

REPORT NO. UIUCDCS-R-93-1799

UILU-ENG-93-1714

**A GRAPHICAL EDITOR PROPOSAL FOR DEVELOPING CONCURRENT,
HIERARCHICAL, FINITE STATE MACHINES**

by

Paul Jay Lucas

January 1993

A Graphical Editor Proposal for Developing Concurrent, Hierarchical, Finite State Machines

PAUL J. LUCAS*

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL
61801-2987, U.S.A.

SUMMARY

Concurrent, hierarchical, finite state machines (CHSMs), a realization of statecharts, can be a valuable tool for modeling and implementing reactive systems. A language system for building CHSMs has been developed; the next step is to add a graphical-editor to allow the user to draw the statecharts. Attention is paid both to user-interface and implementation issues.

1 Introduction

1.1 Statechart Overview

Statecharts are a graphical notation to represent complex reactive systems more naturally than conventional state-transition diagrams [1] [2] [3] [4]; specifically:

1. Statecharts are hierarchical in that they can have *child* states grouped together, or “nested,” within a *parent* state and thus allow for locality. This allows parent-states to be treated as “black-boxes.”
2. Replicated transitions like those shown in figure 1.1a are eliminated by the introduction of *logical-exclusive-or* state groups just alluded to and shown in figure 1.1b. Henceforth, logical-exclusive-or state groups shall be referred to as *clusters*.

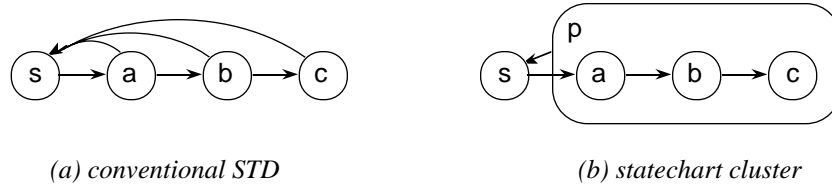


Figure 1.1: Eliminating replicated transitions

Here, states *a*, *b*, and *c* are grouped inside cluster *p*; to be in cluster *p* is to be in only *one* of its child-states *a*, *b*, or *c*. The transition from cluster *p* to state *s* is taken regardless of which child-state the statechart is in; hence, the child-states need no transition to state *s*.

3. The exponential increase in the number of states when new states are added is eliminated by the introduction of *logical-and* state groups as shown in figure 1.2a. Henceforth, logical-and state groups shall be referred to as *sets*.¹ In the figure, states *a* and *b* are

* Current address: AT&T Bell Laboratories, Naperville, IL 60566, U.S.A., paul_j_lucas@att.com

¹ The term “state,” however, will be used universally to refer to states, clusters, and sets (since clusters and sets are *still* states); the terms “cluster” and “set” will only be used when it makes a difference. Also, the term “plain-state” will be used explicitly to mean a non-cluster and non-set state.

child-states of cluster x and states c , d , and e are child-states of cluster y ; clusters x and y , in turn, are child-states of set p ; to be in set p is to be in both child-states x and y . Figure 1.2b shows the conventional (rather messy) STD equivalent.

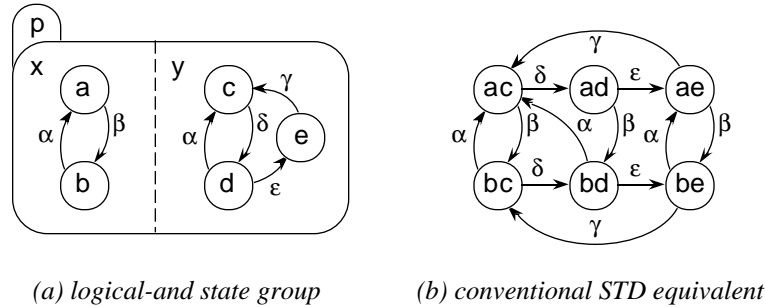


Figure 1.2: Statechart sets

4. Sets also allow concurrency. Assuming the statechart of figure 1.2a is in states b and d to start, if event α occurs, the statechart will simultaneously make transitions to states a and c .

1.2 Concurrent, Hierarchical, Finite State Machines

Concurrent, hierarchical, finite state machines (CHSMs) are a realization of statecharts just as finite state machines are a realization of state-transition diagrams. To develop actual working models of reactive systems, software was written to implement CHSMs [4]. The language system for CHSMs is a hybrid system in the tradition of *lex* [5] and *yacc* [6] where an existing (host) programming language was augmented with additional constructs. The host language chosen for CHSMs was C++ [7]. (A different approach for implementing statecharts was taken by the STATEMATE system [8].)

A CHSM source description has three parts as shown in figure 1.3: declarations, a description, and a user-code section; these are placed into a text file and are separated by the token `%%` (mirroring *yacc* grammar specifications).

```

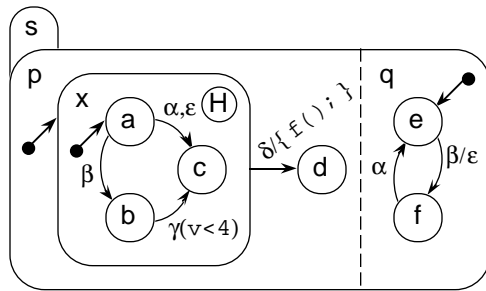
declarations
%%
description
%%
user code

```

Figure 1.3: CHSM source-description

Since CHSM descriptions are integrated with C++ code, the declaration section contains any useful declarations that the user wants to be used in the C++ code fragments such as global variables, constant declarations, etc. The description section would contain (naturally enough) a CHSM description given in the CHSM description language. An example of what the language looks like is given in figure 1.4b where the statechart of figure 1.4a is described.

The user-code section, if given, would contain any function definitions presumably declared in the declarations-section.



(a) sample statechart

```

set s(p,q) is {
  cluster p(x,d) is {
    cluster x(a,b,c) history {
      delta->d %{ f(); %};
    } is {
      state a {
        alpha,epsilon->c;
        beta->b;
      }
      state b { gamma(v<4)->c; }
      state c;
    }
    state d;
  }
  cluster q(e,f) is {
    state e { beta->f/epsilon; }
    state f { alpha->e; }
  }
}

```

(b) CHSM description

Figure 1.4: CHSM description-language example

2 A Graphical Editor

2.1 Overview

The reasons for developing this graphical editor, as opposed to just using STATEMATE, are: (1) STATEMATE is a proprietary, closed system that can not be used to take advantage of the features and extensibility offered by [4], and (2) STATEMATE costs \$50,000 for just the base package.

2.1.1 Look and Feel

A graphical-editor should allow the user to create and edit statecharts. An example of what an editor's window could look like is shown in figure 2.1. The window looks much like many object-based drawing editors'. The window elements, title, frame, scroll-bars, etc., would be those of whatever window-manager the user happens to be using.² The window would also include a tool-palette for the selection and creation of states and transitions and split-bars in order that the view can be split so that the user can look at different parts of the statechart simultaneously.

2.1.2 Appearance of States, Clusters, and Transitions

Plain-states would be drawn as rounded-corner rectangles large enough to enclose their name; clusters and sets would be drawn large enough to enclose their name and all of their child-states. Transitions would be drawn as splines³ with arrow-heads and the name of the first event drawn

² The figures used in this paper have the Macintosh "look-and-feel" since that is what the author has on his desk; the reader can substitute any windowing-system, however.

³ Foley, et al [9], pp. 478-516.

near the center-point of the spline. If there is more than one event, or the event has a condition or the transition has an action, then an ellipsis would be drawn after the first event's name (because the label would be too long otherwise).

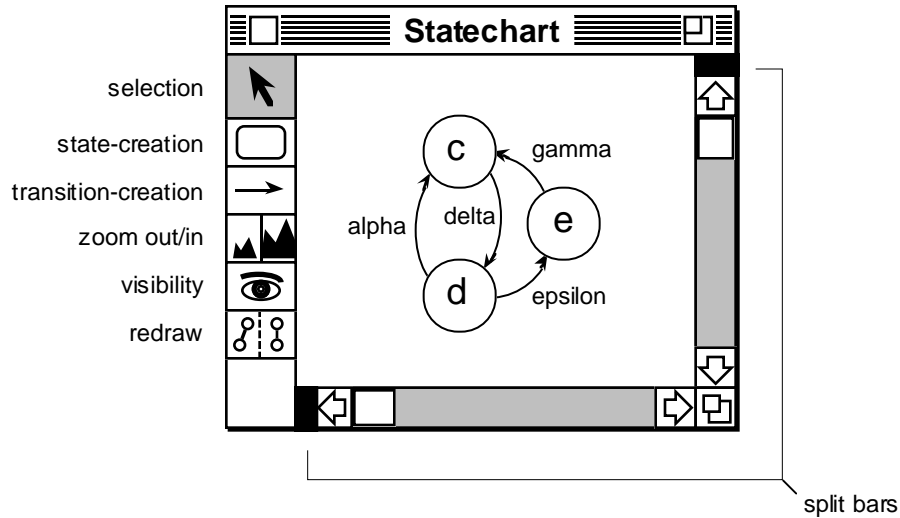


Figure 2.1: Example editor window

2.1.3 Appearance of Sets

Editing sets drawn in the conventional manner would be difficult. Referring to figure 2.2a, set names are drawn in the extra “tabs” that stick out, child-clusters share their borders with sets, and the dashed-line separators are just problematic.

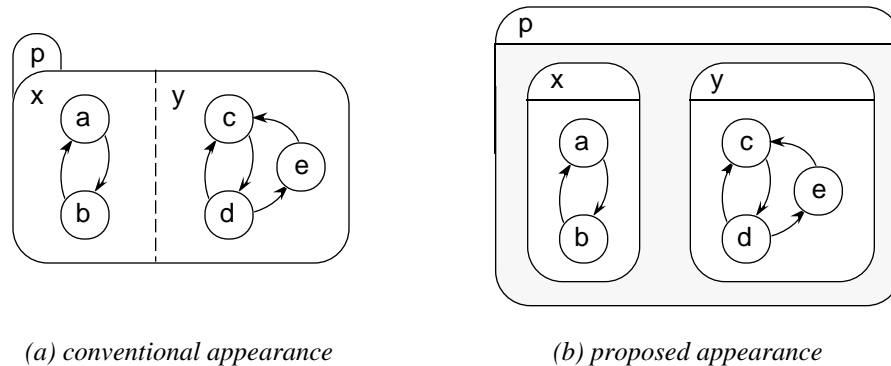


Figure 2.2: Drawing sets

The fact that child-clusters share their borders with sets means that clusters are drawn differently depending on whether their parent state is a set or not (making things difficult for the implementor). The dashed-line separators are problematic: How are they created by the user? What happens to child-states in different components if their separator is removed, i.e., which cluster “absorbs” the other?

A solution is to draw sets like clusters except with a lightly-shaded background like that shown in figure 2.2b. (Also, cluster and set names have been placed into a “title-bar” so as not to interfere with their child-states.) The shading not only distinguishes sets from clusters, but also

reinforces the semantics that child-states of a set can not have transitions between them, i.e., that each child-state is a separate component.

2.1.4 *Editing the Declarations and User-Code Sections*

In addition to allowing the user to create and edit statecharts, there would be commands on one of the editor's menus to edit the declarations and user-code sections of a CHSM description in a textual-editor window spawned by the graphical-editor.

2.2 Tool Description

2.2.1 *Selection Tool*

The selection-tool would be used to select an existing state or transition designating it as the object to which any subsequent action will be performed (§2.3). Selecting an object would display "handles" that would allow the object to be manipulated. Double-clicking a state or transition would pop-up a dialog box allowing the attributes of the object to be edited. The ability to select multiple objects simultaneously would also be supported.

2.2.2 *State-Creation Tool*

The state-creation tool would be used to create states, clusters, or sets. The mouse-pointer would change into the shape of a circle. Clicking in the drawing area would pop-up a dialog box, like that shown in figure 2.3, allowing the attributes of the new state to be specified.

The dialog box is titled "Create/Edit State". It contains the following elements:

- A text input field labeled "Name:" in the upper-left corner.
- Three radio buttons for state type: "State", "Cluster", and "Set". The "Cluster" radio button is selected.
- Three radio buttons for history options: "No History", "History", and "Deep History". The "No History" radio button is selected.
- Buttons for "OK" and "Cancel" on the right side.
- A "Derived Type:" label followed by a text input field containing "None" and a dropdown menu also containing "None".

Figure 2.3: Create/Edit State dialog box

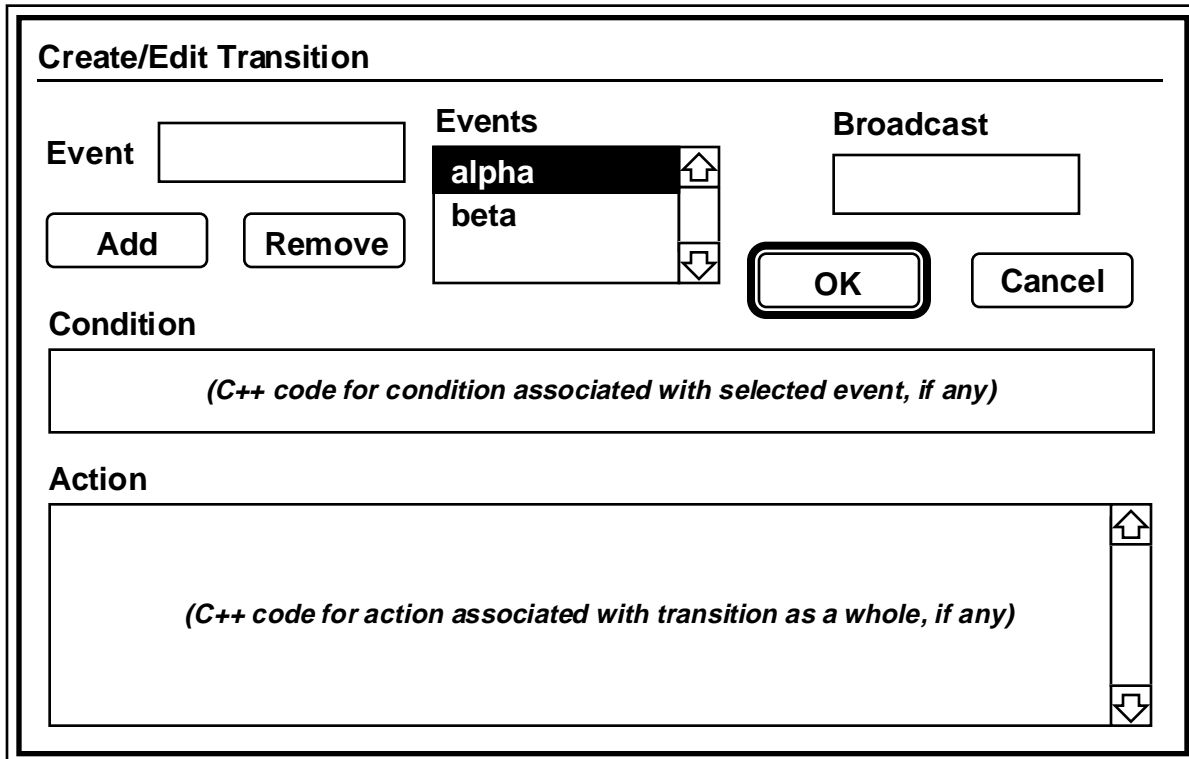
The state's name would be entered in the box provided in the upper-left. The type of state, that is whether the state being created is a plain-state, a cluster, or a set, would be chosen by the selection of one of the radio-buttons. The radio-buttons having to do with history would be disabled and "grayed-out" unless the Cluster radio-button had been selected. At the bottom, there would be a box provided where the name of a derived-type can be entered;⁴ the pop-up menu to the right would list all those derived-types that have been used previously allowing the user to select from those as opposed to having to type the name every time.

⁴ The language-system in [4] allows new kinds of states to be created by the user by using the C++ class-derivation mechanism. For example, the user could create a state that counts how many times it has been entered and exited.

The newly created state would be placed at the point clicked; if it is within a cluster or set, then the new state would be made a new child-state of that cluster or set.

2.2.3 Transition-Creation Tool

The transition-creation tool would be used to create transitions between states. The mouse-pointer would change into the shape of a right-pointing arrow. The user would click in the source state and drag the mouse to the target state; the drawing area would automatically scroll if necessary. Visual feedback would be given as the user drags the mouse in the form of a freehand line.⁵ Releasing the mouse in the target state would erase the line then pop-up a dialog box, like that shown in figure 2.4, allowing the attributes of the transition to be specified.



The dialog box is titled "Create/Edit Transition" and is divided into several sections. At the top left, there is an "Event" text box, an "Add" button, and a "Remove" button. To the right of the "Event" box is an "Events" list box containing "alpha" and "beta", with "alpha" selected and highlighted. Above the "Events" list is a "Broadcast" text box. Below the "Events" list are "OK" and "Cancel" buttons. The "Condition" section contains a large text box with the placeholder text "(C++ code for condition associated with selected event, if any)". The "Action" section contains a larger text box with the placeholder text "(C++ code for action associated with transition as a whole, if any)".

Figure 2.4: Create/Edit Transition dialog box

The name of an event that this transition responds to would be entered in the box provided at the upper-left. The Add button would then be clicked to add it to the list of events to the right; the button would be disabled unless there were something entered for the event name. If the event has a condition associated with it, then the C++ code for it can be entered in the box provided in the middle. Similarly, if the transition has an action associated with it, the C++ code for it can be entered as well; the broadcast-event associated with the transition, if any, can be entered in the box provided in the upper-right.⁶ Events can have the C++ code for their conditions edited by selecting their name from the list of events, or they can be removed from the transition entirely by clicking the Remove button.

⁵ A straight, "rubber-banding" line used in other drawing editors could not be used here since that would not allow a transition to be drawn from a state to itself.

⁶ A broadcast-event is an event that can optionally be broadcasted to the entire statechart upon making a transition with the desire being that other transitions will take place upon receiving it.

Clicking OK would then dismiss the dialog box and draw a smooth arrow (spline) from the source to the target state.

2.2.4 Zoom Tool

The zoom tool would allow the user to “zoom in” on areas of a statechart to focus attention on a particular component and, correspondingly, to “zoom out” to view the “big picture.” For zooming-in, the last point clicked at would designate the center of the area to be zoomed into.

2.2.5 Visibility Tool

The visibility tool would allow clusters and sets to have their child-states “hidden,” i.e., not drawn. Once certain components have been completed, hiding their contents would reduce clutter and allow statecharts to be redrawn faster. This tool is not a tool in the same sense as the aforementioned tools where the mouse-pointer changes shape and it is used to create an object; instead, this tool is really a button that operates on the currently-selected state.

The tool would toggle the visibility of the selected cluster or set; hidden states could be drawn as shown in figure 2.5b. Non-local transitions, like those between states *e* and *s*, would be drawn as shown to indicate that they go to or come from “some” child-state of cluster *y* and not *y* itself.

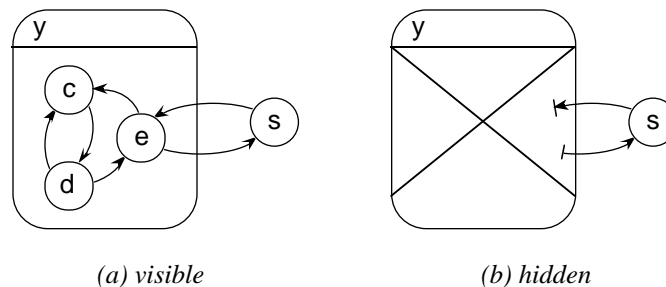


Figure 2.5: Hidden states

The overall size of the state would remain the same until the next time the user requests the statechart to be redrawn; once that occurs, the size of the state could be reduced (since its child-states are not being drawn) and thereby occupy less space in the overall drawing. If such a reduced state were to have its child-states made visible, it would, of course, have to return to its former size; doing so, however, may force the statechart to be redrawn because the state may no longer fit into the area that it’s currently in, i.e., surrounding states would have to be pushed farther away.

To edit a hidden state’s child-states, it could either be made visible and edited as usual, or a cluster or set could be edited in a separate window just showing the child-states.

Hidden states also show potential for lending themselves to form “component libraries,” i.e., collections of previously-developed clusters and sets that are used often. Once such components have been developed, their “details” (child-states) need not be of any concern to the user who just wishes to reuse them, i.e., just “plug” various components together.⁷ The graphical editor could also have commands for managing such component libraries.

2.2.6 Redraw Tool

After the user adds several states and transitions, the statechart may become “messy.” The redraw-tool would be used to have the entire statechart redrawn by the editor in a “tidier”

⁷ Although the user would have to know about broadcast-events that are generated, if any.

manner. (The subject of automatic graph drawing—a statechart is a specialized graph—will be fully addressed in §3.)

2.3 Editing

2.3.1 Moving & Resizing States and Transitions

A state can be moved by clicking on it and dragging the mouse. When released, all transitions attached to the state would move accordingly so as to remain attached. States could be moved in and out of clusters and sets; states would not be allowed to “straddle” other state boundaries.

Plain-states would not be able to be resized since their proper size only depends on the length of their name. Clusters and sets, however, would be allowed to be “stretched” to make room for additional child-states that the user may want to add by clicking on and dragging one of the state’s “handles”; clusters and sets would not be permitted to be made smaller than the bounding-box of all their child-states.

Similarly, transitions could be reshaped by clicking on and dragging one of the handles associated with a control-point for the spline. When transitions are reshaped, the endpoints of the arrow, i.e., the points at the source and target states, will remain fixed; only the middle-part will be permitted to be reshaped.

2.3.2 Cut, Copy, & Paste

The features of Cut, Copy, and Paste found in object-based drawing editors would be available. Cutting a state would remove it (and all of its child-states, if any) and all of the transitions connected to it (them) from other states. Copying a state would copy it (and all of its child-states and local transitions, if any) but not any transition connected to it (them). Pasting a state would place the last state cut or copied (plus its child-states and local transitions, if any) into the statechart in the area where the user happens to be looking at the time.

2.4 File Format

Once created, the editor would “compile” the statechart drawing into its CHSM language description and place it into a text file to be the input for the CHSM compiler.⁸ The file would also contain additional, editor-specific information embedded in “special” comments; being comments, they will be ignored by the CHSM compiler. Some of the editor-specific information would be things like the coordinates and sizes of states and the control-points of the splines used for transitions. An example is shown in figure 2.6; special comments are either `//#` or `/*#` ... `*/` followed by a keyword that the graphical-editor would recognize.

```
state a {
  // #xywh: (x,y) , width , height
  beta-> /* #points: (x1,y1) , (x2,y2) , . . . , (xn,yn) */ b;
}
```

Figure 2.6: Special comments example

Coordinates would be specified in *points* thereby being system-independent. This would also facilitate translation into PostScript for printing statecharts.

⁸ This implies that a copy of the CHSM compiler would be “folded” into the graphical editor in addition to the CHSM compiler being a stand-alone program.

3 Implementation

3.1 Windowing Systems

Obviously, a windowing-system is needed to support a graphical-editor application, complete with windows, dialog-boxes, buttons, menus, etc. Since the CHSM language system runs under the Unix operating system, a windowing-system that also runs under Unix makes the most sense. A wide-spread system that runs on many hardware platforms is the X Windows system [10]. However, programming for X Windows directly is fairly complex; user-interface tool-kits make the job easier.

The InterViews tool-kit is a library of C++ classes that define common interactive objects and handle user-interaction and operating-system events [11]. The window-system, in this case X Windows, is made entirely abstract from the application. Even with a tool-kit such as InterViews, however, the task of writing any application using a graphical user interface is still daunting.

Unidraw is a framework designed specifically for creating domain-specific graphical-editors and is built on top of the InterViews tool-kit [12]. By “domain-specific,” it is meant that Unidraw allows customized objects and relationships between objects to be defined on the part of the application. In the case of the statechart editor, the customized objects would be states, clusters, sets, and transitions, and the relationships between objects would be parent/child states and transitions between states.

Unidraw also supports multiple views, both on-screen and external. External views are used to store or represent a drawing in a file. For the case of the statechart editor, this would be the commented statechart description in the CHSM language or perhaps even a PostScript file for printing.

3.2 Drawing Statecharts

3.2.1 Goals

Drawing statecharts—which are specialized graphs—automatically will be the most difficult part of the statechart editor. The reason that they should be able to be drawn automatically is to “clean-up” a statechart after it has undergone revision. Ideally, statecharts should be drawn in an aesthetically-pleasing manner. The criteria for being aesthetically-pleasing are:

1. Draw related states close together. States are related if they have the same parent-state or they have transitions between them. This makes transitions shorter and easier to follow.
2. Draw transitions with as few crossings as possible.
3. Draw transitions such that they do not “hit” states. It is suggested that “graceful,” elliptical arcs be used.

3.2.2 Drawing Clusters

There has been much research on drawing graphs since doing so while satisfying the aesthetic criteria is NP-hard [13]. A good portion of the research deals with drawing hierarchical graphs [14] [15] [16]. In this context, the term “hierarchical” means that the graphs typically have a root node and the remaining nodes are successively positioned into levels (ranked) until leaf nodes are reached.⁹ Although statecharts are hierarchical, the hierarchy is drawn by nesting rather than ranking by level; therefore, algorithms of this class are not directly useful. However, [16] does provide many useful ideas—and detailed pseudo-code—on drawing graphs in general.

There are algorithms that do have some potential, however. In [17] and [18], the idea of viewing transitions as springs is presented. Nodes have a natural repulsive force between them;

⁹ States correspond to nodes (or vertices) in the literature; transitions correspond to edges.

they are only brought together by the attractive force exerted by the springs (transitions). The desired layout of the graph results when the total energy of all the nodes and springs (transitions) of the graph is at a minimum, i.e., all the nodes would “gravitate” into place.

Another idea with potential is presented in [19] and [20] where nodes are placed in circular, ring layouts. These two ideas can be combined: Place related nodes on a ring and allow them to repel each other, tempered by the attractive force of the transitions; nodes that are more related, because of more transitions between them (more springs), would be placed closer on the ring.

Since statecharts are nested, the number of nodes representing child-states of a cluster will probably be small (no more than several states). To minimize transition crossings, a heuristic of placing no more than three states on a ring could be used since there can be no crossings with only three states. Also, the ordering of the three states on a ring is irrelevant since there is really only one ordering—other orderings are either rotations or reflections of it. Additional states, if any, would be placed onto larger, concentric rings, gravitating toward those states in the inner ring to which they have transitions. An example layout is shown in figure 3.1a with the underlying rings.

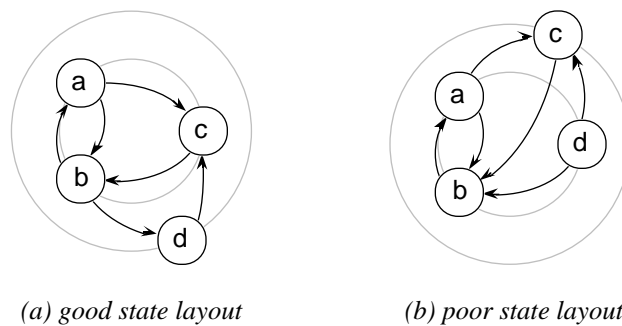


Figure 3.1: Node layout using springs and rings

The problem now becomes deciding which states to place onto which rings. A heuristic that can be used here would be to place states with higher *degree* (ignoring non-local and self transitions) on inner rings.¹⁰ In figure 3.1a, if states *c* and *d* were placed on opposite rings, the layout shown in figure 3.1b would result where this is regarded as a poorer layout than that in figure 3.1a. Using this heuristic, the number of inter-ring transitions will also be minimized (state *c* has a degree of 3 and state *d* has a degree of 2).

This layout scheme may not have to check explicitly for (and subsequently try to reduce) transition-crossings; it is hoped that it is sufficient to draw statecharts and that their layout will “just work out” in practice.

The nesting and the small numbers of states in a cluster would virtually allow them to be laid out independently of the rest of the graph and quickly; however, non-local transitions, like the one from state *b* to state *d* in figure 3.2a, affect the placement of states.

Rather than moving cluster *x* or state *d* so that states *b* and *d* are closer (probably affecting other transitions to cluster *x* and state *d*), the entire set of child-states *a*, *b*, and *c* can be rotated about the center of their ring inside of cluster *x* as shown in figure 3.2b, i.e., the transition (spring) between states *b* and *d* would pull them closer together.

3.2.3 Drawing Sets

Using the simpler appearance of sets (§2.1.3) and the fact that there are no transitions between child-states of sets, their placement within a set is arbitrary (except for non-local transitions; this could be handled similarly to the way a cluster’s child-states are, i.e., by rotation). The layout goal for sets would then be to “pack” child-states into as small an area as possible, but also into a

¹⁰ The *degree* of a node is the number of transitions entering and exiting it.

region as square as possible for aesthetic reasons. This is a variation of the “bin-packing” problem, which, unfortunately, is also NP-complete. There has been much research on bin-packing algorithms that use heuristics to achieve good packings in polynomial time [21]; of these, the ones that consider packing in two dimensions are relevant. These algorithms, given a fixed width for a bin, try to pack rectangles such that the height of the bin is minimized. (For the case of drawing sets, the rectangles correspond to the bounding-boxes of their child-states.)

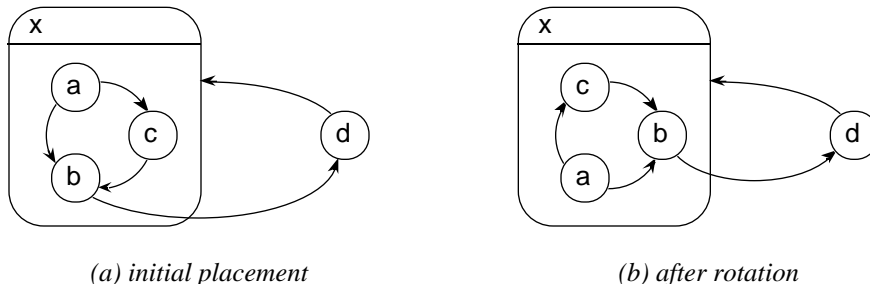


Figure 3.2: Adjusting cluster layout for non-local transitions

The first algorithm considered was the bottom-left, decreasing-width (BLDW) algorithm where rectangles are packed as tightly as possible as its name suggests [22]. Bin-packing algorithms are measured in terms of their worst-case performance as an asymptotic ratio between the heights of the resulting and optimally-packed bins; for BLDW, this ratio is 2.¹¹

The second algorithm considered was the first-fit, decreasing-height (FFDH) algorithm. This one differs from BLDW in that it is a “level” algorithm where rectangles are placed into the lowest level in which they will fit; the height of each level is determined by the tallest rectangle on that level which, because of the decreasing height ordering, is the first.¹² An example is shown in figure 3.3.

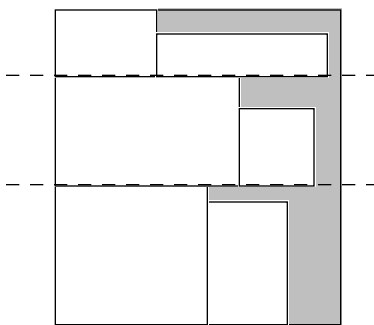


Figure 3.3: Example of first-fit, non-decreasing “level” algorithm

At first glance, it might seem that FFDH would be worse than BLDW because of the wasted space in each level above rectangles other than the tallest; however, it turns out that this space is bounded and that FFDH is *better* than BLDW with a performance ratio of 1.7 [21] [23].

The two algorithms presented disallow rectangles to be rotated 90° since it is stated in [21] that the performance bounds are in terms of area; however, in [23], it is suggested that a heuristic

¹¹ Packing rectangles into a bin such that they are as close to the bottom and left as possible seems fairly intuitive (bottom-right is the same, just reflected); however, why decreasing-width? It turns out that of the four possible orderings (increasing-width, increasing-height, decreasing-width, and decreasing-height), only decreasing-width has a worst-case bound, i.e., the ratio for the others is ∞ .

¹² A good analogy is filling a book-case with adjustable shelves.

where all the rectangles are oriented such that they are taller than they are wide may, according to the performance theorems, yield better results for smaller widths.

The two algorithms presented assume a fixed-width bin; to strive for our “square as possible” aesthetic goal, the following adaptation is suggested: Alternate the direction of packing between *up* and *right* as indicated by the orientation of the “gap” formed between the rectangles placed so far and the smallest bounding-square; if the gap is taller than wide, go right, otherwise, go up. This is illustrated in figure 3.4; the next rectangle to be packed would be to the right of the first rectangle. Switches in packing direction would occur only when the bounding-square is forced to “stretch” as a result of a rectangle not fitting into any available level. This algorithm shall be known alternating first-fit, decreasing-height (AFFDH).

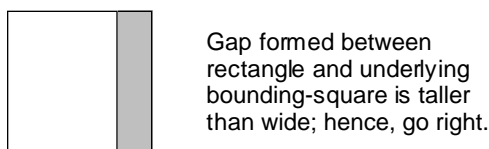


Figure 3.4: Determination of direction to pack subsequent rectangles

Since this algorithm progresses in two directions, the decreasing-height ordering is *relative* to the current direction: when going right, the ordering will remain decreasing by height; when going up, however, the ordering will be decreasing by width since, if everything is rotated by 90°, decreasing-width is equivalent to decreasing-height. Given an initial, unordered set of rectangles, like those shown in figure 3.5a, we shall create two orderings like those shown in figure 3.5b; the rectangles are numbered for reference.

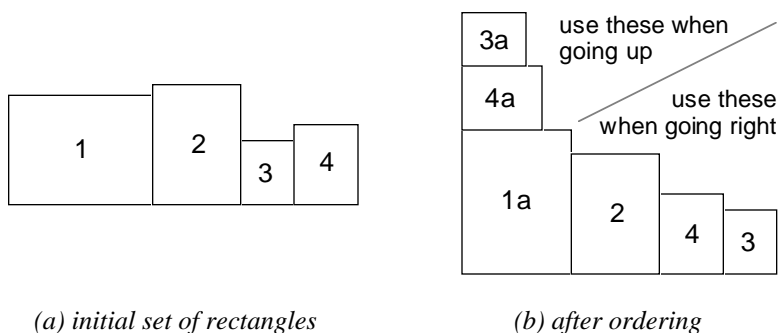


Figure 3.5: Rectangle orderings

For the *right* ordering, the rectangles are rotated 90° so that they are all taller than wide and ordered by decreasing-height; in the case of the given set of rectangles, only rectangle 1 actually needs to be rotated (yielding rectangle 1a) and rectangle 4 is placed before 3. For the *up* ordering, rectangles with number greater than 2 are “copied” then rotated so that they are wider than tall, and ordered by decreasing-width; in the case of the given set of rectangles, numbers 3 and 4 are copied, rotated, and ordered yielding rectangles 3a and 4a. As the algorithm progresses, when a rectangle is packed, its “twin” rectangle, the one created by copying then rotating it, will be deleted from the (other) list of rectangles remaining to be packed.

Since the original FFDH algorithm uses decreasing heights, this algorithm shall start off using the tallest rectangle and proceed to the right initially since the gap formed between it and the underlying bounding-square will be taller than wide. Because of this, the second rectangle to be packed (rectangle 2 in the figure), will always be placed to the right in its vertical orientation; hence, there is no need to copy either the first or second rectangles (that is why there is no rectangle 2a in the figure).

The first two steps just mentioned are illustrated in figure 3.6a,b; the dashed line demarcates the levels. In (b), the bounding-square has been “stretched” to accommodate rectangle 2; now the gap formed is wider than tall so the algorithm will switch to packing upwards; correspondingly, the orientation of the level-line also changed. (There are now two levels.) The next rectangle to be packed from figure 3.5b (from the up-list) is number 4a; it will not fit into rectangle 1a’s level, but it will fit into number 2’s resulting in the packing shown in (c). Still packing upwards, the next rectangle is number 3a and it will fit into rectangle-1a’s level resulting in the packing shown in (d).

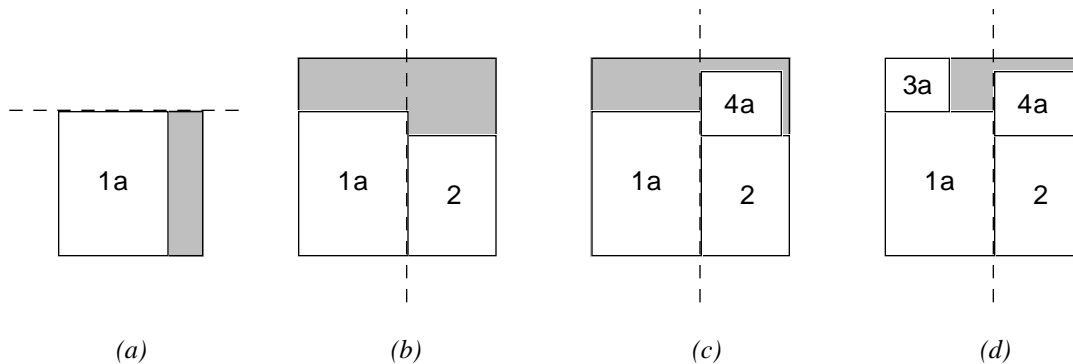


Figure 3.6: Progression of the bin-packing algorithm

Given that the number of rectangles (child-states) per set is presumed to be small in practice, it is assumed that AFFDH based on FFDH will yield good results. If it turns out that algorithms better than FFDH are needed as the starting point, they can be found in [24] [25]; however, these algorithms are considerably more complex.

3.2.4 Commentary

Even though graph-drawing and bin-packing are NP-complete problems, it may not actually matter in the case of drawing statecharts. Since the number of child-states per cluster or set is presumed to be small in practice and computers are getting faster, it may be reasonable to use “brute-force” algorithms where all the possible permutations are tried and the best is selected.

3.3 Related Work

In [26], Paulisch also discusses the user-interface, hierarchical graph-layout, and file-format issues raised here in some detail for developing her own graph editor, EDGE (Extensible Directed Graph Editor). Described in chapter 8, EDGE is written in C++ under X Windows and can use any one of four graph layout algorithms. Even though none of these would be suitable for drawing statecharts, EDGE claims to be extensible in that other layout algorithms can be added. EDGE also supports multiple views, hidden nodes, and writing a textual representation of the graph to a file. Graphs are saved in GRL, a Graph Representation Language, but it too claims extensibility by allowing external input/output routines to be used; presumably, routines could be written to read and write the CHSM description language.

EDGE defines internal data-structures for nodes, edges, etc.; all that would be necessary would be to add or modify EDGE source code for drawing statecharts. Paulisch herself states:

When constructing a new application, it is often simpler and more elegant to integrate application and EDGE data-structures by extending EDGE’s basic class definitions. The application may add new attributes and operations and even redefine existing operations.

In developing a graphical editor for statecharts, the EDGE system is worthy of serious consideration.

4 Future Work

This paper has proposed how a graphical-editor for statecharts should look and function and provided directions for its implementation using either InterViews and Unidraw or EDGE, and also presented a set of graph-drawing and bin-packing heuristics for drawing statecharts. Future work would naturally be to implement such a system and see how it would live up to expectations.

References

- [1] David Harel, et al. “On the Formal Semantics of Statecharts.” *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, IEEE Press, NY, 1987. pp. 54–64.
- [2] David Harel. “On Visual Formalisms.” *Communications of the ACM*, vol. 31, no. 5, May 1988. pp. 514–530.
- [3] ——. “Statecharts: A Visual Formalism for Complex Systems.” *Science of Computer Programming*, vol. 8, 1987. pp. 231–274.
- [4] Paul J. Lucas. “An Object-Oriented Language System for Implementing Concurrent, Hierarchical, Finite State Machines.” *M.S. Thesis*, University of Illinois at Urbana-Champaign, Department of Computer Science, 1993.
- [5] M. E. Lesk. “Lex—A Lexical Analyzer Generator.” *Computing Science Technical Report 39*, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [6] S. C. Johnson. “Yacc—Yet Another Compiler Compiler.” *Computing Science Technical Report 32*, AT&T Bell Laboratories, Murray Hill, NJ, 1975.
- [7] Bjarne Stroustrup. *The C++ Programming Language*, 2nd ed. Addison-Wesley, Reading, MA, 1991.
- [8] David Harel, et al. “STATEMATE: A Working Environment for the Development of Complex Reactive Systems.” *IEEE Transactions on Software Engineering*, vol. 16, no. 4, April 1990. pp. 403–414.
- [9] James D. Foley, et al. *Computer Graphics: Principles and Practice*, 2nd ed. Addison-Wesley, Reading, MA, 1990.
- [10] Robert W. Scheifler and James Gettys. “The X Window System.” *ACM Transactions on Graphics*, vol. 5, no. 2, August 1983. pp. 57–69.
- [11] Mark A. Linton, John M. Vlissides, and Paul R. Calder. “Composing User Interfaces with InterViews.” *Computer*, vol. 22, no. 2, February 1989. pp. 8–22.
- [12] John M. Vlissides and Mark A. Linton. “Unidraw: A Framework for Building Domain-Specific Graphical Editors.” *ACM Transactions on Information Systems*, vol. 8, no. 3, July 1990. pp. 237–268.
- [13] Peter Eades and Roberto Tamassia. “Algorithms for Drawing Graphs: An Annotated Bibliography.” Technical Report No. CS-89-09 (revised version), Brown University, Department of Computer Science, Providence, RI, October 1989.
- [14] John N. Warfield. “Crossing Theory and Hierarchy Mapping.” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-7, no. 7, July 1977. pp. 505–523.
- [15] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiro Toda. “Methods for Visual Understanding of Hierarchical System Structures.” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. SMC-11, no. 2, February 1981. pp. 109–125.
- [16] E. R. Gansner, E. Koutsofios, S. C. North, and K. P. Vo. “A Technique for Drawing Directed Graphs.” AT&T Bell Laboratories, Murray Hill, NJ, 1993.
- [17] Peter Eades. “A Heuristic for Graph Drawing.” *Congressus Numerantium*, vol. 42, 1984. pp. 149–160.

- [18] Toshima Kamada and Satoru Kawai. “An Algorithm for Drawing General Undirected Graphs.” *Information Processing Letters*, vol. 31, 1989. pp. 7–15.
- [19] Marcello G. Reggiani and Franco E. Marchetti. “A Proposed Method for Representing Hierarchies.” *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 18, no. 1, January / February 1988. pp. 2–8.
- [20] Margaret A. Bernard. “On the Automated Drawing of Graphs.” *Proceedings of the Third Caribbean Conference on Combinatorics and Computing*, 1981. pp. 43–55.
- [21] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. “Approximation Algorithms for Bin Packing—An Updated Survey.” *Algorithm Design for Computer System Design*, G. Ausiello, M. Lucertini, and P. Serafini, editors, Springer-Verlag, NY, 1984. pp. 49–106.
- [22] Brenda S. Baker, E. G. Coffman, Jr., and Ronald L. Rivest. “Orthogonal Packings in Two Dimensions.” *SIAM Journal on Computing*, vol. 9, no. 4, November 1980. pp. 846–855.
- [23] E. G. Coffman, Jr., M. R. Garey, D. S. Johnson, and R. E. Tarjan. “Performance Bounds for Level-Oriented Two-Dimensional Packing Algorithms.” *SIAM Journal on Computing*, vol. 9, no. 4, November 1980. pp. 808–826.
- [24] Brenda S. Baker, Donna J. Brown, and Howard P. Katseff. “A $5/4$ Algorithm for Two-Dimensional Packing.” *Journal of Algorithms*, vol. 2, 1981. pp. 348–368.
- [25] Igal Golan. “Performance Bounds for Orthogonal Oriented Two-Dimensional Packing Algorithms.” *SIAM Journal on Computing*, vol. 10, no. 3, August 1981. pp. 571–582.
- [26] Frances Newbery Paulisch. “The Design of an Extendible Graph Editor.” *Ph.D. Dissertation*, Karlsruhe University, January 1992.