Fabrik
A Visual Programming Environment

Dan Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, Ken Doyle
Apple Computer Inc.
20525 Mariani Avenue
Cupertino, CA 95014

Abstract: Fabrik is a visual programming environment - a kit of computational and user-interface components that can be "wired" together to build new components and useful applications. Fabrik diagrams utilize bidirectional dataflow connections as a shorthand for multiple paths of flow. Built on object-oriented foundations, Fabrik components can compute arbitrary objects as outputs. Music and animation can be programmed in this way and the user interface can even be extended by generating graphical structures that depend on other data. An interactive type system guards against meaningless connections. As with simple dataflow, each Fabrik component can be compiled into an object with access methods corresponding to each of the possible paths of data propagation.

1 Kits and Concrete Manipulation

A kit is a set of primitive components, together with a framework for connecting the components to do new and interesting things. If objects built with the kit can in turn be used to augment the original set of components, then the range of application becomes very large, limited only by the capability of the primitive components and the manner of their interconnection. The kit approach has been around for a long time, manifest in the subroutine libraries of the last three decades. However, the ability to browse through, and experiment with the available components was extremely primitive, owing to the textual orientation of underlying computing environments during those early years.

With the advent of iconic user interfaces, nontechnical users – those not trained to appreciate invisible objects and connections – are able to work *concretely* (by pointing at an image) with data and functional components. While iconic interfaces have had little effect on conventional programming practice, they have the potential to greatly facilitate programming with kits. A box with connectors can represent a function and its parameter list. The insides of a box can be descriptive of a function or it can be an active piece of the user interface. A connecting line connotes both the passage of a value and sequential dependence. A single intuitive visual metaphor thus encompasses the acts of browsing, testing, connecting, and finally using the components in the library and those built from them. We feel that concrete manipulation can offer users untrained in programming the kind of control that has previously been available only to professional programmers.

2 Programming with a Kit

The Fabrik library includes computational elements found in most programming libraries, such as arithmetic and string manipulation, file access, and logic. These appear as boxes with connectors which can be wired together to build new functions. This concrete access to functional composition is important, but it is only part of what is needed to build a complete application.

A look at a modern application program reveals many idiomatic components which make up its user interface. Windows, panels with editable text, lists of selectable items, choice buttons, scroll bars, and menus are a few that appear fequently. As with simple computational elements, the Fabrik library provides equally immediate access to components which provide such high-level capability and which support concrete user interaction.

The remainder of this section presents an example which illustrates the support for browsing, construction, testing and packaging of a complete application in Fabrik. Many details relating to the user interface have been postponed to a later section to preserve the flow of what is an extremely simple and immediate process.

Goal: To build a simple file browser with the following capability (see figure 1f):

- User can type a name pattern in one panel,
- Second panel shows file names that match,
- A third panel shows the text of a file if the user selects its name.

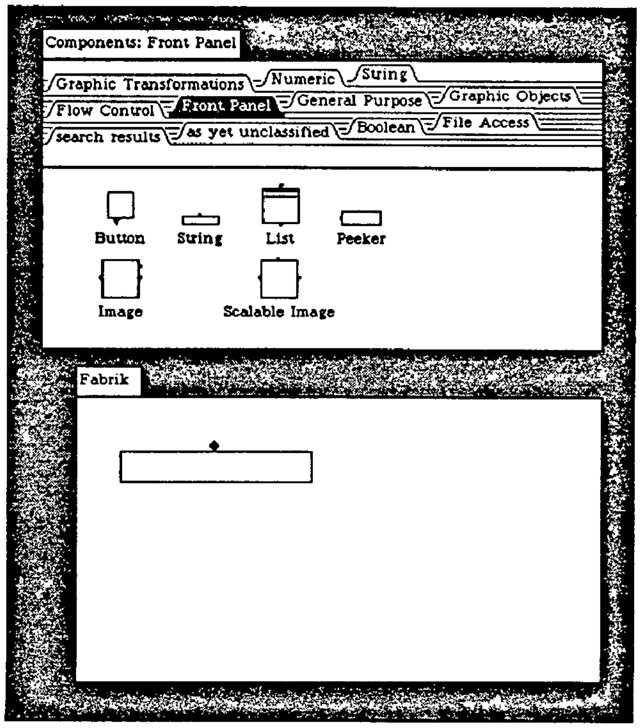


Figure 1a. A String Viewer has been dragged from the Parts Bin to a new Construction Window.

Figure 1a shows a Fabrik *Parts Bin* in the top window, and a new *Construction Window* below it. The process of building applications in Fabrik is as simple as dragging copies of components from the Parts Bin into a Construction Window, and hooking together the components by their connectors. In the first figure, a *String Viewer* has been copied from the *Front Panel* section of the Parts Bin to the Construction window to allow the user to type text.

The application in figure 1a, though trivial, is already usable. One can type and edit text within the String Viewer, with full support for font changes and justification. Upon typing <enter>, the current text appears as output at the pin on the top but nothing else happens because the output pin is not connected to anything. If another text component were wired to the one shown, then the text would appear in that one as well.

In figure 1b, the author has typed the word 'memo' in the String Viewer, in order to prepare the

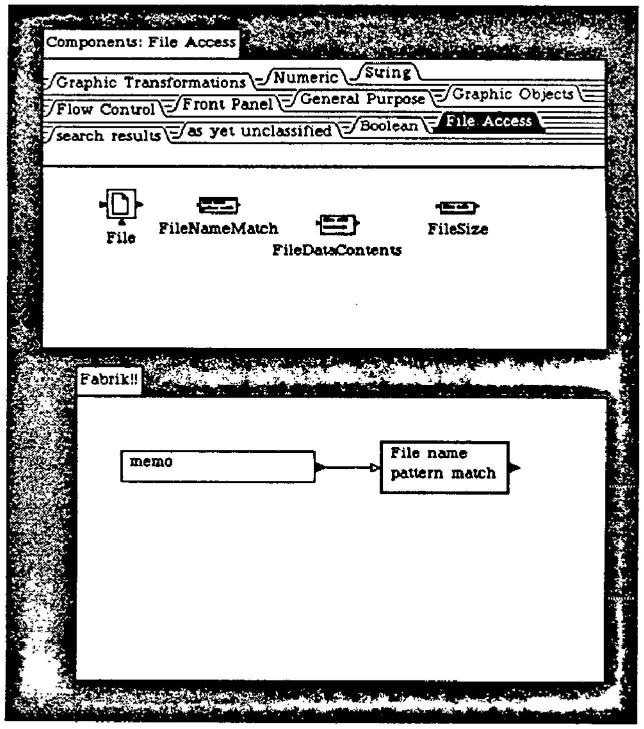


Figure 1b. The String Viewer has been hooked to a File Name Matcher. The resulting list of file names is waiting at the output pin.

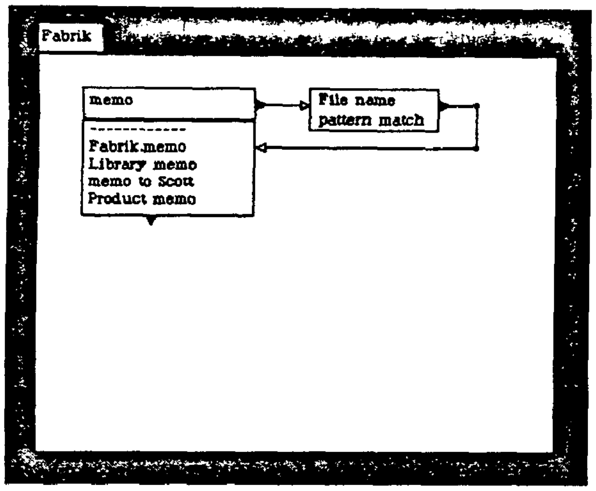


Figure 1c. A List Viewer has been installed and attached to view the list of file names.

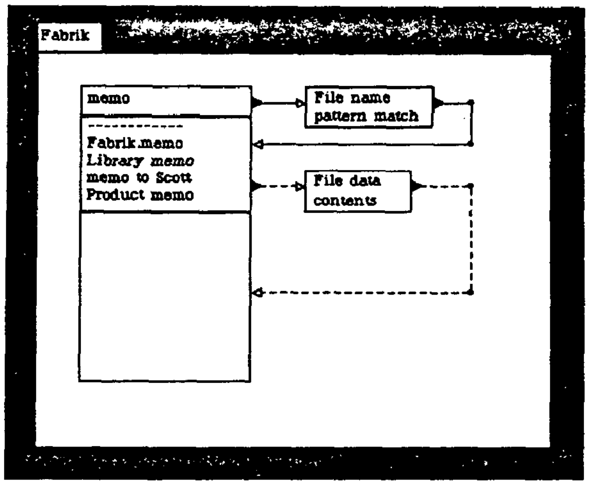


Figure 1d. The selected file name output is now hooked up via a File Contents extractor to a String Viewer.

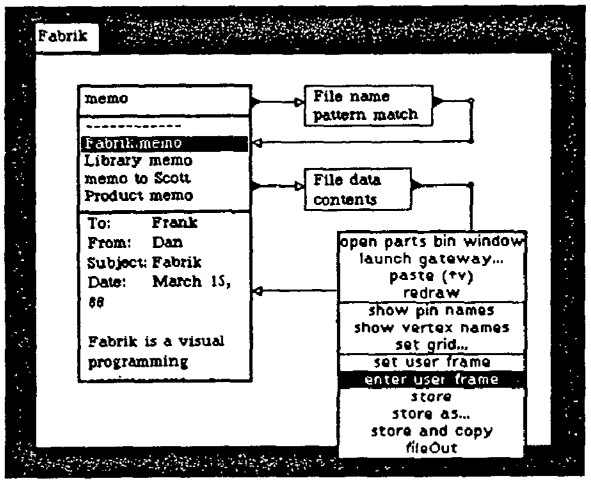
example to search for files with those letters in their names. He then selected the *File Access* category of the parts bin, dragged a *File Name Pattern Matcher* to the Construction Window, and hooked that component to the String Viewer. He also slid the top pin of the String Viewer to one side for neatness.

In figure 1c, the author has obtained a *List Viewer* from the Front Panel category of the Parts Bin to view the list of file names matching the string, 'memo'. As soon as the File Name Matcher output is connected to the List Viewer input, the list of names appears as shown.

In figure 1d, the desired functionality of our sample application is completed by installing a *File Data Contents* extractor and a second String Viewer. These are hooked up in the obvious manner to view the text when a file name is selected. The connection appears dashed at this point in the construction because, with no file name selected, the value is *nil.* Fabrik takes care in this situation to track invalidity so that no component is triggered with invalid data.

Figure 1e shows that after a file name has been selected, it propagates through the File Data Contents module, turning the connecting lines solid, and finally displaying the text of the selected file as desired. In roughly five simple steps, the desired application has been programmed, and is ready for use. The only problem is that it is still surrounded by a small scaffold of computational components and their connections.

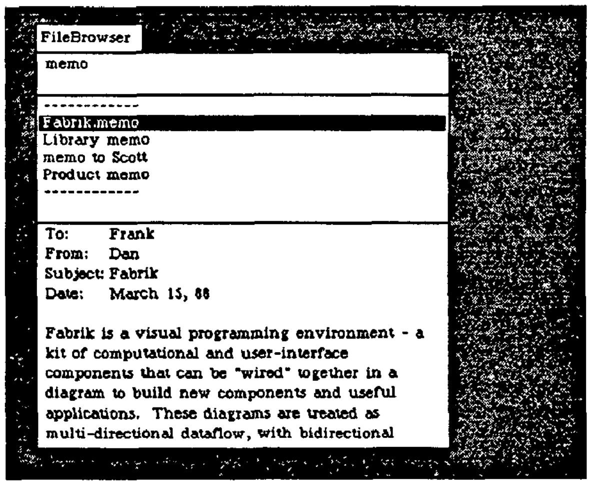
Fabrik allows a subregion of each diagram to be designated as the *user frame*. This has been done with the left three panels in figure 1e, as can be seen form the heavy border around their periphery. Once the user frame has been designated, a menu command is available to *enter* that frame as shown in the figure. This command instructs Fabrik to restrict its view to only the designated



To: Frank
From: Dan
Subject: Fabrik
Date: March 15, 88

Fabrik is a visual programming

Figure 1e. The left three panels have been selected as the "user frame," and a menu command lets one "enter" that frame.



memo

Fabrik.memo
Library memo
memo to Scott
Product memo

To: Frank
From: Dan
Subject: Fabrik
Date: March 15, 88

Fabrik is a visual programming environment - a kit of computational and user-interface components that can be "wired" together in a diagram to build new components and useful applications. These diagrams are treated as multi-directional dataflow, with bidirectional

Figure 1f. The construction scaffold is now hidden and the application can be launched as a reframable window ready for normal use.

components, and to make the result available as a normal window in the environment, capable of reframing, multiple instances, and all the other comforts of the supporting environment.

Figure 1f shows our file browser application after entering the user frame. An application such as this can easily be assembled in less than five minutes. Moreover all of the original scaffolding can later be retrieved for documentation or as the basis for a revision. This ease of "opening the hood" adds to the potential reusability of Fabrik software.

3 About the User Interface

As shown in the preceding example, an author defines his application by directly manipulating its visual representation. He selects appropriate components from the library, places them in a Fabrik window, and connects them up to achieve the desired functionality and appearance. A Fabrik component appears as a rectangle, usually with one or more connectors, called "pins," on its periphery. Some components are purely computational. Others provide user interface functions within their rectangles. Some pins are used to gather input for the component, others to channel output, and others (bidirectional pins) are able to pass data in either direction. The pins can be moved about the periphery to simplify wiring.

The Fabrik Parts Bin is organized as a file drawer with index tabs as shown in figures 1a and 1b. Components in the library appear here in miniature to save space. The file drawer idiom used for the Parts Bin was chosen for its rapid access to many categories without the complexity of generally nested windows. To use a component, the author uses the mouse to "grab" it from the Parts Bin, to drag it over the desired layout window, and to place it at the desired location by releasing the mouse button. The "part" being laid out grows to actual size as it leaves the Parts Bin, to facilitate layout in the destination window.

Components are connected using a concrete "wiring" interface, which serves to connect pins of different components. To wire from a to b, the author clicks on *a*, and then clicks again on *b*. It is not necessary to wire directly from one pin to another. Each click down establishes a "vertex" in the wiring diagram, a point where the wire can be bent or additional wires attached, making it easier to produce readable diagrams. Numerous features in the user interface allow this wiring to be changed for either aesthetic or logical reasons.

As the Fabrik author lays out his components, the tableau he is creating is always 'alive'. The appearance and behavior of the application being built are always directly manifest. This is in contrast to the conventional cycle of editing source code, compiling, fixing syntax errors, recompiling, linking, loading, then test-running an application.

If the author is building a new component, as opposed to a stand-alone application, he will need to add pins to the window border that serve as gateways for data to flow into and out of the component. Internal dataflow semantics of the component are expressed by wiring between gate-

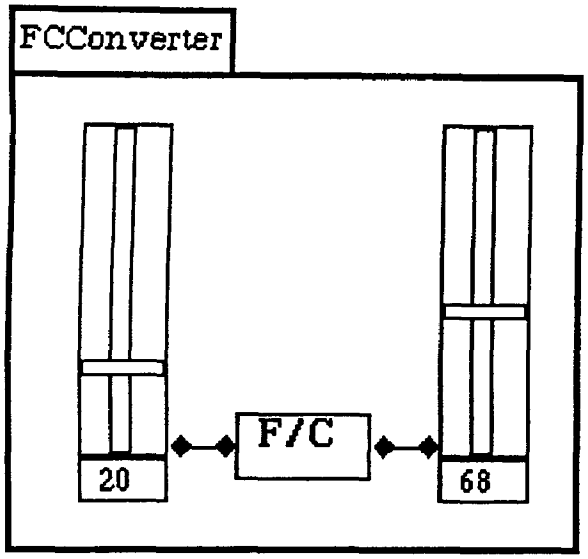


Figure 2a. Bidirectional diagram using two Slider controls to achieve a Fahrenheit-to-Centigrade converter.

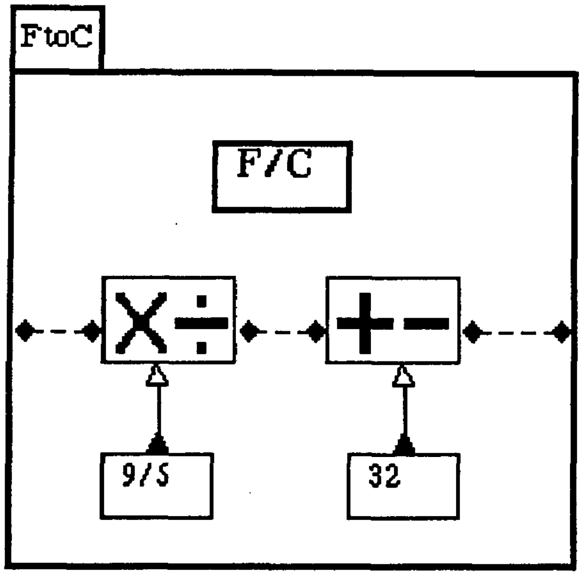


Figure 2b. Internal diagram for the F/C component used in the diagram at left.

ways and the pins of the sub-components. The gateways later appear as the pins on the periphery of the component when it is used to build other components and applications.

The direct manipulation of components to assert relationships virtually eliminates syntax errors from the development process. The only error possible in building a Fabrik application is to connect two pins that shouldn't be connected. Fabrik checks for incompatible modality (output to output or input to input), as well as for type mismatch (see section 8) before allowing a wire to be connected. Permission to connect is communicated to the user through apparent attraction and repulsion during the wiring process.

The attentive reader will notice that the bidirectional pin on the String Viewer in figure 1a was coerced to output-only as a result of being connected to the input of the file component in figure 1b. Such directional coercions are handled by Fabrik automatically.

Fabrik's user interface, browsing, programming and debugging support are discussed in full detail in a companion paper [Ludolph].

4 Bidirectionality

In Fabrik we have chosen dataflow as the underlying model of computation. It is presented to the user in a loop-free and therefore timeless and declarative model. Dataflow is often considered to be incompatible with bidirectionality because bidirectional diagrams appear to have loops in them. However, with some care, most of the benefits of bidirectionality can be achieved in a system based on dataflow.

The key observation about most uses of bidirectionality is that they are simply a shorthand for multiple paths of dataflow. Except in complex situations with subtle constraints, the different paths may be treated completely independently.

Figure 2 above shows Fabrik diagrams for a bidirectional temperature converter, an example borrowed from Thinglab[Borning]. Both the outer application and the numerical conversion resolve simply into a left-to-right flow for input on the left, and *vice-versa* for input on the right.

Bidirectionality enhances the intuitive aspect of Fabrik's concrete constructions by reducing the amount of wiring as well as by reducing the number of components needed in the library. We consider support for bidirectional behavior to be crucial in any system used to build user interfaces, since most visual metaphors used for output have a natural interpretation for input as well.

It is, of course, possible to construct confusing or ambiguous diagrams with bidirectional components. For instance, if the Times/Divide component in figure 2b were bidirectional at all three pins, there would be abiguity as to whether a change in one temperature caused a change in the other temperature or a change in the conversion constant! We have chosen to leave Fabrik users exposed to such possible confusion since the benefits are so compelling. Possible solutions range from simple restrictions, such as allowing no more than two bidirectional pins per component, to employing more powerful techniques as in Thinglab.

5 Synthetic graphics

We have seen in the foregoing examples how existing components can be combined to make new applications and how, through the use of external connectors, the new applications can act as components themselves, thus augmenting the Fabrik library and increasing the power of the system as a whole. It is obvious that, given a reasonable basic set of arithmetic and string-handling components, most simple programming functions can be built up on Fabrik. What is less apparent at first is that, by including a basic set of components capable of producing images, Fabrik assumes the ability to synthesize any computable image. In this way, Fabrik applications are as extensible in their user interfaces as they are in their numerical and textual manipulations, thus enabling simple construction of applications and components such as a bar-chart, a scroll-bar, or an animation sequence.

Fabrik's graphical capabilities center around graphemes, objects that represent images. Graphemes can be simple, such as lines, circles or bitmaps, or they may be transformed or combined with other graphemes through overlays, clipping, rotation and so on. The ability to carry a variety of different graphical objects on a wire relies critically on the polymorphism of the underlying object-oriented programming language (Smalltalk) in which Fabrik is implemented.

Some Fabrik components take in non-graphical data (such as magnitudes, points, vectors, style specifications) and generate graphemes as output. Thus, a rectangle creator component takes in two points for its opposite corners and an optional style (border color, border width, interior color), and produces a grapheme representing the image of a rectangle satisfying the input parameters. Various other such components create lines, ovals, bitmaps, display text, etc.

A second group of graphical components provide graphical transformations of general applicability. These include components to scale, translate, rotate, hide, and invert a grapheme, and to merge multiple graphemes in various ways.

Graphemes are viewed with graphic viewers. Each viewer defines its own local coordinate space. The coordinates carried by the grapheme define its location in that local space. Whenever the input changes, the grapheme is redisplayed. Different graphic viewers offer such features as automatically scaling their contents, emitting the final bitmap resulting from the incoming (complex) grapheme, allowing their contents to "pop up" on top of the current screen layer, and so on.

Finally, interaction is supported in this world of synthetic graphics by sensor components that sensitize a grapheme or collection of graphemes to user input, such as mouse clicks and location within the bounding box of the grapheme. The Mouse component has one output pin (on the right

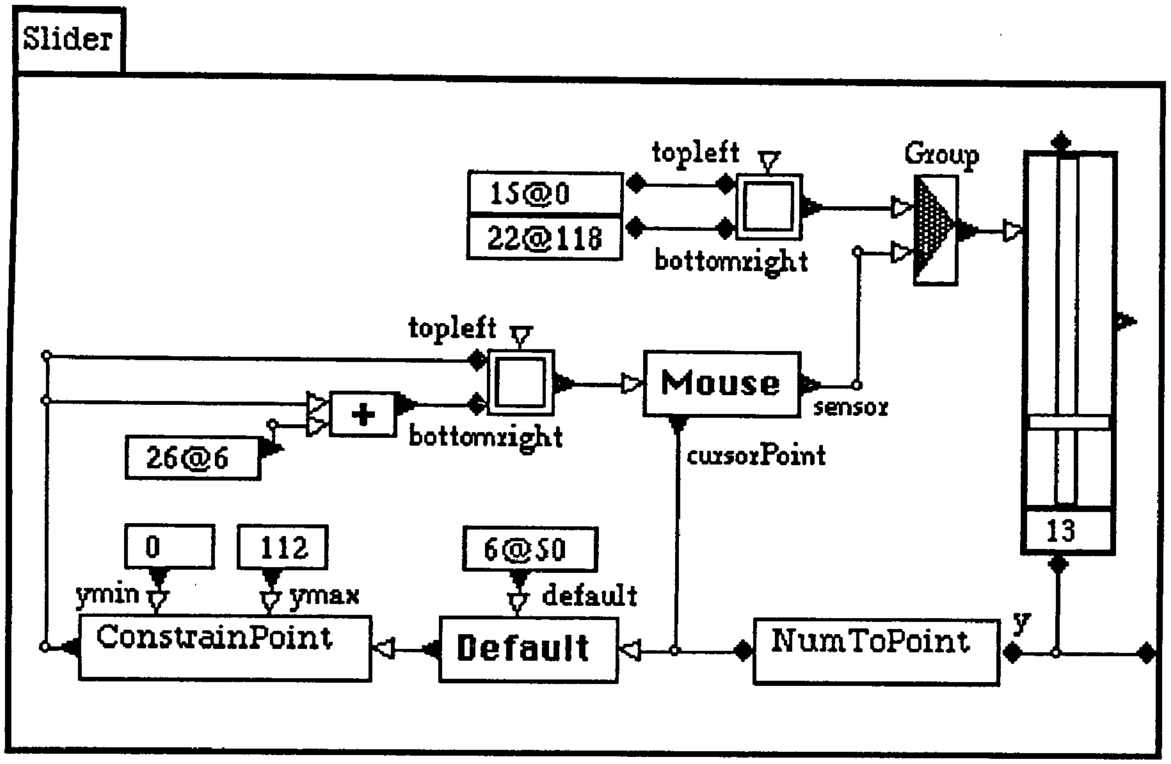


Figure 3. A Fabrik diagram computes the image for the slider in figure 2. The Mouse component sensitizes the slider image to support input as well as output.

in figures 3 and 4) which emits a *sensitized* version of the input grapheme whenever it changes. The output on the bottom is activated when a grapheme viewer detects user input, at which time it emits the cursor location. Various sensors support tracking of different button states, keyboard input, and so on.

Figure 3 shows the complete Fabrik diagram for the slider component used in the Fahrenheit-to-Centigrade example of figure 2a. Several labels have been made visible, using an option available in the Fabrik interface. To the right, the slider is visible in a graphic viewer that automatically scales the incoming grapheme to fit. The slider is composed of two rectangular graphemes, a tall slender one generated by the rectangle creator grapheme with input points 15@0 and 22@118 (its top-left and lower-right corners), and a horizontal, mouse-sensitized rectangular grapheme with the size 26@6 whose location is determined by the program. Below the slider is a small number component that displays the current value, 13.

When the user positions the cursor over the horizontal rectangle in the graphic viewer and presses the mouse button, the mouse component emits the viewer-relative location of the cursor out the bottom pin. The new cursor location causes two components, NumToPoint and Default, to fire. Default passes non-nil inputs through to its output unchanged. If the input is nil here, the value, 6@50 is output. ConstrainPoint is a user-built component that limits the Y value of the point to the range 0-112 and replaces the X value with 6. The resulting point is fed unchanged to the origin pin (upper-left corner) of the rectange creator and, after adding 26@6, to the corner point (lower-right corner). The apparent loop through the Mouse component is not really a violation of dataflow, since the cursor output is not triggered by incoming data, but only by user input.

The new rectangle grapheme created as a result is sensitized to the mouse, merged with the vertical rectangle, scaled and displayed. Sensitization to the mouse means that clicking the mouse button in

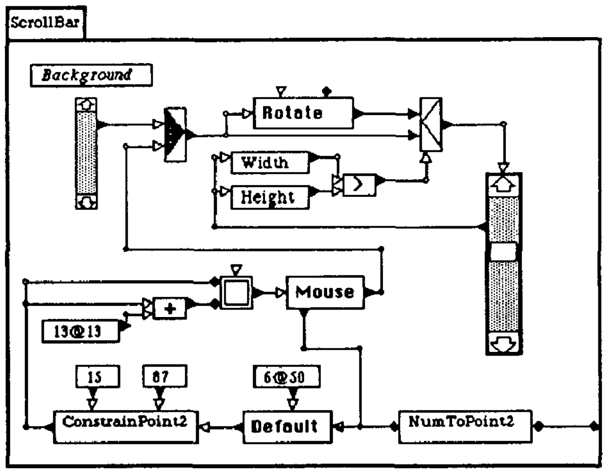


Figure 4a. A simple scrollbar diagram. Logic is provided for rotating when the image is wider than high.

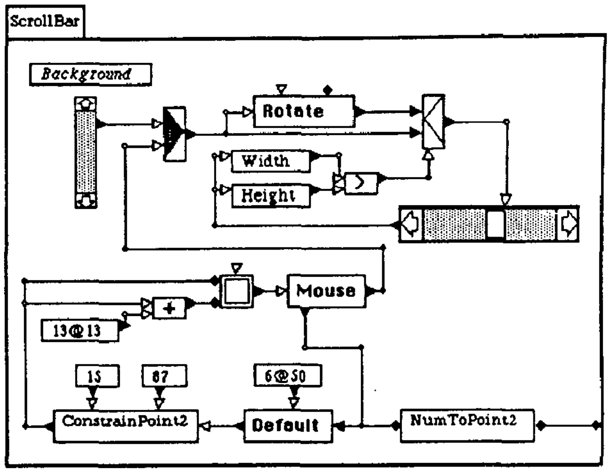


Figure 4b. Demonstrating the rotation logic. All display and mouse-tracking operations are properly scaled and rotated.

the area of the slide will cause the mouse coordinate to be emitted from the bottom pin of the Mouse component. NumToPoint is a user-built component that converts the Y value of the input point to an integer which is both displayed in the small number component below the graphic viewer and passed through the gateway on the right wall of the window to anything wired to it. As the user moves the mouse with the button still pressed, the cycle is repeated.

Bidirectionality permits the slider to be used as an output indicator as well as an input control. If the user types into the small number component or a value flows in through the external gateway, it is displayed in the number component, flows through NumToPoint which changes the integer value into a point, through Default (no change), through ConstrainPoint, etc., and causes the small horizontal rectangle to be repositioned and redisplayed at the appropriate location in the graphic viewer.

The use of a general algebra of images in Fabrik's synthetic graphics adds considerable leverage to each application. For instance, the scrollbar example in figure 4 produces either a vertical or a horizontal image, depending upon the aspect ratio of its framed image.

In this case, the background image, another Fabrik diagram, is scaled and rotated as well as the slider. Mouse-tracking coordinates are automatically transformed through this logic, leaving the scrollbar diagram relatively uncluttered.

From these examples of synthetic graphics it is apparent how Fabrik deals with other dynamic media. A musical score or an animation sequence can equally well be synthesized as a set of primitive elements combined and transformed by other higher-level functions.

6 The Draw component

As shown above, images can be generated by connecting various grapheme creators and viewers. A step toward more direct manipulation of graphical material is the *Draw* component. Currently a primitive component, it allows the user to draw graphic objects as in a normal drawing program, while the corresponding Fabrik diagram is automatically laid down and connected. The user can further adjust the location and size of the graphic objects right in the Draw component. These changes are immediately shown as updated point values that are inputs to the grapheme creators. The Fabrik diagram generated is exactly the

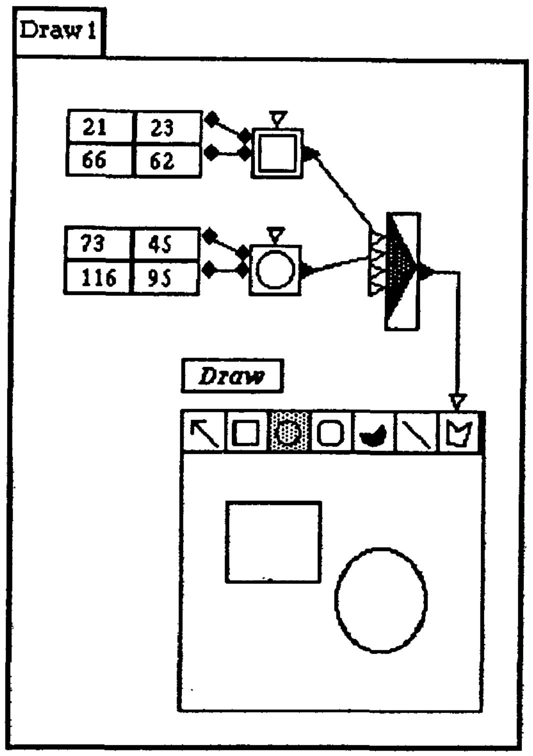


Figure 5a. The Draw component automatically lays out diagrams as the user creates a drawing.

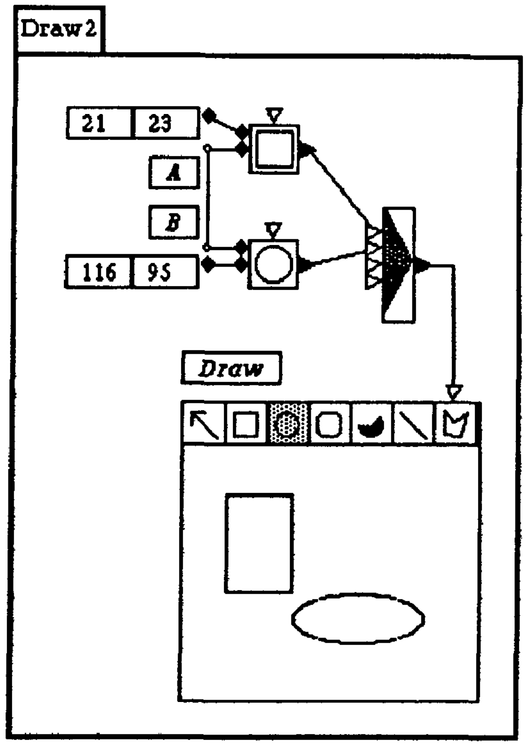


Figure 5b. By editing the generative diagram, the top-left of the oval is tied to the bottom-right of the rectangle.

same as one assembled from scratch, and hence it can be edited to place additional constraints on the image.

In the example of figure 5a, the only component the user actually laid out was the the one marked "Draw"; the rest of the diagram was automatically constructed as consequences of the user drawing within that component. By tying vertices A and B together in figure 5b, the top-left corner of the oval is constrained to have the same location as the bottom-right corner of the rectangle. Moving either the oval or the rectangle will cause the other to resize and maintain the constraint.

7 Iteration

Fabrik components that wish to communicate with the outside world do so by means of *gateways*, pins that provide the link between data outside the component and data within the component. Most often, these gateways provide a simple handoff of data from the outside to the inside (inbound gateways) or from inside the component to the outside world (outbound gateways).

Certain kinds of gateways can provide iterative functionality, however. For example, in figure 6, the inner component expects a collection of numbers to be fed to it. Its inbound collection gateway on the left disaggregates the collection into its elements, and the interior of the iterator is fired once in turn for each element.

On the right of the iterator, a similar outbound gateway collects up the values it receives from each firing of the iterator, and when the last iteration is done, it aggregates the data it has received into a new collection. In the example above, the collected rectangle graphemes appear as the bar chart image in the viewer at the bottom.

In Fabrik, every component fires to completion when instructed to, and iterators are no exception. The firing of an iterator may involve many complete passes through it. The number of passes is

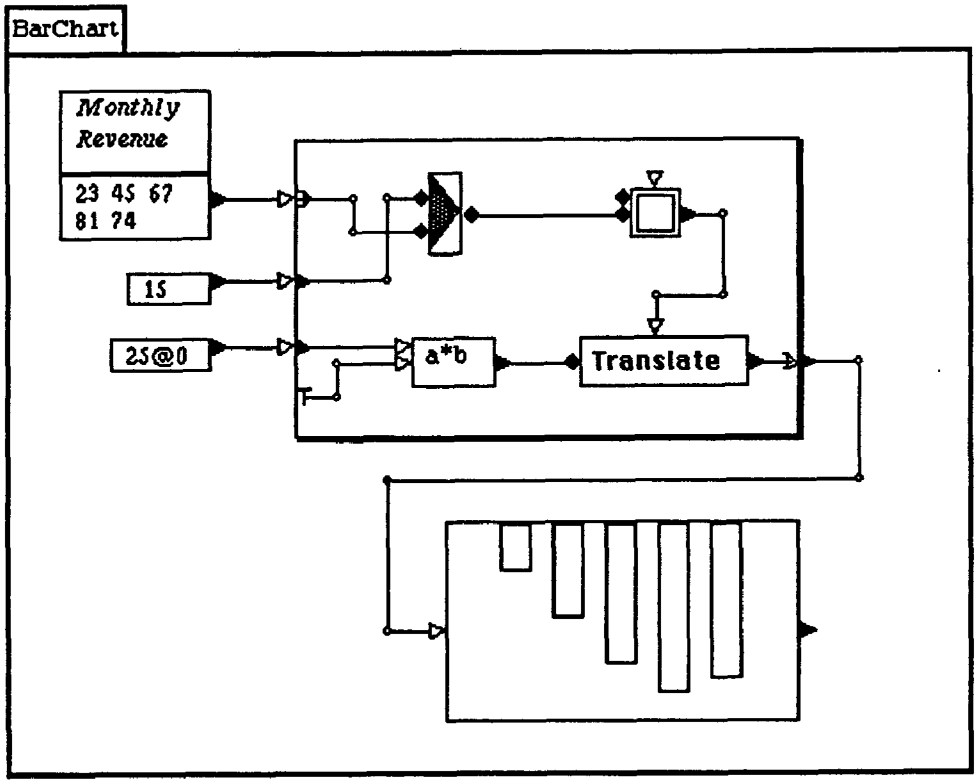


Figure 6. Streaming gateways provide iterative capability. Here numbers are converted to rectangles, resulting in a simple bar chart.

determined by both the component topology and the actual data values arriving. Global state within the iterator between passes is preserved by the semantics of the gateways, and a particular kind of gateway structure allows mutable values global to the iteration process to be used.

Within any particular pass through an iterator, strict rules of dataflow scoping prevail so that predictability and system integrity are assured. Fabrik's approach to iteration in a dataflow context was inspired by the Show And Tell Language [Kimura]. The library includes a full repertoire of iterative control gateways, such as conditional terminators and the iteration count used for horizontal spreading in the above example.

The bar chart example illustrates the use of iteration to produce graphical aggregates. This same approach is the key to Fabrik's handling of music and animation.

8 Type System

In order to validate (or prohibit) wiring attempts in Fabrik, a type is associated with each component pin. Fabrik currently supports primitive (record) types, bundled types, array types, and enumerated types. Primitive types are defined by the primitive data operations in the system, e.g., Number, Boolean, Character, Grapheme, etc. Bundled types are defined via bundler components that allow non-homogeneous element types. Each element type can in turn be any of the above types. The order is defined by the connections to the prong pins of the bundler component. Array types must have homogeneous element types, although their size does not matter for compatibility. An enumerated type defines a set of actual values allowed.

Each primitive component assigns a type for each of its pins restricting input and output to specified types. When a user places the mouse over a pin, the pin name and type are shown to assist connection of the correct pin. When a pin is about to be connected to another pin, type checking is performed. If the two types match, the connection is allowed; otherwise, a message is displayed in a status panel and the connection is not made. A primitive type can only match with other similar primitive types. A bundled type can match with another bundled type or array type. Two bundled types are compatible if their sizes are the same and each of the element types matches in order. A bundled type can match with an array type only if all the element types of the bundled type are the same and match with that of the array type. The result type is the array type. An enumerated type can match with a primitive type if the primitive type is the same as the type of the data allowed in the enumerated type, and data is actually checked if the primitive typed data is available. Some components have unspecified types, *i.e.*, their pins can be connected to any typed pins. The result type of such a connection is the specified type or remains unspecified if both types are unspecified.

User-built components can have their pins designated with types which are either implicitly inferred from the connections to the corresponding gateways or explicitly assigned to be some known types.

Components with unspecified type also have the ability to propagate types to other pins once a connection is made to a typed pin. For example, the selector component that selects from a set of inputs has all pins designated to be unspecified but of the same type (except the selection pin which is of Number type). Therefore when one of these pins is connected to a typed pin, that type is propagated to all the other unspecified pins of the selector, so that other connections to the selector will only be allowed if they carry that same type. Similarly when a wire to the selector is cut, its pin type reverts to unspecified or is inferred from other pins if there are still other connections. Thus even with the flexibility of unspecified types, every pin in the complete Fabrik diagrams will be inferred to have a type. This makes the diagrams simple and easy to understand.

9 Compilation

Compaction of representation and speed of execution are the reasons for undertaking compilation in Fabrik. The task of compilation here is to map the semantics of a diagram, as embodied in its interpretive behaviour when manipulated in the Fabrik layout editor, into the definition of a new class whose instances behave as specified by the diagram.

The current version of Fabrik uses the ability of Smalltalk to compile new classes and methods dynamically, and hence the code generated, as illustrated below, consists of Smalltalk methods.

Every Fabrik diagram contains a particular set of subparts (themselves all Fabrik components) and a particular set of what may be thought of as data slots, where each data slot corresponds to the datum beneath the "copper" of one particular wire. Instances of a compiled class representing a Fabrik diagram hold, in instance storage, the subparts and the data slots. Compiled methods mostly read from and store to these instance variables, making for efficient code.

The consequences of any data perturbation in a Fabrik diagram can be mapped, after a topological sort, into a linear dataflow path — a sequence of subparts to be traversed in a specified order and in specified manners. The compiled method representing any particular path carries out the corresponding series of traversals.

Compilation example 1: The FtoC

Looking at the internal structure of the FtoC component shown in section 4, we notice that the diagram has four subcomponents – the two bidirectional arithmetical components TimesDivide and PlusMinus (both of them in turn being components built with Fabrik) and the two boxes that generate the conversion constants 9/5 and 32.

Additionally, there can be seen to be five different pieces of data flowing in the wires, starting with, at center left, the datum that represents the Centigrade value, and including two invariant data constants, one intermediate result, and the Farenheit value.

After compilation, the FtoC would be represented by an object with the instance structure shown in figure 7a. Note that there is one instance variable for each constituent subcomponent, and one instance variable for each data slot in the component's scope.

Note that part of the responsibility of the traversal code of any component is to export its computed output data to the surrounding domain. Thus, for example, in method p1:, the call

c3 p1: x

will result in an updated value appearing on instance variable v3. In this case, v3 represents an intermediate result, which will be seen to be used subsequently as an input argument in the call

c5 p1: v3.

The calls to export:outChannel: handle exporting of values computed during traversal to the world outside for the FtoC itself.

Compilation example 2: The FCConverter

Turning to the FCConverter, we have an even simpler diagram, and hence an even simpler instance structure. In this case there are no traversal methods, since no data arrives from external components connected to the edge of this one. Instead, one needs "radiation" methods, which propagate data originating spontaneously from within nested subcomponents. Note that in this example, either of the sliders can generate new dataflow by being stimulated, via the user interface, from either keyboard input or mouse actions.

inst var name	description
c1	a TimesDivide component
c2	generates the constant 9/5
c3	a PlusMinus component
c4	generates the constant 32
v1	holds Centigrade value
v2	holds the constant 9/5
v3	holds intermediate result between PlusMinus & TimesDivide
v4	holds the constant 32
v5	holds Farenheit value

Figure 7a Instance structure for FtoC application

```
p1: x "called when a new Centigrade value
              arrives from the left"
   c3 p1: x.  "traverse the Times/Divide from left"
   c5 p1: v3. "traverse the Plus/Minus from left"
   self export: v5 outChannel: 2
              "ship V5 value out output channel 2"

p2: x "called when a new Farenheit value
              arrives from the right"
   c5 p2: x.  "traverse the Plus/Minus from right"
   c3 p2: v3. "traverse the Times/Divide from right"
   self export: v1 outChannel: 1
              "ship V1 value out output channel 1"
```

Figure 7b Compiled methods for FtoC application

inst var name	description
c1	the left Slider
c2	the FtoC
c3	the right Slider
v1	holds Centigrade value
v2	holds Farenheit value

Figure 8a Instance structure for compiled FCConverter

```
e1p1: x  "takes value x, emitted from the left
               slider, and propagates it rightward"
    c2 p1: x.  "convert from Centigrade to
               Farenheit; load result on v2"
    c3 p1: v2  "sends the value through the
               right slider"

e2p1: x  "takes value x, emitted from the right
               slider, and propagates it leftward"
    c2 p2: x.  "convert from F to C; will leave
               computed C value on v1"
    c1 p1: v1  "sends the value through the
               left slider"
```

Figure 8b Compiled methods for FCConverter

Note that in this example, the methods p1: and p2: invoked for subcomponent named c2, which is the FtoC, are precisely the methods p1: and p2: illustrated for the FtoC in the previous example.

Compilation example 3: The Slider

The Slider, shown in figure 3, combines elements of the two preceding compilation examples, in that it must respond both to traversal (new value arriving on an external input pin) and to internal change from the user interface (user drags the slider's bar, or types in a new value in the slider's numeric readout panel). Since it has a considerably more complex internal structure, we shall only illustrate the instance structure (figure 9a), one traversal method, and one radiation method (figure 9b).

Note that there is a common subsequence of six instructions between the two illustrated methods. Combining such common subsequences into separate methods is just one of a number of possibilities for codesize compression and code optimization arising from the basic scheme illustrated.

inst var name	description
c1	a Point component (15●0)
c2	a Point component (22●118)
c3	a RectangleCreator
c4	a GraphemeMerger
c5	a Mouse component
c6	a NumToPoint component
c7	an Expression component (6●50)
c8	a Default component
c9	an Expression component (0)
c10	an Expression component (112)
c11	a ConstrainPoint component
c12	an Expression component (26●6)
c13	an Adder
c14	a RectangleCreator
c15	a Display component
c16	an Integer component
v1	15●0
v2	22●118
v3	default style for fixed rectangle
v4	*fixed rectangle grapheme*
v5	grapheme output from Mouse component
v6	*sensor location from Mouse component or from NumToPoint*
v7	button-state output from Mouse component
v8	selected or default point
v9	6●50
v10	0
v11	112
v12	constrained point
v13	26●6
v14	corner for movable rectangle
v15	default style for movable rectangle
v16	movable-rectangle grapheme
v17	*merged grapheme*
v18	bitmap out from Display component
v19	computed magnitude

Figure 9a Instance structure for compiled Slider

```
p1: x  "invoked when a new magnitude enters the Slider from outside"
    c16 p1: x.                      "textual display of value at base of slider"
    c6 p2: x.                       "convert to a point"
    c8 p2: v6.                      "use 6@50 if nil comes in from outside"
    c11 p3: v8.                     "constrain point to fit within slider"
    c13 p1: v12.                    "compute corner for movable rectangle"
    c14 p1: v12 p2: v14 p3: v15.    "generate grapheme for movable rectangle"
    c5 p1: v16.                     "bundle rectangle with an input sensor"
    c4 p2: v5.                      "now merge with the fixed vertical rect"
    c15 p1: v17.                    "display the merged grapheme"

e5p2: x "invoked upon mouse-click in the slider's display area"
    c8 p2: v6.                      "use 6@50 if nil comes in from outside"
    c11 p3: v8.                     "constrain point to fit within slider"
    c13 p1: v12.                    "compute corner for movable rectangle"
    c14 p1: v12 p2: v14 p3: v15.    "generate grapheme for movable rectangle"
    c5 p1: v16.                     "bundle rectangle with an input sensor"
    c4 p2: v5.                      "now merge with the fixed vertical rect"
    c15 p1: v17.                    "display the merged grapheme"
    c6 p1: x.                       "convert to a number"
    c16 p1: v19.                    "display magnitude at base of slider"
```

Figure 9b Two methods for compiled Slider

```
p1: x
    "Pre-iteration processing"
    c6 import: x.                      "import the new collection and set up"
    c7 prepareToIterate                "initialize the collection-out gateway"

    "Actual iteration"
    self iterate:                      "repeat the following block until done..."
        [c6 fireOnce.                  "inject the next collection element"
        c5 fireOnce.                   "inject the current tick-count"
        c8 p2: v10 p3: v9.             "fire the bundler component"
        c3 p1: v4 p2: v3 p3: v7.       "create a new rectangle"
        c4 p1: v1 p2: v8.              "evaluate a * b"
        c2 p1: v6 p2: v2.              "translate the grapheme"
        c7 p1: v5].                    "accumulate at the collection-out gateway"

    "Post-iteration processing"
    c7 export                          "export the resulting array"
```

Figure 10 Compiled method involving iteration

Compilation example 4: Iteration

Compilation of components that iterate requires extra mechanism. Iterating traversals are decomposed into three processing phases: pre-iterative processing, actual iteration, and post-iterative processing. The compiled code for the actual iteration features invocation of "self iterate:" with a block as its argument. The block describes a single traversal, and its contents are similar to other traversal code, with special code added to carry out the functions of the iterating gateways.

For example, here is the code compiled for firing the iterator illustrated in the BarChart example in Section 7, with "x" representing the new array to be charted:

10 History and Status

Experience with Fabrik suggests that a successful visual programming kit requires only three things: Specification of an effective visual and computational interface for each component, interactive access to an interesting (network) library of existing components, and the ability to use and combine these components interactively to build new library components and finished applications. The examples and discussion above detail Fabrik's contribution in the areas of bidirectionality, synthetic graphics, iteration, type checking and compilation.

Fabrik began with an attempt to mix arbitrary layout and cell types in an object-oriented spreadsheet. The spreadsheet approach broke down with the complex expressions needed for synthetic graphics and other generative structures. The wiring approach addressed this problem and also opened the way for bidirectional constructions.

The initial Fabrik prototype was developed in Smalltalk within the Advanced Technology Group of Apple in 1985, and was demonstrated widely within Apple in Spring of 1986. The type system was added during the Winter of 1987 and compilation was completed in the Spring of 1988.

An important next step in this investigation is one of scale: to assemble a library sufficient to accomodate a large class of applications, and to support networking of this library so that many people can borrow from and experiment with each other's work.

References

[Kimura] T.D. Kimura and P. McLain, "Show and Tell™ User's Manual," Tech. Report WUCS-86-4, Department of Computer Science, Washington University, St. Louis, MO, March 1986.

[Borning] A.H. Borning, "ThingLab, A Constraint-Oriented Simulation Laboratory," Tech. Report SSL-79-3, Xerox Palo Alto Research Center, Palo Alto, CA, July 1979.

[Smith] D.N. Smith, "InterCONS: Interface CONstruction Set," Tech. Report RC 13108, IBM T.J. Watson Research Center, Yorktown Heights, NY, September 1987.

[Labview] "LabVIEW™ Demonstration Manual," National Instruments, Corp. Austin, Texas, 1987.

[Gould] L. Gould and W. Finzer, "Programming by Rehearsal", Tech. Report SCL-84-1, Xerox Palo Alto Research Center, Palo Alto, CA, May 1984.

[Ludolph] F. Ludolph, D. Ingalls, Y. Chow, S. Wallace, "The Fabrik Programming Environment," to be published in proceedings of the IEEE Workshop on Visual Languages, 1988.

[Metaphor] "Metaphor Capsule Development," Metaphor Computer Systems, Mountain View, CA.