The Fabrik Programming Environment

Frank Ludolph, Yu-Ying Chow, Dan Ingalls, Scott Wallace, Ken Doyle

Apple Computer Inc.
20525 Mariani Avenue
Cupertino, CA 95014

Abstract

Fabrik is an experimental interactive graphical programming environment designed to simplify the programming process by integrating the user interface, the programming language and its representation, and the environmental languages used to construct and debug programs. The programming language uses a functional, bidirectional data-flow model that trivializes syntax and eliminates the need for some traditional programming abstractions. Program synthesis is simplified by the use of aggregate and application-specific operations, modifiable examples, and the direct construction of graphical elements. The user interface includes several features designed to ease the construction and editing of the program graphs. Understanding of both individual functions and program operation are aided by immediate execution and feedback as the program is edited.

Keywords: visual programming, data-flow, direct manipulation, programming-with-example, user interface

Introduction

Fabrik is designed to simplify programming, a difficult task requiring large investments of time and effort. Lewis and Olson [1] summarize many of the difficulties associated with programming, among them the concepts of control-flow and variables, the use of multiple abstract representations, and the complexity of program synthesis. Their analysis of the spreadsheet programming paradigm suggests that its success is a result of many factors including a familiar, concrete, visible representation, suppression of the inner world of computation, automatic consistency maintenance, aggregate and high-level operations, and immediate feedback. They state that the features that make spreadsheets relatively easy to use are applications of some of the principles of cognition, such as the use of familiar representations and analogies to aid understanding, and the use of immediate feedback to aid problem-solving.

BASIC was the first widely used end-user programming environment. It achieved this status, in part, because it too contained features that were applications of those cognitive principles, e.g. immediate statement execution, easy access to the inner world of computation, a more concrete representation of variables, a few aggregate operations, and a simplified edit-compile-execute-debug cycle.

The developers of spreadsheet and BASIC programming environments also recognized that programming involves more than just the programming language. Both integrated to some extent the environmental languages used to construct, debug and modify programs and spreadsheets virtually eliminated the edit-compile-link-execute-debug cycle.

While BASIC and spreadsheets have taken significant steps to simplify programming, more needs be done. Generally speaking, spreadsheets have little support for creation and use of high-level, user-defined functions, and BASIC has only a simple subroutine facility. Debugging is still difficult in spite of the inner world suppression because the control-flow, multi-assignment properties of BASIC require the programmer to know the execution history in order to interpret the current state of the program, and neither spreadsheets nor BASIC provides for a broad, visible, concrete presentation of program relationships.

Fabrik is a visual environment that attempts to make programming more accessible to the casual and novice programmer by supplementing and extending the concepts that made spreadsheets and BASIC successful. The remainder of this paper describes elements of the Fabrik language and programming environment that could significantly simplify the programming task. A companion paper addresses the topics of synthetic graphics and compilation, and provides a more detailed discussion of the language elements described in the next section [2].

Language Overview

Fabrik is an interactive environment based on an augmented structural data-flow model. Programs are represented as data-flow graphs of interconnected function icons, called *components*. This basic approach, used by many other systems [3-8], was chosen because a graphical representation presents the programmer with a concrete view of the relationships between data and functions. Data-flow graphs provide good support for user-defined abstractions because they

can be hooked together and nested. On the other hand, control-flow graphs, e.g. flow charts, usually treat the access of data stored in variables as a side-effect, and side-effects make the merging of graphs more difficult. In fact, the need for variables to store temporary results disappears altogether from data-flow graphs.

The two prevailing models of data-flow are the token and structure models [3]. In the token model, data is viewed as a token that is absorbed by a node, transformed, and passed on to downstream nodes. Iteration is accomplished by creating loops in the graph to recycle tokens. In the structural model the data is not absorbed but remains for the lifetime of the execution. Nodes generate new output data based on the inputs. Streams of data are treated as a structure, e.g. lists or trees. Iteration is accomplished without loops by aggregate or recursive functions that operate on an entire structure. The maintenance of the execution history and the elimination of loops remove much of a program's dynamic operation and gives the structural model and, as a result, Fabrik a "timelessness" that makes program operation easier to understand and simplifies debugging.

Bidirectional Data-flow

Fabrik enhances the traditional data-flow model with bidirectional data-flow. Components have connection points or *pins* around their periphery. Input and output pins are shown as triangles that point toward or away from the component respectively. Bidirectional components have diamond-shaped pins which may function either as input or output depending on the direction of the dataflow. This extension permits the construction of components that combine several related functions, typically a function and its inverse, within a single package.

The direction of the data-flow through a bidirectional pin is established by the modality of the pins to which it is connected. For example, if a bidirectional pin is connected to an output-pin, it will function as an input pin. The actual function performed by the component is the one whose input and output specification matches the data-flow. The packaging of multiple functions in a single package results in a fewer number of system components.

Bidirectional diagrams result from the use of bidirectional components. Input entering a component at the left of a diagram, such as through a type-in box, may flow through the diagram left-to-right, while input entering a component at the right may flow right-to-left through the same diagram components and connections. This bidirectionality provides a simple local constraint mechanism [9] and results in diagrams with fewer components and connections.

Syntax

The syntax of text-based languages is always problematical for the inexperienced or casual programmer. The syntax for graphs is very simple, node-arc-node. Using a mouse, Fabrik components are dragged from a parts bin onto the Fabrik diagram and released. The mouse is then used to draw connections between the pins of different components. Potential syntax errors are limited to attempts by the user to connect incompatible pins, e.g. input to input, graphic to numeric, and connections that result in loops. If the programmer attempts such a connection, Fabrik refuses to make the connection and informs the user of the incompatibility. Thus every Fabrik diagram is always syntactically correct.

Diagrams that have unconnected pins may still execute. A component with one or more unconnected input pins may be able to generate output value(s) either because an unconnected input pin might have a defined default value, or because some function of the component can compute without using the pin's value. If a component cannot compute, the values on the output pins are invalid and this invalidity is passed on to the connected input pins, overriding any default value defined for the pin. Connections that carry an invalid value is shown as a dashed line.

This section described techniques that Fabrik uses to simplify the programming language. The structural data-flow model promotes understanding of program operation with its timelessness and simplified model of iteration. The use of diagrams rather than linear text simplifies the syntax and makes the program relationships visible and concrete. And bidirectionality reduces both system and program component counts and the number of program connections.

Building A Simple Analog Clock

The Fabrik programming environment includes a basic set of predefined components. The components perform arithmetic, string and graphic manipulation, file access, and generate common graphical elements such as rectangles, ovals, lines, polygons and bitmaps. Additional sets or *kits* of application specific components can be added as needed. For example, a modern application program has many idiomatic graphic elements that make up its user interface. A user interface kit might contain components such as views, panels with editable text, lists of selectable items, choice buttons, scroll bars, and menus that can be combined in various ways when building new applications.

Some of the components are *primitive*, implementing system-defined functions. The rest are *built* from primitive and possibly other built components, and they can be altered by the user. Primitive and built components are essentially indistinguishable from each other in appearance and use.

In this section, we illustrate the Fabrik programming process and the support for browsing, constructing, testing and packaging a complete application by building a simple analog clock. Figure 1a shows a Fabrik Parts Bin window (above) and a Fabrik Construction Window (below). At the bottom of the

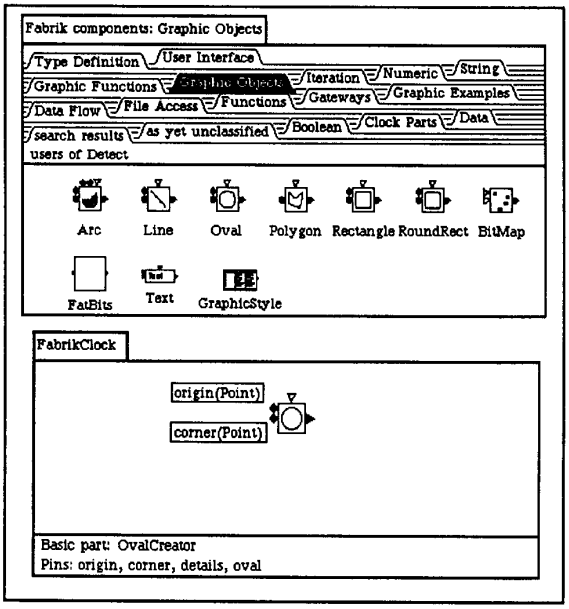

Figure 1a. An Oval Creator has been dragged from the Parts Bin (above) to a new Construction Window (below).

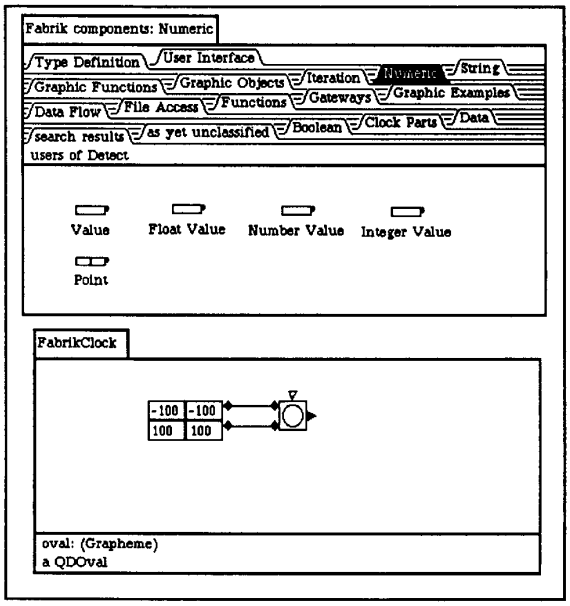

Figure 1b. Two Points have been connected to the Oval Creator and locations of the top-left corner and bottom-right corner have been entered. The oval grapheme is waiting at the output pin of the Oval Creator.

Construction Window is a *status panel* that displays information about the selected component, pin, or vertex, and feedback about editing activity such as error messages about attempts to connect incompatible pins. The operation of the Parts Bin is described below under *Finding Components*.

To build a Fabrik application, the user drags components from the Parts Bin into a Construction Window, and connects their pins together. In the first figure, an Oval component has been copied from the Graphic Objects category of the Parts Bin to the Construction window to be used as the clock face. The Oval has two pins on the left, for the locations of the top-left and bottom-right corners of a box that would contain the oval, and a pin on top for specifying border width, border pattern and inside pattern. The oval grapheme of the specified size and appearance is output from the pin at the right.

In figure 1b, the author has dragged two Point components from the Numeric category of the Parts Bin and connected them to the Oval. During connection, a pin's name and type (as shown in figure 1a) automatically pops-up when the cursor is over the pin. In this figure the author has entered two points for the top left corner and the bottom right corner of the clock face, i.e. the clock face will have a radius of 100 pixels.

In figure 1c, the author has dragged both Group and ScalableDisplay components from the Graphic Functions category of the Parts Bin. A Group can merge any number of graphemes. When the Group was copied, the user specified that it should have four input pins, although pins can be added and deleted at any time. As soon as the Oval output was connected to the Group input and the Group output connected to the ScalableDisplay input, the clock face was displayed. The ScalableDisplay, one of several graphic viewers in the library, automatically scales the input grapheme to the size of the viewer.

In figure 1d, a ClockHand component has been copied from the Clock Parts category of the Parts Bin three times and the output of each connected to input pins on the Group. The three ClockHand components will produce the second, minute and hour hand graphemes. Inputs to a ClockHand component are two

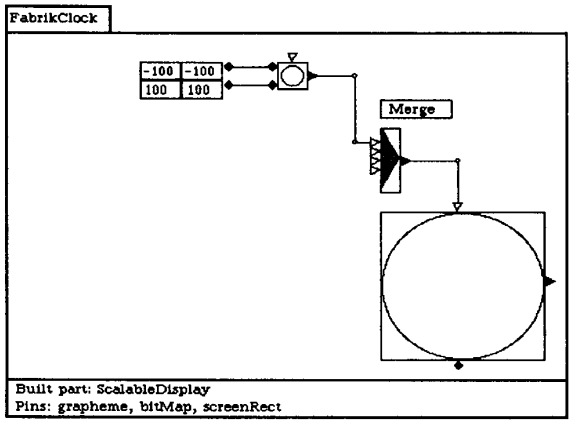

Figure 1c. A Group and a ScalableDisplay have been installed and attached to view the clock face.

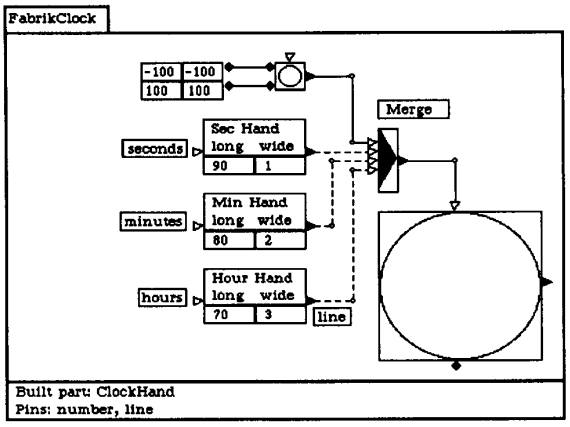

Figure 1d. Three ClockHand generators have been laid down and connected to the Group to produce second, minute and hour hands.

type-in boxes to customize the length and width of its hand display and an input number between 0 and 59. In the clock example, the second hand (the top one) will display as 90 pixels long and 1 pixel wide, the minute hand 80 long and 2 wide, and the hour hand 70 long and 3 wide. The connections between the ClockHand outputs and the Group appear dashed at this point in the construction because, with no hands produced yet, the values are invalid. Fabrik tracks invalidity so that no component executes with invalid input data.

In figure 1e a Time component has been added and connected to the Second ClockHand and Minute ClockHand. Activated by the connections to their inputs the Second and Minute ClockHands generate their graphical clock-hand outputs and the dashed output connection lines become solid. The clock-hands are merged with the clock face and displayed. Our clock is running and the clock hands are moving as the time

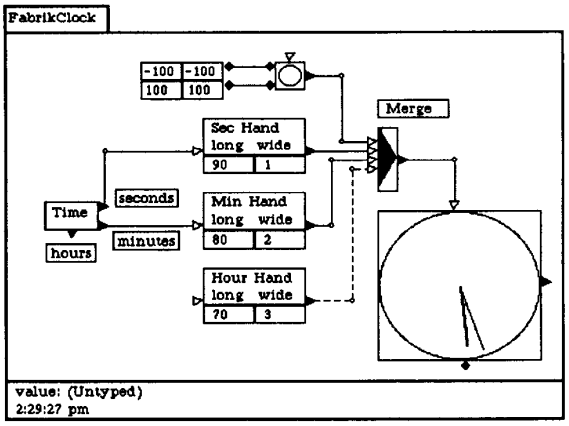

Figure 1e. The Time component that generates the seconds, minutes and hours has been hooked up to produce the display of the second and minute hands.

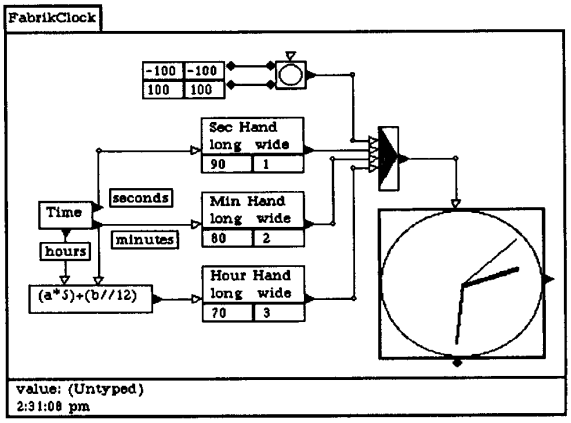

Figure 1f. To show the hour hand in proper position a Formula has been attached that converts the hours in between 0 and 11 to between 0 and 59 depending the minutes value.

changes. Only the hour hand remains to be added.

Figure 1f shows a Formula component added to convert the hours from the Time component, an number between 0 and 11, to a number between 0 and 59 for the Hour ClockHand. It also adds in an offset for the number of elapsed minutes. The Formula component evaluates an expression (currently in Smalltalk-80 syntax) typed-in by the user. Here the first argument pin ('a' in the expression) is connected to the hour output pin from the Time component and the second argument pin ('b') is connected to the minutes pin. The hour-hand grapheme propagates and is displayed.

The desired application has been programmed and is now fully functional. However, the clock display is surrounded by the computational components and their connections. Fabrik allows a subregion of a diagram to be designated as the *user frame*. This has been done with the ScalableDisplay in figure 1f, and the user frame is shown as a heavy border around its periphery. Once the user frame has been designated, a menu command is used to *enter* the frame. This command instructs Fabrik to restrict the view to only

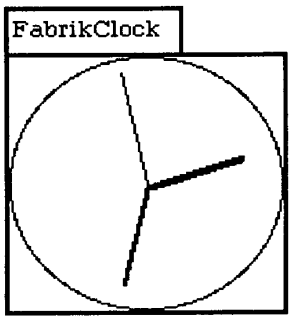

Figure 1g The connection diagram has been hidden and the clock can be launched and its window can be resized just like a normal application.

the designated components, and to make the result visible in a standard application window. Figure 1g shows our clock after entering the user frame and being enlarged for better viewing.

An application such as this can easily be assembled in a short amount of time. 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 and provides the end user with the ability to tailor an application to his specific needs.

The Fabrik Environment

The Fabrik environment has many elements: a component library, library management functions, file system interface, a graphical language-based editor, an interactive run-time system with change-triggered recomputation, and a set of debugging tools. Each element has some user interface aspects.

User Interface Guidelines

The intended Fabrik user interface was to be generally Macintosh-like though the experimental nature of the Fabrik project, implementation in Smalltalk-80, and the requirements specific to the programming task encouraged us to try new alternatives. The interface was to be single-handed, mouse-based, visible, concrete, and direct. Many alternatives were tried. Often competing approaches were implemented and made switchable by a flag setting in order to understand each alternative better and discover personal preferences. Although the experience is anecdotal, we feel it has value and some of the more interesting alternatives are described below.

One-Handed and Two-Handed Input

Since the early days of the mouse, there has been a continuous debate between those that prefer to use keyboard-based interfaces and those that prefer mouse-based interfaces. But there is also a more muted discussion on the mouse side about one- and two-handed input. The most common forms of two-handed input are the use of command-keys as an alternative to menus and the use of modifier keys to alter the function of the mouse button. In addition, Buxton [11] and others suggest the use of additional devices, such as the touch pad, to be controlled by the alternate, "non-mouse" hand. The reasoning behind two-handed input is that it can increase the bandwidth from person to machine, alleviating a common performance bottleneck caused by today's faster machines and user-event driven applications.

Fabrik was initially designed to be run single-handed and, to that end, techniques not common to the Macintosh, e.g. the gesture menus described below, have been used. The occasional new operation that seemed to require the use of a second-hand was always augmented with single-handed alternatives soon afterward.

More than one user, shown only the single-handed methods, complained of having to "sit on their left hand." An apparently conflicting experience of the implementation group, who were all quite familiar with the two-handed alternatives, was that even those that pressed most for two-handed input tended toward single-handed use. Subsequent observation showed that there was only limited opportunity during editing to use the second hand and that it tended to move away from the keyboard to more relaxed positions when not used for a period of time.

As a result of this experience, the user interface goals have been modified to provide a two-pronged approach to mouse use: design a single-handed interface which is complete, visible, and as efficient as possible, and augment it with two-handed alternatives that focus on improved performance and continuous use. Where modifier keys are used in conjunction with the mouse button, there has been an attempt to define a consistent set of modifier operations that can be applied to all objects. For example, the shift key is used for multiple selection and to constrain drawing operations, the command key is used with alpha keys to issue commands and with the mouse button to pop-up a menu appropriate to the object under the cursor, and two other keys are used for moving and drawing.

Building Programs

Program synthesis, the process of constructing a program from component parts, is difficult. The programmer must have full knowledge of what each of the parts does, how the parts interact, and plans for combining parts to perform common functions such as counting, traversing data structures, etc. Most languages also require the programmer to translate from "what" is to be done to "how" to do it. In spite of the difficulties, synthesis is used to write programs because decades of use have demonstrated its extraordinary flexibility.

Fabrik provides support to ease many of the inherent difficulties. Structural data-flow reduces the "how" by relieving the programmer of the need to specify much of the arbitrary sequencing associated with control-flow. Built-in high-level and aggregate components reduce the need for the programmer to learn and remember plans. Immediate execution gives rapid feedback about the in-diagram operation of components. And on-line support for locating, documenting, and examining in-diagram use of components aid in component selection.

Fabrik also supports alternatives to synthesis: modification and programming-with-example. Given a large library of working components, it might often be easier to find and modify a similar component rather than build a new one from scratch. Important to this approach is support for rapidly locating the potential base component. When a suitable, modifiable example cannot be easily located, the user might construct a (portion of a) new program by giving

examples as when defining the programs graphic elements and their behavior [12]. In Fabrik the programmer draws graphical elements within a Draw component which automatically generates the corresponding Fabrik diagram as the elements are drawn. (See "The Draw Component" below.)

Finding Components

The Parts Bin is divided into sections indicated by folder tabs. Fabrik maintains two permanent folders, "search results" and "as yet unclassified." The user can add, delete, and rename other folders at will. All components are kept in the Parts Bin and appears in at least one folder. The user can put copies of a component in additional folders or delete a component from a folder.

Fabrik currently supports three kinds of searches: partial-name, keyword, and content. Each places copies of the selected components in the "search results" folder. The results remain in the folder until the next search is performed. The partial-name search is a simple form that all components that contain the given string anywhere in their names. The keyword search selects all components that include any of the given keywords anywhere in component's on-line text description or within comment blocks in the built component's diagram.

The content search selects built components that use a specified component, either primitive or built. (This is similar to opening a browser on a method's "senders" in Smalltalk-80 [15].) The content search is an important form of on-line documentation about a component. The user can open the diagrams of components found by a content search to see how the component of interest is used and, since Fabrik is interactive, what the component actually does within the diagram.

The Parts Bin provides an adequate organizing scheme for about 100 components. The current search facilities work reasonably well for this quantity, but for larger quantities additional facilities such as alphabetical and chronological listings, multiple parts bins, and a pin-type based topological search will be necessary.

Editing

A complete set of efficient techniques for editing diagrams does not yet exist, but it is useful to remember that editing text programs has not always been easy. Over the years sequence numbers have disappeared, simplifying statement insertion, and tools like group select, indent, and move have made it easier to coordinate logical organization and visual layout. As program interfaces become more complex, new tools are being developed that serve as learning and memory aids because, ultimately, the program text still must be accurately typed-in.

Drawing and CAD systems will likely be a rich source of new techniques during the next few years. There is some expectation that automatic aids, e.g. the routing of connections, will be important. They may well be, but it is interesting to note that the widely used editing aids for text-based languages today are the ones that provide efficient and precise manual control with minimal automated support.

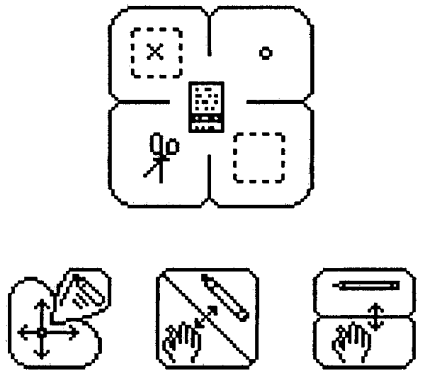

Figure 2. Gesture Menus. The lower three show alternatives menus for the same function.

The individual techniques used in Fabrik are not generally unique. Two not in common use, relocatable pins and directional gestures, are described below.

Pin Positioning

Fabrik components have pins that are used as connection points around their periphery. In many other diagram-based systems the connection points are invisible, fixed, or position dependent [4-8]. In Fabrik, the user can reposition a component's pins. This freedom of pin placement allows components to be placed directly adjacent to each other, as when creating a user frame, and enables the user to simplify a diagram's wiring by moving pins to reduce the number of twists and turns a connection must take. Although a pin that has been moved cannot be recognized by its location, identification by this means is only marginally useful given a large number of components. Moving the cursor over the pin gives immediate positive identification.

Directional Gestures

Fabrik uses simple directional gestures to increase the number of commands associated with the mouse button. The user initiates a gesture-based operation by positioning the cursor over an object, pressing the button and moving in a specific direction. Once the initial direction, hence operation, is established, the mouse may then be moved in any direction to complete the operation. If no significant cursor movement occurs within a half-second after the mouse button is pressed, a pop-up "gesture menu," such as those shown in figure 2, appears under the cursor. The short delay prevents flashing during normal editing but provides prompting for less commonly used operations.

For example, a gesture menu is used to both move and connect pins and vertices because their small,

eight-pixel-square size is not large enough to allow separate areas to be designated for each operation. A connection is initiated by clicking on the pin/vertex and moving in one direction, while relocation is initiated by clicking and moving is a different direction. An additional set of gestures is defined for the editing window background and includes operations such as group select and delete.

The intent was to create a very fast form of command invocation using just one hand and is somewhat different from both pie menus [13] and full gestures [14]. Gestures require more complex movements that are difficult to perform with a mouse and are slow compared to a directional gesture. Pie and other forms of pop-up menus require the mouse button to be released over specific area to invoke the command, an action inappropriate for certain actions such as moving and drawing. There is also a visual feedback loop from eye to hand that slows command invocation.

Although no formal user-testing has been performed, experience suggests that the design of gesture menus is fairly critical. Four unique directions seems the maximum for fast error-free operation using a mouse, though wedges as small as 45 degrees may have acceptably low error-rates. The near-reflex response of learned movements to pre-conscious intent requires that common operations appearing on more than one menu must use the same direction regardless of context or high error rates will occur. And the asymmetric movement of the human wrist means that left-hand/right-hand differences should be considered.

User reception to gestures used in this way has been mixed. While established Fabrik users have few complaints, new users split about 50-50 initially until they learn to first establish the operation, then the direction. Even among some experienced users the feeling persists that other approaches might be more appropriate for operations that inherently include arbitrary direction, e.g. moving and drawing.

Running and Debugging

A Fabrik diagram is always active, that is, every time a connection or value is changed, the related subgraph recomputes its values. This immediate feedback after each action promotes understanding of both the function of individual components as they are added to the diagram and the operation of the diagram as a whole. The edit-compile-execute-debug cycle is greatly simplified since the process of editing automatically includes "compiling" and execution. (Fabrik is interpretive at diagram level, but built components are compiled when placed in the parts bin.)

Fabrik's structural data-flow model and graphical representation do much to simplify debugging. As mentioned above, the input and output values of each component persist following execution. Connections that have no value are shown as dashed lines (see figure 1d), immediately indicating to the the user inactive portions of the diagram. The user can click on any pin or vertex and see its value in the status panel at the bottom of the construction window. In addition, small value display boxes can be attached to pins and vertices to monitor values continuously at several points simultaneously.

Should the monitored values cause the user to suspect a faulty user-built subcomponent, it can be opened into an active edit view of its own to permit both monitoring values during operation and direct editing of the subcomponent. Edits are immediately reflected in the execution of the program. The changes may be saved, either as an updated version of the component or as a different component.

Encapsulation and the User Frame

When a Fabrik diagram has been completed the user places a user frame around the part of the diagram that is to be visible in its component/application form, as described in the Clock example above, and saves the diagram thus encapsulating its function as a reusable component that is available in the parts bin. If the user frame includes components that accept keyboard input or display components, they will be active in the component form, that is, the user can type in to them or they will display the results of their internal computation as appropriate.

An alternative to the user frame adopted by other systems [7] is the use of a second window that displays only the user visible elements of the diagram while the first window holds the entire diagram. This approach allows components to be arranged for display in an arbitrary fashion without impacting the diagram layout but at the cost of additional windows and a mechanism to relate visible components in the display view to the corresponding component in the diagram. We felt that these costs added too much complexity and so chose the simpler user frame, an approach we are comfortable with but do not yet have sufficient experience to validate.

The Draw Component

The Draw component is a special component that automatically generates the program diagram for graphical display while the user draws the graphical objects in the Draw component. It frees the user from having to find, place, and connect the corresponding graphic components and operations. It allows users to design graphical displays in a very natural way, much as a normal drawing program. Adjustments to the location and size of the graphic objects in the Draw component are immediately reflected in the diagram. Alternatively, the ability to enter specific values into the diagram using the keyboard gives the user the precise control that can be difficult using just the mouse. The Fabrik diagram generated is identical in structure to one assembled from scratch so the user can edit it directly to add additional components and

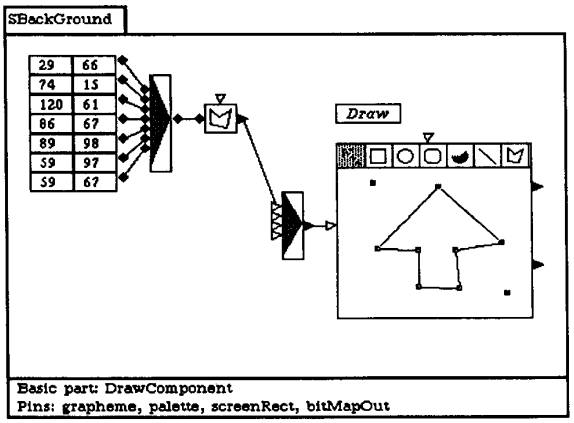

Figure 3a. A Draw component has been laid down and the polygon tool selected, and the sketch of an arrow has been drawn in the Draw component.

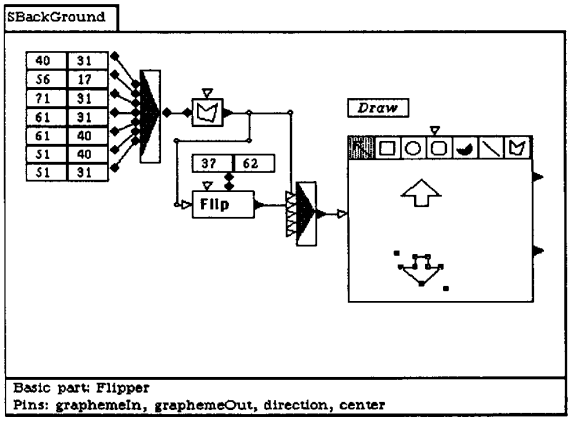

Figure 3b. The arrow has been adjusted and a Flip component has been laid down to generate the down arrow.

connections that are not possible via the Draw component.

We demonstrate the Draw component by creating a scroll bar background that can later be combined with the thumbing logic for a usable scroll bar. The complete scroll bar diagram and the discussion of synthetic graphics and mouse sensitivity are in a companion paper [2].

The Draw component has a palette at the top for selecting the graphical objects to be drawn. In the example of figure 3a, the only component the user actually laid down was the the one marked "Draw". A scroll bar contains an upward-arrow, a downward-arrow and some rectangles. The polygon tool (the rightmost one in the palette) was selected first and a rough sketch of the up-arrow drawn. When the arrow drawing was finished, the rest of the diagram was automatically constructed by Fabrik. The selection tool (the leftmost one in the palette) was then selected. The control points for further adjusts are shown in the figure.

In figure 3b the author has adjusted the up-arrow to get the correct shape, size and location. To produce the down-arrow, Flip and Point components were dragged from the parts bin and connected as shown. The Point was connected to the "center" pin (top right one) of the Flip component, but no values were entered. (By default a grapheme is flipped vertically about its center.) When the output of the Flip was connected to the second input pin of the Group, the down-arrow appeared on top of the up-arrow. Then the down-arrow was dragged downward to the location shown in the figure which automatically updated the Point value to the corresponding value for the center of the Flip.

In figure 3c a rectangle tool (the second one from left in the palette) is selected first to create the rectangle (the Rectangle above "Up Arrow Rect") that surrounds the up-arrow. Then the same logic for flipping the up-arrow is copied to flip this rectangle so that a second rectangle is created to surround the down-arrow. At this point, the order of the Group inputs is such that the arrows are displayed under their bounding rectangles. To reverse this ordering the author invokes a menu command in the Group component to rotate the connections downward twice. The corresponding portions of the diagram were manually rearranged using a group-move for neatness.

The the gray area (the Rectangle next to "Gray Area") was generated by creating another rectangle in the Draw component in between the two arrows' bounding rectangles. Under the area marked "Style", the color specification of this rectangle was generated with a GraphicStyle component, selecting '1' for the border width, 'black' for the border color and 'light gray' for the inside color, and then connecting its output pin to the graphic style pin (top) of the Rectangle.

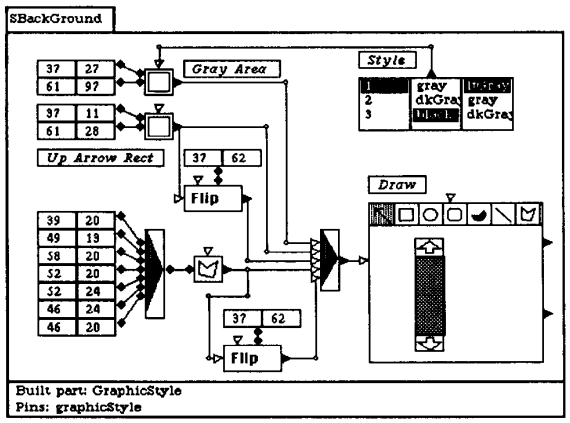

Figure 3c. The bounding rectangles have been created in the Draw component and another Flip component was added for the down arrow's rectangle.

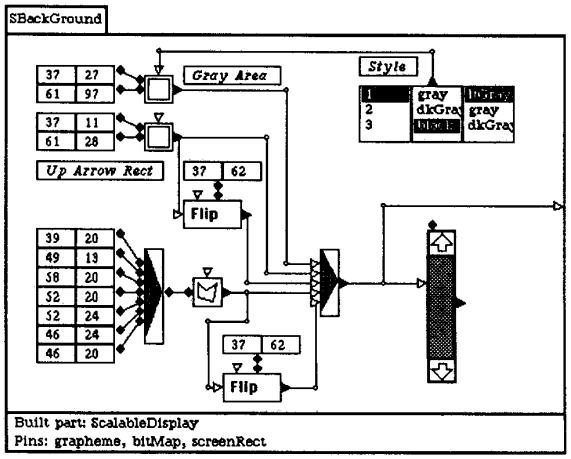

Figure 3d. The Draw component has been replaced by a ScalableDisplay so that the scroll bar will be automatically scaled to the viewer size. An output gateway has been added, and the diagram is ready to be used to build a scroll bar.

To complete the scroll bar for use with the thumbing logic, the author replaced the Draw component with a ScalableDisplay as shown in the figure 3d. An output gateway was added to the right side of the window and connected to provide the scroll bar grapheme to the user of this component. Now with a user frame around the ScalableDisplay, this component is ready to be stored into the parts bin for use with other components.

History and Experience

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. Graphical layout, as described above, addressed this problem and also opened the way for bidirectional dependence.

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. Two more feature-laden and library-rich versions were made available to a limited audience by Spring of 1987. The type system was added during the Winter of 1987 and compilation was completed in the Spring of 1988.

Rapid system evolution and poor performance have limited experience with the programming environment to the implementation of demonstration programs, e.g. the clock, configurable analog gauges, and simple animated simulations, simple parsers, bar charts, and the full set of relational database operators. Now that the system is more stable and compilation has improved the performance, it is possible to attack more interesting problems.

An important next step in this investigation is to assemble a library of components sufficient to accommodate 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. This broader group of users and programs will help to test the ideas expressed above, investigate solutions to the problems of diagram complexity, and evolve the user interface more rapidly.

References

[1] C. Lewis and G.M. Olson, "Can Principles of Cognition Lower the Barriers to Programming?" in *Empirical Studies of Programmers (Vol 2)*, Ablex, 1987.

[2] D. Ingalls, S. Wallace, Y-Y. Chow, F. Ludolph, K. Doyle, "Fabrik: A Visual Programming Environment," to appear in *1988 OOPSLA Conference Proceedings.*

[3] A.L. Davis and R.M.Keller, "Data flow Program Graphs," *IEEE Computer*, Feb. 1982, pp 26-41.

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

[5] I. Yoshimoto, N. Monden, M. Hirakawa, M. Tanaka, T. Ichikawa, "Interactive Iconic Programming Facility in Hi-Visual," *1986 IEEE Workshop on Visual Languages*, pp 34-41.

[6] ProGraph™, on-line documentation, 1988.

[7] *LabVIEW™ Demonstration Manual*, National Instruments, Corp. Austin, Texas, 1987.

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

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

[10] D.C. Smith, *Pygmalion*, ISR 40, Birkhauser Verlag Basel, 1977.

[11] W. Buxton and B.A. Myers, "A Study in Two-Handed Input," *Proc. CHI'86 Human Factors in Computer Systems*, pp 321-326.

[12] B.A. Myers, "The State of the Art in Visual Programming and Program Visualization," Tech. Report CMU-CS-88-114, Computer Science Department, Carnegie Mellon University, Pittsburg, PA. 1988.

[13] J. Callahan, D. Hopkins, M. Weiser, B. Shneiderman, "An Empirical Comparison of Pie vs. Linear Menus," *Proc. CHI '88 Human Factors in Computer Systems*, pp. 95-100.

[14] M. Lamb and V. Buckley, "New Techniques for Gesture-Based Dialog," *Human-Computer Interaction - INTERACT '84*, North-Holland, Amsterdam, 1984, pp. 135-138.

[15] A. Goldberg, *Smalltalk-80, The Interactive Programming Environment*, pp 178-179, Addison-Wesley, 1984