OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

Grif: A Graphical Programming Language for Robotics

David W. Sandberg
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

86-60-6

Grif: A Graphical Programming Language for Robotics

David W. Sandberg
Oregon State University

September 16, 1986

1 Introduction

Robot manipulators are being used increasingly in manufacturing. A major part of the cost of a robot system is the cost of developing the software needed to control the robot. Grif is an interactive graphical programming system intended to reduce program development costs for robotics. Grif is interactive to handle the iterative nature of programming robots[3]. Grif is graphical to better express the control functions of the program. Grif is flexible since new graphical primitives can be defined when existing primitives prove inadequate. Since new primitives are defined textually, the programmer has a choice of using graphical or textual programming.

In manufacturing assembly lines, the major part of programming involves specifying how the components of the assembly line are to be controlled. The first assembly lines were controlled with relays. Relay-ladder-network diagrams[1] (see Figure 1) were developed to program these assembly lines. Today, relays have largely been replaced by programmable logic controllers that simulate the behavior of relays with a microprocessor. Relay-ladder-network diagrams are still used to program programmable logic controllers. Ladder networks are good at expressing the control functions of an assembly line and are a natural form of expression for the engineers who design the assembly line. Unfortunately, ladder networks are rather poor at handling data and are often hard to decipher because of the lack of mnemonic labels for inputs and outputs.

When robots are added to the assembly line, data becomes important for expressing the locations of parts, but data is still less important than the control functions. Ladder networks are no longer adequate for this purpose. Robots are usually programmed using the technology that has been developed for digital computers[3,4,10]. This technology assumes that data manipulation is central. A technology where control is more central is better for programming robots.

Graphics has been the preferred method for expressing control flow. For example, flow charts are used to express control flow in programs, Petri nets[6] are used to express control flow in parallel systems, Ladder networks are used to express the control in assembly lines, and graphs are used to define finite state machines. A graphical form should also be a good choice for expressing the control of a robot.

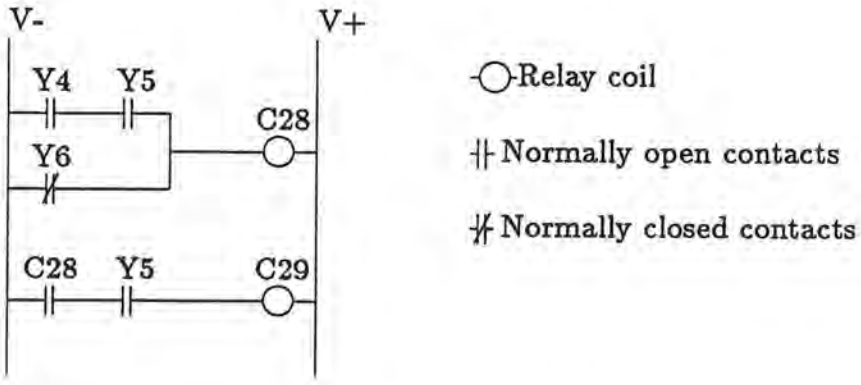

Figure 1: An example of a ladder network diagram. This "ladder" has two "rungs". The boolean logic equivalent is $Y4 \cdot Y5 + \overline{Y6} = C28$; $C28 + Y5 = C29$.

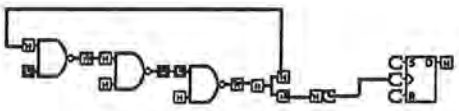

Figure 2: An example Grif program.

A graphical form has other benefits[7]. The human mind is strongly visually oriented and can acquire information faster from a complex picture than by reading text. A graphical form is better suited for expressing the possible parallelism than sequential text. A picture can be more easily animated to express dynamically changing information. A good graphical system should also be easier for a novice to use since a picture is often a more concrete object than a textual representation.

2 A Simple Example.

Figure 2 contains an example of a Grif program. The basic components of a Grif program are modules. Each nand gate in Figure 2 is a module. Each module has some output ports that are connected to input ports of other modules. Modules communicate by sending messages over these connections. The ports are typed according to the type of data sent over the port. Figure 2 contains two different kinds of ports, one for boolean data and the other for control information. (Control information is a signal that an event has occurred.) The kind of port can be determined by the symbol that is used to represent it. Boolean ports are represented by H or L. Control ports are represented by C. The programmer can

interactively send data to a port by pointing at it and typing an appropriate key. Each module contains one or more processes that monitor the input ports and send data out the output ports. In addition a module may have a process that updates the display when the module changes state.

The nand gates in Figure 2 have the obvious function. The fourth module from the left duplicates its input on its two outputs. The fifth module from the left sends a control signal when its input changes from low to high. The module on the far right is a flip-flop. The state of the Grif program in Figure 2 is stable since the unconnected input to the nand at the left is low. If the programmer pointed to this port and typed H, the state would become unstable, and the programmer would see the boolean ports changing back and forth from high to low.

Building the program in Figure 2 is straightforward using the Grif editor. The modules are created by selecting them from a menu, pointing to a location to put them, and clicking a mouse button. The ports are connected by pointing to the input port and clicking a mouse button, pointing to the intermediate routing points, and finally pointing to the output port and clicking a mouse button. If a module has extensive state, a module specific menu is provided to edit the state.

3 A More Complicated Example

Figure 3 contains a more complicated Grif program. This program controls a robot that takes parts from one tray and places them in another. Control flows primarily from left to right through the figure. Figure 4 explains the functions of the ports of some of the modules used in Figure 3. The modules labeled with M are move commands to the robot. Each of these modules specifies where to move by reading the point present on its port, and adding an internal delta to the point. This delta makes makes it easy to specify the approach to the slot relative to the slot in the tray. The move is performed when a control signal is sent to its input port on the left. When the move is completed, a control signal is sent out its output port on the right.

The tray modules supply the location data to the move modules. The tray is described by the number of slots in the tray, the location of first slot, and the location of the last slot. Each time a control signal is sent to the port on the upper right corner, the location of the next slot is sent to the move module. When the tray is empty, a high is sent out the boolean port on the tray module. This signal is converted into a control signal which sets a flipflop which controls a switch which redirects a control signal so that the robot arm will move to a parking position.

Figure 3 contains an abstract module to represent the control around the destination tray. This module is expanded in the lower right of Figure 3. An abstract module is created by opening a new edit window and creating the contents of the abstract module like any other Grif program. Next, an abstract module is created by selecting the *Amod* menu item. Ports can then be copied from the other modules onto the abstract module. This abstract module is then copied into a buffer by selecting the *copy* menu item. The abstract module can now be pasted into other edit windows by selecting the *paste* menu item. The contents of an abstract module can be inspected by pointing at the module and

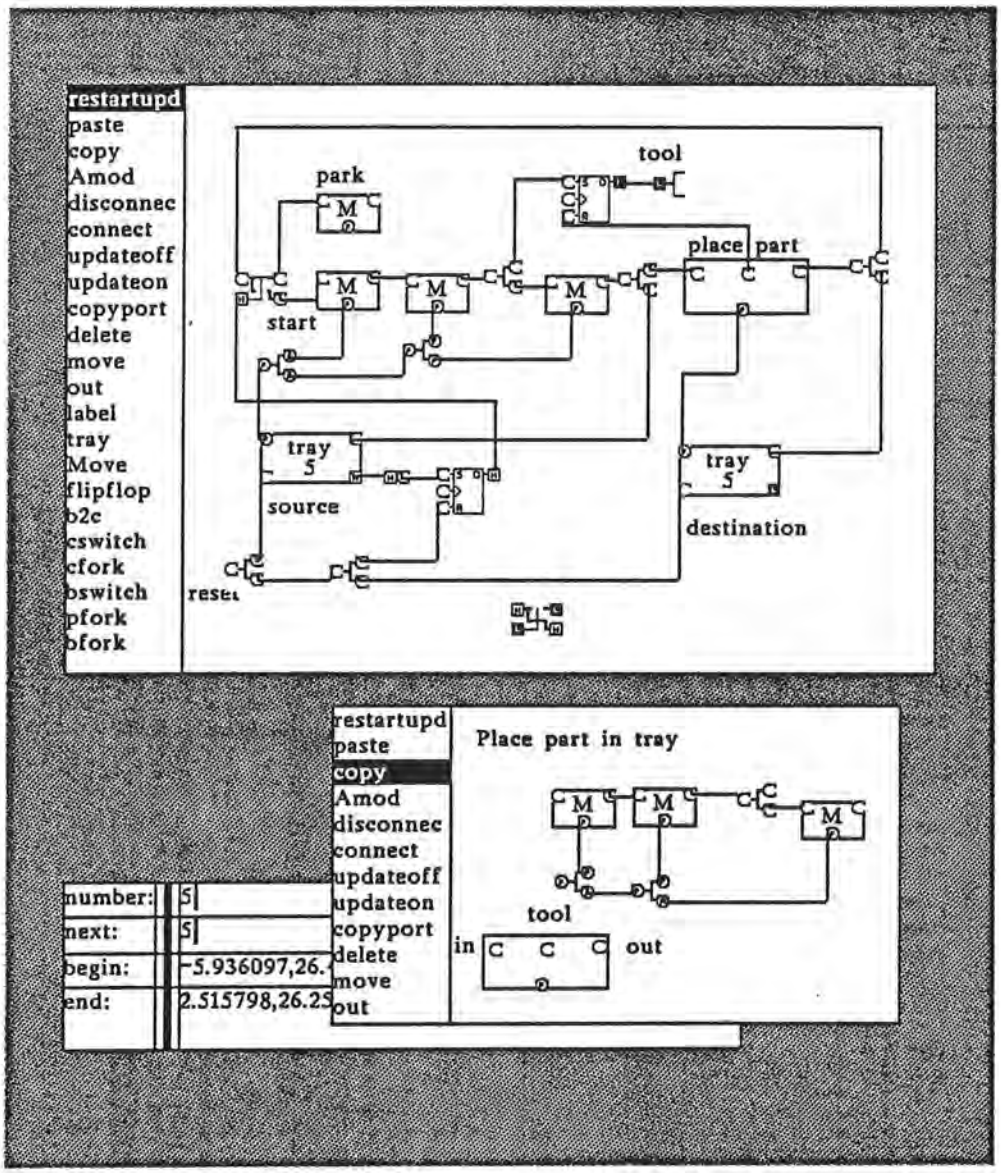

Figure 3: A Grif program for a robot.

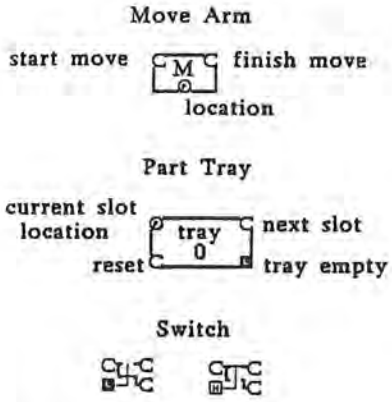

Figure 4: The functions of some ports.

selecting *open* from the menu.

The lower left corner of Figure 3 contains an editor for changing the internal state of the source tray. The window was created by pointing at the tray and selecting *edit* from the menu. Similar editors are available for the move modules.

4 Defining new types of modules and ports.

Since Grif is implemented on top of the experimental language X2[8,9], X2 was the natural choice for expressing the module definitions. Another language could also be used for expressing the module definitions without much difficulty. X2 syntax is difficult to read for those not familiar with it, so we are using a Pascal-like syntax here.

Defining a simple module is not difficult. Figure 5 gives the definition of the flipflop module used in Figure 2. A type is declared to represent the module, three processes are defined to perform the actions, and a function is declared to create a new module. The major difficulties involve synchronizing the processes correctly.

Defining new types of ports is also easy. More time may be spend designing the icons to represent the type than writing the code. The boolean port type is defined in Figure 6.

5 Implementation

As mentioned earlier, Grif is implemented on top of X2. X2 is an object-oriented language that can best be described as a cross between CLU[5] and Smalltalk[2]. Grif would be very difficult to implement in many languages because the implementation makes heavy use of procedure types, process types, and parameterized types. The powerful abstraction tools that are in X2 allowed Grif to be built in a short time(approximately one man-month.)

```
{Declare type to represent module}
type flipflop=^record
 D:port(bool);  {Port is a parameterized type.}
 S:port(control); R:port(control); T:port(control);
 update:notifier; {Used to notify display that state has changed.}
end;

procedure Taction(f:flipflop); {Action to toggle flip-flop}
var b:control;
begin
 repeat
  b:=readwait(f.T);
  send(f.D, not readnowait(f.D)); {Send a message to f.D.}
  signal(f.update); {Notify display that state has changed.}
 until false
end; {Taction}

procedure Raction(f:flipflop); {Action to reset flip-flop.}
var b:control;
begin
 repeat b:=readwait(f.R); send(f.D,false); signal(f.update); until false;
end; {Raction}

procedure Saction(f:flipflop); {Action to set flip-flop.}
var b:control;
begin
 repeat b:=readwait(f.S); send(f.D,true); signal(f.update); until false;
end; {Saction}

function newff: gmodule(flipflop); {Create new module.}
var mg: gmodule(flipflop); b: flipflop;
begin
 {initialize fields}
 new(b); b.D:=newport; b.R:=newport; b.T:=newport;
 b.S:=newport; b.update:=newnotifier;
 {start up processes}
 newprocess(Saction,b); newprocess(Raction,b); newprocess(Taction,b);
 {Supply information on how to display module.  flipflopform is a global variable that
 contains a bitmap to represent the module. It was created using a form editor.}
 mg:=newmodule(b,newmenu,flipflopform,b.update);
 {The third parameter to addport indicates whether the port is an output port.
  (Addport is overloaded, that is, it refers to two different procedures.)}
 addport(mg,S,false,0@0); {0@0 gives port location}
 addport(mg,R,false,0@13);
 addport(mg,T,false,0@13);
 addport(mg,D,true,40@0);
 return(mg);
end; {newff}
```

Figure 5: Defining a new kind of module.

```
procedure addport(mg:gmodule(f); p:proc(f,name(port(bool)));
                  out:bool; {Is the port an output port?}
                  location:point);
where f is free; {Indicates f stands for any type.}
begin
 addport(mg,p, displayat ,"bool",out,location,true,boolportcontrol);
 {True in the argument list indicates that the port is to be redisplayed when
  the state changes.}
end; {addinport}

procedure boolportcontrol(p:port(bool); c:char);
{Defines user interaction with boolean ports.}
begin
 if c ="h" then send(p,true) else send(p,false)
end; {boolportcontrol}

procedure displayat(b:port(bool); p:point; clip: rect);
{Defines how to display a port of type bool. Boolon and booloff are two forms
 that define the symbols used to display boolean ports.}
var
 f:form;
begin
 if readnowait(b) then f:=boolon else f:=booloff
 copy(f,display,p); {Copy form f to display at point p.}
end; {displayat}
```

Figure 6: Defining a new kind of port.

A Grif program is represented as a list of module descriptions and a list of connection descriptions. Each module description has a bitmap and a location to display the bitmap on the screen. Each module description also has a module-specific menu, a list of port descriptions, and a list of update descriptions. Each port description contains a procedure to display the port, a location to display the port, the type of the port, whether the port is an input or output port, the address of the port, and a control function that defines the user interaction with that particular port. Each update description describes how to display part of the module when the state changes. Each update description contains a module, the location of the update relative to the module, how to display the update, and the notifier that indicates when the state has changed. When the module is created a process is automatically created that monitors the notifiers and performs the update when needed.

A connection description contains descriptions of the input port of the connection and the output port of the connection, and a list of the intermediate points through which the connection is to be routed. When a connection is made, the output port is thrown away and the input port is shared between the two modules.

A notifier is a counter. Each time a change is made the counter is incremented. Processes can wait until the counter exceeds a given value. A process normally waits until the counter exceeds the value the counter had when the last update was begun.

A port is a semaphore and a data value. When data is sent to a port, the data value is set and a signal is done on the semaphore. When data is read from a port, a wait is done on the semaphore and then the data is read. A read without waiting just returns the data value.

The operations of the editor are implemented with straightforward manipulations of the above data structures. The choice of data structures was not as straightforward. Grif was rewritten three times to arrive at the current version.

6 Conclusion

Grif has been connected to a Intelledex MicroSmooth 440 robot and a few simple programs have been written. More extensive experience is needed to develop a good set of primitive modules. Graphics does seem to be better than text for expressing the parallelism inherent in control systems. On the other hand, some concepts are expressed more concisely with text. For example, arithmetic expressions have a very concise text form. Grif's ability to intermix textual and graphical programming allows the programmer to use the more appropriate form. Grif allows the programmer to determine and modify the state of a program more easily than is possible in a system that only uses text. Grif shows promise of making robots easier to program, but more experience is needed before any firm conclusions can be drawn.

7 Acknowledgements

I wish to thank Intelledex for providing access to their robots and Jim Campbell of Intelledex for numerous valuable discussions.

8 References

[1] Ernest O. Doebelin. *Control System Principles and Design.* John Wiley and Sons, 1985.

[2] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983.

[3] Ron Goldman. *Design of an Interactive Manipulator Programming Environment.* UMI Research Press, Ann Arbor, Michigan, 1985.

[4] C. S. G. Lee, R. C. Gonzalez, K. S. Fu, eds. *Tutorial on Robotics.* IEEE, 1984.

[5] Barbara Liskov, et al. *CLU Reference Manual.* MIT Technical Report, MIT/LCS/TR-225, 1979.

[6] James L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, 1981.

[7] Georg Raeder. A Survey of Current Graphical Programming Techniques. *IEEE Computer Magazine,* August 1985, 11-25.

[8] David W. Sandberg. *The Design of the Programming Language X-2.* OSU Department of Computer Science Technical Report 85-60-1, 1985

[9] David W. Sandberg. *An Alternative to Subclassing.* To appear in Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon. September 1986.

[10] Wesley E. Snyder. *Industrial Robots: Computer Interfacing and Control.* Prentice-Hall, 1985.