Annotating Software With Arbitrary Performance Characteristics

Scott Meyers

Computer Science Department
Brown University
Providence, Rhode Island 02912

Technical Report No. CS–89–01

January 10, 1989

Annotating Software With Arbitrary Performance Characteristics

Scott Meyers

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

January 10, 1989

Abstract

Characterizations of software performance are usually restricted to discussions of the software's size and speed, but these two criteria fail to take into account other important aspects of software performance, such as reliability, portability, and robustness. In this paper, we present a methodology for annotating software with arbitrary performance characteristics, and we describe how such annotations might be used as the basis for a system that could automatically select optimal implementations for software abstractions. The selection of optimal implementations requires that fundamentally incomparable criteria (such as speed and portability) be unified, and we develop a convention for achieving this unification. We also discuss how the idea of using arbitrary performance characterizations can be applied to other domains, including hardware design and general-purpose planning.

1 Introduction

Software performance is typically measured in terms of only two criteria: size and speed. These metrics are usually expressed as worst-case asymptotic limits ("Big Oh") or as empirical results obtained by running a given problem on a given machine with a given set of resources. Certainly the time and memory requirements of a program are important, but reliance on these measures alone reflects a bias in favor of conserving hardware resources and implies that other aspects of software performance are of little interest.

Such is not always the case. Consider, for example, the selection of software to go aboard a satellite. Given a choice of programs to perform a given task, each of

which meets the speed and memory limitations of the satellite, the ultimate selection is likely to be made more on the basis of reliability or robustness, rather than size or speed. In fact, given the importance of avoiding software malfunctions in this case, it may actually be preferable to choose the most reliable program, even if that means redesigning the hardware to incorporate more resources for the software.

In this paper we propose a methodology for annotating software entities with arbitrary performance characteristics. These entities will typically be implementations for abstractions used in programs, e.g., data abstractions, procedural abstractions, iterative abstractions, etc. We also discuss how this methodology could serve as the basis for a system that automatically chooses the most appropriate software entity to perform a specified task, and we describe some of the important issues that such a system would have to address.

Previous work in the area of automatically selecting implementations for software abstractions includes that by Low [Low76] and by Rovner [Rov79], both of whom worked on systems to automatically select implementations for associative data structures in the LEAP [FR69] sublanguage of SAIL [Smi76]. Their work was subject to a host of restrictions, however, including the facts that (1) only data abstractions that were primitives in the language were considered, (2) each abstract variable could have only one implementation in a program, and (3) the only criterion for selecting "good" implementations was that of minimizing the estimated space-time product. The methodology introduced here suffers from none of these shortcomings.

Sherman's language Paragon [She83] allows implementations for abstractions to be annotated with arbitrary performance information and allows programmers to specify arbitrary strategies for selecting implementations, but offers no methodology for using these capabilities. The declarative approach proposed here is substantially easier to use than that provided by Paragon, and it makes no special demands on a compiler. In contrast, Sherman's approach is entirely procedural, and it requires that a complete language interpreter be available during compilation.

2 Evaluations

The methodology we propose is based on software *evaluations*. An evaluation allows a programmer to annotate an implementation of an abstraction with information on its performance using whatever metrics make sense to the programmer. Each annotation consists of a *criterion* and a *value*. A criterion identifies which aspect of the software's performance is being characterized (e.g., its speed, its size, its reliability, etc.), and the value identifies how well it rates for that aspect. Table 1 gives examples of some possible criteria and values.

Evaluations can exist at many levels of implementation for an abstraction. For example, consider the abstract data type *set*. A set might be implemented as either

Criterion	Value	Criterion	Value
Asymptotic Runtime	$\log n$	Asymptotic Memory	2^n
Exception Handling	Poor	Maintainability	Excellent
Author	Meyers	Reported Bugs	11
Verified	True	Year Written	1982

Table 1: Sample Criteria and Values

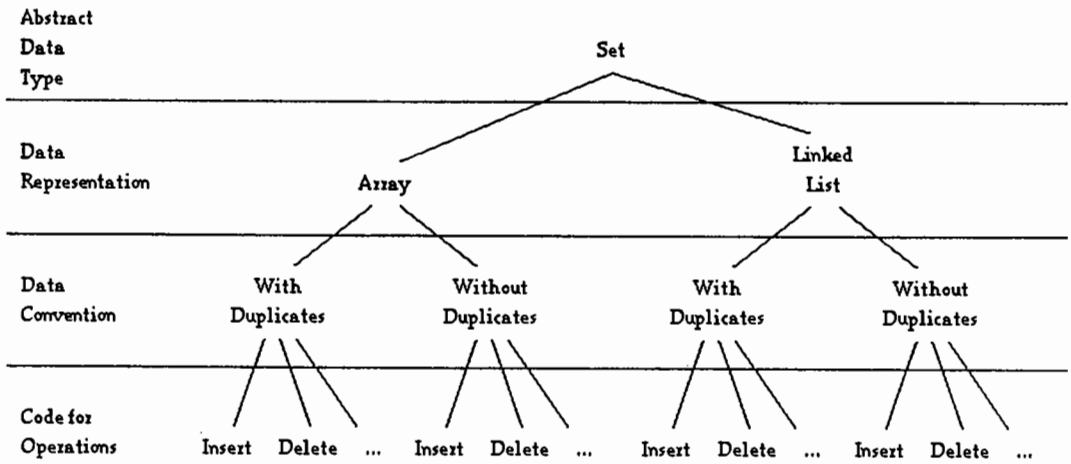

Figure 1: Implementation Hierarchy for *set*

an array or a linked list, and each of these representations might or might not store dupicate entries. This gives rise to a three-level hierarchy of implementations, as shown in Figure 1. The top level of the hierarchy is the data representation (array versus linked list); the middle level is the data convention (duplicates versus no duplicates); the bottom level is the code for the individual operations on the data type (e.g., *insert*, *delete*, etc.). Each of these levels could have evaluation information associated with it. For example, an evaluation at the representation level might indicate whether the cardinality of the set was bounded, as it would be with an array implemenation; an evaluation at the convention level might indicate performance characteristics of certain sets of operations used together [1]; and an evaluation at the code level might specify memory requirements or exception-handling capabil-

[1]This is useful for describing the behavior of groups of operations that have been subject to amortized analysis, such as Tarjan's *makeset*, *find*, and *link* operations for the abstract data type *disjoint set* [Tar83].

ities for a particular code-level implementation of an abstract operation. For the remainder of this report, we limit our discussion to evaluations at the bottom of this kind of hierarchy, but the methodology described here can be extended to include evaluations at many levels.

Formally, an evaluation is a set of pairs $\{(C_1, V_1), \ldots, (C_n, V_n)\}$, where each C_i is a criterion and V_i is its value. Each criterion is associated with a particular type T; T defines the domain from which the values for V must be chosen.

The following operations must be defined for each evaluation type:

- The *min* and *max* functions choose the "smallest" and "biggest" value, respectively, from their arguments. Because these functions must be uniquely defined, evaluation type domains must be totally ordered with respect to these functions.

- The *ratio* function takes *min* or *max* and a sequence of values; it returns a sequence indicating the ratio of each value to the "smallest" or "biggest" of the values, respectively. That is, $ratio(Op, [v_1, \ldots, v_n])$, where Op is either *min* or *max*, returns $[r_1, \ldots, r_n]$, where $0 \leq r_i \leq 1$. Intuitively, the ratio r_i returned for a value v_i reveals "how small" or "how big" v_i is compared to the "smallest" or the "biggest" of the arguments to *ratio*. For positive real values, one might expect that

$$r_i = \begin{cases} \frac{v_i}{Op(v_1,\ldots,v_n)} & \text{if } Op = max \\ \frac{Op(v_1,\ldots,v_n)}{v_i} & \text{if } Op = min, \end{cases}$$

  but *ratio* is free to compute its result in any way that makes sense for the type of its arguments.

In addition, because the choice of the best implementation for an abstraction is often influenced by how that abstraction is used in a program, it is useful to define the following operators, which make it possible to reason about evaluations in the context of program flow graphs:

- The *sequence* operator, signified by "$\oplus$," is used to determine the value of a criterion for two operations executed one after the other. It does this in terms of the criterion values for each of the individual operations. If Op_1 and Op_2 are operations with criterion values $value(Op_1)$ and $value(Op_2)$, then $value([Op_1, Op_2]) = value(Op_1) \oplus value(Op_2)$.

- The *loop* operator, signified by "$*$," is used to determine the value of a criterion for an operation in a loop. If an operation Op is in a loop that is expected to be executed n times, then $value(Op^n) = n * value(Op)$.

Operation	Criterion		
	Reliability	Runtime	Meets Mil. Specs.
$Op_1 \oplus Op_2$	$Op_1 \cdot Op_2$	$max(Op_1, Op_2)$	$Op_1 \wedge Op_2$
$n * Op$	Op	$O(1)$ if n is $O(1)$ and Op is $O(1)$ Op if only n is $O(1)$ n if only Op is $O(1)$ $n \cdot Op$ otherwise	Op
$Op_1 \diamond Op_2$	$min(Op_1, Op_2)$	$max(Op_1, Op_2)$	$Op_1 \wedge Op_2$

Table 2: Sample Definitions for *Sequence*, *Loop*, and *Branch*

- The *branch* operator, signified by "$\diamond$," is used to determine the value of a criterion when one of two operations may be executed; this corresponds to an `if-then-else` construct. When either Op_1 or Op_2 will be executed, $value(Op_1 \text{ or } Op_2) = value(Op_1) \diamond value(Op_2)$.

Because it is cumbersome to always write $value(Op)$ to indicate the value of a criterion for a particular operation, we will usually refer simply to Op where the context makes the meaning clear. Hence, we will write "$Op_1 \oplus Op_2$" to refer to the value of the *sequence* operator, "$n * Op$" for the *loop* operator, $min(Op_1, \ldots, Op_n)$, etc.

None of these operators is particular about the order of its operands. *Sequence*, *loop*, and *branch* must be commutative; *min* and *max* are insensitive to the order of their arguments; and a permutation of the sequence given to *ratio* results in a corresponding permutation in its output.

This is not an exhaustive list of useful operations. Other operations may be defined, and these additional operators may be used in desirability formulas, described in the next section. It will frequently be convenient to define domain-specific operations, such as the arithmetic operators for numerical domains.

As an example, imagine an implementation of an abstraction that has the evaluation

$$\{ \text{(Reliability, .82), (Runtime, } O(n^2)\text{), (Meets Mil. Specs., True) } \},$$

where Reliability is of type *real* in the range [0.0, 1.0] and represents the probability of an operation executing correctly; Runtime is of type *Big Oh* (which has values such as $O(1), O(\log n), O(n), O(n \log n)$, etc.); and Meets Mil. Specs. has type *boolean*. *Sequence*, *loop*, and *branch* might be defined as shown in Table 2. These definitions reflect the intuitive meanings of program reliability, asymptotic runtime,

and adherence to military specifications. For Reliability, the *sequence* operator is defined to return the product of the reliabilities of its constituent operators, since the chance of two operations successively succeeding is the product of their individual chances of success; the *loop* operator's definition of exponentiation is analogous. The *branch* operator for Reliability is defined to return the minimum of the reliabilities of its arguments because a software system can surely be no more reliable than its least reliable operation. For Runtime, the definitions for *sequence*, *loop*, and *branch* reflect the usual *Big Oh* semantics. Finally, the Meets Mil. Specs. definitions correspond to the fact that a program as a whole meets military specifications if and only if each of its constituent operations does.

Coming up with definitions for *min* and *max* for this example is straightforward, but *ratio* is a bit more troublesome. Reliability is measured on a linear scale, so it is possible to use the simple numerical ratio presented earlier for that criterion. That still leaves the problem of finding ratios of values for Big Oh and boolean criteria. What is the ratio of an $O(n)$ algorithm to an $O(n \log n)$ algorithm? How "True" is "False?" An appropriate solution to the former problem ideally calls for an analysis of the expected input size and distribution, i.e., a good characterization of "n," while the latter problem can only be solved by adoption of a convention, e.g., that $ratio([True, False]) = [1, 0]$.

The difficulties that can arise in defining functions like *ratio* for types like Big Oh and boolean make clear that functions for evaluations may not always be easy to implement. Nonetheless, their problematic aspects mirror similar conditions in the real world. When software engineers are faced with choosing between a Quicksort and an Insertion sort, they don't simply choose the Quicksort outright, even though they know that its expected-case asymptotic runtime is superior to that of Insertion Sort. Instead, they first try to get some idea of the data that will need to be sorted. After all, when the number of elements is small enough, Quicksort itself may well end up calling a routine to perform an Insertion Sort! Coming up with reasonable conventions for the computation of functions like *ratio* under these conditions is quite challenging.

3 Desirability Formulas

Evaluations allow programmers to characterize certain aspects of software behavior; *desirability formulas* allow programmers to specify, in terms of evaluations, what behavior would characterize the most desirable concrete implementation of an abstract program. For example, desirability formulas allow programmers to indicate that the desired programs:

- Are as fast as possible.
- Have a minimum space-time product.

- Are as reliable as possible given that their runtime is no more than $O(n^2)$.
- Meet military specifications, are the fastest, and are the most maintainable, where maintainability is twice as important as speed.

The last example is particularly noteworthy. First, it implicitly acknowledges that compromises may be necessary. After all, it may not be possible to get an implementation that offers both the fastest performance and the greatest maintainability. Second, the example illustrates the fact that it is natural for programmers to indicate the relative importance of criteria, even when the criteria are not directly comparable. This reflects software engineering decisions in the real world, where it makes perfect sense to say that maintainability is twice as important as speed, even though it may be difficult to define the units of maintainability or to say how such units stack up against units of speed.

Desirability formulas consist of zero or more predicates to be satisfied and at most one expression. Predicates simply act as filters on potential implementations; all implementations that satisfy the predicates are considered equally acceptable. An expression in a desirability formula is a set of terms

$$\{w_1 E_1, \ldots, w_n E_n\},$$

where $w_1 \ldots w_n$ are weighting factors and $E_1 \ldots E_n$ are evaluation expressions. The weighting factors may be positive or negative: positive values indicate that the corresponding E_i should be maximized, negative values indicate that it should be minimized. Hence, "the maximum A and the minimum B" can be expressed as $\{A, -B\}$, and "the least runtime and the most maintainability, where maintainability is twice as important as speed" becomes $\{-R, 2 \cdot M\}$, where R represents runtime and M represents maintainability.

An expression E_i must involve only values of a single type and may use any operations defined on that type. Thus, the desirability formula for "the smallest space-time product" is $\{-(S \cdot T)\}$, where S and T have the obvious meanings and are of the same type, e.g., Big Oh.

Weighting factors take on significance only when it is impossible to find a set of implementations for abstractions that is optimal for all E_i. That is, the fact that one criterion is more important than another makes no difference unless we are forced to trade off a decrease in desirability in one criterion for an increase in desirability in another. When this is the case, we proceed as follows. First, for each E_i in a desirability formula, we find the most desirable value M_i across all possible implementations. Given implementations $I_1 \ldots I_m$, we call V_{ij} the value of E_i in implementation I_j, and we let R_{ij} be the ratio of V_{ij} to M_i (determined by *ratio*); we define $R_{ij} = 0$ if I_j lacks a value for one or more of the criteria needed to evaluate E_i. (Such values might be missing because, for example, I_j has no information on

a criterion used in E_i.) The implementation to choose is the one that minimizes

$$\sum_{\substack{1 \le i \le n \\ 1 \le j \le m}} |w_i|(1 - R_{ij}).$$

Here we take the absolute value of w_i, because R_{ij} is a relative measure, rather than an absolute measure. Hence we only need the magnitude of the weight, not the the weight itself.

This sum indicates how to unify the potentially disparate units over which the various E_i are defined. Rather than deal with the units themselves, we deal with what fraction of the optimal value a particular implementation has. As a result, no attempt is made to compare apples to oranges, but instead the convention is adopted that – given equal weighting factors – a given reduction in the quality of apples is equivalent to the same reduction, on a fractional basis, in the quality of oranges.

4 Using Desirability Formulas

Desirability formulas could serve as a declarative interface to a system that would automatically choose optimal implementations for software abstractions. In such a system, which could be part of a "smart" optimizing compiler, the user would express the ideal characteristics of a particular program by writing a desirability formula, and the system would choose implementations for the program's abstractions based on the evaluations of the available implementations. In what follows, we describe some of the primary issues that such a system would have to address.

Given a variable x of of an abstract data type with a set of implementations I in a program P, it might seem that the best implementation for x is the one that optimizes x's desirability formula in the context of P, but determining a truly *optimal* implementation for x requires that one consider the possibility that x take on different implementations at different points in the program. Hence, the best implementation for x is not a single element of I, but is instead a *mapping* from points in P to elements of I.

Unfortunately, the most desirable implementation for x in P cannot be determined by simply calculating the most desirable implementations for x in the individual operations in P. Instead, one must examine how the operations interact. For example, imagine that P uses x in two operations, Op_1 and Op_2. Imagine further that x has two possible implementations, I_1 and I_2, and that Op_1 is most desirably implemented using I_1 while Op_2 is most desirably implemented using I_2. Under these conditions, the most desirable implementation for x in P might be I_1, I_2, or some combination of the two. The best implementation for x will depend on at least the following factors:

1. How much better I_1 is than I_2 for Op_1.

2. How much better I_2 is than I_1 for Op_2.

3. The desirability of the translation $x_1 \rightarrow x_2$.[2]

4. The desirability of the translation $x_2 \rightarrow x_1$.

5. The control-flow environment of Op_1 and Op_2, e.g., whether either or both are in a loop. If a loop is present, is it nested inside other loops? How many times is the loop expected to be executed?

6. The control-flow relationship between Op_1 and Op_2, e.g., whether they are part of a sequence of statements that is always executed together or whether they fall on opposite branches of a conditional, etc.

The end result is that the desirability of an implementation for an abstract variable can only be determined by examining the flow of control through the program and by calculating the greatest desirability possible for *all possible combinations* of implementations for the variable in the program.

Consider the control-flow graph in Figure 2. This graph contains thirteen nodes, each of which corresponds to an operation to be performed. Using the notation introduced earlier for operation sequences, for looping, and for branching, it is possible to express the desirability D of an implementation in the context of this flow graph as the expression

$$\begin{aligned} D &= Op_1 \oplus Op_2 \oplus ((n * (Op_3 \oplus Op_4 \oplus Op_5)) \diamond (Op_6 \oplus (Op_7 \diamond (Op_8 \oplus Op_9 \oplus \\ &\quad Op_{10})))) \oplus Op_{11} \oplus Op_{12} \oplus Op_{13}, \end{aligned}$$

where n is the estimated number of executions of the loop.

Assume that x is involved only in operations Op_1, Op_4, Op_{10}, and Op_{12}. To find the optimal implementation for x, it is necessary to optimize D. The first step in this calculation is to remove from the expression for D all terms not affecting x's implementation; in this case, all operations except Op_1, Op_4, Op_{10}, and Op_{12}. Formally, since these operations cannot affect the choice of the best implementation for x, they are assigned maximum desirability, denoted by $\top$. By definition, $\top$ is an identity element for *sequence*, *loop*, and *branch*, so

$$\begin{aligned} D &= Op_1 \oplus \top \oplus ((n * (\top \oplus Op_4 \oplus \top)) \diamond (\top \oplus (\top \diamond (\top \oplus \top \oplus Op_{10})))) \oplus \\ &\quad \top \oplus Op_{12} \oplus \top \\ &= Op_1 \oplus ((n * Op_4) \diamond Op_{10}) \oplus Op_{12} \end{aligned}$$

[2] $x_\alpha \rightarrow x_\beta$ represents a *translation function* that converts x from implementation α to implementation β.

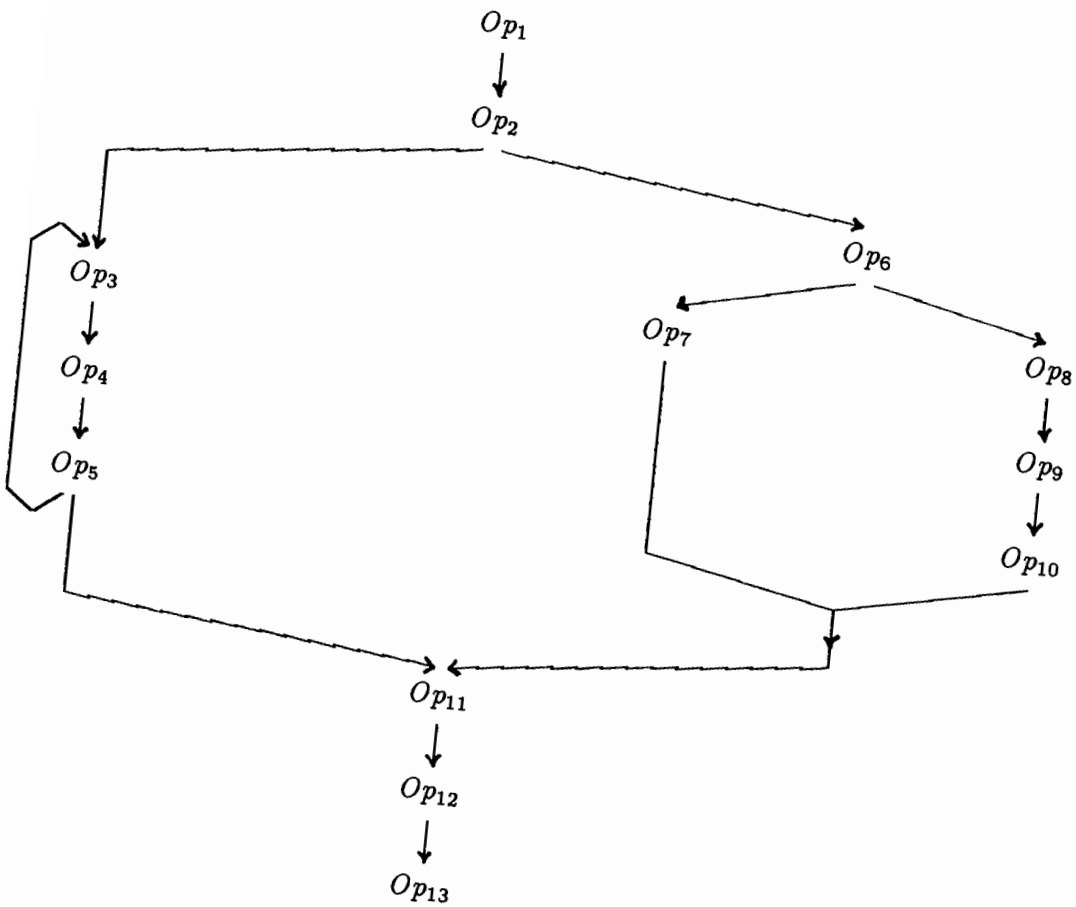

Figure 2: Control-Flow Graph

D could be optimized by examining all possible combinations of implementations for x in Op_1, Op_4, Op_{10}, and Op_{12}, but, in general, that would be computationally prohibitive because the number of possible combinations grows exponentially with the cardinality of I and the number of relevant operations.

The situation becomes even more complex when subprogram calls are taken into account. Consider a subprogram S which takes x as an actual parameter. During execution of $S(x)$, x can conceivably take on up to three different implementations: its implementation *before* the call, its implementation *during* the call (i.e., inside S), and its implementation *after* the call. Call these three implementations x_b, x_d, and x_a, respectively. Given particular implementations for x_b, x_d, and x_a, the overall desirability of $S(x)$, then, can be expressed as

$$(x_b \rightarrow x_d) \oplus S(x_d) \oplus (x_d \rightarrow x_a).$$

Clearly, a call to S will cause a set of operations to be executed, but which operations will be executed depends on the implementation of x, since the implementation for S will typically depend on how x is implemented. Because of this uncertainty, one must consider all possible combinations of implementations for S and implementations for x. In addition, one must consider how the implementations for x_b and x_a affect the implementation for x_d, since it may be more desirable in a global sense to achieve a lesser local desirability for $S(x)$ in order to save the cost of a translation function before and/or after the call to S.

In general, the call graph of a program imposes a hierarchy on the selection process[3], as does the implementation hierarchy for abstractions (i.e., one abstraction may be implemented in terms of other abstractions), and both hierarchies must be examined at once if a truly optimal set of implementations is to be determined. Unfortunately, all algorithms we have so far examined reduce to exhaustive search, which is clearly unacceptable as the basis for a practical system for automatically selecting implementations. An algorithm based on recasting the selection process at each level of the hierarchy as a shortest-path problem reduces the average-time complexity of the problem somewhat (because there are efficient algorithms for computing the shortest path through a directed graph), but this approach does not seem to generalize well to subprograms involving more than one abstract parameter.

5 Discussion

One might hope that by relaxing the constraint to find the *optimal* set of implementations, it might be easier to develop an automatic selection algorithm. If one

[3]Recursion in the call graph is a point of concern, but there are various methods for dealing with such recursion. The simplest method is to simply prohibit recursive dependencies among evaluations.

assumes that some abstractions in a program have incomplete implementations, for example, then one might be interested in a selection algorithm that could efficiently find *feasible* solutions, i.e., *any* complete implementation. It seems to be the case, however, that this problem, too, becomes intractable in the presence of subprogram calls with more than one abstract parameter.

It is disappointing that it should be so difficult to develop efficient algorithms for selecting globally optimal (or feasible) implementations for software abstractions. That it is, however, does not change the fact that the idea of annotating software entities with arbitrary performance characteristics is a useful and flexible one. For example, one might choose implementations for abstractions by looking only at the set of operations in which the abstractions are involved (ignoring the context of the operations), and by then choosing the implementations with the greatest desireability over that set. This is an approach that is similar to that taken by Low and by Rovner, but is more flexible because it is not limited to language primitives and because it applies to arbitrary selection criteria. Another possibility would be to employ a heuristic search for "good" implementations, using the value of a desireability formula as the "goodness measure" to guide the search.

The problem discussed in this paper, that of selecting implementations for software abstractions, is an instance of a more general question: given a choice between solutions to a given problem, each of which "works," which solution should be used? Designers of digital systems face this question when they must choose between IC families to implement their designs, and they face it again when they partition their systems into functional or manufacturable components [Pay81]. Similarly, automated planning systems may face several functionally equivalent plans, but they must somehow choose between the plans before putting one into effect [Hor88]. The methodology described in this paper applies directly to these other problems.

6 Summary

The ideas developed in this paper, that of annotating implementations of abstractions with arbitrary declarative performance characteristics, is much more flexible than traditional space/time measures, and it corresponds more closely to the kinds of decisions that face real-world software developers. This methodology recognizes that performance trade-offs must sometimes be made, and it offers an algorithm for combining incomparable measures of performance. It is a methodology that could be used as a basis for selecting good implementations for the abstractions in a software system, but it is adaptable to many other problem domains as well.

References

[FR69] Jerome A. Feldman and Paul D. Rovner. An Algol-Based Associative Lan-

guage. *Communications of the ACM*, 12(8):439 – 449, August 1969.

[Hor88] Eric J. Horvitz. Reasoning Under Varying and Uncertain Resource Constraints. In *Proceedings of AAAI-88, Volume 1*, pages 111 – 116. American Association for Artificial Intelligence, August 1988.

[Low76] James Richard Low. *Automatic Coding: Choice of Data Structures.* Birkhäuser Verlag, 1976.

[Pay81] Thomas S. Payne. *Automated Partioning of Hierarchically Specified Digital Systems.* PhD thesis, Stanford University, July 1981.

[Rov79] Paul D. Rovner. *Automatic Representation Selection for Associative Data Structures.* PhD thesis, University of Rochester, September 1979.

[She83] Mark Steven Sherman. *Paragon: A Language Using Type Hierarchies for the Specification, Implementation and Selection of Abstract Data Types.* PhD thesis, Carnegie-Mellon University, August 1983.

[Smi76] Nancy W. Smith. *Sail Tutorial.* Technical Report STAN-CS-76-575, Stanford University Artificial Intelligence Laboratory, October 1976.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

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

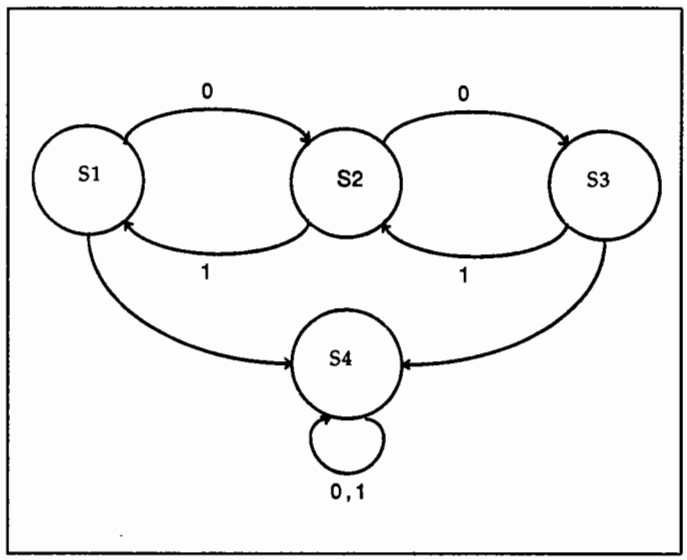

Figure 1: A Customary FSA Diagram

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.

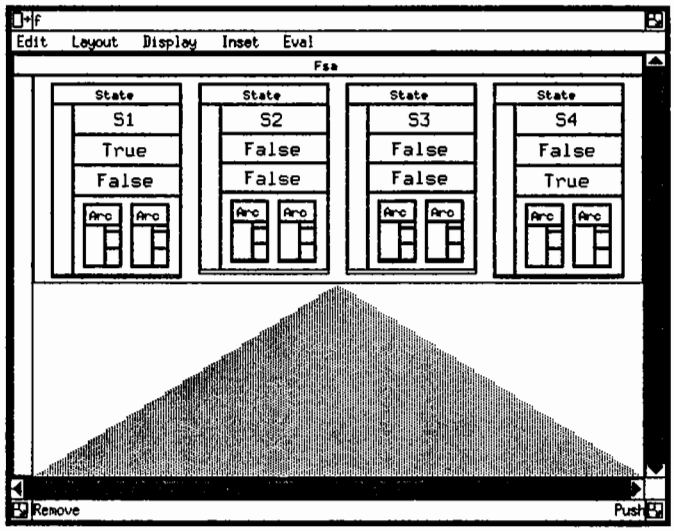

Figure 2: Default Display for f

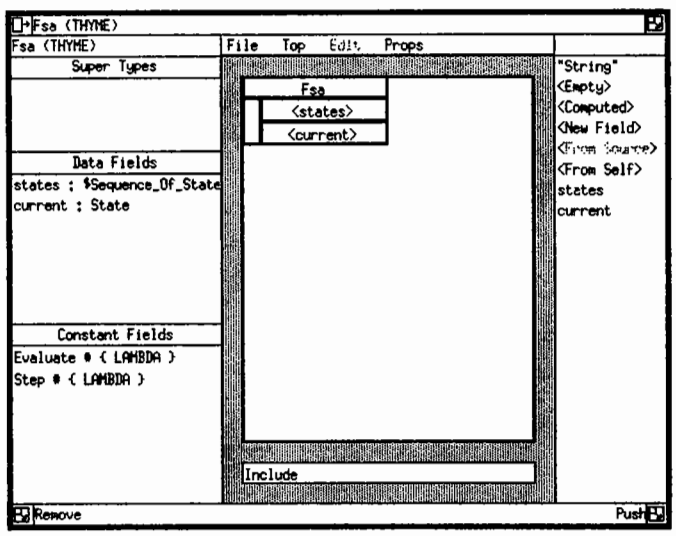

Figure 3: Thyme Display for *Fsa* Showing Default Graphical Representation

- 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.

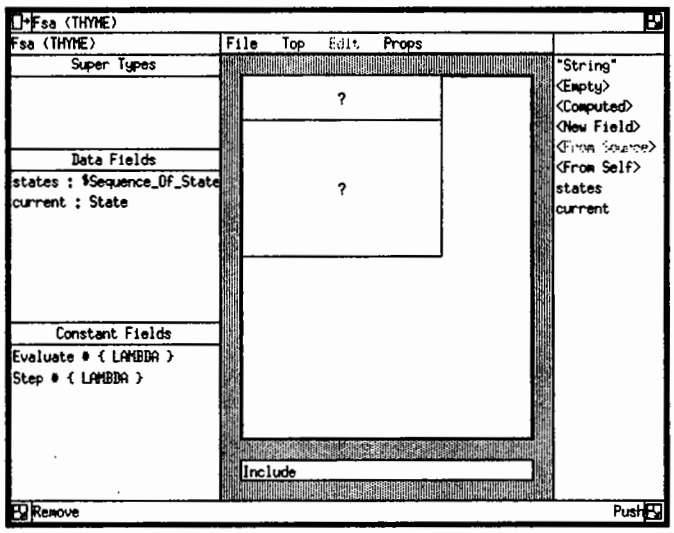

Figure 4: Thyme Display for *Fsa* after Creating a Tiling

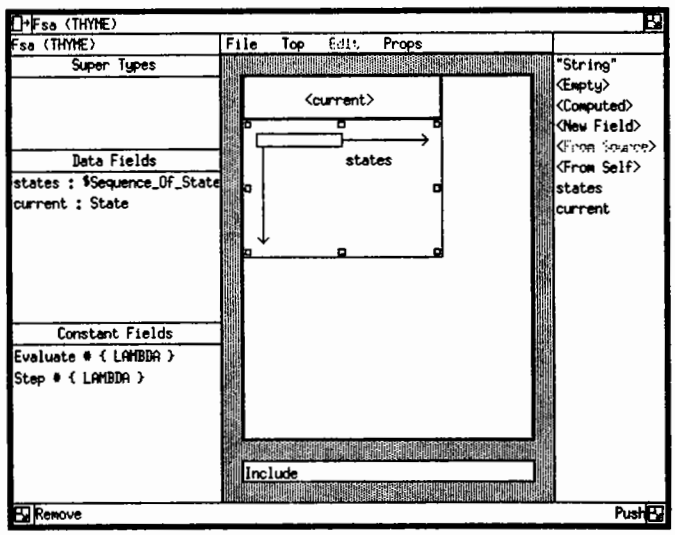

Figure 5: Final Thyme Display for *Fsa*

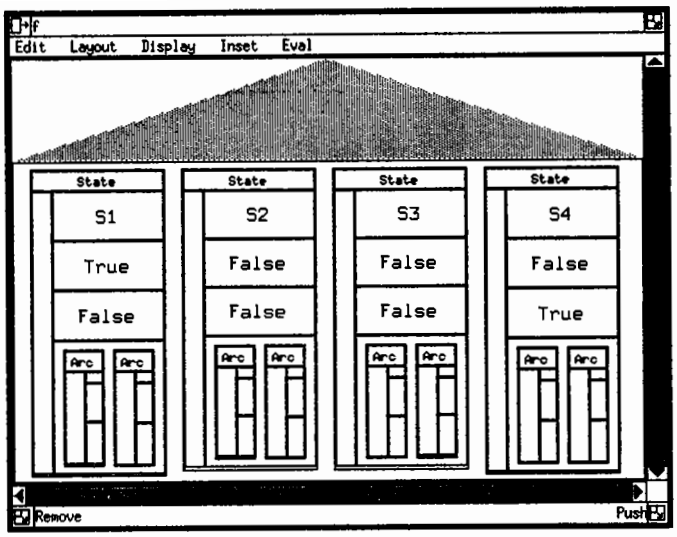

Figure 6: Display of f After Editing *Fsa*

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.

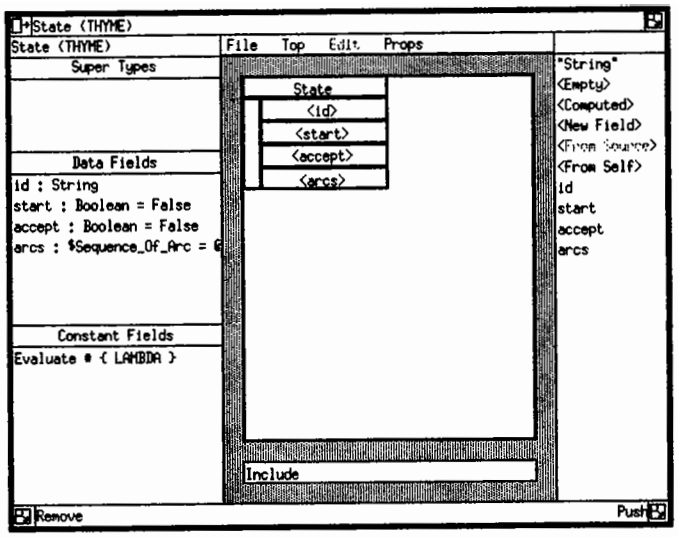

Figure 7: Thyme Display for *State* Showing Default Graphical Representation

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.

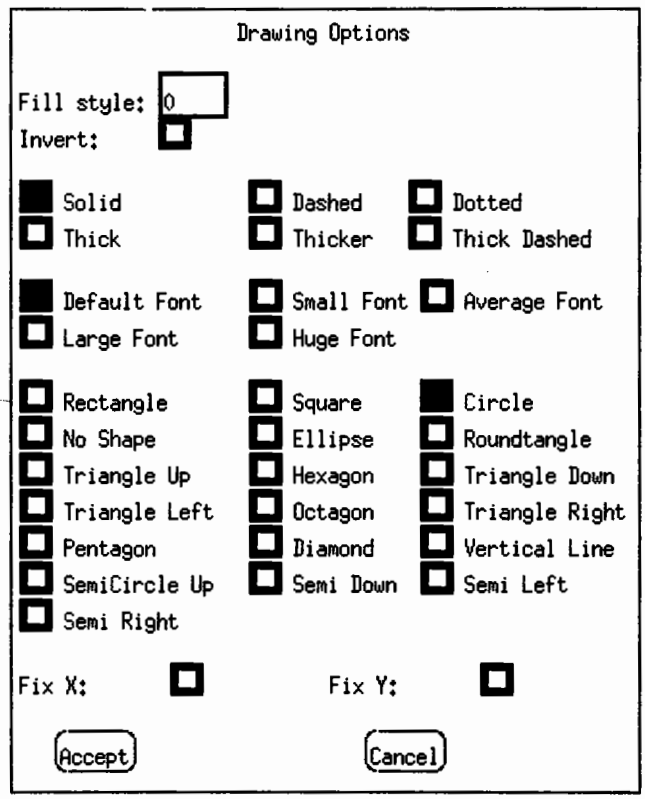

Figure 8: Thyme Options for Box Objects

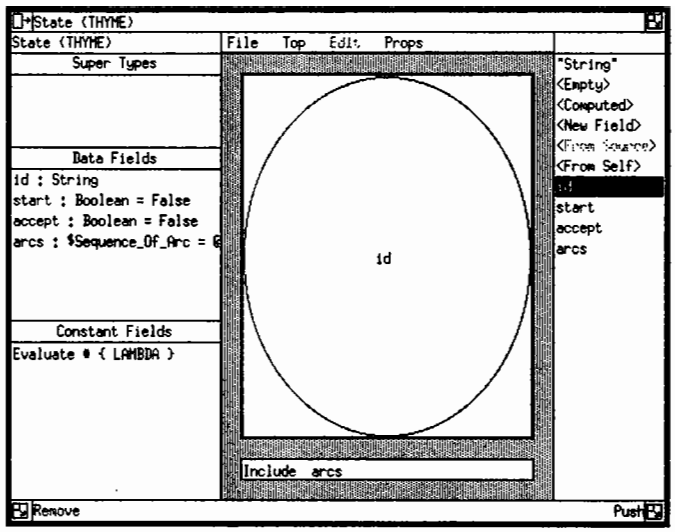

Figure 9: Final Thyme Display for *State*

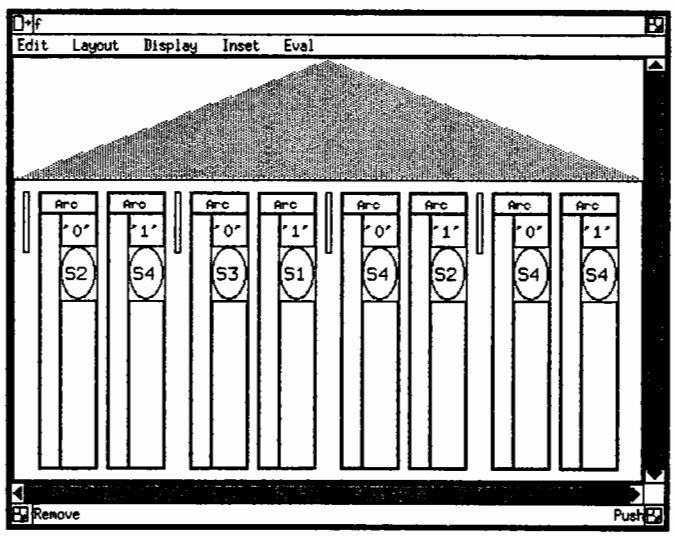

Figure 10: Display of f after Editing *State*

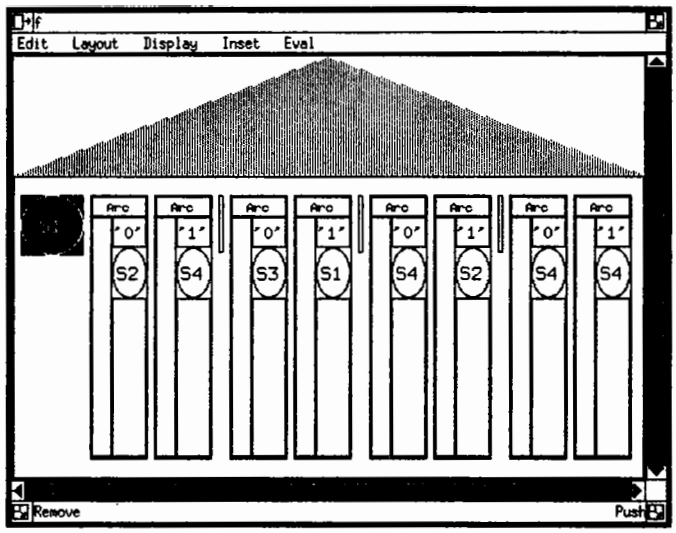

Figure 11: View of f with a State Emphasized

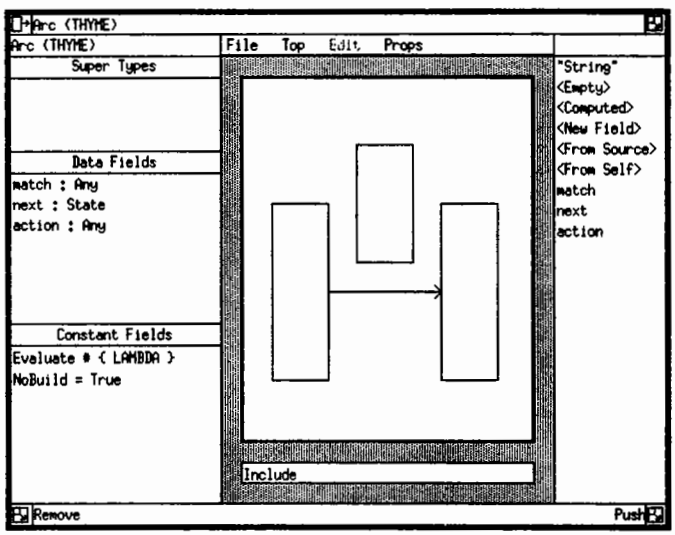

Figure 12: Thyme Display for *Arc* Showing General Arc Representation

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.

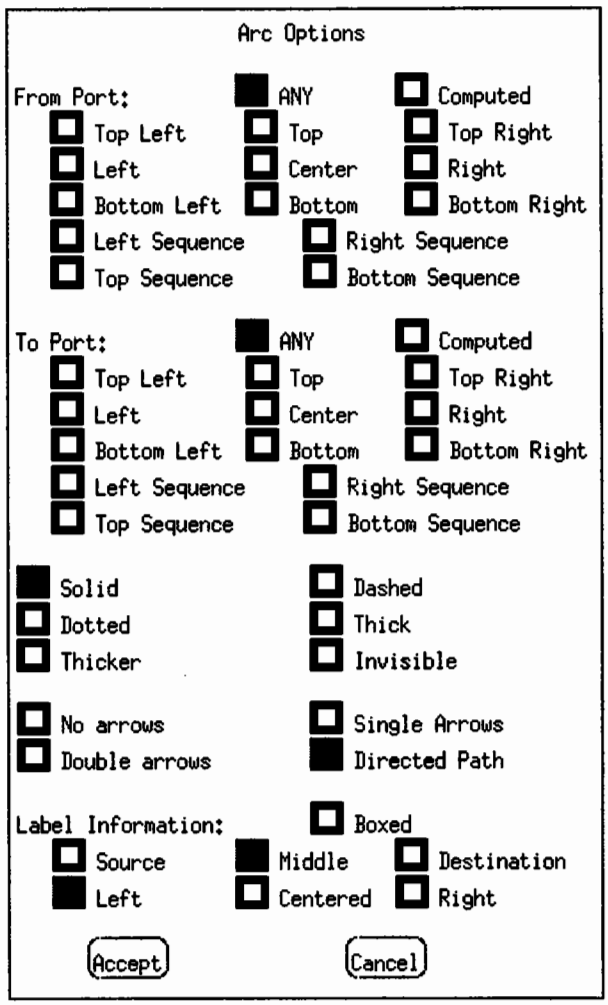

Figure 13: Thyme Options for Arc Objects

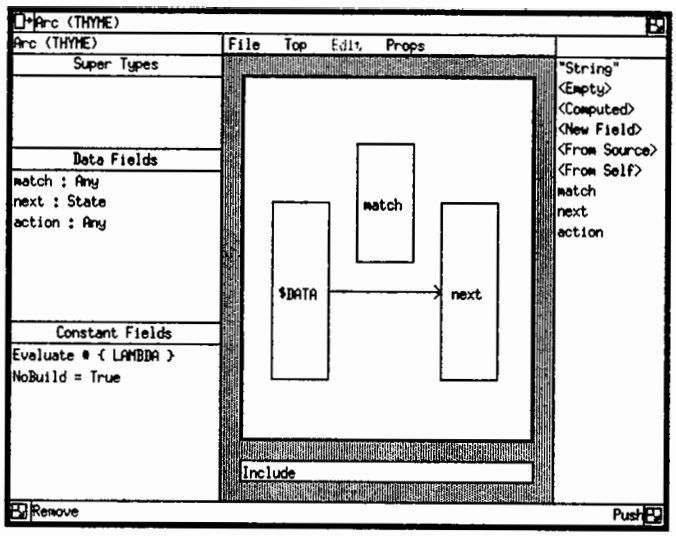

Figure 14: Final Thyme Display for *Arc*

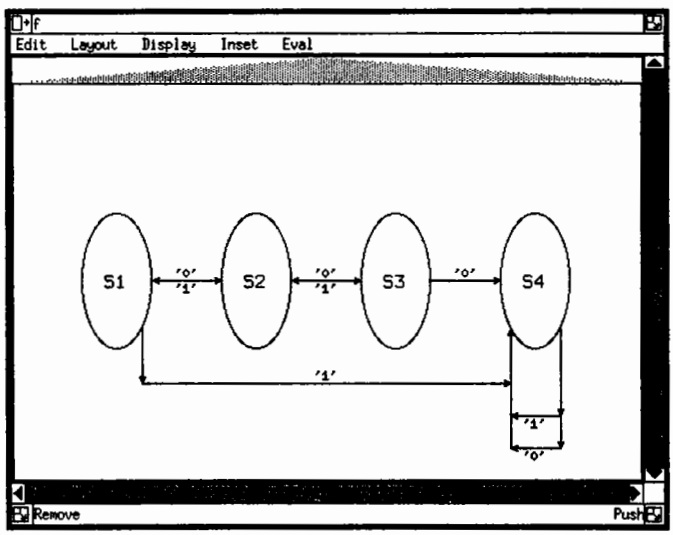

Figure 15: Display of *f* After Editing *Arc*

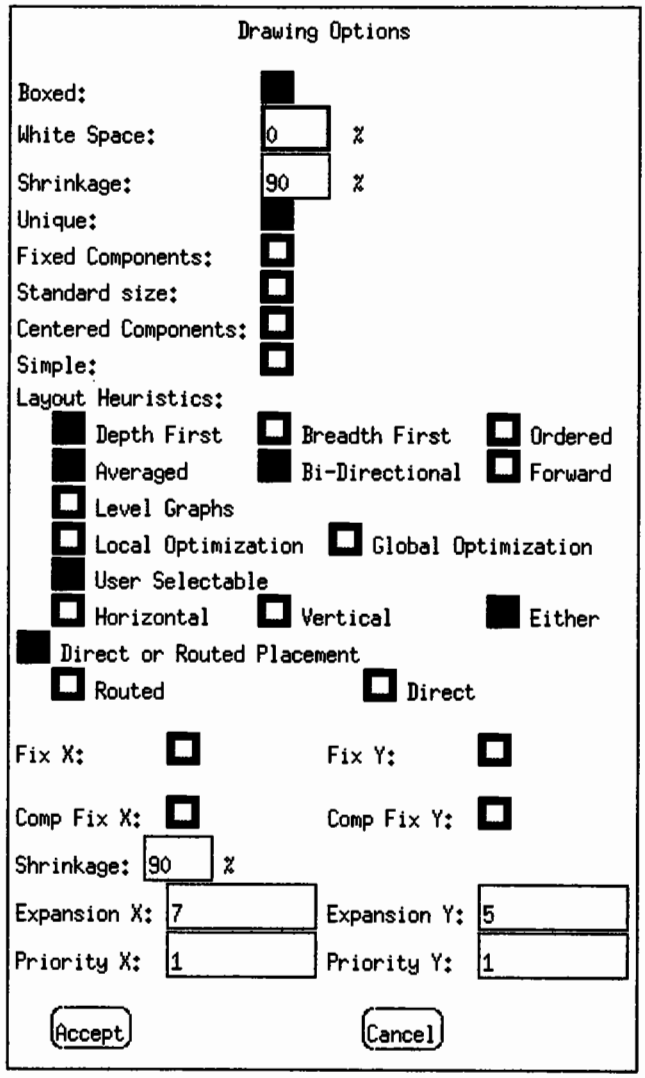

Figure 16: Thyme Options for Layout Objects

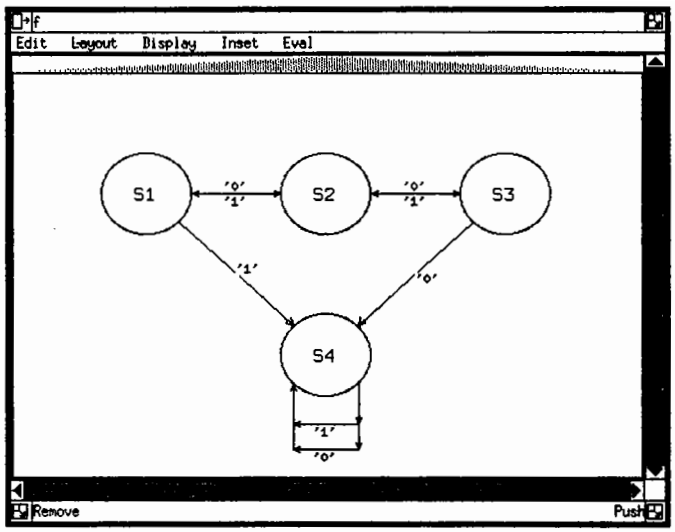

Figure 17: Display of f After Setting Layout Options

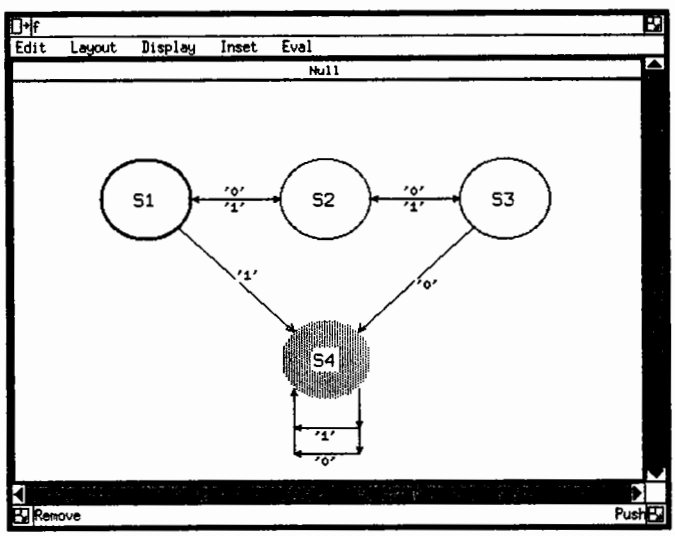

Figure 18: Display of f Using Alternative Representations for *State* Objects

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.