta	Rational	ViSta	Rational	ViSta	Rational	ViSta
Edges Crossings	2	0	6	0	13	0	4	0
Edge Bends	2	5	7	3	20	8	14	7
Edge Label Overlap	10	0	10	0	25	0	17	0
State-Edge Overlap	4	0	0	0	0	0	0	0

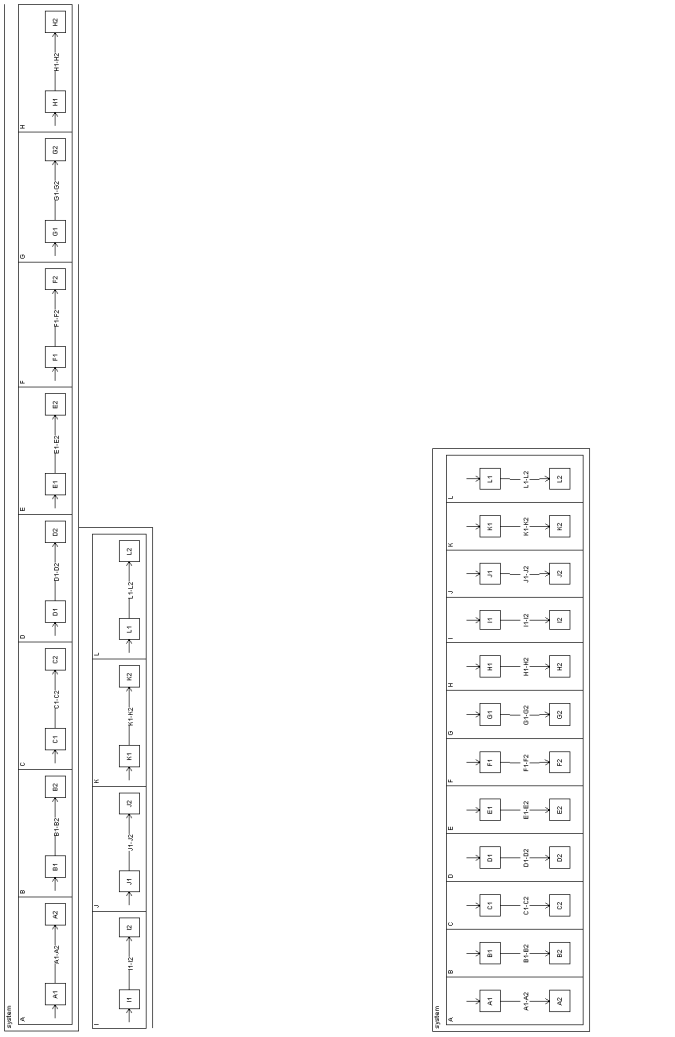
Table 1: Comparison of statecharts drawn by our algorithms

Aesthetic Criteria	Example 5	Example 6	Example 7	Example 8	Example 9	Figure 9
Edges Crossings	0	0	0	0	9	0
Edge Bends	19	18	18	32	23	7
Width	875	1,029	861	1,632	992	1077
Height	1,259	1,722	1,593	1,424	1423	683
W/H Ratio	0.695	0.5975	0.54	1.44	0.697	1.57
Area in pixels	1,102,884	1,771,938	1,371,573	2,323,968	1,411,616	735,591

Table 2: Comparison of statecharts drawn by our algorithms

Secondly, we include drawings of five additional statecharts drawn by Vista in the Appendix. These statecharts show different attributes of Vista and cannot be automatically drawn by Rational Rose 1999. The statistics of these drawings are summarized in Table 2. We notice that, after the application of the tool, the drawings have: (1) few edge-crossings; (2) a low number of edge bends; (3) small area; and (4) a good aspect ratio.

A limitation of our technique is related to the optimization of the drawing area obtained by our current floorplanning. The approach works well if the parent AND state is composed by a few children. If the AND state has a lot of children (as shown in Figure 19.a, which is obtained before applying our floorplanning technique) our algorithm will produce a drawing that will still use one dimension more that the other (see Figure 19.b). More research is needed in this direction.



(a) Original Drawing

(b) Drawing after floorplanning

Figure 19: Statechart that shows the limitations of our tool.

References

[1] G. D. 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. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[3] T. Biedl and M. Kaufman. Area-efficient static and incremental graph drawings. In *Proceedings of the 5th. Annual European Symposium on Algorithms*, volume 1284, pages 37–52. Springer-Verlag, 1997.

[4] T. C. Biedl, B. P. Madden, and I. G. Tollis. The three-phase method: A unified approach to orthogonal graph drawing. In G. D. Battista, editor, *Graph Drawing (Proceedings GD'97)*, pages 391–402. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.

[5] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

[6] R. Castelló. *From Informal Specification to Formalization: An Automated Visualization Approach.* PhD thesis, The University of Texas at Dallas., 2000.

[7] R. Castelló, R. Mili, and I. G. Tollis. Vista: A tool suite for the visualization of statecharts. *to appear on Journal of Systems and Software.*, (0), 2002.

[8] J. Christensen, J. Marks, and S. Shieber. An empirical study of algorithms for Point Feature Label Placement. *ACM Trans. on Graphics*, 14(3):203–232, July 1995.

[9] S. Doddi, M. V. Marathe, A. Mirzaian, B. M. Moret, and B. Zhu. Map Labeling and Its Generalizations. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 148–157, 1997.

[10] P. Eades and Q.-W. Feng. Drawing clustered graphs on an orthogonal grid. In G. D. Battista, editor, *Graph Drawing (Proceedings GD'97)*, pages 146–157. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.

[11] P. Eades, Q.-W. 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.

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

[13] M. Formann and F. Wagner. A packing problem with applications to lettering of maps. In *Proc. 7th Annual ACM Sympos. Comput. Geom.*, pages 281–288, 1991.

[14] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(32):214–230, March 1993.

[15] 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.

[16] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[17] 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.

[18] 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.

[19] E. Imhof. Positioning names on maps. *The American Cartographer*, 2(2):128–144, 1975.

[20] K. G. Kakoulis and I. G. Tollis. An algorithm for labeling edges of hierarchical drawings. In G. D. Battista, editor, *Graph Drawing (Proceedings GD'97)*, pages 169–180. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.

[21] 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.

[22] E. S. Kuh and T. Ohtsuki. Recent advances in VLSI layout. *Proceedings of the IEEE*, 78(2):237–263, 1990.

[23] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout*. John Wiley & Sons, 1990.

[24] 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.

[25] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Computing*, 6(2):183–210, 1995.

[26] S. C. North. Incremental layout in dynadag. In F. J. Brandenburg, editor, *Graph Drawing (Proceedings GD'95)*, pages 409–418. Springer-Verlag, 1996. Lecture Notes in Computer Science 1027.

[27] 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.

[28] A. Papakostas and I. G. Tollis. Interactive orthogonal graph drawing. *IEEE Transactions on Computers*, 47(11):1297–1309, 1998.

[29] 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.

[30] Rational. Rose java. Downloaded from Rational over the Internet. http://www.rational.com. Accessed on November 1999.

[31] 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.

[32] K. Ryall, J. Marks, and S. Shieber. An interactive system for drawing graphs. In S. North, editor, *Graph Drawing (Proceedings GD'96)*, pages 387–393. Springer-Verlag, 1997. Lecture Notes in Computer Science 1190.

[33] J. Seeman. Extending the sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In G. D. Battista, editor, *Graph Drawing (Proceedings GD'97)*, pages 415–424. Springer-Verlag, 1997. Lecture Notes in Computer Science 1353.

[34] L. Stockmeyer. Optimal orientations of cells in slicing floorplan designs. *Information and Control*, (57):91–101, 1983.

[35] 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.

[36] A. S. 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.

[37] F. Wagner and A. Wolff. Map labeling heuristics: Provably good and practically useful. In *Proc. 11th Annual. ACM Sympos. Comput. Geom.*, pages 109–118, 1995.

[38] S. Wimer, I. Koren, and I. Cederbaum. Floorplans, planar graphs and layout. *IEEE Transactions on Circuits and Systems*, pages 267–278, 1988.

[39] P. Yoeli. The logic of automated map lettering. *The Cartographic Journal*, 9(2):99–108, 1972.

Appendix

In this Appendix we show drawings of four example statecharts that were used in table 1. These drawings were automatically generated by Rational Rose and Vista.

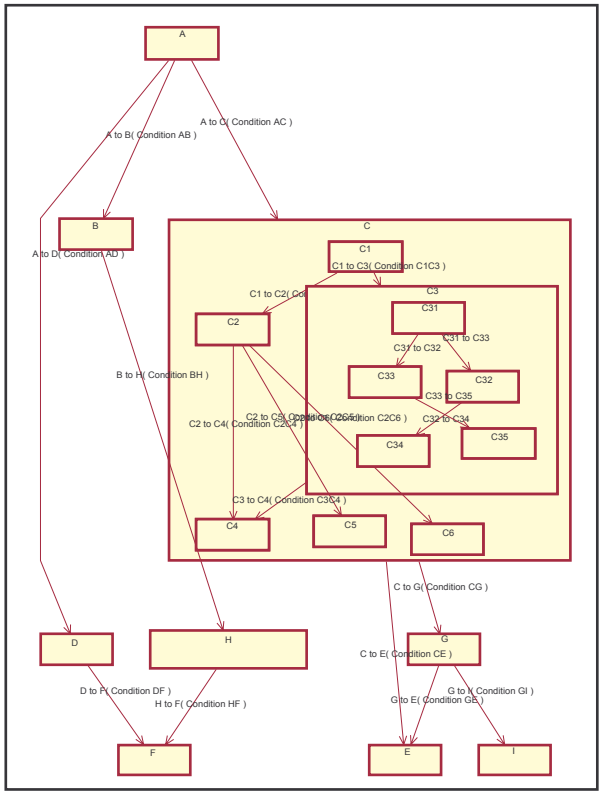


Figure 20: Example 1 generated by Rational Rose.

We also show drawings (Figures 28,29,30,31,32) of five example statecharts that were used in table 2. These drawings were automatically generated by our implementation of the framework presented in this paper.

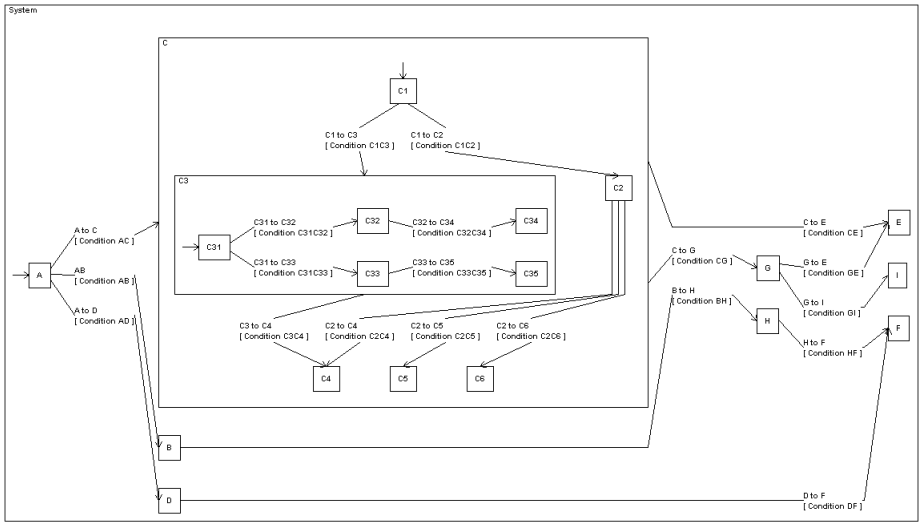


Figure 21: Example 1 generated by ViSta.

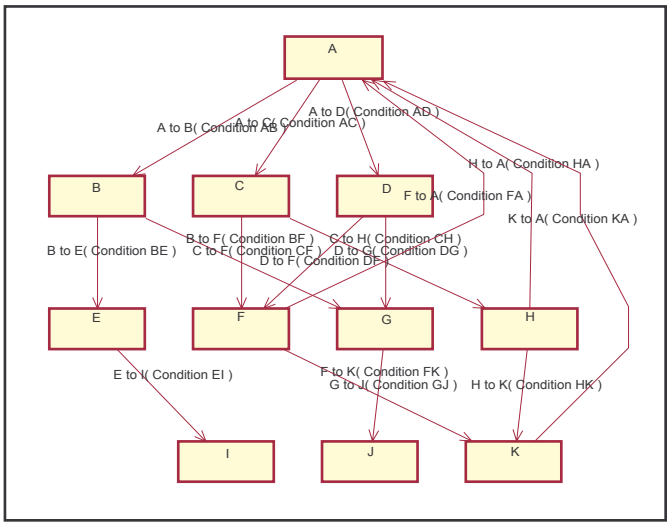


Figure 22: Example 2 generated by Rational Rose.

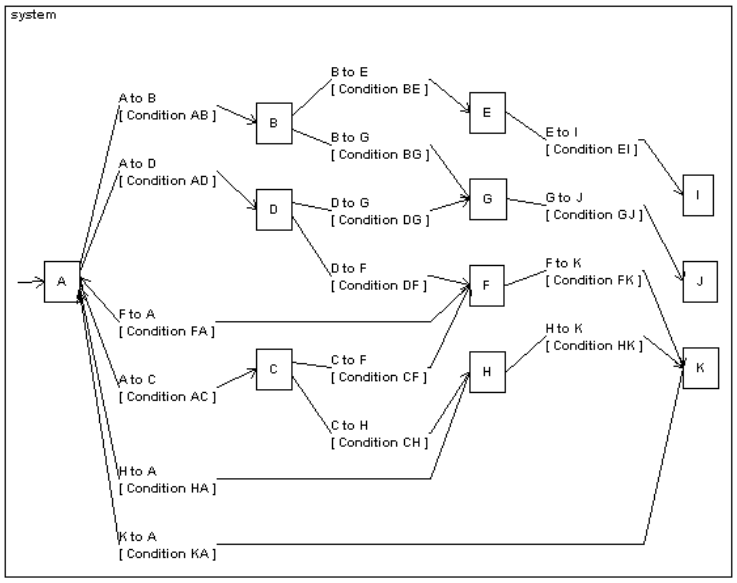


Figure 23: Example 2 generated by ViSta.

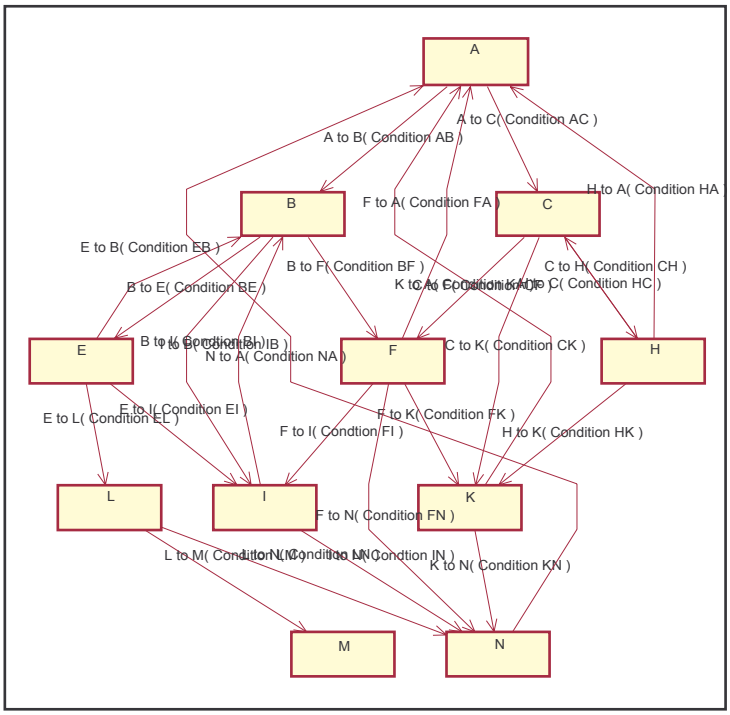


Figure 24: Example 3 generated by Rational Rose.

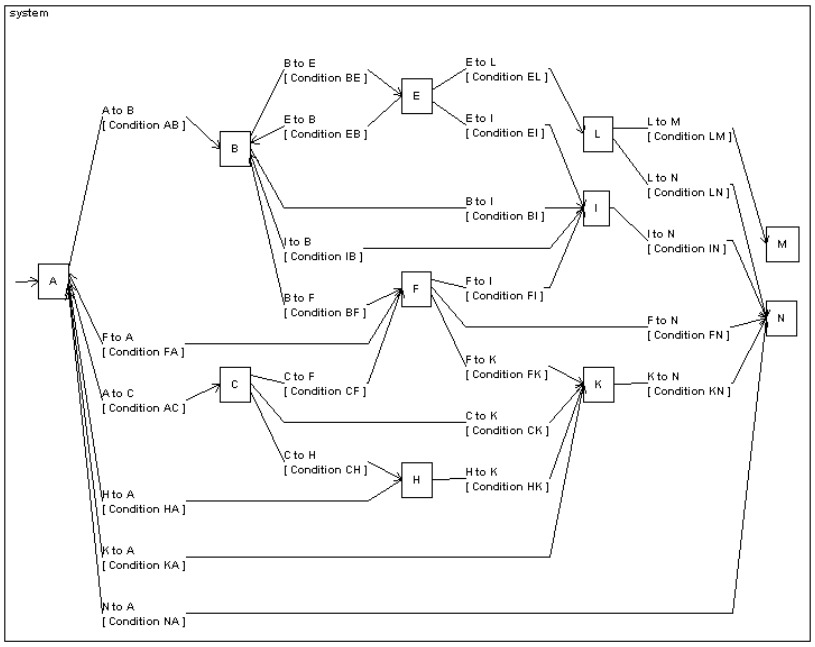


Figure 25: Example 3 generated by ViSta.

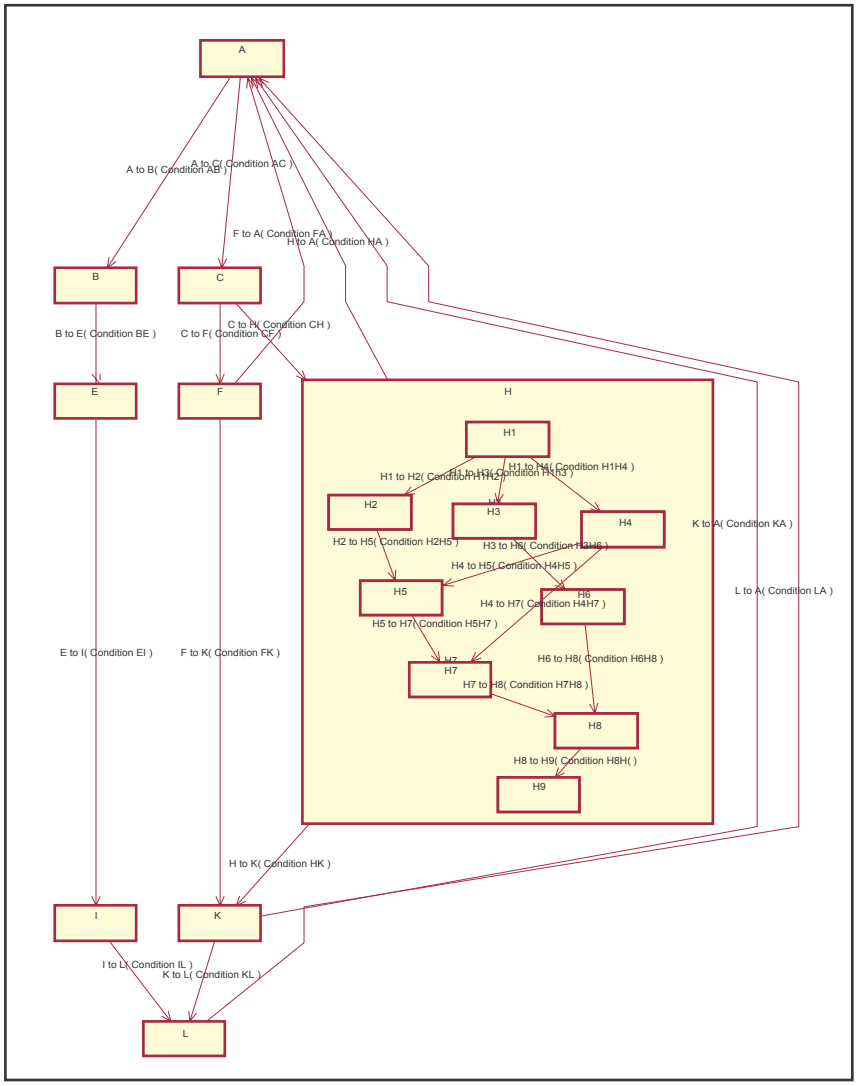


Figure 26: Example 4 generated by Rational Rose.

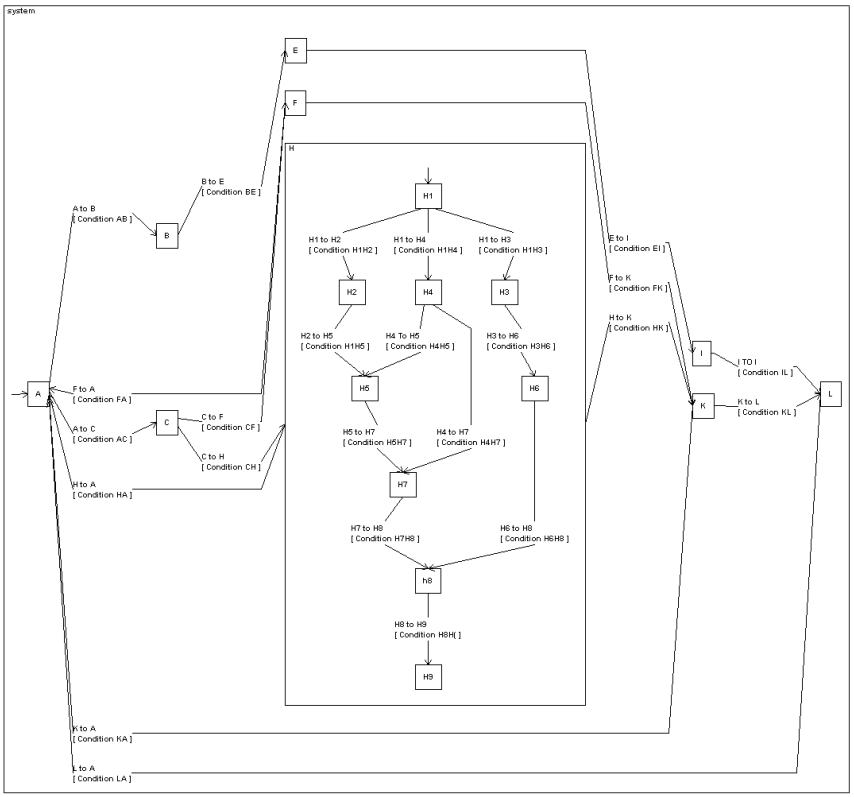


Figure 27: Example 4 generated by ViSta.

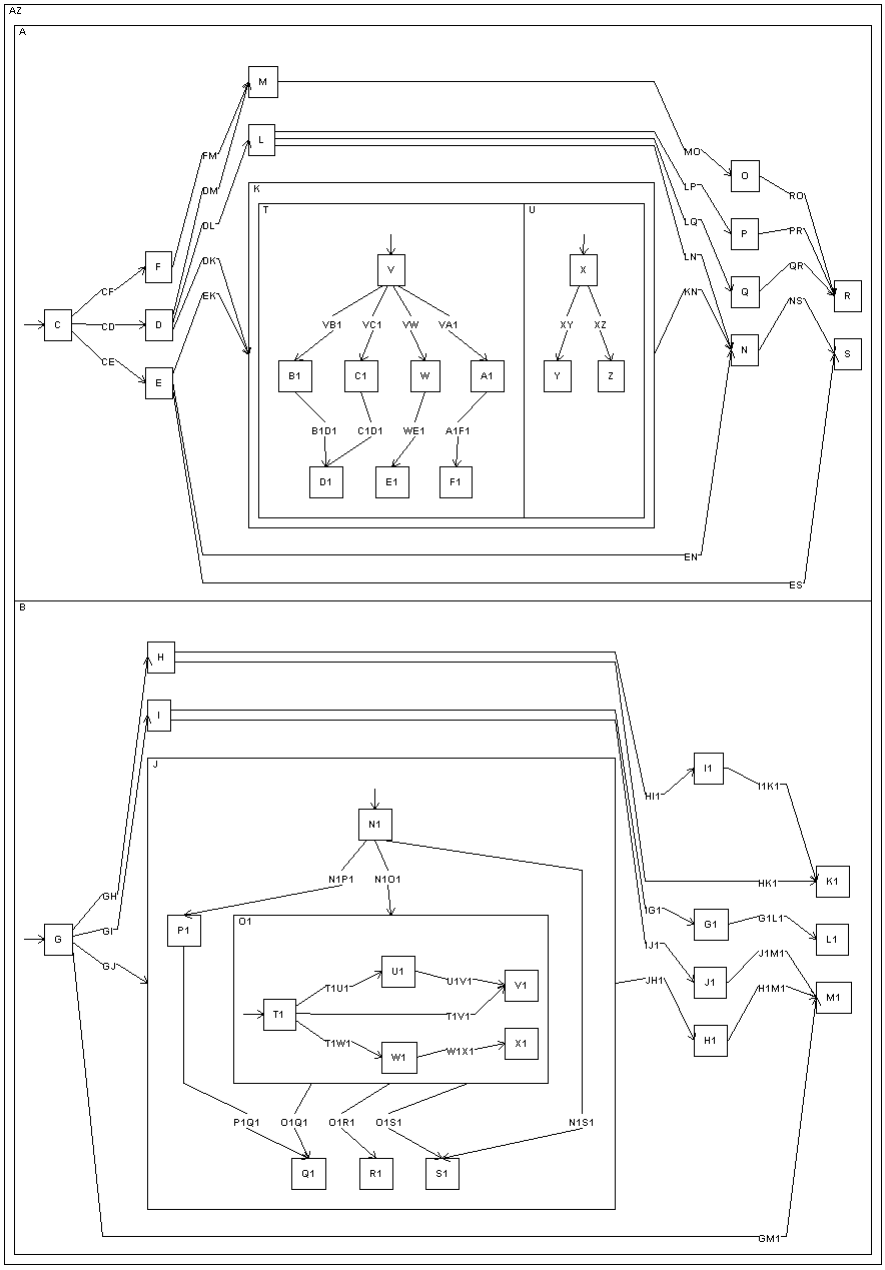


Figure 28: Example 5.

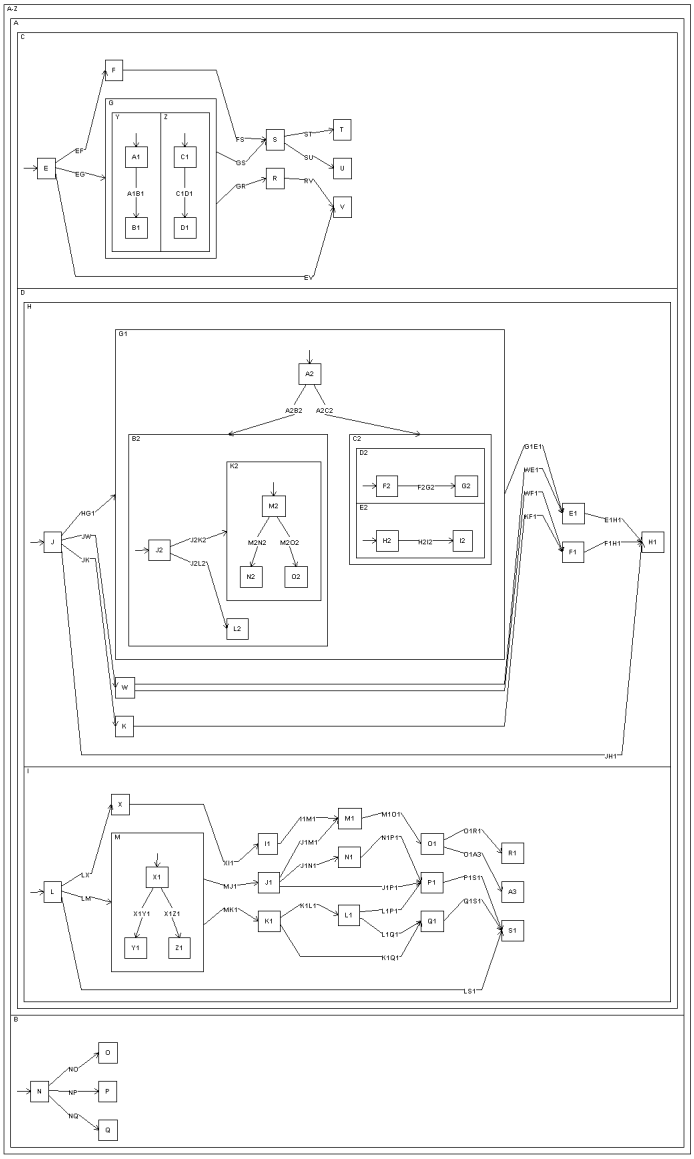


Figure 29: Example 6.

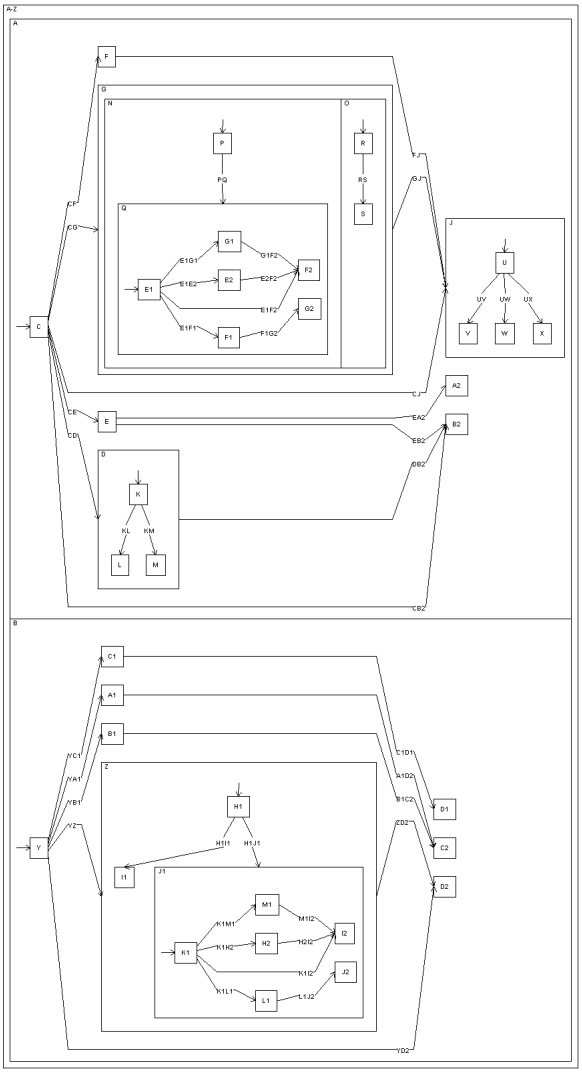


Figure 30: Example 7.

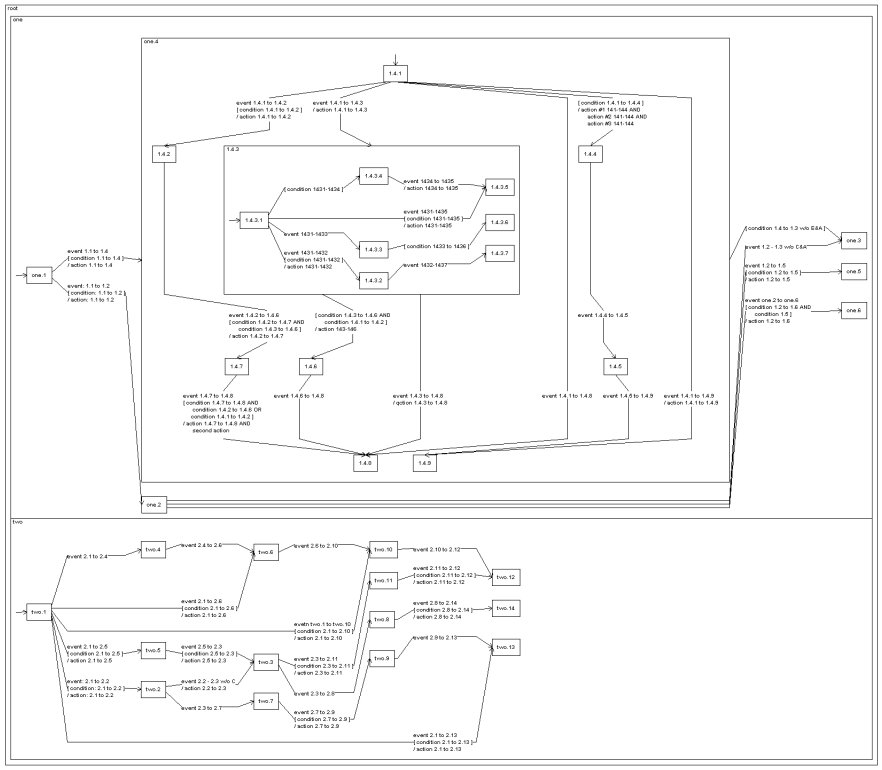


Figure 31: Example 8.

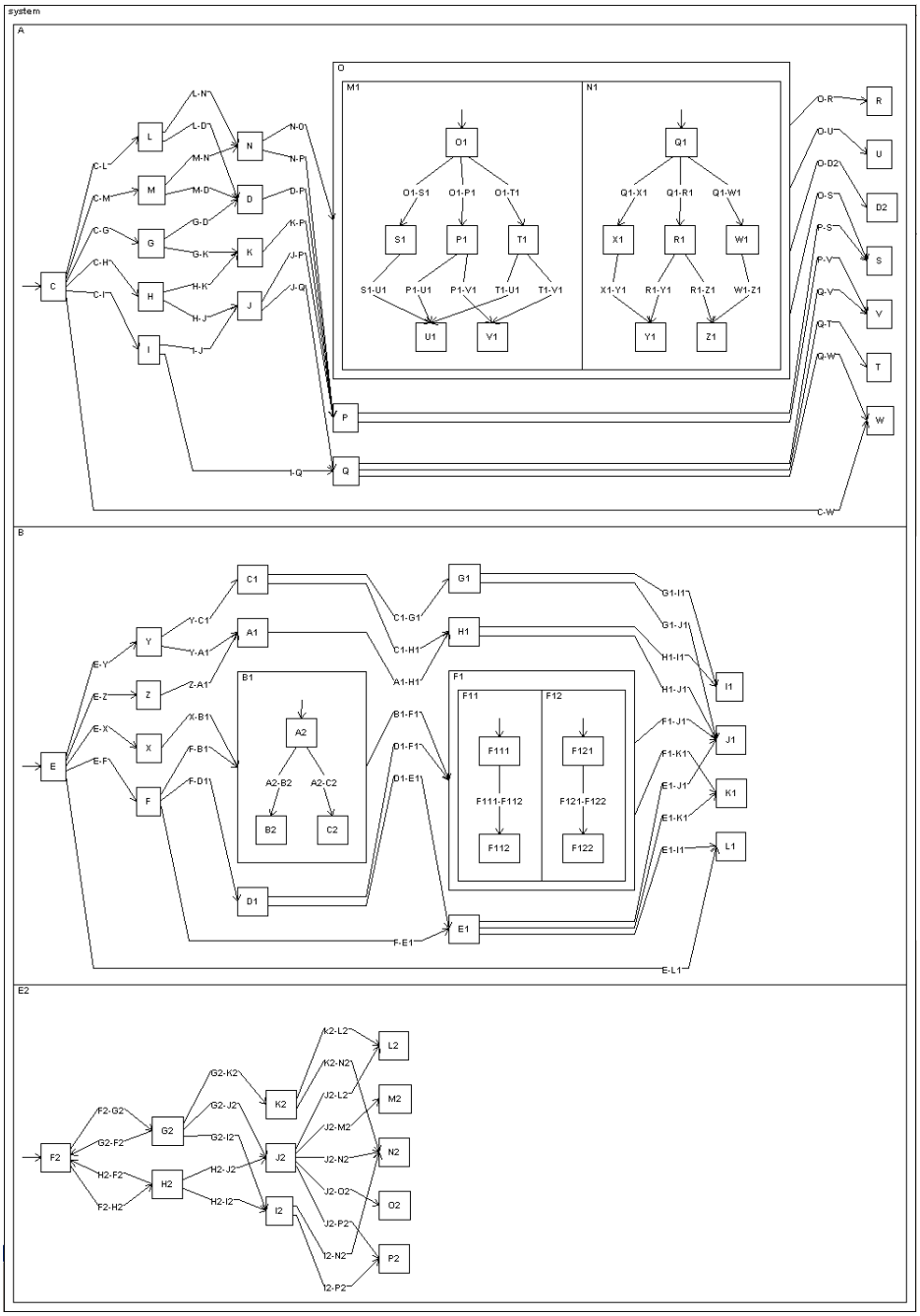


Figure 32: Example 9.