An Interactive Editor for the Statecharts Graphical Language

Stephen Edwards

June 26, 1995

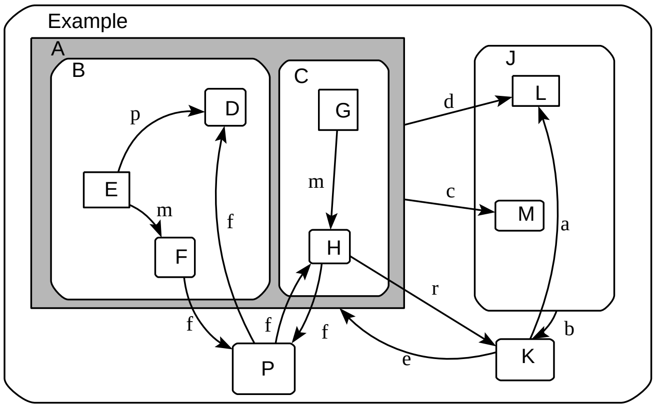

Figure 1: A Statechart

1 Overview

The objective of this project was to produce a graphical, interactive editor for the Statecharts graphical language, which uses hierarchically-arranged finite state machines. I chose this model after users requested I extend my previous finite-state machine editor to include hierarchy.

A Statechart contains two sorts of entities: states, which are boxes with names, and transitions, which are curved arrows connecting two states. A Statechart drawn with my editor is shown in Figure 1.

The editor can

- add and delete states and transitions
- move states and transitions, either preserving hierarchy or modifying it

  When states move, transitions remain attached. Transitions' shapes either come from a single index of curvature, or can be specified arbitrarily by the user.
- edit multiple views of the database, each starting from a different state in the hierarchy, going down a different depth, and with different magnification

  For example, the Statechart of Figure 1 can be displayed with a depth of 2 (Figure 2) and starting with state A (Figure 3) in two windows simultaneously.
- edit attributes (fonts, colors, shape) of states and transitions in dialogs
- read and write the database to disk
- print a view of the database to a PostScript file

This editor is written using McLennan's `[incr Tcl]` package, which brings an object-oriented data model to Tcl. My previous editor, written in straight Tcl, made it clear that the more elaborate database required for the Statecharts editor would be difficult to implement in straight Tcl.

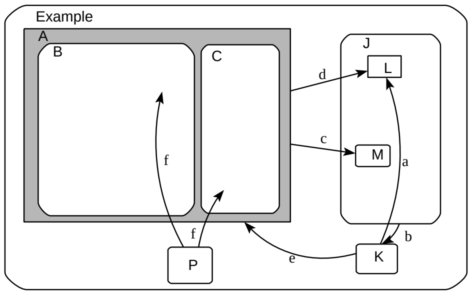

Figure 2: The Statechart of Figure 1 displayed with depth 2

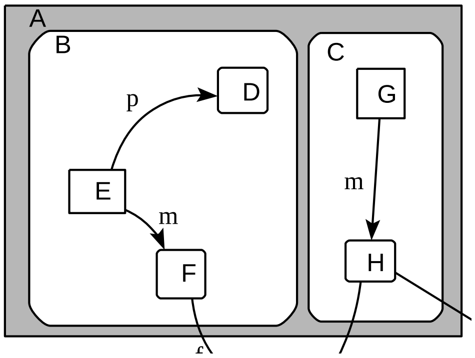

Figure 3: The Statechart of Figure 1 displayed starting with state A

In Section 2, I describe the architecture of the editor in detail by describing the major classes. In Section 3, some later-corrected bad design decisions are listed. Finally, Section 4 contains the conclusions I reached from building this editor.

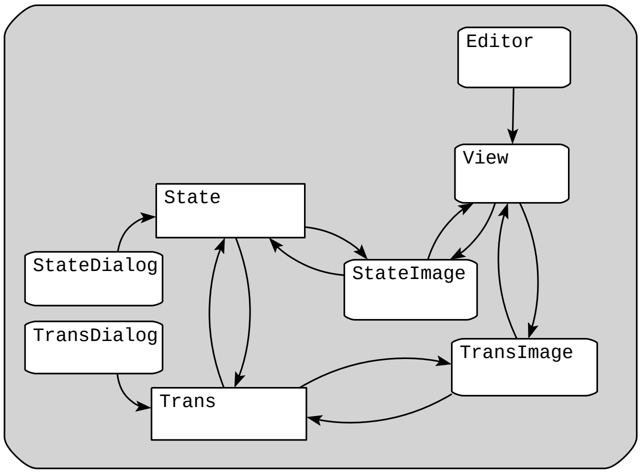

Figure 4: Major classes and their relationships

2 Architecture

The major classes and their relationships are shown in Figure 4. The `State` and `Trans` store non-derived information; these alone are stored and retrieved from disk. Instances of the `StateImage` and `TransImage` classes, derived from instances of `State` and `Trans`, describe how states and transitions are currently being displayed. The `View` class surrounds a canvas widget with the additional information required of a particular view of the database, including a list of all `StateImage` and `TransImage` instances displayed on the canvas. The `Editor` class surrounds a `View` instance, and is responsible for tool selection, controlling what appears in the `View`, and so forth.

2.1 The `State` Class

The `State` class contains

- the state's name

  This can be specified at creation

- the state's parent, none if it is the root state

  This is specified at the state's creation.

- the state's children

  This list is maintained by two methods, `attachChild` and `detachChild`, called by the constructor and destructor for this method. This scheme requires a state's parent to exist before the state can be created. To ensure this when a database is read, the states are written in a tree order.

- display information for the state:

  – a bounding box

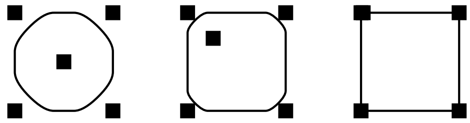

Figure 5: Control for the state's shape: (a) circular ($c_x = c_y = 0.5$) (b) a rounded rectangle ($c_x = c_y = 0.1$) (c) rectangular ($c_x = c_y = 0$)

  – two numbers controlling the shape of the drawn state

  – the color, font, and position of the state's name

  – the color of the state and the fill pattern for its outline

  In one of my early attempts, I attempted to place this information in a separate class (see Section 3). For performance reasons, it was placed in the `State` class. If additional view types were added, the `State` class would grow cluttered—a clean separation would prevent this.

- a list of attached transitions

  Although this information can be recreated from the transitions themselves, it is stored explicitly here for performance reasons—when a state moves, it must inform all of its attached transitions about the move, since they will need to update themselves.

  This information is not saved—the transitions themselves keep their state's lists current using the methods `attachTrans`, `detachTrans`, `attachTransD`, and `detachTransD`. The constructor and destructor for the `Trans` class call these methods.

- a list of attached `StateImage`s

  When a state's information changes (e.g., its position), all attached `StateImages`, which are displaying the state, are informed of the change by the `config` method defined on the `State` class. For a single view type, such as is currently implemented, this scheme is easiest, but as view types increase, this will become error-prone.

  The constructor and destructor for the `StateImage` class calls the methods `attachImage` and `detachImage` defined on the `State` class to maintain this list. This is the best time to update this list, but including it explicitly in the routines is error-prone.

The interesting methods defined on the `State` class are

- `attach` and `detach` methods for child `State`s, `StateImages`, and source and destination `Trans`es

  These methods are used by the constructors and destructors of the other classes to maintain the various lists.

- `recreate`, which returns a string, when executed, that recreates the instance

  The procedure that writes the database to disk use this method.

  Only those values which differ from their defaults are written out explicitly to save space. One danger of this is if the defaults change. A better way would be to write out the defaults at the beginning of the file.

- `select` and `unselect`

  These call the `select` and `unselect` methods for any attached `StateImage` in a given view—selection is by view, not global. Currently, these are slow because the view of each attached `StateImage` must be compared with the requested view.

  These are called by the `select` and `unselect` methods of the view class.

- `move`

  Moving a state is a common operation, so I wrote a special method for it. This moves the two coordinates defining the state's bounding box, the position of the name, instructs all its child states to move themselves, and informs each of the transitions that they have moved using the `moves` and `moved` methods.

2.2 The `StateImage` Class

The `StateImage` class contains

- the image's master `State`

- the `View` in which this image appears

- the canvas IDs for the polygon, the smoothed line, and the name

  No single canvas object was rich enough for what I wanted to display, so I built one. These IDs are used when the state changes; the `config` method modifies the coordinates and attributes of the canvas objects to correctly reflect the state of its master `State`.

- the canvas IDs for the four corner handles, the name handle, and the control handle

  The four control handles allow the user to move each of the four corners of the state. The name handle allows the state's name to be moved. The control handle allows the user to adjust the shape of the state, as shown in Figure 5.

  Since the handles only appear when the state is selected, these IDs do not always exist.

The interesting methods defined on the `StateImage` class are

- `config`

  This takes the display information from the master `State` and adjusts the coordinates and attributes of the various canvas objects comprising the state.

- `select` and `unselect`

  These create and delete the selection handles.

- `arrowB1`, `arrowB1motion`, ...

  Methods for each relevant binding are defined on the image. For example, the `arrowB1` method is invoked when button 1 is pressed with the arrow tool somewhere on one of the objects comprising the `StateImage`. This selects the current object if it is not selected, unselects everything else in the view if wasn't already selected, and remembers which object was clicked.

  Mouse events pass the coordinates of the event and a canvas-object specific piece of data that identifies which canvas object of the `StateImage` received the event. For example, a click on any of the selection handles invokes `arrowB1`, but which selection handle was clicked is also passed to `arrowB1`.

2.3 The `Trans` Class

The `Trans` class contains

- the transition's source and destination states

  Both specified at creation, these are used by the `Trans` constructor to attach itself to its source and destination states.

- the transition's label

  At the moment, this is an arbitrary one-line string, but when this information is used elsewhere, it will have to conform to some format. It is not clear whether this format should be enforced in this program, or in fact, how it should be enforced.

- the color, font, and position of the transition's label

  Much like the display information in the `State` class, this should probably be in a separate class to facilitate adding view types.

- four control points for the spline used to draw the transition

- a single number describing the curvature of the arc

- a flag indicating whether the curvature number should be used to calculate the control points for the spline and the position of the label

  When this flag is set, every time the transition is informed that one its attached states is moved, all four control points and the position of the label are recomputed from the arc curvature number and the display information of

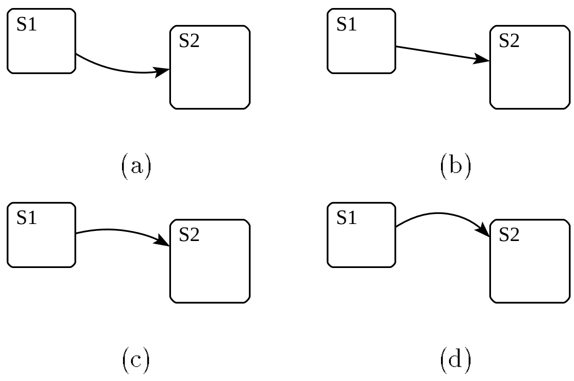

Figure 6: Control for a transition's curvature: (a) $c = -0.1$ (b) $c = 0$ (c) $c = 0.1$ (d) $c = 0.2$

the two states. The curvature number sets how much the spline deviates from a straight line connecting the centers of the two states (Figure 6).

When this flag is cleared, the four control points and the position of the label are allowed to change independently.

- a list of attached `TransImages`

  When a transition's shape changes, all attached `TransImages` are informed. This list is updated and maintained analogously to the list in the `State` class.

The interesting methods defined on the `Trans` class are

- `recreate`

  Like the `recreate` defined on the `State` class, this returns a string that, when executed, recreates the transition. Only values differing from their defaults are written. One twist is that when the fixed curvature flag is set, the control points are not written, since they can be regenerated.

- `snapArc`

  This calculates where the four control points and the position of the label should be based on the specified curvature value and the positions of the source and destination state. This is called by methods defined on the `Trans` class when the position of either state changes.

- `moves` and `moved`

  These are invoked by the `State`'s `move` method, and either call `snapArc` or move the spline's control points. The control point on the source or destination state (hence the two flavors) is moved the given distance, the next nearest is moved 75% of the given distance, the next, 25% of the distance. This way, when both states move the same distance, all four control points are translated that distance, yet when only one state moves, both ends stay attached and the in-between points do something reasonable.

- `select` and `unselect`

  Like their `State` counterparts, they invoke a `select` or `unselect` method for an attached `TransImage` with the given view.

2.4 The `TransImage` Class

The `TransImage` class is analogous to the `StateImage` class, and stores similar data: the master `Trans`, the `View`, a list of canvas IDs for the transition, and a list of canvas IDs for the selection handles. The methods are for the most part similar, although the methods for bindings behave in very different ways.

2.5 The `View` Class

The `View` class can be thought of as a wrapper for the canvas widget. The canvas widget has much, but not all of the required functionality. The biggest omission in the canvas widget is a way to go from canvas objects to objects in the database. I also wanted a zoom feature, something the canvas widget doesn't do elegantly.

The `View` class contains

- the topmost state in the view, and depth to display

  Each view displays a subtree of states, starting at the specified root and going down for a specific depth.

- a zoom and pan

  Converting from database to canvas coordinates involves subtracting the pan and then multiplying by the zoom.

- a list of all state and transition images in the view

  These lists are maintained by the constructors and destructors of the `StateImage` and `TransImage` classes using the `attach` and `detach` methods defined on the `View` class.

- a mapping between canvas IDs and the objects they belong to

  This identifies which image created each canvas object. Since each image is composed of multiple canvas objects, this is essentially a grouping of canvas objects.

- a mapping between canvas IDs and their object-specific types

  Each image is composed of multiple canvas objects, and this distinguishes them by function. For example, each selection handle on a particular image has a different type.

- a list of all selected objects in the view

  Each view has its own set of selected objects: `State` and `Trans` are selected, not their images.

  Doing things this way is somewhat inconvenient: a selection request is passed to the view, which passes it to the object, which passes it to the image. But the view must pass itself to the object, since the object must determine which image will actually be selected.

The alternative is keep a list of selected images and whenever something is done to a list of selected objects, any change is eventually passed back to their masters.

The interesting methods defined on `View` are

- `attach` and `detach Image` and `TransImage`

  This are invoked by the constructors and destructors of the `StateImage` and `TransImage` classes respectively. The one twist is that the detach methods also remove from the selection list the object being detached.

- `create` and `del`

  These are wrappers for the canvas `create` and `delete` method that update the mappings between canvas objects and objects (`StateImage`s, `TransImage`s).

- `addHandle`, `positionHandle`

  These methods add and relocate a single selection handle. The `select` methods defined on the two image classes invoke `addHandle`.

- `select` and `unselect`

  These update the view-specific list of selected objects and invoke the `select` and `unselect` methods defined on the given object.

- `findStates`

  Given a bounding box and either `enclosed` or `overlapping`, this method returns a list of all the `StateImage`s in or near the region. This is used when states are moved and their position in the hierarchy is changed, when new states are added, etc.

- `redraw`

  This destroys all the attached images and regenerates them, invoked when the depth or root state of the view changed.

The system could be extended to support different ways of displaying part of the database. The equivalent of the `StateImage` and `TransImage` classes would be added, and a variation of the `View` class would be needed. Class inheritance would be perfect here: some aspects of the `View` class would be useful for all canvas-oriented views, such as the object ID-to-object arrays and the selection mechanism, but others, such as the zoom mechanism, are view-specific.

2.6 The `StateDialog` and `TransDialog` Classes

The constructor for the `StateDialog` class takes a `State` object as an argument and creates a non-modal dialog that can edit the attributes (name, name font, name color, box fill color, and outline stipple) of that state. The dialog's two buttons are Apply, which copies the attributes in the dialog to the database, changing all views in which the state appears, and Dismiss, which ends the dialog.

The `TransDialog` class behaves similarly, except it edits transition-specific attributes.

2.7 Bindings

There are currently four tools: the arrow tool for moving, selecting, and stretching things; the state tool for creating new states; the arc tool for creating new transitions; and the text tool for editing the text attached to states and transitions. Really, these tools are different ways of interpreting key and mouse events.

New tools are defined using the `newtool` procedure:

```
newtool arrow @$bmd/arrow.xbm left_ptr
```

The arguments are the name of the tool, the bitmap to display on the tool palette, and the cursor to display in the canvas.

The mapping between my names for bindings and X's are defined with an array:

```
set bindings(B1) <Button-1>
set bindings(B1Release) <ButtonRelease-1>
```

This gives the program a list of all possible bindings.

The procedure called when an event occurs is defined in a two-step process:

```
bindproc {%v %x %y} {view x y} {
  set object [$view currentObject]
  if { $object != "" } {
    if { [ $object info method arrowB1motion \
           -body ] != "" } {
        $object arrowB1motion \
           $x $y [$view currentData]
    }
  }
}
bindto arrow B1Motion
bindto arrow ShiftB1Motion
bindto state B2Motion
```

This scheme allows the same procedure to be called for multiple events across multiple tools. In this example, the actual arguments of the bound procedure are the view and the coordinates where the event occurred (`%v %x %y`), the formal arguments are `view`, `x`, and `y` (variable names in the procedure), and the given procedure is bound to the arrow tool's `B1Motion` event, the arrow tool's `ShiftB1Motion` (mouse movement with both the first button and shift key held down), and the state tool's `B2Motion` event.

The procedure in this example first asks the `View` on which object (image) the event occurred. For example, if the user was dragging a handle on a `StateImage`, it would return that particular `StateImage`. Next, if there is an object there, it checks to see if there is a method `arrowB1motion` defined on the object. If so, it invokes the method, passing the coordinates of the event and the type (`currentData`) of the canvas object (e.g., `NWHandle`, `spline`, etc.).

Some actions are best done to the entire selected set, others are best done to a single object. Both schemes can be implemented in this framework. For example, dragging a state should move all selected states. When the method for `arrowB1motion` is given an event coming from the outline of a state, it invokes the `move` method on all the selected states explicitly. Dragging handle, on the other hand, only affect the state or transition whose handle it is.

3 Dead Ends

My first attempt had the information about how to draw a state (position, size, color, etc.) in a class separate from the one with its name and parent. This would have led to inefficiency, since virtually every operation would have to access both of these objects. Additionally, a third class would have been required to store view-specific information, i.e., a canvas, the object IDs for that canvas, etc. Also, it made programming a mess: I would have to remember which object certain attributes are attached to. For example, is the name of the state attached to the structural or visual representation?

The idea behind this may have been sound: isolate the information about a state relevant only for display from that relevant to other uses. However, in this case, it didn't seem wise to start splitting the data so early. Something like this must be done eventually, however, since it might be worthwhile to add additional "views" of the state outside of the single graphical one. For example, a tree-structured view that the user can rearrange might also be desired, in which case splitting the data like this would be a worthwhile technique.

In my second attempt, I combined the two classes and included information about things like canvas object IDs in a single `State` class. This was a wrong turn because it would not support multiple views of the same database—each state could only be responsible for a single set of objects on a single canvas. The canvas, in fact, was a global variable.

In my third attempt, I split the `State` class into two, the `State` class, which has information about the state—its name, position, shape, etc.—and the `StateImage` class, which has information about where the state is currently being displayed—the canvas, the object IDs on that canvas, etc.

In this attempt, I tried a scheme for positioning states relative to a state's parent. The idea was that when a state moved, all of its children were translated implicitly. There were a number of problems with this. First, this made it very time-consuming to display a set of states, since a fairly complex operation had to take place to figure out where one actually was. Rather than try to recompute each position completely from scratch, I computed each state's position and size based only on its parent's position and size, making sure to do this in a breadth-first manner. It was a problem to keep this cache consistent.

Another problem was what to do with the root of a particular window. It did not have a displayed parent, so where would its geometry be derived from? You could create a dummy parent with a given geometry that responded to the root's requests, but this seemed problematic. Instead, I tried treating the root state of each window separately. The root state would have its geometry handed to it, and all the others would be instructed to position themselves relative to this. A problem was making sure that the children were always up-to-date, since they had to be told to position themselves explicitly, effectively removing the utility of scheme to take care of positioning automatically.

In the fourth attempt, I simplified the position model. Now, instead of defining the upper left-hand corner of a state as a fraction of the way from the upper left-hand corner of its parent, I defined it as an absolute distance from the upper left-hand corner of its parent. Width and height which, until the fourth attempt, were defined as fractions of the width and height of the parent, were now given in absolute coordinates. However, this scheme suffered from most of the problems of the last: maintaining consistency remained a problem. Whenever the position of a state was changed, this change was noticed and passed down to the children, which would be told to reposition themselves.

All of this recomputation and cache-coherency maintenance made for very slow code. It was clear that to make the program run with any reasonable speed, something different was required. Eventually, all of this relative position nonsense was replaced by an absolute positioning scheme, modulo the pan and zoom of the current view. But transforming all a view's objects uniformly is much easier than doing each differently, so this turned out to be a partial solution to the speed problem.

4 Conclusions

There should be an environment that makes building multi-view direct manipulation graphical editors easy. This editor suggests what that environment should be like.

The environment should be object-oriented. The object-oriented `[incr Tcl]` package was an immense help in developing this editor: it allowed the easy addition and removal of attributes during the development process, and it made most code much cleaner.

The environment must support relationships between objects. In the current editor, all relationships are maintained explicitly by objects' constructors and destructors. Adding a new relationship requires modifying both classes, and this is error-prone.

The environment must support implicit constraints. All constraints in this editor were coded explicitly, and when new constraints are added, many pieces of code must be changed.

The environment must support finely-grained constraints.

This editor uses a coarse constraint mechanism: when any part of a master changes, the slave is instructed to recompute *everything*. This is wasteful: the complex state shape calculation is expensive, and needless when only a state's name changes. Earlier in the development, there were many more expensive calculations and the editor's speed was unacceptable. Even after removing most of these, it is still slower than I would like.

The environment should have support for undo, a cut-and-paste facility, a grid mode, a "changes have been made" flag, or a good scheme for keyboard bindings across multiple windows (shortcuts, etc.). This editor does not have these.

To make adding view types easy and safe, the database should have its structural information better-separated from view-specific information. At the moment, structural information and view-specific information are stored in the same class. These should be separated so another view can be added without having to modify (and potentially break) the existing views.

I consider this editor to be a partial success. It does many of the difficult things I intended: multiple views, direct manipulation, and rich graphical objects. But I'm disappointed that the system is so slow—I attribute this to using the `[incr Tcl]` package and object-oriented features that make the program easier to write, but slower. In many cases, its speed makes it frustrating to use. The solution to this, I believe, is a more finely-tuned architecture, and things like "pacers" that keep responsiveness high at the expense of image degradation.