Requirements for a Dataflow Diagramming Tool

Version 0.1
January 2006

Contents

1. Introduction

This document defines requirements for a dataflow diagraming tool, called "`dftool`". The tool provides functionality for drawing and editing dataflow diagrams (DFDs). It also supports the translation of DFDs to and from their textual representation in a formal modeling and specification language (FMSL).

Some basic requirements for `dftool` are presented in the users manual for a previous implementation of RSL support tools. Sections 7 and 8 of that manual discuss features of the dataflow editing.

The set of tools described in the old manual were implemented in C and C++ to run in an X Windows environment, and are not portable to other platforms. A key objective of the current dftool project is to design and implement a multi-platform version of the dftool in Java. Initially, the Java-based dftool will be a stand-alone application. At some later point, it may be integrated into an Eclipse environment for FMSL.

The requirements presented here refine and extend the earlier tool's features. Particular new features of note are the following:

a. The new dftool can operate as a fully stand-alone application, in addition to operating as a component of a larger integrated FMSL development environment; the requirements here focus on its stand-alone interface.

b. These requirements have significant enhancements for the connectivity of dataflow nodes in a diagram, in particular the explicit display of input/output ports, and incremental validation of inter-node connections.

c. The FMSL textual representation of a dataflow diagram has been enhanced since the implementation of the earlier RSL-based tool. These new requirements describe the formal mapping from a graphical dataflow diagram to its underlying FMSL specification.

1.1. Problem Statement

The problems to be solved stem from the inadequacies of the earlier version of the dataflow tool, which are:

a. non-portability;

b. smaller-than-desired set of features;

c. lack of support for updated FMSL dataflow syntax and semantics.

1.2. Project Personnel

The primary end users are students in Cal Poly undergraduate software engineering classes, specifically CSC 307, 308, and 309.

The primary developer for January through June 2006 is Cal Poly senior project student Tim Ober, supervised by Gene Fisher. Ober will do design and implementation, Fisher will do requirements.

1.3. Operational Setting

The stand-alone version of `dftool` is designed to run on any major computing platform, in particular Microsoft Windows, Apple Mac OSX, and Linux. The necessary Java support must be installed on the running platform.

As a component of an integrated development environment, the most likely setting will be as a plug-in to the Eclipse IDE. These requirements do not address this aspect of `dftool` integration.

1.4. Impact Analysis

The desired positive impacts are these:

a. increased use of DFDs in software engineering courses

b. better support for formal modeling in FMSL

Regarding the increased use of DFDs in courses, this is desirable insofar as DFD use is appropriate to the class projects and curriculum. The "insofar as" qualification means that DFDs are not necessarily a major part of the current course curricula, however their use can be beneficial in many cases. Having a graphical tool for DFD editing is likely to enhance their utility, in the same way that other kinds of software support tools enhance the utility of the technologies that the tools support.

Potential negative impacts are:

a. DFDs are elevated to an exaggerated level of importance in the specification process

b. students waste time playing with `dftool` in unproductive ways

c. a flawed tool frustrates rather than assists students

The first negative impact is essentially the flip-side of the "insofar as" qualifier above. It can be mitigated by making it clear to students how DFDs fit into the scope of a particular project. This negative impact is based to a large extent on the choice of class projects. In the specific projects that have been chosen by Fisher in recent CSC 308/309 offerings, DFDs are of some use, but not as much as say UML diagrams. However, there are many other projects where the use of DFDs would be prominent, in particular projects chosen by other faculty.

2. Functional Requirements

The `dftool` provides functionality to draw, edit, and execute dataflow diagrams (DFDs). It also provides functionality to translate a DFD to and from its textual representation in a formal modeling and specification language (FMSL).

Following an overview of the `dftool` user interface, details of tool usage are presented in the following scenarios:

- drawing and editing DFD nodes
- drawing and editing DFD edges
- defining hierarchical DFD levels
- data stores and user-supplied data
- graphical annotations to the DFD
- editing node and edge properties
- FMSL source text for nodes and edges
- validation of DFDs
- execution of DFDs
- debugging DFDs
- viewing details
- formating details
- tool options
- file-related commands
- edit commands
- help

2.1. User Interface Overview

When the user initially invokes the stand-alone version of the Dataflow Tool, the screen appears as shown in Figure 1.

Figure 2 is an expansion of the command menus. The `File` menu contains commands for manipulating data files and performing other tool-level functions. '`File New`' creates a new empty DFD drawing canvas, in its own window. '`File Open`' opens an existing DFD from a previously saved file. '`File Close`' closes the currently active canvas, offering to save if it has been modified since opening. '`File Close All`' closes all open canvases, offering to save any that have been modified.

'`File Save`' saves the currently active canvas on the file from which it was opened, or on a new file if it was created from a new display. '`File Save As`' allows the current canvas to be saved on a different file from which it was opened or most recently saved upon. '`File Save All`' saves all currently open canvases.

'`Print Setup`' allows the user to set printing parameters for particular operating environments. '`File Print`' prints the contents of one or more active display windows, per the setup parameters. '`File Exit`' exits the Dataflow Tool, offering to save any modified calendars if necessary.

The `Edit` menu contains commands for manipulating DFD data during editing. '`Edit Undo`' undoes the most recently completed undoable command. '`Edit Redo`' redoes the most recently undone command. The undo and redo commands can be repeated successively to undo/redo a sequence of commands.

'`Edit Cut`' removes and copies the currently selected datum in the current display window. '`Edit Copy`' copies the currently selected datum without removing it. '`Edit Paste`' inserts the most recently cut or copied datum at the currently selected edit point in the current display. '`Edit Delete`' removes the currently selected datum without copying it. '`Edit Select All`' selects all editable data in the current display.

'`Edit Find`' performs a search for a text string. The search is performed in all open canvases. Successive invocations of search with an unchanged search string search for further occurrences of the string until all occurrences are

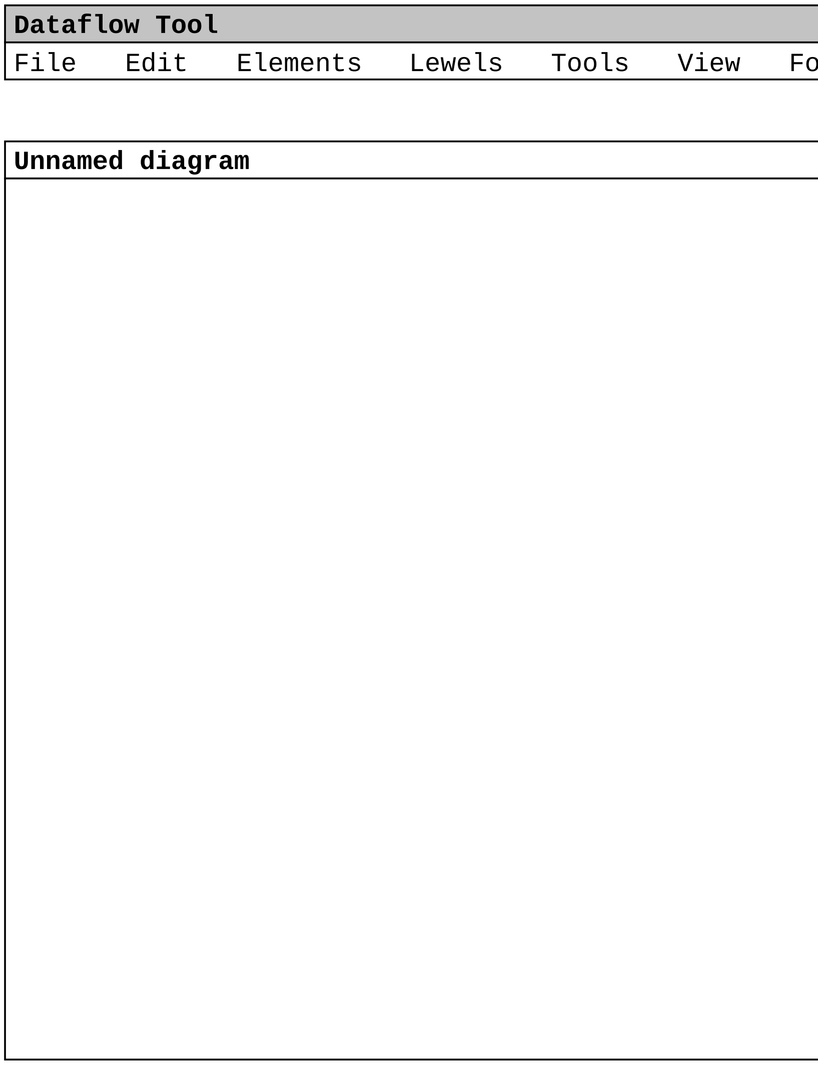

Figure 1: Initial screen.

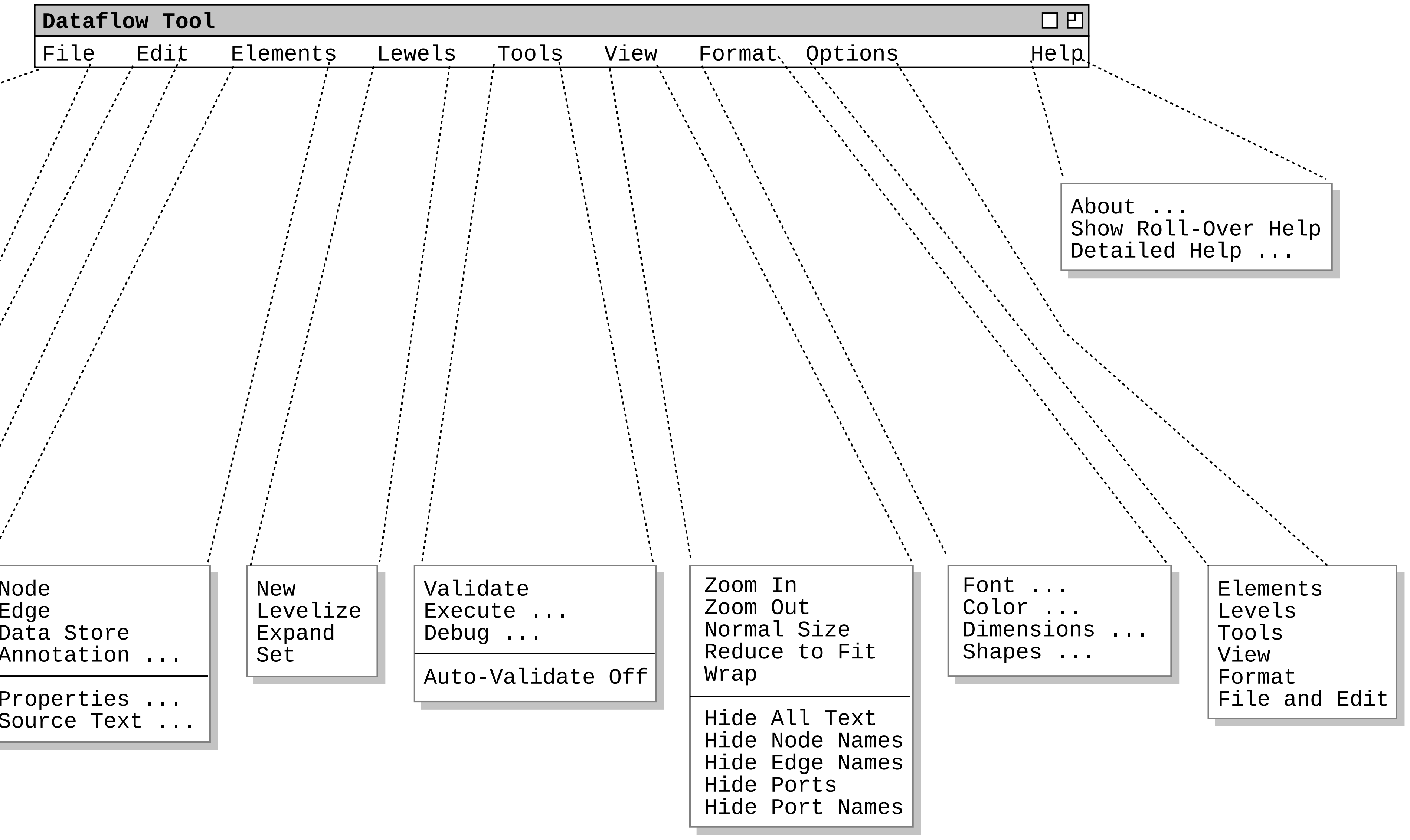
Dataflow Tool
File
Edit
Elements
Lewels
Tools
View
Format
Options
Help
About ...
Show Roll-Over Help
Detailed Help ...
Node
Edge
Data Store
Annotation ...
Properties ...
Source Text ...
New
Levelize
Expand
Set
Validate
Execute ...
Debug ...
Auto-Validate Off
Zoom In
Zoom Out
Normal Size
Reduce to Fit
Wrap
Hide All Text
Hide Node Names
Hide Edge Names
Hide Ports
Hide Port Names
Font ...
Color ...
Dimensions ...
Shapes ...
Elements
Levels
Tools
View
Format
File and Edit

Figure 2: Expanded menus.

found.

The last two commands on the edit menu allow the user to add and delete points to DFD edges. The edges themselves are added to the DFD with the '`Elements->Edge`' command.

The `Elements` menu has commands to add elements to a DFD in the current drawing canvas. The elements are a `Node`, `Edge`, `Data Store`, and free-form graphical `annotation`. The `Properties` command displays a dialog for editing properties of DFD elements. `Source Text` allows the user to edit the FMSL textual form of the DFD show in the drawing canvases.

The `Levels` menu has commands for managing different levels of a DFD. `New` creates a new empty level in a new canvas window. `Levelize` creates a new level out of a user-selected set of nodes in the current canvas. `Expand` opens the level canvas for a selected node. `Set` allows the user to define the expansion of an existing node to be another DFD canvas.

The `Tools` menu provides commands to operate on the DFD and its underlying FMSL specification. `Validate` checks the consistency and other properties of the DFD. `Execute` runs the DFD, providing visual tracing and other features. `Debug` provides commands to set breakpoints, examine runtime data, and other debugging functionality. `Auto-Validate` is a toggle that enables and disables incremental validation as the DFD is edited.

The `View` menu has commands to control the physical display of the DFD. The commands apply to the current canvas or all canvases, based on a user option setting. `Zoom In` and `Zoom Out` enlarge and decrease the size of the DFD display contents. `Normal Size` returns the display to its normal size. `Reduce to Fit` zooms the DFD to a size that fits entirely within the canvas window. `Wrap` changes the size of the display window so the DFD fits in it exactly, changing the zoom level only if the window size exceeds the available space on the computer screen.

The bottom five commands on the `View` menu are toggles that show and hide specific components of the DFD display. `Show/Hide All Text` operates on the text for all nodes, edges, and other elements of the display. `Show/Hide Node Names` operates only on the text of nodes names, leaving other text unaffected. `Show/Hide Edge Names` operates similarly on edge names only. `Show/Hide Ports` operates on the arrow-shaped icons for node ports. `Show/Hide Port Names` operates on port-name text only.

The `Format` menu has commands to control additional display details. `Font` allows the user to set the font for the text elements of the display. `Color` applies to the coloring of display elements. `Dimensions` allows the user to set the default dimensions of display elements. `Shapes` has commands to select the default shapes for nodes and edges.

`Options` commands allow the user to set various tool options. They are organized into categories that correspond to the other command menus.

2.2. Drawing and Editing Diagram Nodes

When the user selects the '`node`' command in the '`Elements`' menu, the tool displays a node-addition cursor under the current mouse position on the dataflow canvas, as shown in Figure 3. The arrow-shaped cursor has a small circle attached to its lower end, indicating that the `dftool` is in node mode. As the user moves the mouse, the cursor follows it on the canvas in the normal way.

To create a node, the user positions the cursor at the desired placement location, then presses and releases the left mouse button at that location. In response, the tool draws the node, with the upper left corner of the smallest enclosing rectangle at the pressed location, as shown in Figure 4. After placing the node, the tool displays an active text cursor in the center of the node, as shown Figure 5. "Active" means that the user may immediately begin typing the text string for the name of the node. Figure 5 shows the user having typed the node name "ActivateAutopilot", with a newline character separating the words "Activate" and "Autopilot".

As the user types, the text remains centered within the node. If the user presses the `Enter` key while typing, a new line of text begins, with the text remaining centered within the node. If as the user types, the horizontal or vertical

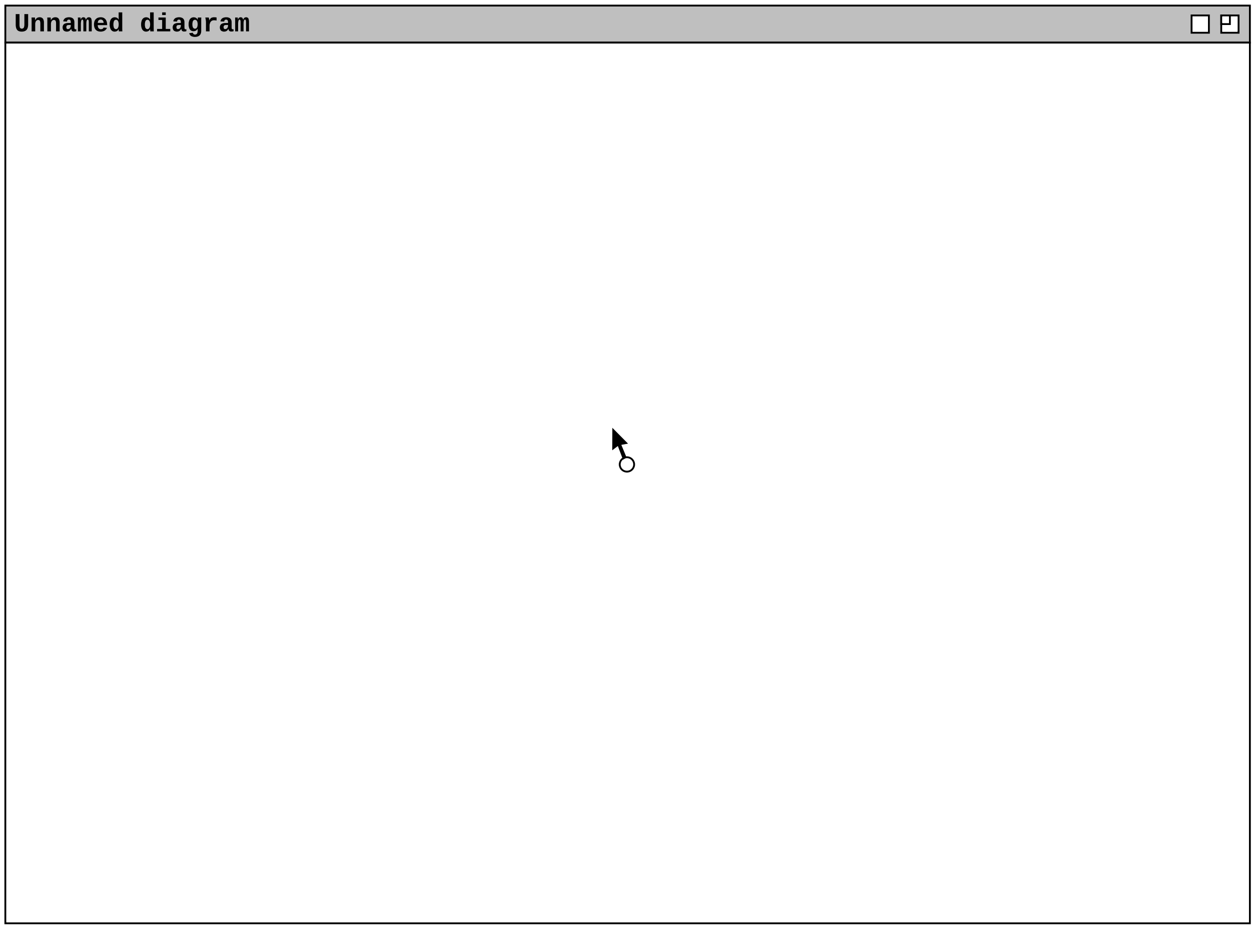

Figure 3: Adding a new node.

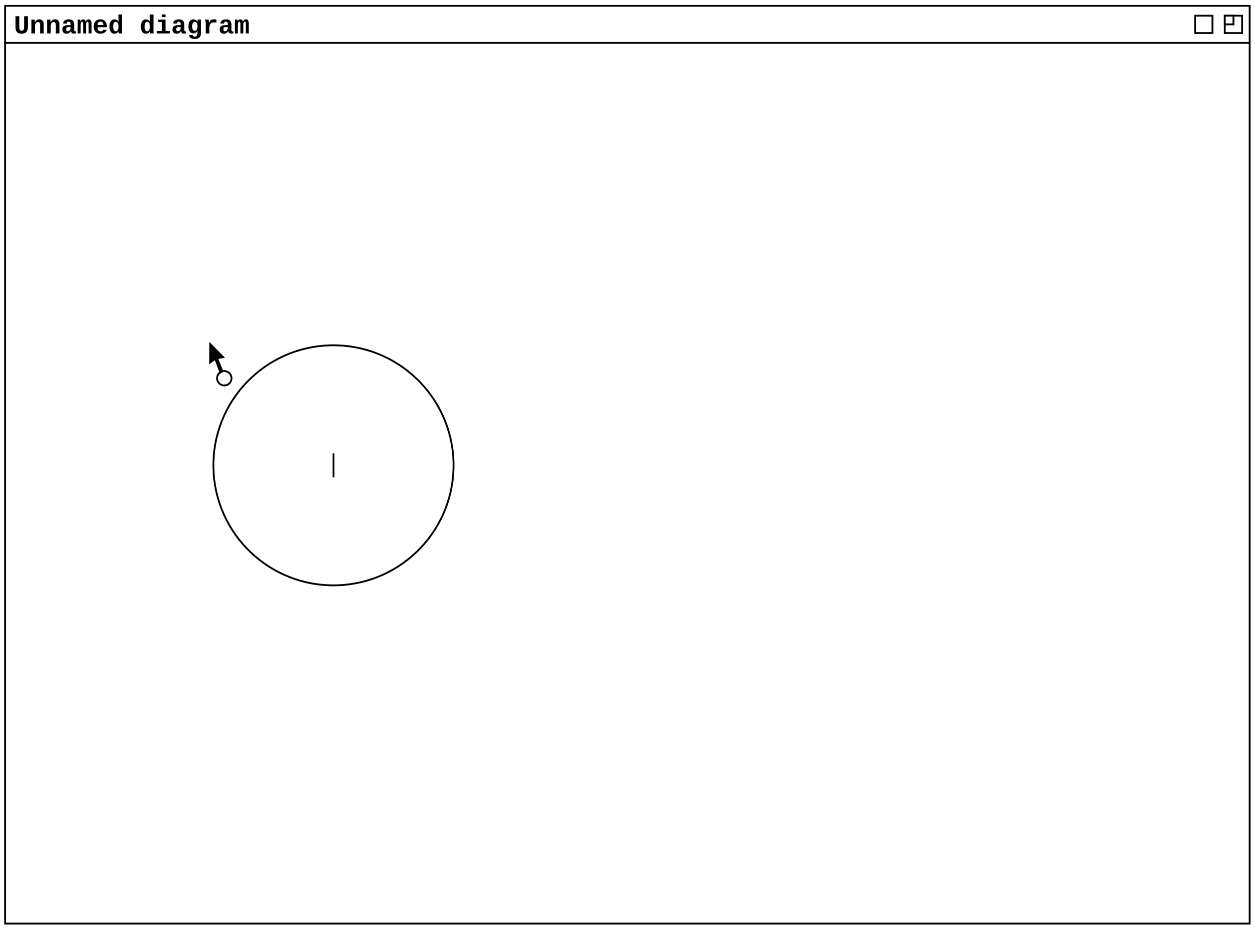

Figure 4: New node placed on the canvas.

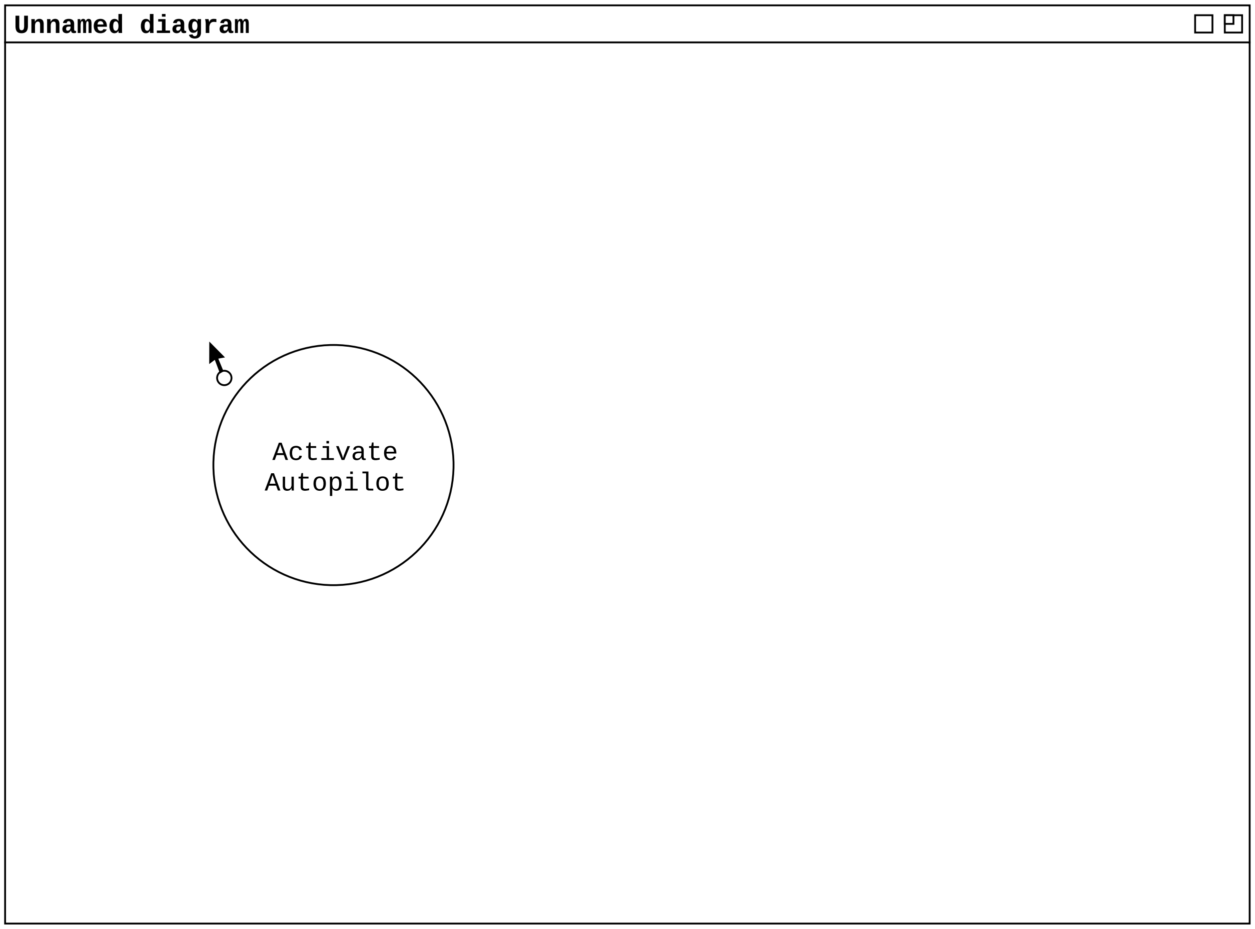

Figure 5: New node placed on the canvas.

size of the text becomes larger than the diameter of the node, the tool dynamically resizes the node so that the text does not extend beyond the node boundary. The minimum distance between any portion of the text and the node boundary is the value of the node-name margin, which by default is 1 em in the current node-name font. The the user can change the default node-name margin with an option setting, as described in ???.

If the user deletes text to make the size of the name smaller, the system dynamically contracts the size of the node, such that the distance between the text and node boundary is no larger than the current node-name margin value. The tool stops node contraction whenever the text reaches a size where it fits within a node of the node diameter setting, which by default is 10 em of the current font. The the user can change the default node diameter with an option setting, as described in ???.

The user signals the completion of node-name typing by clicking the mouse some place other than within the text of the name. If the user performs such a click without having typed any text for the node name, the tool types a default name of the form "`Node`*N*", where *N* is a positive integer ... *(see the phrasing in the cal tool rqmts).*

Node names may contain only the following characters:

- *o* lowercase and uppercase letters
- *o* digits
- *o* underscores
- *o* blank and newline characters (referred to as "whitespace")

Whenever the user types any other character in a node name, the tool ignores the character and sounds a short audible alert.

Formally, the name of a node does not include any whitespace characters. Therefore, the formal name of a node consists of the concatenation of all of its characters with all whitespace removed. So, for example, the following are the same name formally:

```
"ActivateAutopilot"

"Activate Autopilot"


"Activate
   Auto
   pilot"
```

Uppercase and lowercase letters are considered distinct within node names. So, for example, "`ActivateAutopilot`" and "`ActivateAutoPilot`" are two different names, due to the uppercase versus lowercase "p" character.

2.2.1. Sizing New Nodes by Click and Drag

When the user specifies the placement of a new node with a single mouse click, the tool places a node of the current default diameter. If the user places a node with a click and drag operation, the position and size of the node are determined by the location of the initial click plus the length and direction of the drag.

2.2.2. Using Already-Defined Nodes

When entering the text for a node name, the user may type the name of a new node that has not yet been defined, or the name of an already-defined node. *Sketch of the remainder:*

o Allow the normal kind of emacs completion thing here.

o Explain how nodes can be defined by loading an fmsl spec.

o Describe how existing nodes have ins and outs displayed as little iconic ports, shown as ingoing and outgoing arrow heads (illustrated in sketch figure below).

o Probably provide some form of "show port names" command, that displays names of ports, probably in rollover form; i.e., when "show port names" is on, moving the cursor over a port shows its (type) name; or maybe this'll happen when `Elements->Edge` is selected, and to see output ports otherwise, use `Element->Properties`. YES, I think I like the idea of port names appearing roll-over style during edge creation; see the edges section for details.

2.2.3. Port Positioning

The idea I have right now is that the default appearance of port icons will be as follows:

o the icon is a small black right-pointing arrow

o the presumed normal positioning for output icons is on the right of nodes, inputs on the left; "presumed normal" means that the relative ordering of input ports is clockwise from 0 degrees and the relative ordering of outputs is counterclockwise from 0 degrees; that said, there is no restriction on the positioning of ports around the circumference of nodes, including intermixing of input and output ports; the relative ordering rules apply separately to input and output ports, as illustrated in Figure 6

o an output icon is positioned such that the invisible line for its edge is a radius from the center of the circle (ellipse), making the base of the arrow head parallel to a chord on the edge of the circle, with the center point of the arrow head base exactly on the circumference of the circle (ellipse)

o inputs have the tip of the arrow head exactly on the circumference, and are rotated in an analogous way to output arrow heads

o a single input or output port is positioned on the horizontal diameter of the circle (ellipse)

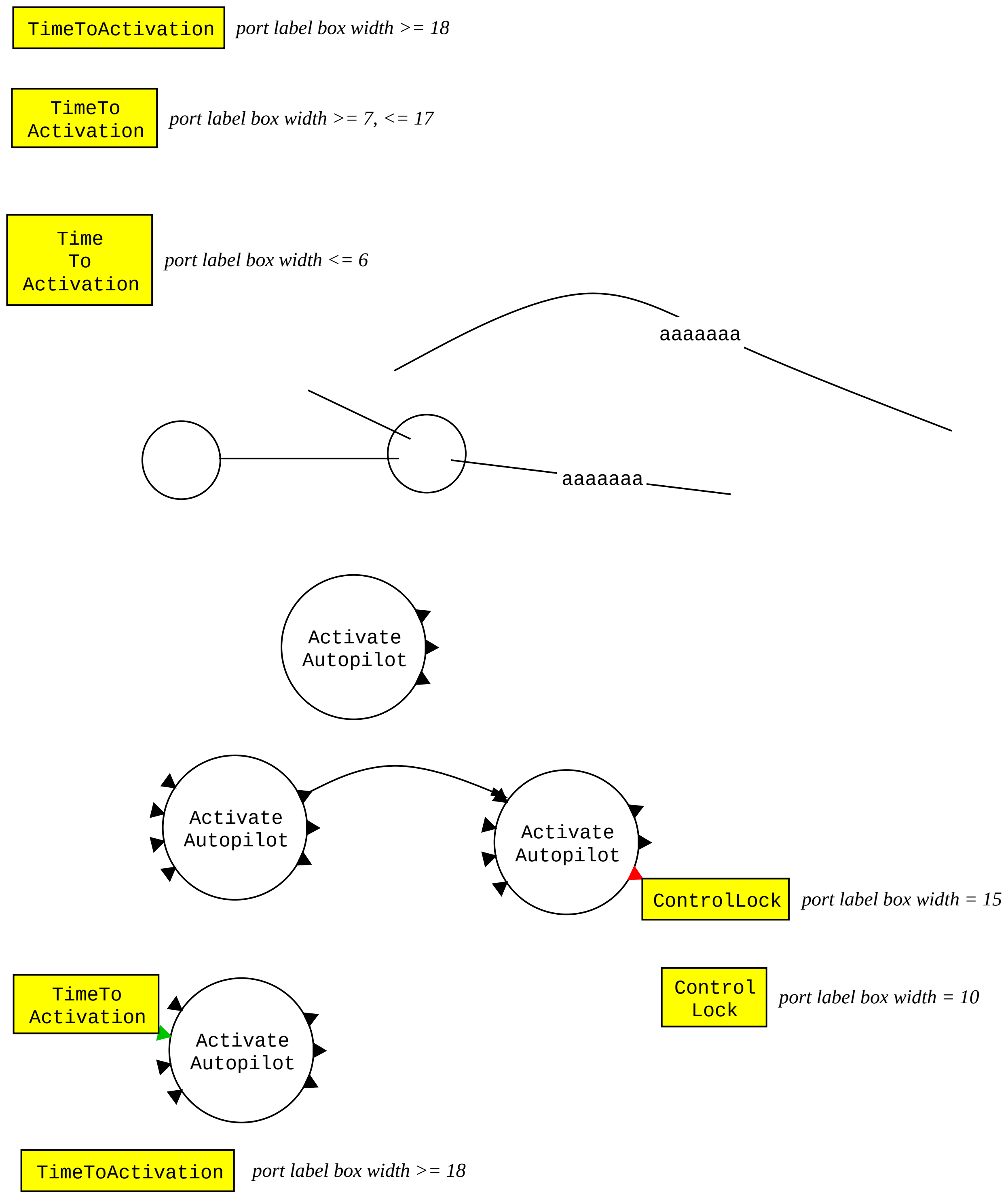

Figure 6: DRAFT -- Example to clarify port ordering (and other aspects of node display).

- *o* multiple outputs are positioned some number of degrees apart
- *o* with an even number of ports, there is nothing on the horizontal axis, just equi-distance above and below it
- *o* we need to come up with a reasonable way to deal with ops that have more ports than can be displayed for a

given size node; some kind of ellipses-labeled drop-down menu, as either the only icon, or the bottommost icon seems like it should (could) work pretty well.

o we need to make sure that there's always enough space between ports to allow new ports to be created, between any two extant ports; AND, it would seem, that the ordinal position of the click to create a new port defines a new input or output in the relative position clicked on; e.g., if the user clicks between ports 3 and 4, then a new parameter is added at position 4; two ways to ensure ample between-port spacing are:

-- define (as a settable option) the minimum distance between two ports, and auto-enlarge the diameter of a node as necessary when new ports are added;

-- some form of overflow menu; open reflection, this seems to be an unworkable solution, since it makes it hard to draw all edges clearly

o to get down and dirty with the precision, it would seem the the degrees of separation have to be based on the fixed base width of the port arrow icons and the diameter of the node

Other port-related functionality:

o When '`Options->Show Ports`' is on, the edge arrowheads are off, and vice versa. Ports can be moved around the circumference of the circle; we may want an option to allow/disallow crossing lines

2.2.4. Moving Nodes

Nodes may not overlap. When the node cursor moves over any part of an existing node, it turns grey.

2.2.5. Resizing Nodes

2.2.6. Reshaping Nodes

2.2.7. Editing the Node Label

Clicking anywhere within label causes cursor to become vertical text-edit bar. This can happen at any time, including during initial creation. Can change the name to something other than an existing node. Can change the name to that of an existing node, as long as it has the same signature of the node with the changing name. In the latter case, the completion list contains only the names of other same-signature nodes. (I think this is pretty cool, actually.)

2.2.8. Repositioning the Label

2.3. Drawing and Editing Diagram Edges

When the user selects the '`Edge`' command in the '`Elements`' menu, the tool displays an edge-addition cursor under the current mouse position on the dataflow canvas, as shown in Figure 7. The cursor has a small edge icon attached to its lower end, indicating that the `dftool` is in edge mode. As the user moves the mouse, the cursor follows it on the canvas in the normal way.

To create an edge, the user starts by moving the mouse to an edge starting position. This position must be on the boundary of a node, including on one of its output ports. Whenever the user moves the mouse to a valid edge starting position, the tool highlights the node by widening its boundary, as shown in Figure 8. To draw the edge, the user clicks and releases the mouse button at the starting position. The user then moves the mouse to the desried edge ending position, then clicks and releases again. The ending position must be on the boundary of a node, including on one of its input ports. Whenever the user moves the mouse off the boundary of a node, the tool unhighlights the node by restoring its boundary to the normal width.

As the user moves the mouse, the tool tracks it with a line that extends from the edge starting position to the current mouse position. During edge drawing, the user is simply moving the mouse, with no button depressed. Figure 9 illustrates the user having dragged the edge cursor approximately halfway between starting and ending positions on the "`Activate Autopilot`" and "`Check Autopilot`" nodes. To complete the edge, the user drags the mouse to a valid ending position, then clicks and releases. Again, the tool widens the boundary of any node that the

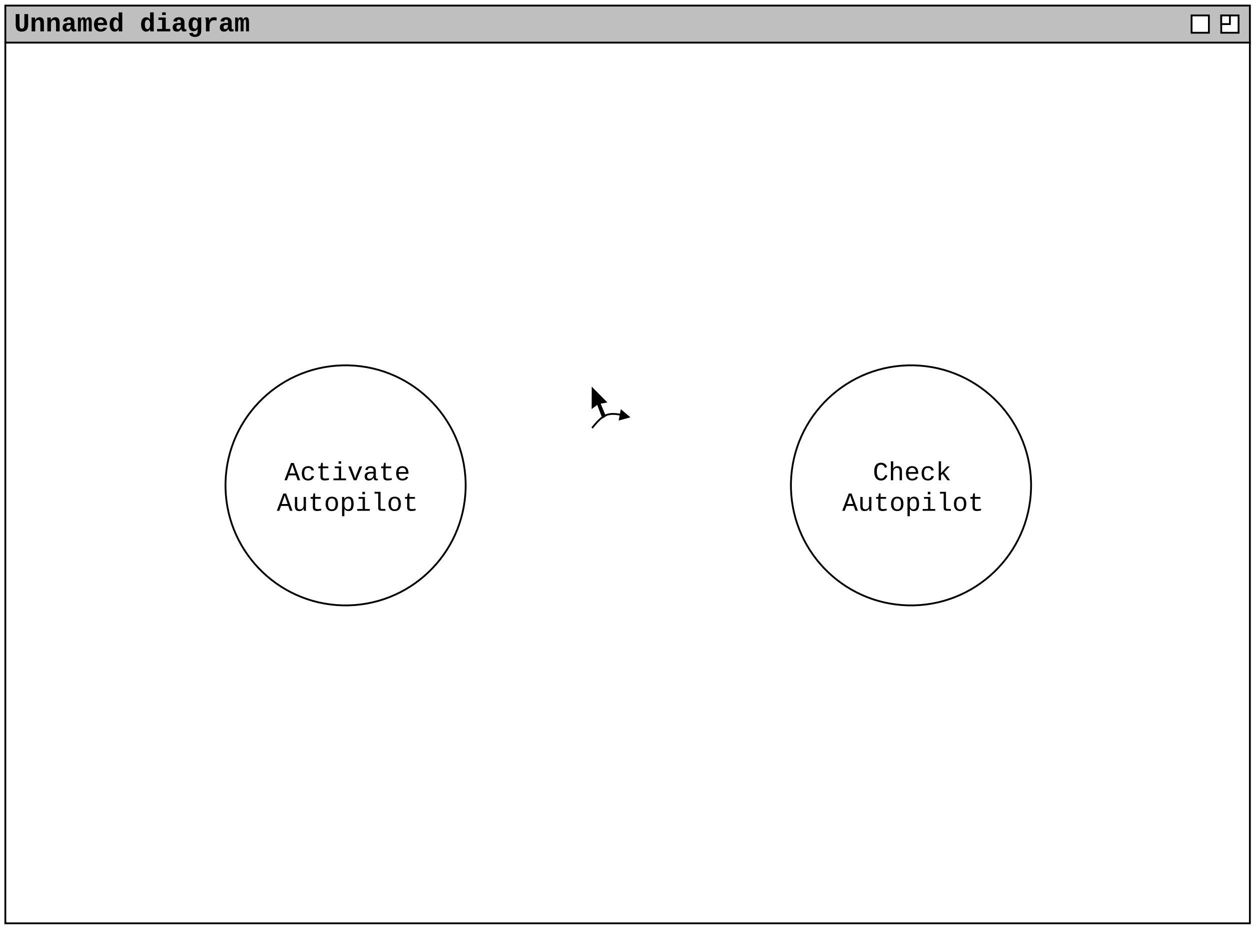

Figure 7: Adding a new edge.

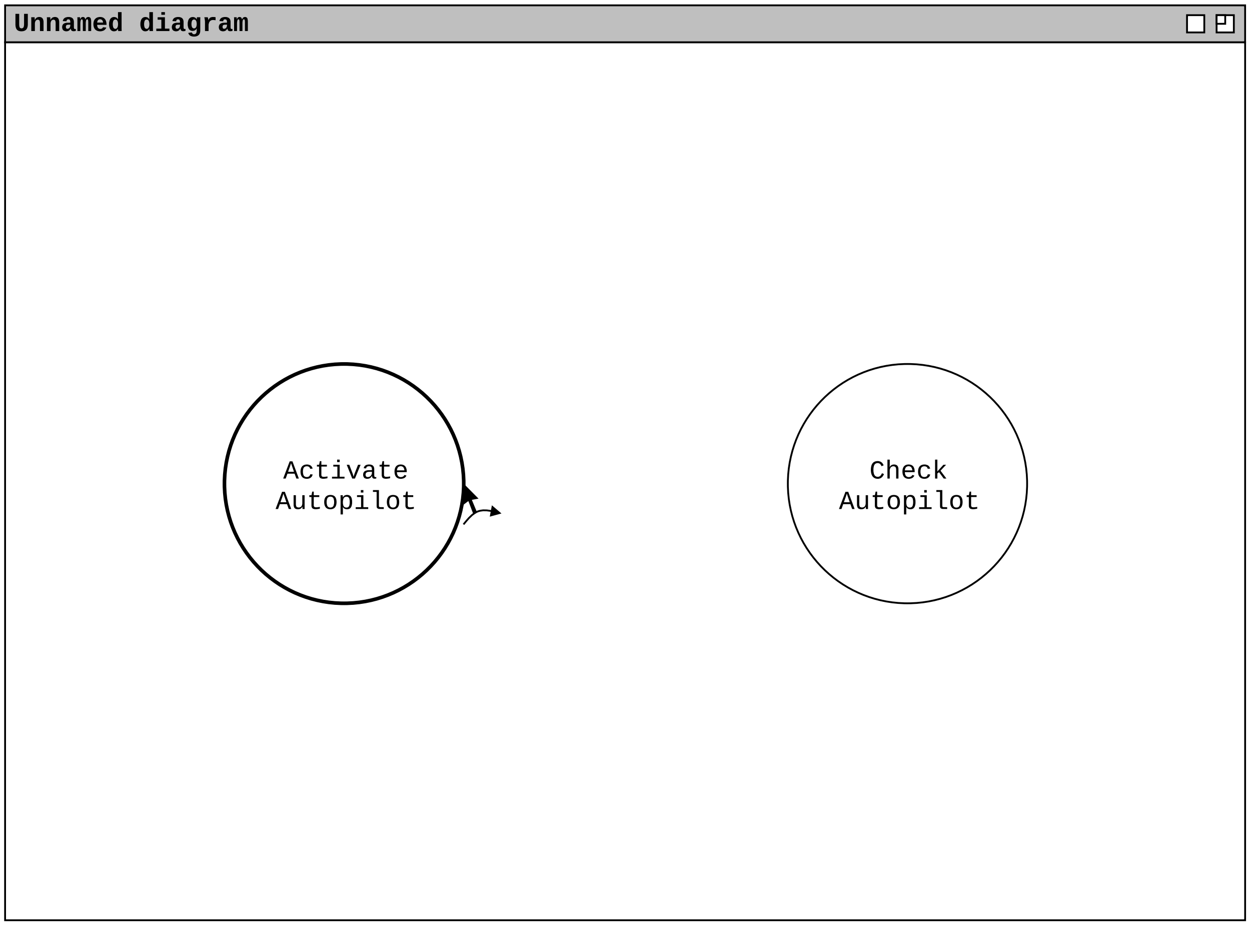

Figure 8: Edge cursor at valid starting position.

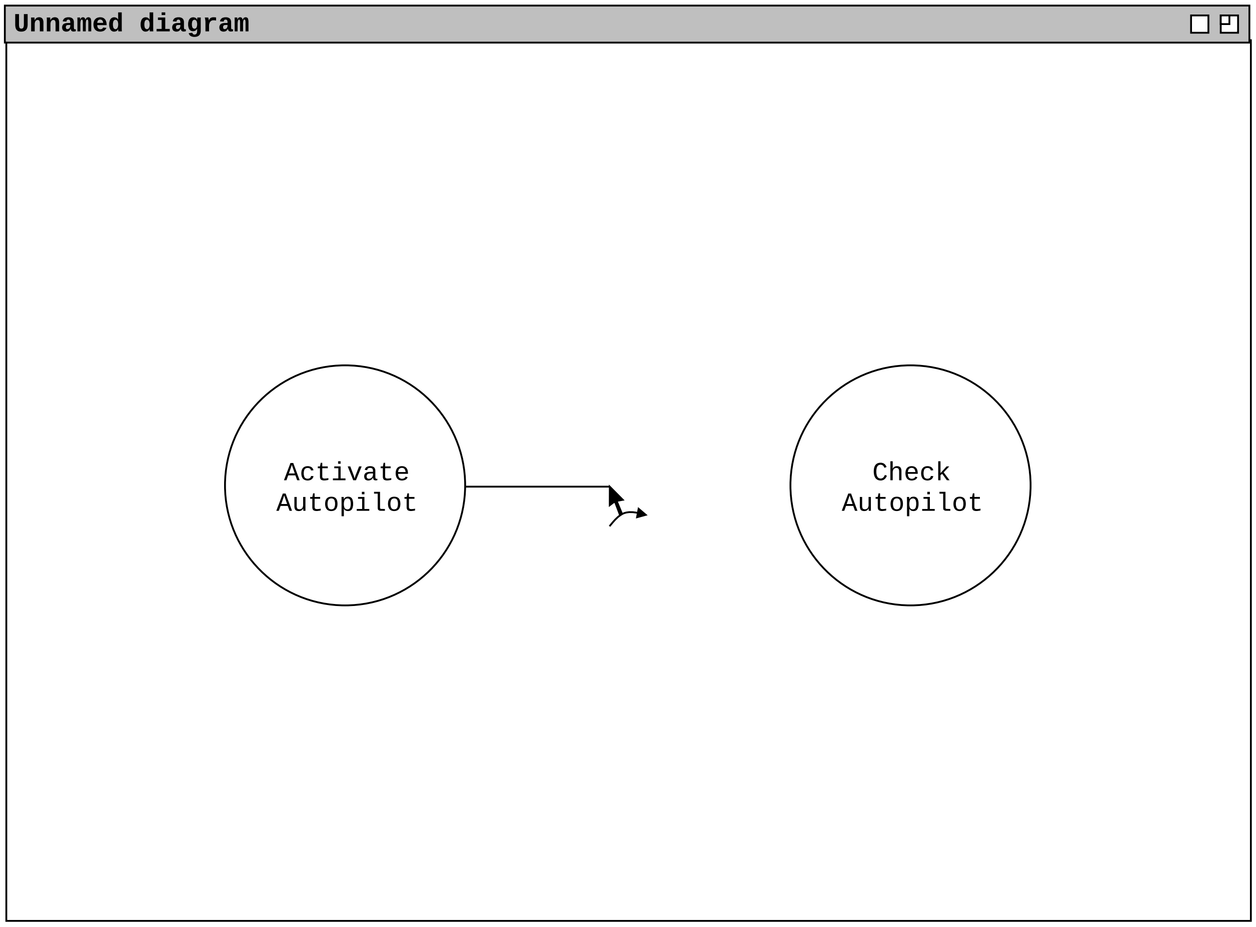

Figure 9: Edge cursor halfway between starting and ending positions.

user reaches with the mouse. In response to user's click, the tool draws the edge line, with an arrowhead on its ending point. The tool then places an active text cursor above the center of the edge, as shown in Figure 10.

Finish details of labeling; there are some notes below.

The user can create three types of edges, based on the canvas locations of the starting and ending positions. The three types are defined in the following table:

Start Position	End Position	Type of Edge
on a node	on a node	node-to-node connection
not on any node	on a node	external input
on a node	not on any node	external output

A node-to-node connection defines an input/output relationship between the two nodes. An external input defines an edge that must have its value provided by an external source, rather than coming from another node. Similarly, an external output defines an edge whose output goes to an external source, rather than to another node. Further details on external data sources are covered in ???.

User may start anywhere within or at the boundary of an node. If the edge-drawing starts within a node, then the starting point of the finished edge is where the edge line intersects the boundary of the node. End point is similar. If the intersection point crosses any part of a port icon, then the edge is considered to be connected to that port (see

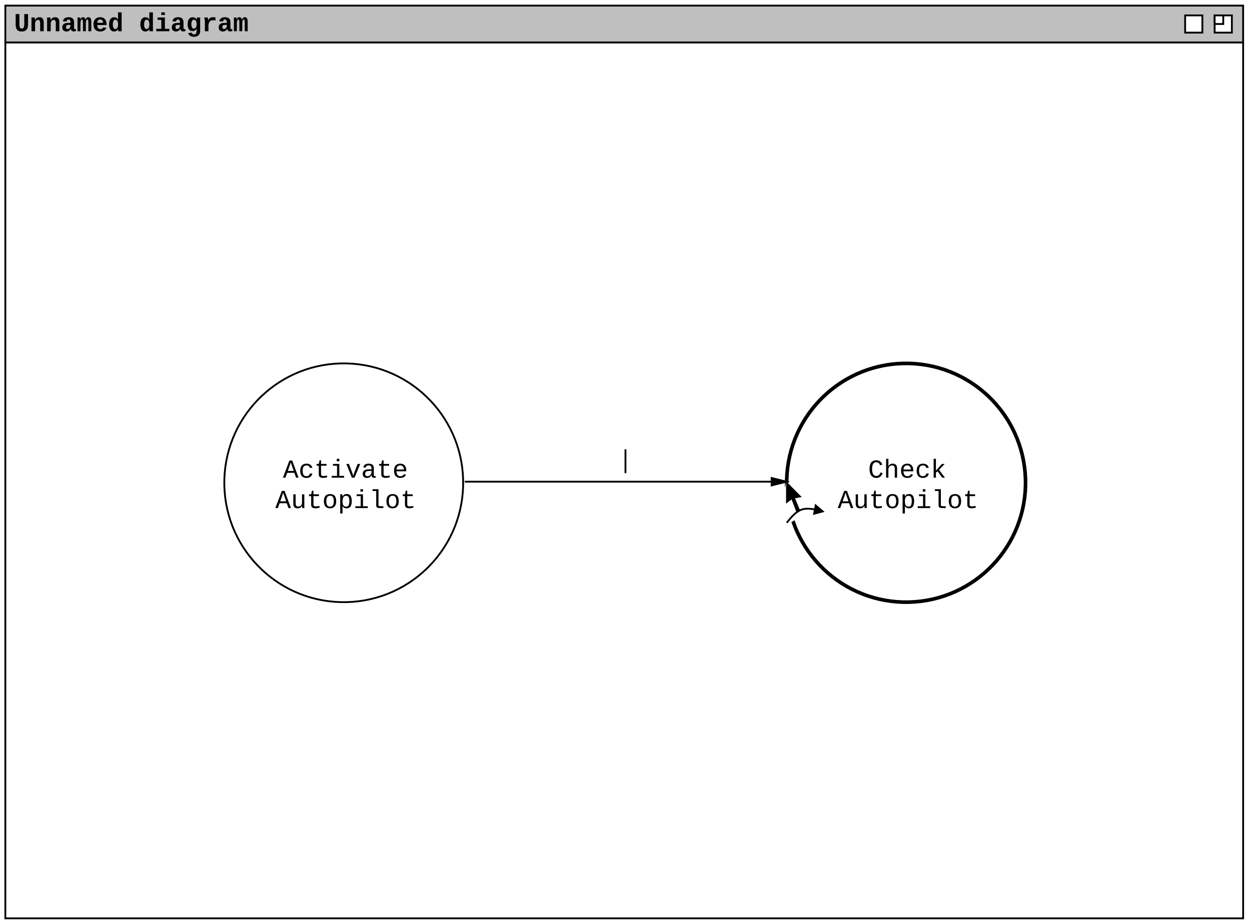

Figure 10: New edged placed on the canvas.

below for more on ports and edges).

Details of label cursor placement: Tool puts text cursor in center of edge, with center defined as the position on the edge equal to 1/2 edge length, as measured along the edge spline. The background of the text is the current bg color, which is white by default. The text color is black by default. The background size is ... . As the user types, the center of the label text remains positioned on center of the edge.

The same type of completing for edge names is provided as for node names. Viz., the names of existing objects are available. If the typed text is not an existing object, than the dftool creates an opaque object of that name and adds it to the underlying fmsl spec.

The following forms of edges are illegal:

1. both starting and ending position are not on a node
2. starting and ending positions forming an edge that crosses any part of a node

To avoid the second form, the user can use multi-point edges, as described in the next section.

It is legal to connect a node to itself, i.e., have the input and output ends of an edge be the same node. To do so graphically, the user must use a multi-point edge.

2.3.1. Edges and Ports

Sketch:

Start Point	End Point	Behavior
off port	off port	new output port created on starting node, new input port created on ending node; type of ports is object named by edge name entered by the user
on port	off port	new input port created on ending node; edge name is auto-filled with type of output port
off port	on port	new output port created on starting node; edge name is auto-filled with type of input port
on port	on port	type of ports must be the same; if so, edge name is auto-filled with that type

The following roll-over behaviors assist with port connections:

o When `Elements->Edge` has been selected but the first click has not yet happened yet, the user can roll over any port to see its type, and whether it's an input or an output. If it's an output, it's name is displayed and it's colored green when rolled over. If it's an input, it's name shows up, but it's colored red. (Note: if we have arrowheads as port icons, then we probably don't need to do the coloring thing; rather, only have the roll-over naming appear for output ports.)

o Once the first click happens, the same roll-over behavior is exhibited. If the first click is on a port, then the second click can only happen on a non-port portion of a node, or an input port of the same type as the output port.

Roll-over display details:

o Labels in the boxes are formated as follows:

-- There is a value for the width of port label boxes, settable as an option; call this *w*.

-- There is a value for the margins, settable as an option; call this *m*.

-- There is a value for the font size of the port label, settable as an option.

-- If the length of the text string for the port type name in the current font is <= *w-2m*, then the text is displayed in one line.

-- If the length of the text string for the port type name in the current font is > *w-2m*, then the text is displayed in multiple lines, center justified, with line breaks inserted at case-shift boundaries, per the standard word-processing fill technique, for example emacs' `justify-paragraph` function.

-- In all cases, the box is shrunk-wrapped around the text, to the current margin setting.

-- The preceding rules imply that if after justification the length of the text string for the port name is > *w-2m*, shrink wrapping results in a box that is wider than *w-2m*.

-- There are values for the following box formating properties, settable as options:

• label font, default black-plain-courier-12

• box background color, default yellow

• box border color, default black

o Label boxes are placed as follow:

-- One of the four corners of the box is chosen as the placement corner, per the following table:

cartesian quadrant of port connection	**box placement corner**
1	lower left
2	upper left
3	upper right
4	lower right

-- for input ports, the placement corner is placed in the center of the base of the port arrow

-- for output ports, the placement corner is placed at the tip of the port arrow

2.3.2. Multi-Point Edges

During the mouse drag operation for an edge, the user may press the right mouse button to add graphic control points to the edge. A graphic control point allows edges to be shaped as curves, to aid in the visual appearance of the diagram. Formally, edges are drawn as b-spline curves, with each control point interpreted according to the standard definition of a b-spline.

Figure 11shows the result of the user drawing an edge by left clicking on the boundary of the source node, right clicking at a middle point while drawing, then left clicking on the boundary of the destination node.

2.3.3. Reshaping Edges

Sketch:

o Select the edge spline to edit it

o Spline handles appear when it's selected, on top of label when necessary

o Label handles do not appear when spline is selected

o Use Format->Hide Label when necessary to clearly see or select a spline when the label is over most of it; an option line auto-hide-edge-label-on-selection might be useful, or auto-grey-edge-label-on-selection.

2.3.4. Editing Edge Labels

When the user selects an edge, the `dftool` draws small "handles" on and around the line or curve for that edge. The handles are control points that the user can drag to reshape the edge.

If an edge is a single straight line, then there are two control points, one on each end of the line. If an edge is a curve, then in addition to the end control points, there are one or more additional control points that defined the curvature of the edge.

The user can click and drag on any edge handle to change its position on the screen, and thereby change the shape of the edge. When the user drags an edge end point the is connected to a node, the movement of the point is constrained to the circumference of the node to which it is attached. When user drags an edge control point that is not connected to a node, it may be moved to any location on the canvas except within a node.

To add or delete control points to an edge, the user selects the '`Add Edge Point`' and `Delete Edge Point`' commands in the '`Edit`' menu. Short cuts for these commands are to hold down `shift` and move the most onto an edge or one of its handles.

To add a new control point, the user selects '`Add Edge Point`' (or holds the shift key) and moves the mouse to a place on an edge that does not have control-point handle. In response, `dftool` augments the mouse pointing cursor with a small plus sign, indicating that the user can click at that point to add a new control point. To remove an

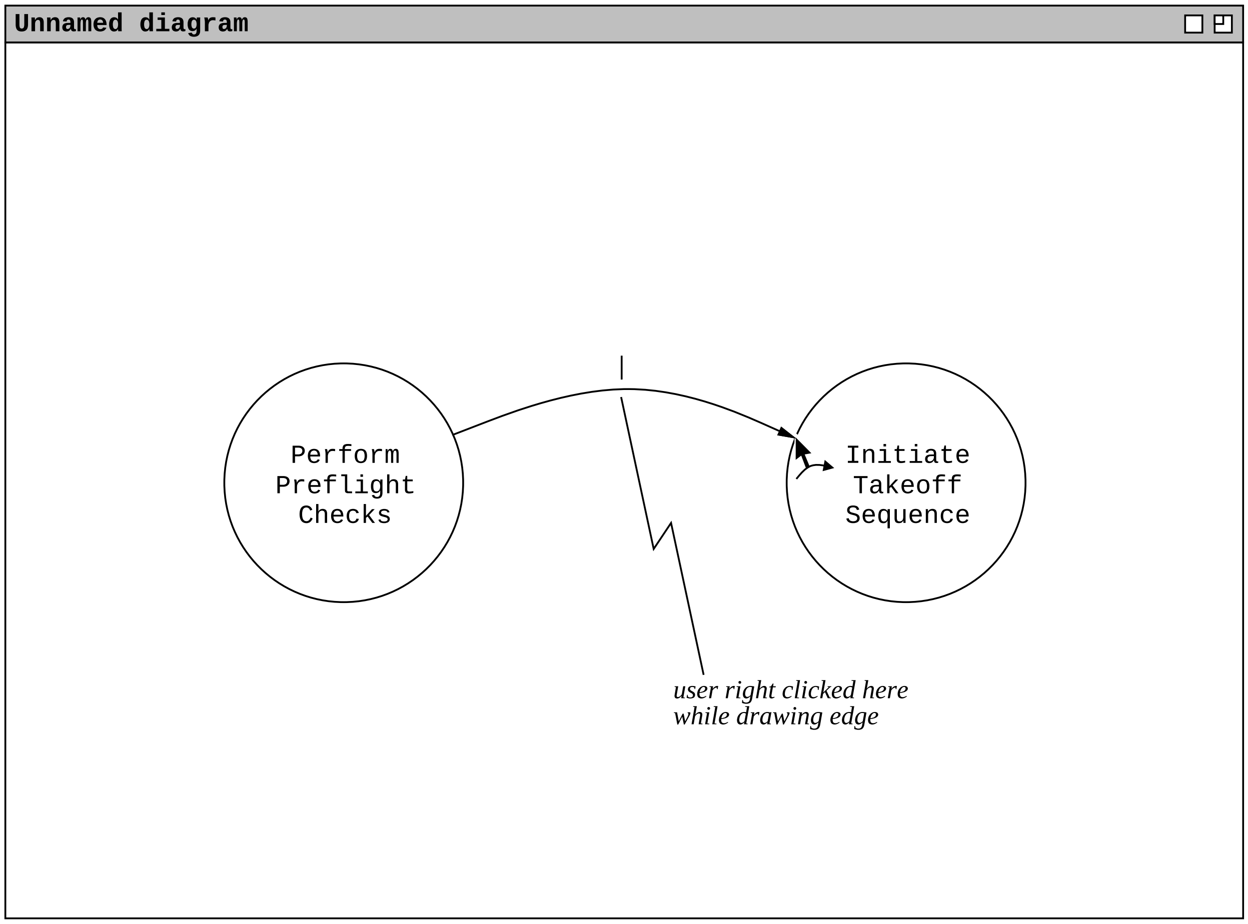
Unnamed diagram
Perform Preflight Checks
Initiate Takeoff Sequence
user right clicked here while drawing edge

Figure 11: Drawing a curved edge.

existing control point, the user selects '`Delete Edge Point`' (or holds the shift key) and moves the mouse to the control to be deleted. In response, `dftool` augments the cursor with a small minus sign, indicating that the user can click to delete that control point. The three states of the mouse cursor adding or deleting an edge point are illustrated in Figure 12.

Notes on label editing, to be more fully elaborated:

o Select edge text to edit or move it.

o When moved, it still belongs to the edge, and it (somehow) stays stuck in some controlled way when the edge spline is reshaped.

o When text is changed, it redefines the types of both connected ports; if this invalidates any other use of the node (operation in the spec), then such invalidation shows up when the spec is checked. It might be nice to have a warning come up when this is possible, but it may be difficult to do it.

2.4. Leveling a Diagram

2.5. Data Stores and User-Supplied Data

Data stores map to concrete `value` definitions in the spec.

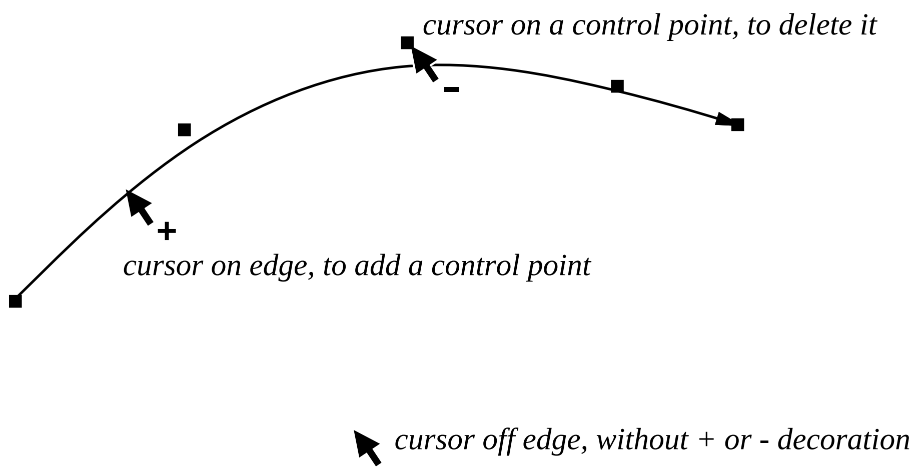
cursor on a control point, to delete it
cursor on edge, to add a control point
cursor off edge, without + or - decoration

Figure 12: Mouse cursor while adding and deleting edge control points.

All external inputs without data stores come from the human user, via a type-specific input dialog. All external outputs without data stores are displayed to the human user, via a type-specific output display. There are two built-in forms for input -- textual terminal-style and GUI. Textual inputs are supplied in the form of an FMSL expression lists, using formal FMSL syntax. GUI inputs are supplied in a tool-generated dialog of a standard form, or by custom user-defined dialogs, in some appropriate form, e.g., defined using `protoj`.

More details coming.

2.6. Diagram Annotations

Provide a very simple set of graphic and text tools to allow the user to put arbitrary annotations on the diagram, for documentation purposes. This functionality is gone if the dftool is provided as a plug-in to a graphic editor. We need to think through this issue fully, but as a stand-alone tool, I think I'm fine with providing simple general graphics functinoality.

2.7. Node and Edge Properties

To view the properties of a node or edge, the user selects the desired element on the screen and then selects the '`Properties`' item in the '`Elements`' menu. When the user selects a node, the tool displaces a node properties dialog of the form shown in Figure 13.

```
-------- Properties of Node "ActivateAutopilot" --------

        Name Text:   Activate
                    Autopilot

        Label Position: center
        Shape: circle
        Canvas Coordinates: 156,292
        Input Dialog: stdin
        Output Display: stdout
```

Figure 13: .

2.8. Source Text for Nodes and Edges

Notes:

o The text editor windows are file-oriented, in that there are separate non-modal windows for each file of the DFD.

o When the user selects '`Elements->Source Text`*', the* `dftool` *displays the file containing the FMSL definition of the operation of the current canvas.*

o There will be rules for the default creation of files if no fmsl files exist prior to the first `File->Save` *operation.*

o Fancy forms of text editing will be part of the Eclipse plug-in; in the stand-alone version, just simple text editing will be fine, with perhaps an option setting that allows the user to choose which text editor to use.

Here is the precise correspondence between fmsl source text and dfd:

DFD	**FMSL**
node	operation, which must be the component of another op unless it's the level 0 node
level	operation, with `components` and `dataflow` attribute
edge	a connection element in a `dataflow` attribute spec, plus a obj declaration of the same name as the edge name
data store	value declaration of the same name as the data store name

We need to do something in the realm of context diagrams, since in fmsl we can't have a dfd just sitting around without an outermost op to contain it. So, I think we need to do something like have each canvas always be a level, with an appropriate name, say in the canvas window banner.

To deal further with the context issue, we need to require that each df canvas corresponds to a unique op definition. This is related to the "inny" versus "outty" in the semi- and fully-formal chapters of the book. The important point is that each dfd canvas has to be defined in terms of an op with components that are effectively refs to other ops, so that the same op can appear in two or more dfd contexts. As it is now, I'm pretty confident that the outty-style notation makes this happen. So, again, what needs to happen in the dftool is that each canvas correspond to a unique op def. We can use overloading and module qualification to deal with two or more canvases having the same name. It may be appropriate, as an option or required, to put the fully qualified, and if necessary signature, in the window banner. The "if necessary" part for the signature can mean that the signature is not necessary if the op is not overloaded in the current module context. And then we have to be clear about how module context is reflected in the dfd display. Here's a crack:

- *o* we load up a spec to get all of the modules and other defs
- *o* we qualify an op name in the window banner in the same way as we do in the data dict, viz., we use the "in module X" form of disambiguation
- *o* we can also use the same form of op overload disambiguation as we do in the data dict (if there is any), or we can do what we said just above, i.e., disambiguate the op name in the dfd canvas banner with its signature if its overloaded in the module scope in which it's defined
- *o* to keep things intuitive for simple dfds, we use the same default-main module convention when we haven't loaded any fmsl into the dftool; in this way, we won't have any module disambiguation at all in canvas window banners, signature disambiguation only when necessary, and disallow two canvases with the same name and signature
- *o* to handle the latter disallow case, we need to figure out at exactly what point the disallowance occurs; I think I'd very much like to opt for it to happen at explicit validation time, rather than having some fancy incremental

checking that detects it when, say, an input or output is added or deleted to the diagram

Editing text potentially invalidates dataflow, but this is detected in the normal way in the text. I.e., errors occur when dataflow connections are type checked. What we need to come up with is the appropriate (aka, clear and helpful) way to display type checking errors related to dataflow in the dfd itself. It may be some variant of the roll-over displays for edge connection. E.g., when there's a type mismatch in any df connection, the edge(s) in the dfd that correspond to that connection are colored red.

2.9. Validation

The following are some notes on the subject that need to be scenarioized.

During the course of diagram editing, the following kinds of invalid conditions can be created. Probably most predominantly the signature of a node is changed such that uses of the operation in some place other than where it's changed go invalid. From a dfd editing perspective, the rule is that changing the signature of a node in *any* diagram changes the fmsl definition of that op's signature. So this means is that we can change an op's signature from more than one place in a multi-canvas dfd. The underlying op definition is that which corresponds to the most recently changed dfd.

Another kind of invalidation may happen when a node or edge name is changed. What we might do to prevent this (perhaps totally) is to disallow changing the name of a node to an existing node with a different signature than that shown in the dfd.

I don't know right this minute if there are other ways dfd editing can invalidate the underlying spec. I'm not (really) falling off here, it's just that it's kind of late and can't think through clearly what other kinds of spec-invalidating things can happen during dfd editing.

When we run `Tools->Validate`, we just let the chips fly where they may. Specifically, we have op defs that correspond to the most recent dfd edits. If an op is used inconsistently in more than one place (within the normal scoping rules of fmsl), then the checker detects the error in the normal way.

What we now have to come up with is the way fmsl checking errors are displayed in the dfds. Here's a crack at the invalid cases that arise, where we care from the dfd perspective:

a. An fmsl `dataflow` attribute fails to type check in some node; in this case, we color the edge(s) that correspond to the type-incorrect df connections in the fmsl

b. We should probably display in some way type checking errors that do not directly correspond to dataflow, because if we have a type-correct spec to start with, we need to let the user know when df editing changes cause problems. For starters, we might just make the display textual, with the general indication an error condition shown in the dfd display by coloring the affected node(s) or edge(s) red.

We do indeed need to deal with the case where we load a spec that is initially invalid. I think we should handle it in pretty much the same way as the docgen tool. Specifically:

a. we can't load a spec with any syntax errors; i.e., after the load discovers one or more syntax errors, it pops up a dialog says the spec cannot be loaded; if we're running stand-alone, there can be an '`Show Errors`' button in the error dialog, that when pressed displays a textual listing of the error messages; in stand-alone mode, there can also be a message that says to use an fmsl editor (IDE) to fix the errors

b. if the spec has type checking errors, we also pop up a dialog, with buttons

```
( Proceed with Load )      ( Cancel Load )      ( Show Errors ... )
```

   If the user chooses "Proceed with Load", a second warning dialog will pop up explaining that validation will continue to fail, unless (somehow) dfd editing fixes the entering errors.

c. If the spec checks fine, then everything's hunky dory for the load, and we can display a "load successful" message if we care to.

2.10. Execution

2.11. Debugging

2.12. Viewing Controls

2.13. Formating

A sketch of functionality follows.

The `Font` command applies to the current selection(s) if any. Otherwise it sets the current working font for all subsequent operations that create text. When the font of one or more selections are changed, the working font remains *UN*changed.

The `Color` command applies to selections and the current working color in the same way as the `Font` command. The following specific coloring contexts are displayed in the color-setting dialog:

node fill ...	on if node selected
node border ...	on if node selected
node label ...	on if node label selected
edge line ...	on if edge selected
edge arrow ...	on if edge selected
edge label ...	on if edge label selected
port fill ...	on if port selected
port border ...	on if port selected

The designation "on if X selected" means the color selection is enabled if one or more objects of type X is selected on the canvas. E.g., if a node is selected on the canvas, than the first three color choosers are enabled, the others are not. If items of more than one type are selected on the canvas, then a union of the color choosers is enabled. E.g., if a node an edge are both selected, than the first six color choosers are enabled. If no item is selected on the canvas, then all color choosers are enabled,

The following are shape selections:
o Node Shape -- circle, elipse, other
o Edge Shape -- curve, multiline

The last five `Format` menu items are show/hide toggles.

At one point, we had ungroup/regroup as formatting commands, with these specs:

```
Ungroup -- enabled when a grouped node or edge is selected
Regroup -- enabled when one or more ungrouped components of a previously
           ungrouped node or edge is selected
Dimensions ... -- diameter, margin
```

I think I prefer the style that's now described in the nodes and edges sections where the ungrouping is automatic. A final decision on this awaits completion of the node and edge scenarios.

2.14. Options

I have in mind options as in the Calendar Tool. Details to come.

• equalize node sizes: find largest required node size and make all nodes that size, where "largest required" is based on size of all labels, meaning there may be an extant node that is larger, but it won't be the "larest required" if it's larger than its fully shrink-wrapped size
• show/hide ports
• show/hide text -> all, node labels, edge labels
• show/hide roll-over port names

The following list looks like an color option thing, where the ‘`Format->Color` menu applies to the current canvas selection(s); node background, node border, node label, edge line, edge arrowhead, edge label. Hmm, maybe not, since we want consistent menu behavior, and the ability to set the current color, but probably should be able to do so for a particular context.

Submenu (or tab pane) details:

```
Node Shape -- circle, elipse, other
Edge Shape -- curve, multiline
Ungroup -- enabled when a grouped node or edge is selected
Regroup -- enabled when one or more ungrouped components of a previously
           ungrouped node or edge is selected
Dimensions ... -- diameter, margin
Font ... -- the normal font stuff
```

Other options:

-

2.15. File Commands

I think the convention of two files with "`.fmsl`*" and "*`.dfd`*" extentions will work well. The fmsl file has the formal dataflow defs, and the dfd file has the graphical information, which is: node shapes, node positions, etc.*

2.16. Edit Commands

2.17. Help