Creating Graphical Languages in Garden

Steven P. Reiss
Scott Meyers

Computer Science Department
Brown University
Providence, Rhode Island 02912

Technical Report No. CS–89–02

January 10, 1989

Creating Graphical Languages in Garden*

Steven P. Reiss and Scott Meyers

Department of Computer Science
Brown University, Box 1910
Providence, RI 02912
spr@cs.brown.edu and sdm@cs.brown.edu

January 10, 1989

Abstract

Garden is an environment for conceptual programming. Conceptual programming means that programmers can create, view, and manipulate systems directly in terms of their conceptualizations of the systems. Such conceptualizations are frequently graphical in nature, and one of Garden's greatest strengths is the ease with which it allows programmers to define graphical languages. In this paper we provide a step-by-step construction of a graphical FSA language. The language can be created in a matter of minutes, and does not require any use of the keyboard.

Introduction

One approach to improving the software development process is to allow programmers to work directly with their ideas of how a system should operate. Programmers should be able to design, code, debug, and maintain their systems using the conceptual models that naturally arise during the creation of the systems. A conceptual programming environment is one that allows programmers to formally define and use their concepts directly and consistently throughout the software lifecycle.

Garden is a workstation-based software development environment that provides a number of features that facilitate conceptual programming. These features include:

- An object-oriented basis supporting multiple inheritance that includes both programs and data;
- A permanent, shared storage facility that supports nested transactions, multiple concurrent users, and multiple concurrent versions;

*Support for this research was provided by the NSF under grant DCR 8605567; by DARPA under contract N00014-83-K-0146, ARPA order 6320; and by the Digital Equipment Corporation under agreement 393.

- Automatic graphical display of complex programs and data. Such displays are dynamic and interactive, and can be used as a foundation for program visualization and visual programming;
- Multiple threads of control for parallel processing;
- Support for automatic dependency storage and propagation;
- Multiple, nested name spaces for segmenting large systems;
- Optional, strong type checking;
- A variety of window-based textual and graphical editors for objects;
- Execution by interpretation, by compiling into a C program, or directly from C code.

This paper does not focus on these features; they have been described elsewhere. Instead, it is devoted to a detailed demonstration of how a graphical language is created in Garden. For readers interested in information on other aspects of Garden, general overviews can be found in [Rei87c, RGR86, Rei87a, Rei86a]; its object-oriented basis is detailed in [Rei87b, Rei86b]; its mechanism for automatically displaying objects and for maintaining consistency between objects and their graphical images is described in [RP87]; and more examples of graphical language definition (but with fewer details) can be found in [Rei86c]. A reference and programmer's manual for Garden is also available from the authors.

Creation of an FSA Language

Finite State Automata (FSAs) have many uses in programming, including user interface design and control logic. Let us suppose that we wish to create a new graphical programming language, one that allows us to create FSA programs by drawing and editing the customary diagrams, such as the one shown in figure 1.

Let us further suppose that we have already defined the data structures and semantics of FSAs as shown in appendix A, which shows the definitions in Garden's Lisp-like textual language. In those definitions, an object of type *Fsa* consists of a list of states (the states of the FSA) and a current state (the state the FSA is currently in). An object of type *State* consists of an identifier, a list of arcs going out of the state, and flags that indicate whether the state is a start state or an accepting state. An *Arc* object consists of an object to match in order to enable the arc, a state to move to if the match is successful, and an action to be taken if the arc is followed.

Finally, let us suppose that we have textually defined an example FSA object f as shown in appendix B, which is equivalent to the FSA depicted in figure 1. In short, let us assume that we have already done everything we would normally do in a textual programming language if we were to be working with FSAs.

To make a graphical language out of what we've already got, we need do nothing at all, because Garden provides a default display format for all objects. The default display for f is shown in

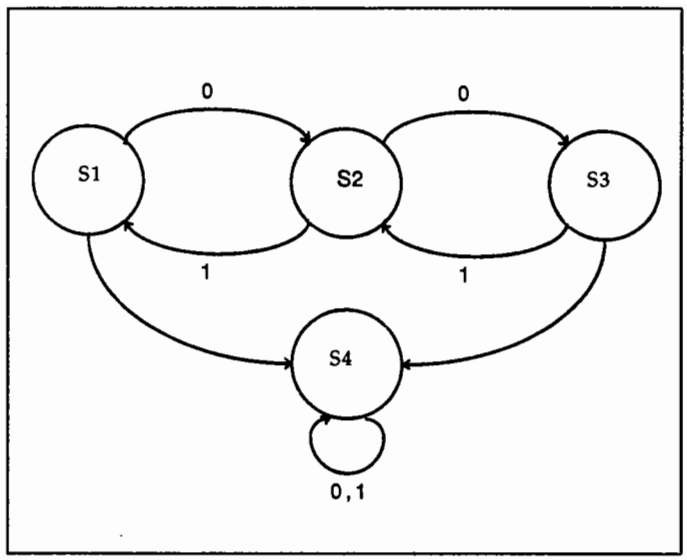

Figure 1: A Customary FSA Diagram

figure 2. Unfortunately, this doesn't look much like our conceptualization of an FSA, so we will need to define new graphical representations for the object types *Fsa, State*, and *Arc*.

Garden's type editor is called Thyme, and invoking Thyme on the type *Fsa* and then asking to see the default graphical representation results in the display shown in figure 3. The Thyme display is divided into five main areas;

- The window for editing **Super Types** is located in the upper left part of the display. Here supertypes of the current type can be added and removed. The list of current supertypes is also displayed here, but *Fsa* has no supertypes in this example.

- At the middle left of the display is the window for editing **Data Fields**, which are the instance variables for the type. For *Fsa*, two data fields are listed: *states* and *current*.

- The **Constant Fields** window at the lower left of the display is used to edit type variables, i.e., those variables which are constant for all objects of a type. The most common use for such variables is to define methods (operations) for the type. In this example, there are two such fields, *Evaluate* and *Step*. Both are lambda expressions (methods). *Evaluate* defines how to evaluate an FSA, and *Step* defines how to take a single step in an FSA.

- The window surrounded by the large grey border is where the graphical representation for the type is displayed. In this example, Thyme is showing the default representation for an FSA, which is a record-like format giving the name of the type and the values of its data fields. This record-like format is the default representation for most types in Garden.

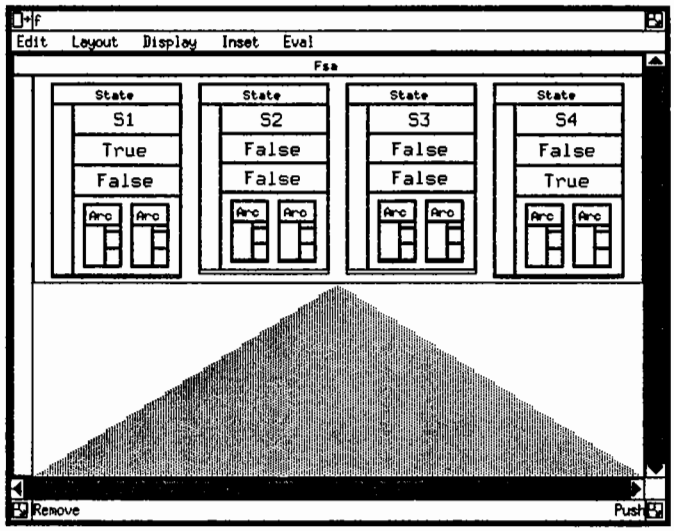

Figure 2: Default Display for f

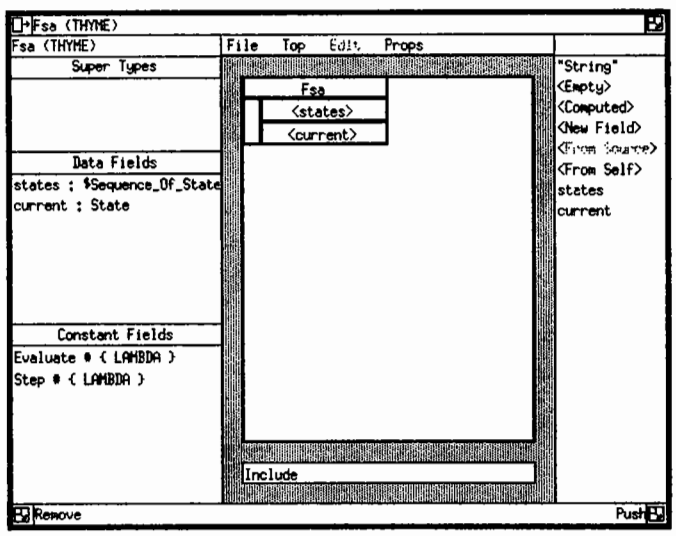

Figure 3: Thyme Display for *Fsa* Showing Default Graphical Representation

- The window to the right of the graphics window is a menu of contents for fields in the graphical display. For example, the choice **String** is used to put constant strings in the display, **Empty** is used for blank fields, **Computed** allows user-defined functions to be called to produce an object that should be displayed in a field, etc. The use of this menu window will become clearer as we edit the types used in an FSA.

Our concept of an FSA is one of a graph-like structure, so we will need to replace the default representation produced by Thyme with a representation that more accurately reflects our conceptualization. We would like to view an FSA as a graph showing the structure of the automaton, plus a field telling us the current state of the FSA.

We begin by choosing a command under the **File** menu to clear the existing graphics, then by choosing the option **Tiled** under the **Top** menu to define our graphical representation for *Fsa* as a Tiled object. There are four basic kinds of graphical representations in Garden – Tilings, Layouts, Boxes, and Arcs – and we shall use each in our development of a graphical FSA language[1]. Tiled objects are used for representations consisting of a set of nonoverlapping rectangular fields.

We use the mouse to create two adjacent tiles, a smaller one on top to contain the name of the current state, and a larger one below to contain the FSA itself. The resulting display is shown in figure 4. To associate the current state with the top tile, we click on the tile to choose it, then click on **current** in the menu of possible field contents; Thyme automatically centers the tile's contents inside the tile. To tell Thyme to display the FSA as a graph in the lower tile, we click on the lower tile to choose it, then we choose the **Layout** option under the **Edit** menu to make the tile a layout object. In general, layout objects are used to display graph-like structures. Finally, we choose <**states**> from the list of field contents as the contents of the layout, since we want to see a graph containing the states of the FSA. This allows us to see the arcs, too, since our definition of an FSA includes a list of arcs for each state. The final Thyme display for *Fsa* is shown in figure 5.

With this new definition of how to display an FSA, Garden now displays *f* as shown in figure 6. The result is not substantially different from figure 2, but it has changed in two ways. First, the area allocated for the current state, shown in both figures as a grey triangle[2], is now smaller than the area allocated for the graph; this is just as we defined it in figure 4. Second, the current state is now located above the graph, whereas in figure 2 it was located below the graph.

We next turn our attention to the graphical representation for *State* objects; the default display for such objects is shown in figure 7. Again, this fails to match our conception of how a state should appear – we think of states as circular nodes in a graph. To instruct Thyme to change the graphical representation of a state to match our conception, we clear the default graphics, then choose the **Box** option under the **Top** menu to make our representation for *State* a box object. Box objects are simple shapes like boxes and circles, but Garden supports a wide variety of boxes, including variations in the shape of the box, the line style used to display the box, and the font used if text is to be displayed inside the box. Figure 8 shows the options available when defining a box, and

[1]These correspond to the four basic object *flavors* in Gelo, the underlying graphical layout package used by Garden. Details on Gelo can be found in [RP87].

[2]The grey triangle signifies that the value of the current state is *null*. This is Garden's default graphical format for the special value *null*, but it could easily be redefined, just as we redefined the graphical display for an *Fsa* object.

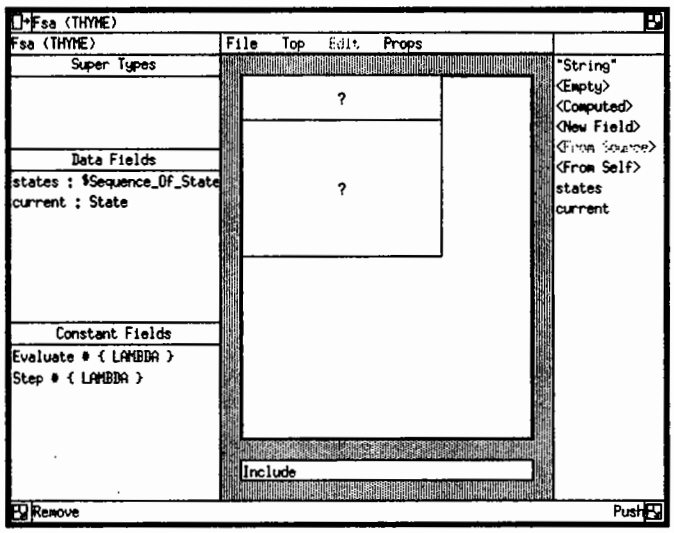

Figure 4: Thyme Display for *Fsa* after Creating a Tiling

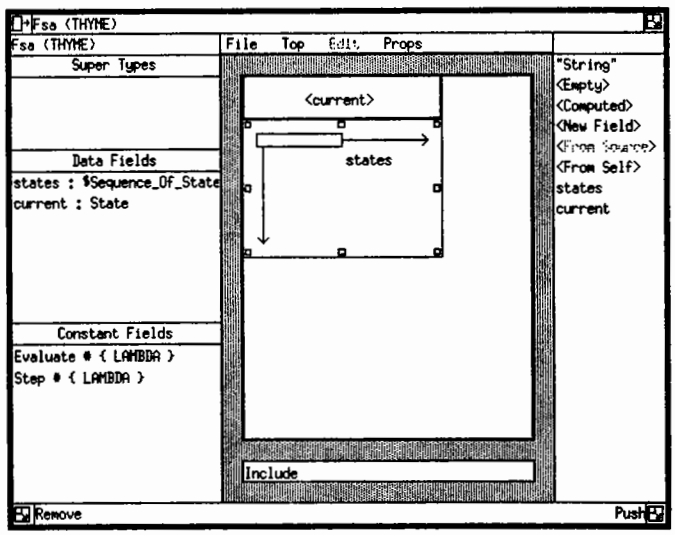

Figure 5: Final Thyme Display for *Fsa*

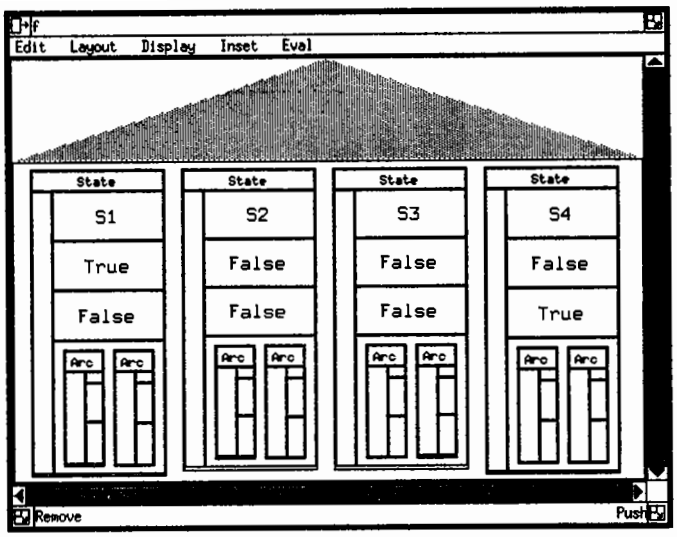

Figure 6: Display of f After Editing *Fsa*

indicates which ones were chosen in our example to get a *State* object to be drawn as a circle.

Having indicated that *State*s are to be drawn as circular boxes, we indicate that the name (*id* field) of the state should be drawn in the circle by clicking on the circle, then clicking on **id** in the menu of field contents; Thyme automatically centers the name in the box.

When we draw *State* objects in an FSA, we wish to see the arcs coming out of the states, too. As we have currently defined things, however, Garden will only display the name of each state inside the appropriate circle; the arcs will not be displayed. What we want to do is to add the arcs to the objects to be placed in the layout tile of the FSA, even though the arcs are structurally part of the FSA's states. To accomplish this, we click on the **Include** box in the graphical part of Thyme, then click on **arcs** in the menu of field choices. This tells Thyme to include the state's arcs in the objects to be displayed – to behave as if they were not part of a state, but instead part of the layout in which the state will find itself. The final Thyme display for *State* objects is shown in figure 9.

Figure 10 shows how Garden displays f after editing the graphical representation of *State* objects. This is quite different from what we had earlier, but at first glance it doesn't seem that we are making a lot of headway toward our goal of viewing FSAs as graphs. A closer look reveals that we are making some progress, however. We can see that we have eight arc objects in the view, which is the number of arcs in f, and that each arc object contains a state depicted as a circle[3]

[3]In this paper, all the "circles" appear as ovals because the aspect ratio of the surrounding windows isn't 1:1. Garden automatically scales drawings independently in X and Y so as to use the entire window area available to it. This behavior can be overridden by the user so that objects that are defined to be circles will always be drawn as true circles.

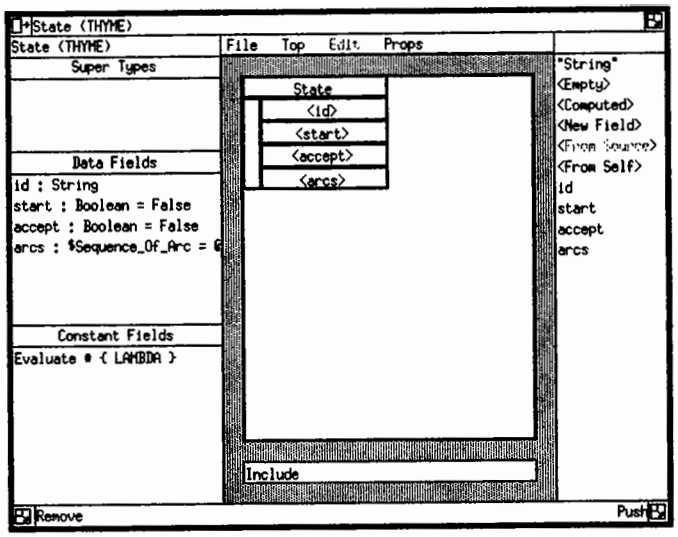

Figure 7: Thyme Display for *State* Showing Default Graphical Representation

with the name of a state inside. In addition, if we emphasize one of the small rectangles next to the arc objects (see figure 11), we see that it is actually a *State* object drawn in the way we expect; it is seen as a rectangle in figure 10 only because it is too small to display as a circle[4].

Our problem is that the FSAs arcs aren't being drawn as we're used to seeing them; they're being drawn in the same record-style format that Garden assigns as its default for most types. To get Garden to draw arcs more to our liking, we invoke Thyme on the type *Arc* and choose the **Arc** option under the **Top** menu. Thyme then offers the display shown in figure 12. Arc objects have three fields: the source of the arc (the field on the left), the sink of the arc (the field on the right), and the label to be placed on the arc (the field in the middle).

For our arcs to be drawn correctly, we need the source of the arc to be the object containing the arc (the state it comes from), the sink of the arc to be the arc's *next* field, and the label of the arc to be its *match* field. We assign these by clicking on the each field in turn to select it, then by choosing the appropriate item in the field contents menu. For the arc source we choose **<From Source>**; for the sink we choose **next**; and for the label we choose **match**.

Garden can draw arcs in a great number of formats, with the user given control over where the arc source and sink attach to their respective objects, how the arc line is drawn, what kind of arrowhead it should have, and where the label should be located. Figure 13 shows the options available when defining an arc, and indicates which ones were chosen in our example to get an

[4]Garden automatically elides information that is too small to display. In this case, an oval of the correct size looks just like a rectangle, and the state name is elided. The graphical editor in Garden allows individual objects to be emphasized (enlarged) or deemphasized (reduced) under user control by simple menu choices and mouse clicks, however, so it is easy to emphasize the small rectangles in figure 10 and verify that they are really *State* objects.

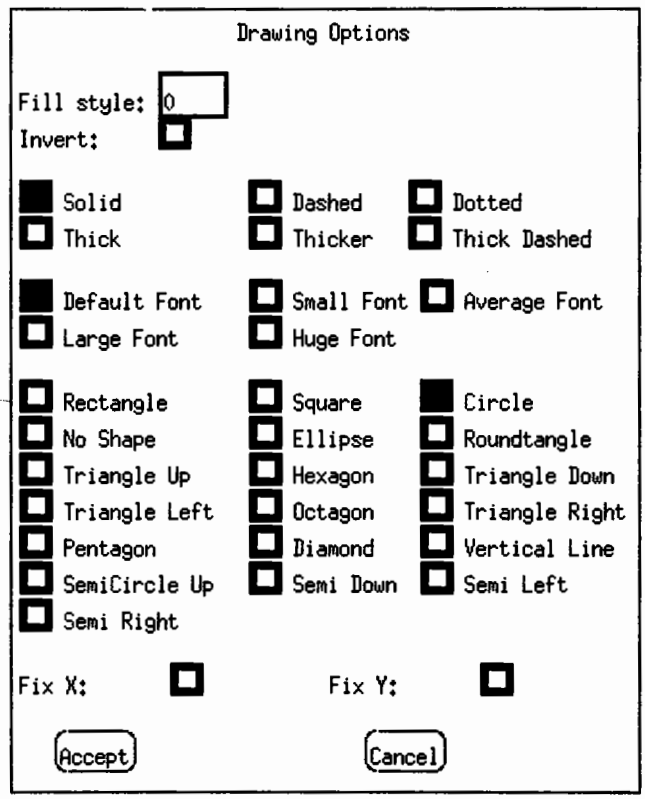

Figure 8: Thyme Options for Box Objects

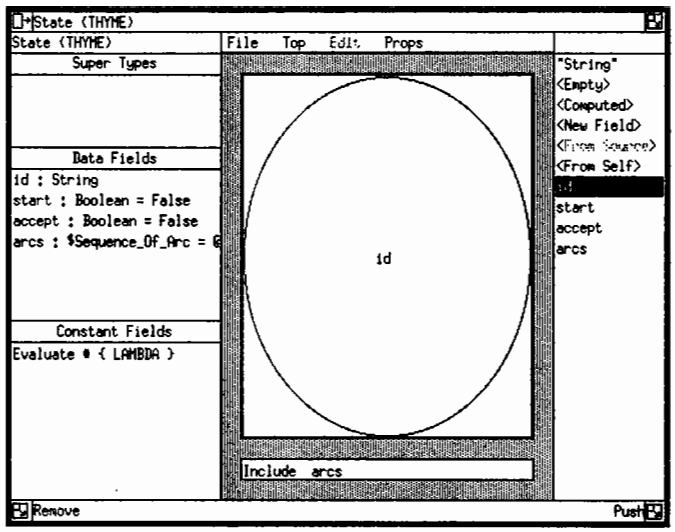

Figure 9: Final Thyme Display for *State*

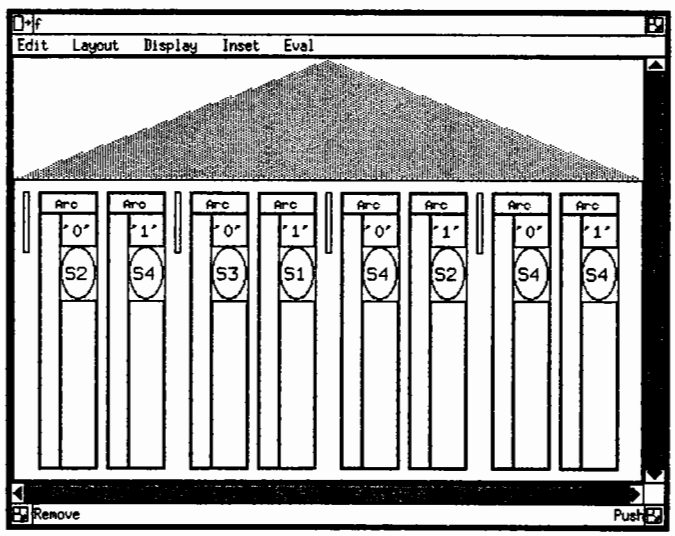

Figure 10: Display of f after Editing *State*

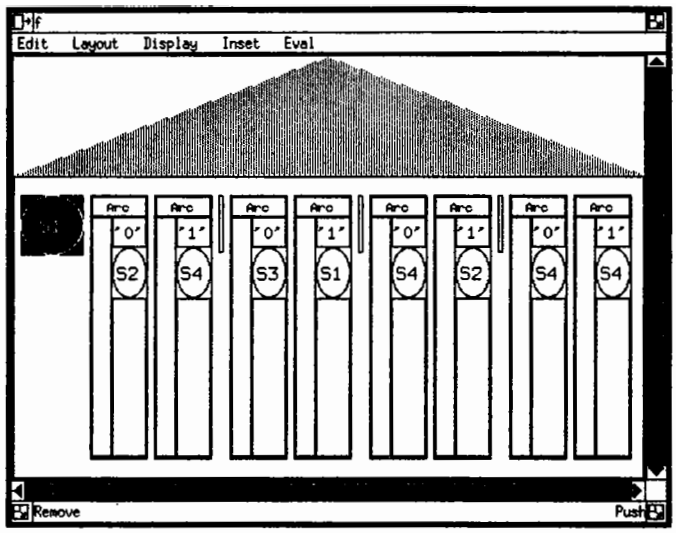

Figure 11: View of f with a State Emphasized

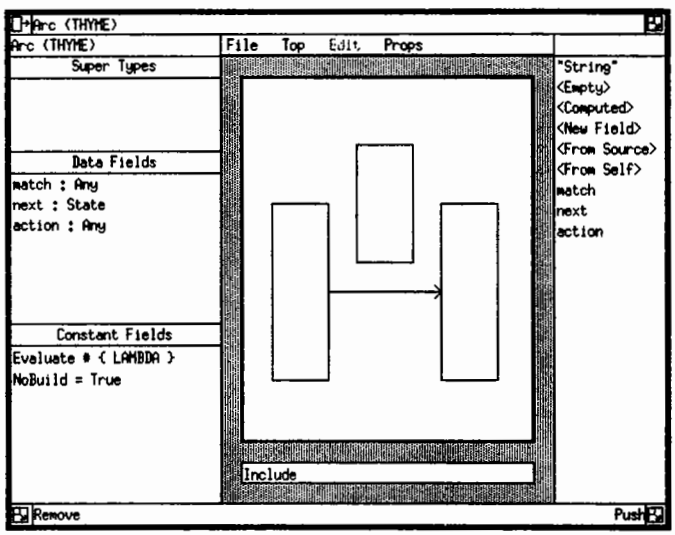

Figure 12: Thyme Display for *Arc* Showing General Arc Representation

Arc object to be drawn as a solid single-headed arrow with its attachment points automatically computed by Garden. The resulting Thyme display is shown in figure 14, and the effect this has on f's display is shown in figure 15.

This is much closer to our conceptualization – the diagram is unmistakably an FSA. Still, the layout could be improved: it's not entirely obvious that there is an arc from S1 to S4. Fortunately, Garden gives us some control over the layout heuristics used when drawing Layout objects. If we go back and edit the layout tile of the graphical representation for type *Fsa*, we are presented with the set of options shown in figure 16. Setting the options as shown there gives us the drawing for f shown in figure 17, which is our final program.

It is important to remember that the result of our efforts is not just a way of viewing the program f, but is instead a complete *graphical programming language*. We can now create new FSA programs by drawing nodes and arcs and connecting them together[5]. We can edit existing *Fsa* objects graphically, even if they were defined textually. We can also execute our *Fsa* objects and watch the execution graphically, since Garden automatically highlights the object currently being evaluated. Finally, we can do all this in a graphical language that was defined in about 15 minutes and that used only a mouse – no typing at the keyboard was required[6].

Of course, we could embellish our graphical FSA language if we wanted to. For example, it might be nice to graphically distinguish start states and accepting states, or we might want to eliminate from our display the textual field showing the current state of the FSA, replacing it with some kind of graphical mechanism to indicate which state is current. Garden makes these sorts of customizations possible through the use of *alternatives*. Thyme allows the user to specify any number of alternative graphical representations for an object, associating with each one a boolean function that determines which alternative should be used at any given time. For example, figure 18 shows f with start states drawn as circles with thick borders and accepting states drawn as filled circles.

Discussion

As we have seen, Garden makes it easy to create graphical languages. Garden is also flexible enough to allow the creation of a wide variety of such languages. Among the other languages already defined within Garden are those for programming in Petri Nets[Rei87c], Flowcharts[Rei86c], Data Flow Diagrams[Rei86c], and a CSP-like parallel language[Rei87b].

As always, there is much room for improvement. The graphical interface is weak in three respects. First, the program layouts in Garden are produced completely automatically and are often not aesthetically pleasing; there is no way for a programmer to "tweak" the pictures to make

[5]We do not mean that we can make general drawings of circles and arrows and then have Garden parse the resulting picture into a program composed of nodes and arcs. Instead, we mean that we can create objects of type *Fsa* by drawing pictures composed of *State* objects and *Arc* objects. The more general problem of graphical parsing is a current research topic at Brown.

[6]Technically, we did use the keyboard to specify to Thyme which objects we wanted to edit each time we invoked the type editor. Garden has ways to do this using only the mouse, but it is often easier to simply specify objects by name. For the actual editing operations discussed in this paper, however, everything was done with the mouse alone.

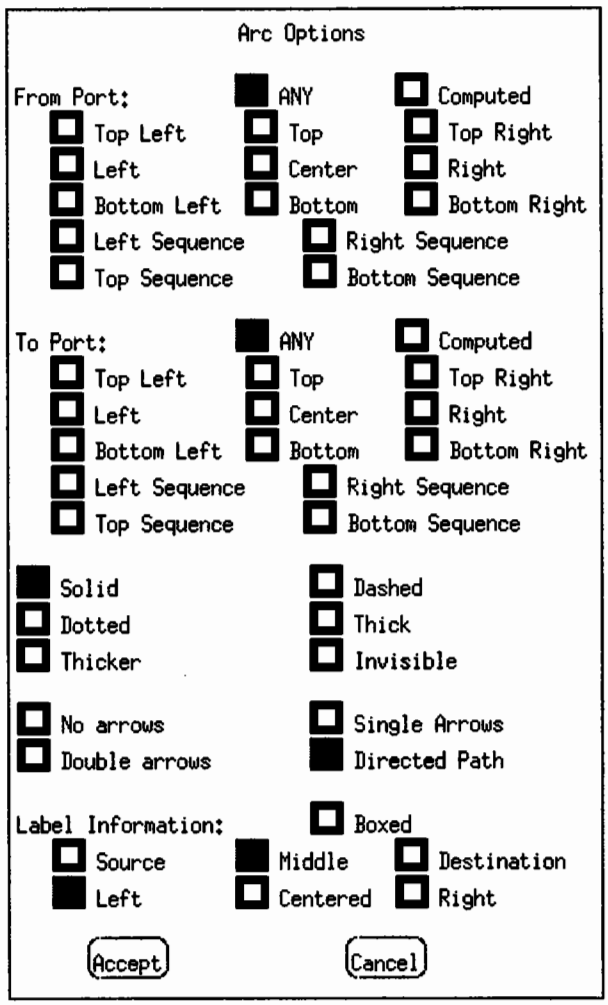

Figure 13: Thyme Options for Arc Objects

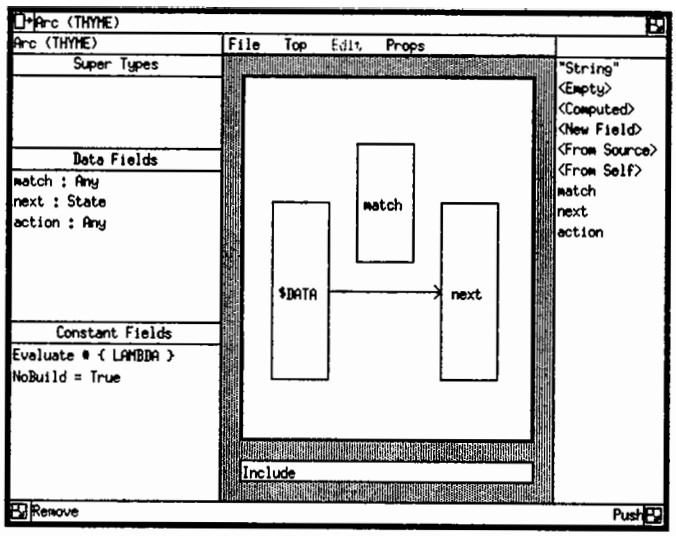

Figure 14: Final Thyme Display for *Arc*

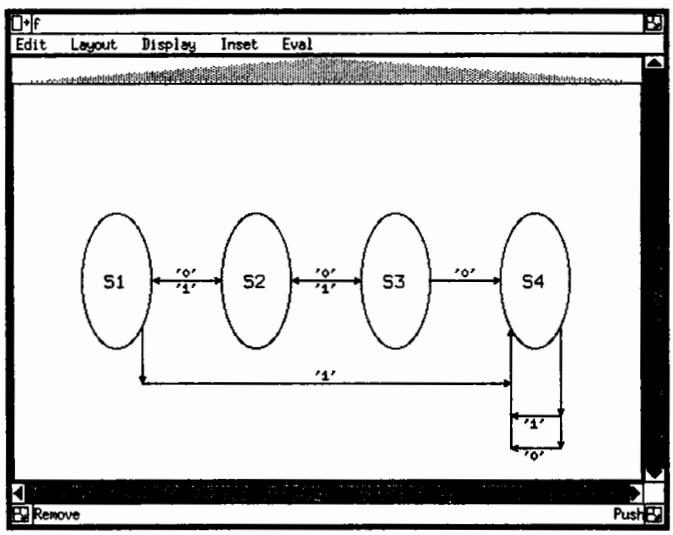

Figure 15: Display of *f* After Editing *Arc*

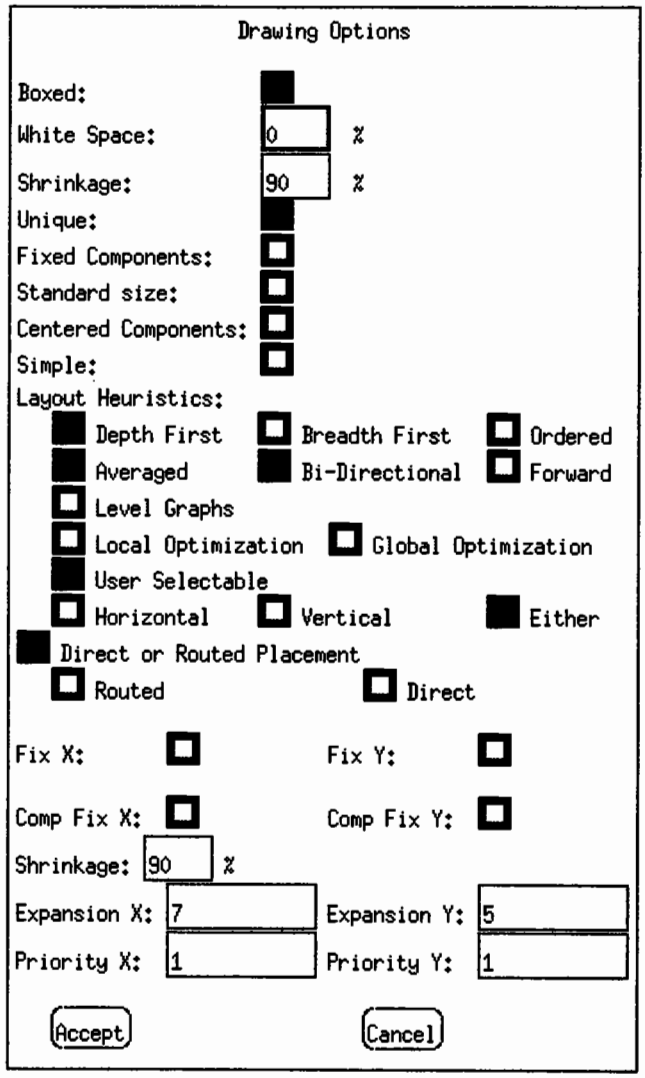

Figure 16: Thyme Options for Layout Objects

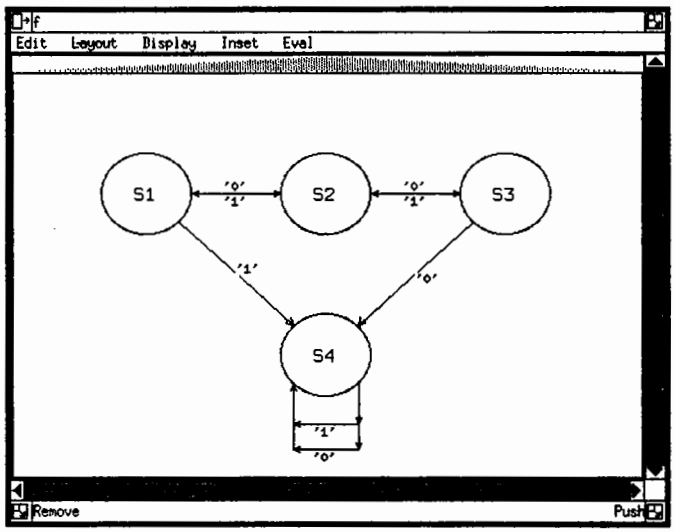

Figure 17: Display of f After Setting Layout Options

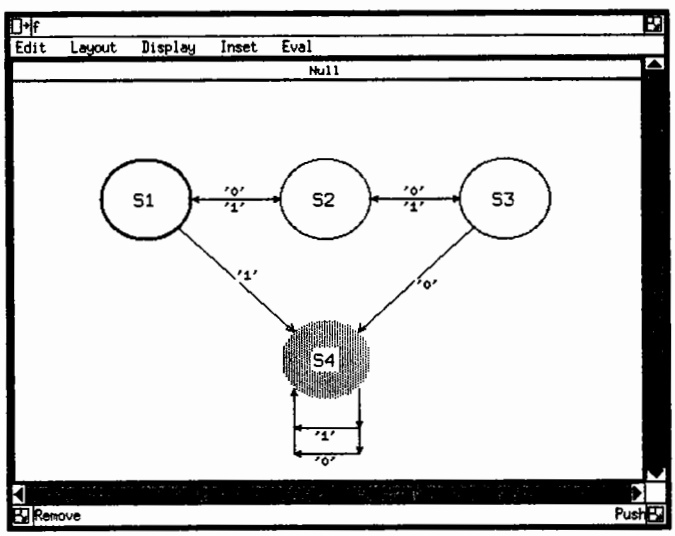

Figure 18: Display of f Using Alternative Representations for *State* Objects

them look just the way s/he wants. For example, our FSA f has overlapping arcs between nodes in figure 15, but our original conceptualization in figure 1 doesn't. To solve this problem, we are currently exploring the possibility of using better heuristics for layouts and of finding ways to let programmers have some direct control over layouts. Second, the graphical program editor is essentally a structured editor for the underlying object. We would prefer, in many cases, that this editor be a general-purpose drawing package that allows users to simply draw the program; we would then parse the resulting picture. Finally, both Thyme and the graphical program editor are designed to work best on hierarchical drawings; programs with graphical relationships between different parts of the layout hierarchy can be difficult or impossible to specify. We intend to redesign the underlying Garden display mechanisms so that they accommodate such languages more easily.

A Textual Definition of FSA Semantics

```
--
-- Finite State Automata definitions
--

@FSA@START

(SCOPE_SET @FSA)

--
-- Types
--

(TYPE_FORWARD "Fsa")
(TYPE_FORWARD "State")
(TYPE_FORWARD "Arc")

(TYPE_DEFINE "Fsa{states:#State,current:State}")
(TYPE_DEFINE "State{id:String,start:Boolean,accept:Boolean,arcs:#Arc}")
(TYPE_DEFINE "Arc{match:?,next:State,action:?}")

--
-- Special symbols
--

(SETQ START '@FSA@START)
(SETQ END '@FSA@END)

--
-- Evaluation Routines
--

(FORWARD Fsa_EVAL Fsa_STEP State_EVAL Arc_EVAL FIND_START)

(DE Fsa_EVAL <f s>
    {* <l>
(SETQ l Null)
(COND (EQ (TYPE_OF s) String)
  (SETQ l (STRING_TO_CHARS s))
      (INQ_SUBTYPE_OF (TYPE_OF s) Sequence)
  (SETQ l s)
      (SETQ l (LIST_NEW s)))
(FIELD_SET (FIND_START f) f 'current)
(LISTLOOP l '{* LAMBDA <v> ('Step f v) *})
(COND (NULL ('current f)) False
      ('accept ('current f)))
```

```
     *}
)
(TYPE_CONSTANT Fsa 'Evaluate Fsa_EVAL True)

(DE Fsa_STEP <f v>
    (COND (NULL ('current f)) Null
  (FIELD_SET (EVAL ('current f) v) f 'current))
)
(TYPE_CONSTANT Fsa 'Step Fsa_STEP True)

(DE State_EVAL <s v>
    (LISTFIND ('arcs s) '{* LAMBDA <a> (EVAL a v) *})
)
(TYPE_CONSTANT State 'Evaluate State_EVAL True)

(DE Arc_EVAL <a v>
    (COND (NEQ v ('match a)) Null
  (SEQ (COND (NOT_NULL ('action a)) (EVAL ('action a) v))
       ('next a)))
)
(TYPE_CONSTANT Arc 'Evaluate Arc_EVAL True)

(DE FIND_START <f>
    (LISTFIND ('states f) '{* LAMBDA <s> (COND ('start s) s Null) *})
)

--
-- Input/Output Definitions
--

(SET_FIELD_PROP FIELD_NestIgnore State 'arcs True)

(SET_FIELD_PROP FIELD_InIgnore Fsa 'states True)
(SET_FIELD_PROP FIELD_InIgnore Fsa 'current True)
(SET_FIELD_PROP FIELD_InIgnore State 'arcs True)
(SET_FIELD_PROP FIELD_InQuoted Arc 'action True)

(TYPE_CONSTANT Arc 'NoBuild True False)
```

B Textual Definition of an Example FSA

```
(SETQ s1: (BUILD State "S1" True False))
(SETQ s2: (BUILD State "S2" False False))
(SETQ s3: (BUILD State "S3" False False))
(SETQ s4: (BUILD State "S4" False True))

(SETQ a1: (BUILD Arc '0' s2 DPRINT))
(SETQ a2: (BUILD Arc '1' s4 DPRINT))
(FIELD_SET (LIST_NEW a1 a2) s1 'arcs)

(SETQ a3: (BUILD Arc '0' s3 DPRINT))
(SETQ a4: (BUILD Arc '1' s1 DPRINT))
(FIELD_SET (LIST_NEW a3 a4) s2 'arcs)

(SETQ a5: (BUILD Arc '0' s4 DPRINT))
(SETQ a6: (BUILD Arc '1' s2 DPRINT))
(FIELD_SET (LIST_NEW a5 a6) s3 'arcs)

(SETQ a7: (BUILD Arc '0' s4 DPRINT))
(SETQ a8: (BUILD Arc '1' s4 DPRINT))
(FIELD_SET (LIST_NEW a7 a8) s4 'arcs)

(SETQ f: (BUILD Fsa (LIST_NEW s1 s2 s3 s4)))
```

References

[Rei86a] Steven P. Reiss. Garden tools: Support for graphical programming. In Reidar Conradi, Tor M. Didriksen, and Dag H. Wanvik, editors, *Lecture Notes in Computer Science 244*, pages 59–72. Springer-Verlag, June 1986. Proceedings of an International Workshop on Advanced Programming Environments, held in Trondheim, Norway. Also available as Brown University Computer Science Department Technical Report No. CS-86-18, April 1986.

[Rei86b] Steven P. Reiss. An object-oriented framework for graphical programming. Technical Report CS-86-17, Brown University, March 1986.

[Rei86c] Steven P. Reiss. Visual languages and the garden system. In P. Gorny and M. J. Tauber, editors, *Lecture Notes in Computer Science 282*, pages 178–198. Springer-Verlag, May 1986. Proceedings of the 5th Interdisciplinary Workshop in Informatics and Psychology, held in Schärding, Austria. Also available as Brown University Computer Science Department Technical Report No. CS-86-16, September 1986.

[Rei87a] Steven P. Reiss. A conceptual programming environment. In *Proceedings of the 9th International Conference on Software Engineering*, pages 225–235. Computer Society Press of the IEEE, April 1987. Held in Monterey, California, USA. Also available as Brown University Computer Science Department Technical Report No. CS-86-20, August 1986.

[Rei87b] Steven P. Reiss. An object-oriented framework for conceptual programming. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 189–218. The MIT Press, 1987.

[Rei87c] Steven P. Reiss. Working in the garden environment for conceptual programming. *IEEE Software*, pages 16–27, November 1987.

[RGR86] Steven P. Reiss, Eric J. Golin, and Robert V. Rubin. Prototyping visual languages with the garden system. In *1986 IEEE Computer Society Workshop on Visual Languages*, pages 81–90. IEEE Computer Society, June 1986. Held in Dallas, Texas, USA.

[RP87] Steven P. Reiss and Joe N. Pato. Displaying program and data structures. In *Proceedings of the 20th Hawaii International Conference on System Sciences*, January 1987. Held in Honolulu, Hawaii, USA. Also available as Brown University Computer Science Department Technical Report No. CS-86-19, April 1986.