Programmable Design Environments: Integrating End-User Programming with Domain-Oriented Assistance

Michael Eisenberg and Gerhard Fischer
Department of Computer Science and Institute of Cognitive Science
Campus Box 430
University of Colorado, Boulder CO 80309
(duck, gerhard)@cs.colorado.edu; (303-) 492-8091

ABSTRACT
Programmable design environments (PDEs) are computational environments that integrate the conceptual frameworks and components of (a) design environments and (b) programmable applications. The integration of these two approaches provides elements (such as software "critics" and "query-able objects") that assist users in learning both the application and its domain; in addition, an interactive "application-enriched" end-user programming environment stresses the values of expressiveness and modifiability. By way of illustration, we present a newly-developed programmable design environment, *SchemeChart*, for the domain of charting and information displays.

KEYWORDS
Programmable design environments, end-user programming, programmable applications, domain-oriented design environments, critics.

INTRODUCTION
In recent years, familiarity with software applications has become a *sine qua non* for professionals in a variety of complex domains: architects, electrical engineers, biochemists, statisticians, and film directors (among many others) all now depend for their livelihood on the mastery of various collections of applications. These applications, in order to be at all useful, must provide domain workers with complex, powerful functionality; but, in doing so, these systems likewise increase the cognitive cost of mastering the new capabilities and resources that they offer. Moreover, the users of most of these applications soon discover that "software is not soft": i.e., that the behavior of a given application cannot be changed or meaningfully extended without substantial reprogramming. The result is that most applications offer only a rather illusory and selective power: new users are not provided with support in learning and mastering the features of the application, while experienced users are not given the expressive range needed to augment, personalize, and rethink those features.

Over the last few years, we and other researchers have investigated conceptual frameworks to address this problem. This paper describes, and illustrates by example, one such framework: a strategy for the creation of programmable design environments (PDEs). Briefly, programmable design environments are computational systems that integrate elements of two software design paradigms that we have each propounded separately—namely, domain-oriented design environments [6, 7, 8, 9] and programmable applications [5]. The former paradigm stresses the utility of "scaffolding" within applications—techniques (such as software "critics") that assist new users in gaining expertise with an application and its domain. The latter paradigm advocates the inclusion of end-user programming environments within applications (and emphasizes in particular the combination of programming with direct manipulation interfaces).

The following section of this paper presents (in somewhat polemical fashion) the motivation behind the development of PDEs, and argues the potentially ominous nature of current observable trends in the realm of commercial software design. The third section describes the architecture of PDEs, and discusses in greater detail their evolution from the two component frameworks mentioned above. The fourth section presents, by way of illustration, a system named *SchemeChart*: this is a PDE for the creation of charts and information displays. We conclude in the fifth section with a discussion of ongoing and related work.

THE MOTIVATION BEHIND PDES
A perusal of the local newsstand—and the inevitable monthly crop of magazines devoted in large part to software advertisements and reviews—provides a compelling picture of burgeoning complexity in the evolution of software applications. One recent ad for a popular graphics program, for instance, noted that the latest release of the program had added over 100 "feature enhancements"; a subsequent review of the same program

noted that it "packs in the features." Other recent reviews include similar phrases: "dozens of new commands and... features"; "packed with page-layout features"; "more-sophisticated scheduling features"; and so forth.[1] It is in fact rare to find a description of a newly-updated program that does not report an increase in the number of features, or that reports such an increase as anything other than a distinctly positive development.

Depending on one's view, however, it is possible to read both good and bad news into these reports. On the one hand, the reviews and advertisements give evidence of an astonishing pace of development in application software, and an outpouring of creativity in design. But there is a disturbing note as well: will the sixth iteration of our paint program eventually include 500 new features; and if so, how will users ever accommodate them into their own work? Moreover, will those 500 new features really improve our understanding of (e.g.) graphics or design— or will they simply overwhelm us with choices that we can never hope to explore?

The culture of software development reflected in these reports thus places a heavy emphasis on the elaboration of large, varied, and extensive feature sets. Clearly, some of these newly-added features will prove useful; but collectively, they signal a troubling trend in the development of applications. On the one hand, they promote a style of use in which the endless exploration of new features takes on more importance than the patient creative mastery of an expressive medium lending itself to abstraction and composition; at the same time, they distract designers' attention from the more fundamental task of constructing tools that offer the possibility of long-term growth and creativity. The impression ostensibly conveyed by the popular computer press is one of a plethora of ever-expanding choice; yet the feeling that often results is one not of freedom but of overload.[2] Moreover, the problems posed by this style of software design are felt both by beginning users (who are daunted by the sheer quantity of features provided) and by long-term users (who, over time, are inevitably frustrated by the absence of *some* advanced feature that the software designers neglected to include).

PROGRAMMABLE DESIGN ENVIRONMENTS

Considerations such as these reflect the motivation behind the development of PDEs. While the growth of complexity in software applications may be an unavoidable (and in some ways beneficial) trend, both advanced and beginning users need techniques with which to cope with that complexity.

For the benefit of long-term, advanced users, we argue that PDEs should include application-specific interactive programming environments, as eloquently advocated by Nardi [17]:

"We have only scratched the surface of what would be possible if end users could freely program their own applications... As has been shown time and again, no matter how much designers and programmers try to anticipate and provide for what users will need, the effort always falls short because it is impossible to know in advance what may be needed.... End users should have the ability to create customizations, extensions, and applications..." {p. 3}

Pursuing Nardi's argument, the provision of end-user programming environments may thus be seen as one means of combatting the explosion of features described in the previous section: if advanced users are given a medium in which to build their own extensions as each task requires, there is no need for the (in any event futile) attempt to anticipate every possible task by means of an associated special-purpose interface feature.

Basing an application-design strategy *solely* on the inclusion of end-user programming environments, however, is insufficient; arguably, programming environments might be seen not as an antidote to complexity, but rather as an additional source of complexity in applications. For beginning users, then, it is important for PDEs to include "scaffolding" elements that assist the user both in learning the application itself and in learning the (potentially complex) domain around which the application is built.

In summary, then, PDEs are designed to cope with complexity from a variety of different angles by integrating a number of distinct elements: (a) an "application-enriched" programming environment, (b) a "critiquing component" that monitors the user's work and occasionally offers suggestions for changes or tutorial assistance, (c) a "catalog" of illustrative or exemplary work that the user can employ as a starting point for his or her own work, and (d) embedded tutorial components that the user can access for learning about the application or domain. The first of these elements is primarily aimed at alleviating the problems of complexity faced by the advanced user; while collectively, the last three of these elements might be viewed as alleviating complexity for the beginner.

The Evolution of Programmable Design Environments

Programmable design environments represent not only the integration of various software-based elements; more broadly, they represent a combination of two strategies for application design that we have propounded separately—

[1]Sources for these quotes are *PC Computing*, July 1993; *Byte*, September 1993; *PC World*, May 1993; *Mac Computing*, 1993; and *MacUser*, June 1993, respectively.

[2]In the terminology of Norman [18], these expanding feature sets might be said to support a style of use that stresses "experiential cognition" at the expense of "reflective cognition."

programmable applications and domain-oriented design environments. These two "ancestor paradigms" merit some discussion here, since the development of our current interest in PDEs was motivated in large part by considering the respective strengths and weaknesses of these two individual approaches.

Programmable applications are systems that combine direct manipulation interfaces with interactive programming environments. SchemePaint [5] is a working prototype and illustration of a programmable application: it is a graphics application that combines a "Macintosh-style" direct manipulation interface with a graphics-enriched Scheme programming environment. The direct manipulation portion [12] of the application is designed to help users explore the basic functionality of the system and employ their "extra-linguistic" skills of hand-eye coordination. The programming environment is designed to provide users with extensibility and expressive range. This portion of the application is constructed around a collection of embedded graphics "sub-languages" (e.g., a "turtle language", a "dynamical systems language", a "tiling-patterns language", and so forth) that allow users to express graphical ideas by writing short, simple programs. The use of SchemePaint by artists has shown that they can create works that would be near-impossible to achieve either by "pure" direct manipulation or by "pure" programming alone.

While programmable applications do, then, overcome some of the limitations of stand-alone direct manipulation systems and end-user programming environments (for a more detailed analysis see [5]), they have their own characteristic shortcomings. First, programmable applications provide insufficient support and feedback to help the user achieve quality artifacts (that is, the use of SchemePaint has shown that while gifted artists can do interesting things with it, this is far from true for less experienced and talented users). Second, these applications provide little support in learning the programming language (in the current case, Scheme), or in assimilating useful programming patterns ("cliches") related to the particular domain-specific sub-languages provided with the system. Finally, programmable applications such as SchemePaint do not support case-based "memories" of good designs (thereby limiting support for design by modification).

Domain-oriented design environments are systems that integrate construction and argumentation (in Schoen's terminology, they support "reflection-in-action" [22]). This integration is made possible by the presence of software critics [7] that analyze an artifact under construction, signal breakdown situations, and provide entry points to the space of relevant argumentation directly relevant to construction situations. The interweaving of construction and argumentation is critical: stand-alone argumentation systems (without the presence of an artifact as the focus of argumentation) [2] are unable to contextualize discussion to the design task at hand, while stand-alone construction systems provide no computational support for analyzing, commenting upon, and critiquing designs.

While design environments have proven to be a powerful concept in a large number of domains [6], they themselves are not free of characteristic problems. Their main shortcomings reside in the problems alluded to in the quote from Nardi [17] earlier: namely, they provide inadequate support for design tasks not foreseen by the creator of the design environment, and thus fall short in transcending the limits of envisioned activities.

These two approaches, then, appear to lend themselves well to an additional step of conceptual integration. PDEs, by combining elements of both design strategies, are intended to overcome their respective limitations. Unlike programmable applications, PDEs include critiquing, catalog, and tutorial elements; unlike design environments, PDEs include an end-user programming language. The following section describes a working prototype of this concept.

A PROGRAMMABLE DESIGN ENVIRONMENT FOR CHART CREATION

SchemeChart is an application for the creation of charts, graphs, and information displays.[3] The program includes a direct manipulation interface for selecting the type of chart that the user wishes to create, for editing newly-created charts by hand, and for performing a variety of standard graphics functions (such as filling and drawing lines); an interpreter for an extended Scheme language suited for the construction of a wide variety of chart types; and a number of critiquing, sample-selection, and tutorial elements (to be described shortly).

Figure 1 depicts a screen view of SchemeChart in the course of a typical chart-design task. Briefly, the **SchemeChart** window is the area in which new charts are constructed and edited; the **Paint Tools** window presents a palette of standard tools for choosing color, pen width, and so forth; the **Charts** window provides an initial "coarse-grained" selection of chart types from which to choose (e.g., bar charts, scatter plots, pie charts, etc.); the **Samples** window presents more specific varieties of the charts selected in the **Charts** window; and the **transcript** window provides an interpreter for the enriched Scheme system. These windows are always displayed and thus comprise the "standard set" for the application; the figure also shows a window labelled **Trapezoidal Bar Chart Examples** which appears in the course of the specific scenario to be discussed below.

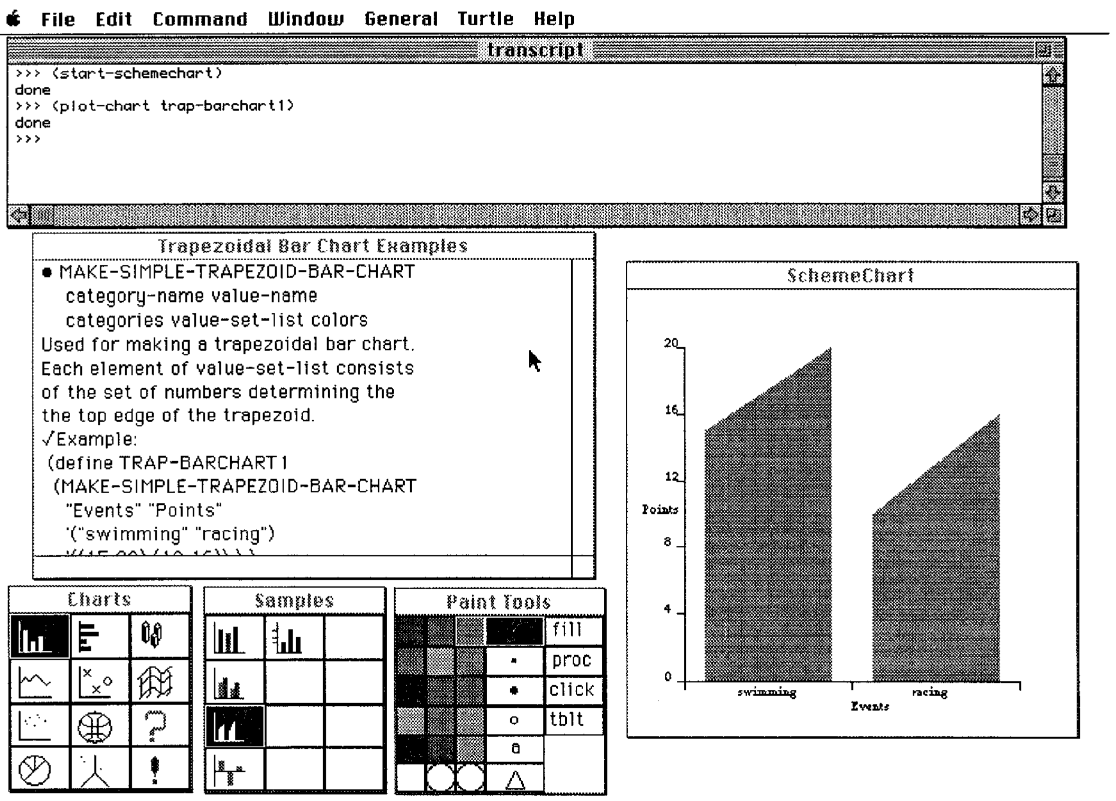
In the Figure 1 scenario, the user has decided that she wishes to construct a bar chart; she selects the bar chart icon from the window labeled **Charts** in the figure. Once

[3]The application is written in MacScheme (Lightship Software, Mountain View, CA) and runs on all Apple Macintosh Computers.

this basic type is selected, a variety of specific bar chart variations is iconically depicted in the **Samples** window. (These include charts with multicolored bars, bars going above and below the horizontal axis, bars accompanied by "y-axis tick marks" to facilitate the reading of height values, and so forth.) The user selects a particular bar chart icon from the set presented—this one depicting "trapezoidal" bar charts with non-horizontal uppermost lines.

Once the selection of trapezoidal bar charts has been performed, the user can access, via menu selection, a scrollable text window containing an explanation of the icon's meaning (along with some tutorial description of the rationale behind using this particular type of chart—in this instance, trapezoidal bar charts are useful for depicting maximum and minimum range values as opposed to the single data values denoted by standard bar chart levels). The user can similarly request a text window in which sample SchemeChart procedures and expressions are shown for the creation of bar charts of this type. This is the scenario shown in Figure 1: here, the user has examined the sample language expression in the window labelled **Trapezoidal Bar Chart Examples**. The sample code in this window can be edited (if so desired) and evaluated; the user has in fact evaluated the original (unchanged) sample expression and plotted the resulting chart just to get a sense of the sample procedure's meaning. This technique of programming via example modification is similar to that advocated by Lewis and Olson [13], and illustrated by MacLean *et al.* in their work on editable "interface buttons" [15]. The use of a browsable application-specific iconic "catalog" to locate examples is designed to address the problems of example-location raised by Nardi [17].

Figure 2 shows a continuation of this scenario. The user has first edited the original trapezoidal bar chart example to include new data values; she then reevaluates the newly-edited expression. When this action is performed, the user receives an "alert signal": the exclamation point in the **Charts** window flashes several times to indicate that a system critic has detected a potential problem in the graph under construction. (In this case, the critic is associated with the particular procedure being invoked.)



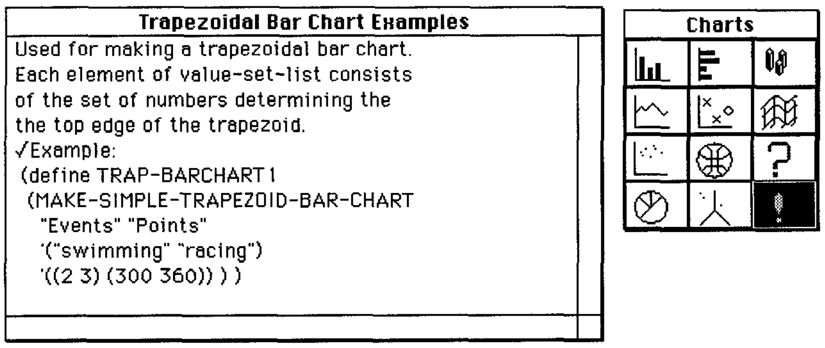
```
>>> (start-schemechart)
done
>>> (plot-chart trap-barchart1)
done
>>>
```

```
• MAKE-SIMPLE-TRAPEZOID-BAR-CHART
   category-name value-name
   categories value-set-list colors
Used for making a trapezoidal bar chart.
Each element of value-set-list consists
of the set of numbers determining the
the top edge of the trapezoid.
√Example:
 (define TRAP-BARCHART1
  (MAKE-SIMPLE-TRAPEZOID-BAR-CHART
   "Events" "Points"
   '("swimming" "racing")
```

Figure 1. A screen view of the SchemeChart application. Charts are created in the **SchemeChart** window at bottom right; the **Paint Tools** window provides standard paint functionality; the **Charts** window provides an overview of standard chart types from which to choose; and, for a given graph choice, a catalog of specific examples is provided in the **Samples** window. Finally, the **transcript** window at top provides an "application-enriched" Scheme interpreter. In this figure, the user has selected a particular (trapezoidal) type of bar chart from the **Samples** window, and has used a menu option to access relevant programming examples for this type of chart; the examples are shown in the window labelled **Trapezoidal Bar Chart Examples**. The user has evaluated the sample expression to produce the trapezoidal bar chart shown.

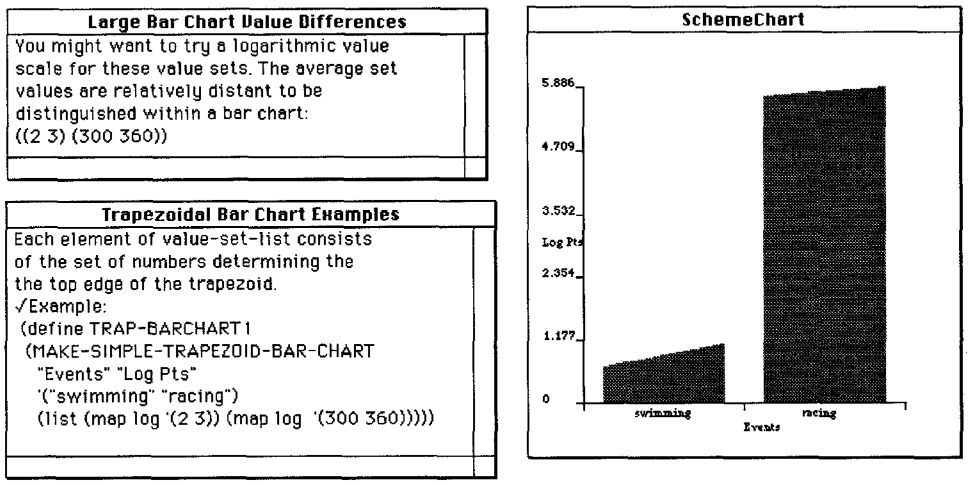
The user now has the option of ignoring the critic's intrusion or requesting (again via menu selection) to view the text associated with the newly-invoked critic. In Figure 2, the user has in fact elected to request the critic's response; the **Large Bar Chart Value Differences** window thus invoked presents the system's critique of the newly-evaluated expression. In this case, the discrepancy between the two bars to be plotted (more accurately, the discrepancy between the average values of the smaller bar and larger bar) is too large to warrant presentation in a bar chart, which typically is used to display more moderate distinctions between values. The critic goes on to suggest that an alternate (e.g. logarithmic) vertical scale might be preferable for displaying these values. In Figure 2, the user has responded by redoing the original language expression so that it plots the natural logs of the given numeric values; when this rewritten expression is now evaluated, the flashing critic alert does not appear, thus indicating that the system has found no reason to suggest changes in the user's construction.

In Figure 3, the user has continued her work by using the system's "query mode" (denoted by the question-mark icon in the **Charts** window). Here, the trapezoidal bar chart has been plotted; by selecting the query mode option, the user can now select (via mouse) portions of the newly-drawn chart. In Figure 3, the user has dragged the mouse over the y-axis of the graph; the system highlights the axis to show that it is a "query-able" object. When the mouse button is released at this point the user is presented with a text window listing a variety of SchemeChart procedures that can be used to change axes (e.g., by changing their length, color, or starting and ending points; selecting the given procedure name yields further description). Similar procedure-description windows may be viewed for (among other elements) axis labels and tick-marks; thus, by invoking the query mode the user is able to work "backward" from a newly-created artifact to the relevant portions of the application's language vocabulary. This technique is similar to that developed by Redmiles [20] in the context of examining programming examples for software reuse.


Trapezoidal Bar Chart Examples

Used for making a trapezoidal bar chart.
Each element of value-set-list consists
of the set of numbers determining the
the top edge of the trapezoid.
√Example:
```
(define TRAP-BARCHART1
 (MAKE-SIMPLE-TRAPEZOID-BAR-CHART
   "Events" "Points"
   '("swimming" "racing")
   '((2 3) (300 360)) ) )
```

Figure 2a. The user edits the sample expression in the window at left to create a new trapezoidal bar chart; when this new expression is evaluated, the "critic alert signal" (the exclamation point in the **Charts** window) flashes several times to indicate that a potential problem has been spotted with this newly-created chart.

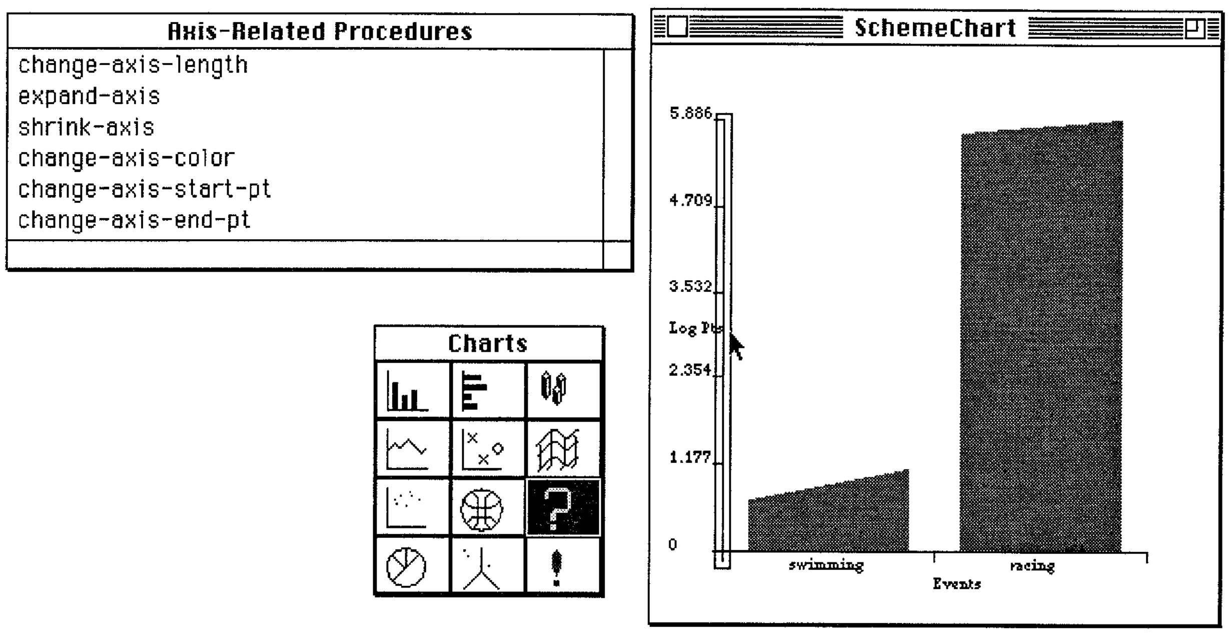

Large Bar Chart Value Differences

You might want to try a logarithmic value
scale for these value sets. The average set
values are relatively distant to be
distinguished within a bar chart:
((2 3) (300 360))

Trapezoidal Bar Chart Examples

Each element of value-set-list consists
of the set of numbers determining the
the top edge of the trapezoid.
√Example:
```
(define TRAP-BARCHART1
 (MAKE-SIMPLE-TRAPEZOID-BAR-CHART
   "Events" "Log Pts"
   '("swimming" "racing")
   (list (map log '(2 3)) (map log '(300 360)))))
```

Figure 2b. The user requests the critic message (at top left), which indicates that the differences between plotted bar chart values is extremely large in this case; the critic suggests a logarithmic value scale. The user rewrites the expression (using the Scheme log function to alter the plotted numeric values) and plots the new chart at right.



Axis-Related Procedures
change-axis-length
expand-axis
shrink-axis
change-axis-color
change-axis-start-pt
change-axis-end-pt

Figure 3. Here, the user selects "query mode" (the question mark icon in the **Charts** window). By dragging the mouse over a particular element of the newly-created graph (in this case, the y-axis), the user can access a list of language procedures relevant to the manipulation of this element.

In summary, then, the programming environment within SchemeChart—containing as it does an enriched Scheme dialect for the representation and display of chart objects—provides a medium in which advanced users can create a wide variety of designs[4]; while the system's critics, iconic catalogs, menu-accessible tutorial material, and "query-able objects" provide means for learning both about the language and about appropriate techniques (or at least techniques comprising a form of conventional wisdom [21, 23]) for designing charts and graphs.

ONGOING AND RELATED WORK

As a software-design strategy, the notion of PDEs reflects (and represents a response to) a variety of influences from related work. In providing user-accessible programming languages, PDEs reflect an outlook similar to those of the Logo [19] and Boxer [4] language-design efforts (though with a greater emphasis on application construction and domain orientation). In this same spirit, the decision to employ Scheme as the base language for our PDE—as opposed to some application-specific *ad hoc* language—is admittedly controversial, though the arguments surrounding this issue are far from clear-cut [5]; the choice of Scheme inevitably reflects the style and influence of Abelson and Sussman [1], and their argument for creating interactive "embedded languages" (domain-enriched dialects) that exploit an underlying general-purpose language environment. Again, many of the specific decisions in SchemeChart diverge from those, e.g., of the "programming-by-example" community [3] (in that SchemeChart provides an explicit programming language); but the ideals of providing users with techniques for modifying and extending applications are shared in both approaches. Mackay [14] and Gantt and Nardi [10] respectively provide cautionary and encouraging empirical case studies of how user-modifiable systems are employed and appropriated within organizations. Finally, in their focus on integrating critiquing, argumentation, and design activities, PDEs reflect most strongly the theoretical framework of Schoen [22], who portrays design activity as productive interplay between both a tacit and formalized (or verbalized) understanding of designs under construction.

PDEs, as complex applications in their own right, raise many issues involving usability and learning (e.g., how users learn programming languages and application domains) which we have begun to investigate; as in our earlier efforts, we expect user studies to prove invaluable both in rethinking our ideas and exposing our mistakes. Our hope is that PDEs can eventually suggest means for mitigating (if not, in a perfect world, resolving) the apparent tension between expressiveness and learnability in application design.

[4]In fact, the language has been used to generate a variety of advanced special-purpose graphs including 3d bar and surface charts, bar charts with "fading colors" to indicate uncertainty, triangular charts, and many others.

ACKNOWLEDGMENTS
The authors would like to thank the members of the Human-Computer Communication group at the University of Colorado who contributed to the conceptual frameworks and systems discussed in this article. The research was supported by the National Science Foundation under grants IRI-9015441 and MDR-9253425. The first author is supported in addition by a National Science Foundation Young Investigator grant (IRI-9258684).

REFERENCES
1. Abelson, H. and Sussman, G.J., with Sussman, J. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA 1980.

2. Conklin, J. and Begeman, M. gIBIS: A Hypertext Tool for Exploratory Policy Discussion. *ACM Transactions on Office Information Systems*, Vol. 6, No. 4, October 1988, pp. 303-331.

3. Cypher, A. ed. *Watch What I Do.* MIT Press, Cambridge, MA 1993.

4. diSessa, A. and Abelson, H. Boxer: A Reconstructible Computational Medium. *Communications of the ACM*, Vol. 29, No. 9, September 1986, pp. 859-868.

5. Eisenberg, M. Programmable Applications: Interpreter Meets Interface. Artificial Intelligence Laboratory Technical Report 1325, MIT, 1991.

6. Fischer, G., Grudin, J., Lemke, A.C., McCall, R., Ostwald, J. , Reeves, B. and Shipman, F. Supporting Indirect, Collaborative Design with Integrated Knowledge-Based Design Environments *Human Computer Interaction*, Vol. 7, No. 3, 1992, (in press).

7. Fischer, G., Lemke, A.C., Mastaglio, T. and Morch, A. The Role of Critiquing in Cooperative Problem Solving. *ACM Transactions on Information Systems*, Vol. 9, No. 2, 1991, pp. 123-151.

8. Fischer, G., Lemke, A.C., McCall, R. and Morch, A. Making Argumentation Serve Design. *Human Computer Interaction*, Vol. 6, No. 3-4, 1991, pp. 393-419.

9. Fischer, G., Lemke, A.C. Construction Kits and Design Environments: Steps Toward Human Problem-Domain Communication. *Human-Computer Interaction*, Vol. 3, No. 3, 1988, pp. 179-222.

10. Gantt, M. and Nardi, B. A. Gardeners and Gurus: Patterns of Cooperation Among CAD Users. *CHI '92 Conference Proceedings*, pp. 107-117.

11. Girgensohn, A. End-User Modifiability in Knowledge-Based Design Environments. Technical Report, Department of Computer Science, University of Colorado CU-CS-595-92, 1992.

12. Hutchins, E.L., Hollan, J. D. and Norman, D. A. Direct Manipulation Interfaces. In D.A. Norman, S.W. Draper (eds.), *User Centered System Design, New Perspectives on Human-Computer Interaction*, Lawrence Erlbaum Associates, Hillsdale, NJ, 1986, pp. 87-124.

13. Lewis, C. and Olson, G. Can Principles of Cognition Lower the Barrier to Programming? In *Empirical Studies of Programmers, Second Workshop.* Olson, G.; Sheppard, S.; and Soloway, E. eds. Ablex, NJ 1987

14. Mackay, W.E. Triggers and Barriers to Customizing Software. *CHI '91 Conference Proceedings*, pp. 153-160.

15. MacLean, A.; Carter, K.; Lovstrand, L.; and Moran, T. User-Tailorable Systems: Pressing the Issues with Buttons. *CHI '90 Conference Proceedings*, pp. 175-182.

16. McCall, R. PHI: A Conceptual Foundation for Design Hypermedia. *Design Studies*, Vol. 12, No. 1, 1991, pp. 30-41.

17. Nardi, B. *A Small Matter of Programming.* MIT Press, Cambridge, MA 1993.

18. Norman, D. A. *Things That Make Us Smart.* Addison-Wesley Publishing Company, Reading, MA 1993.

19. Papert, S. *Mindstorms.* Basic Books, New York, 1980.

20. Redmiles, D. F. Reducing the Variability of Programmers' Performance Through Explained Examples, INTERCHI '93 Conference Proceedings, pp. 67-73.

21. Robertson, Bruce. *How to Draw Charts and Diagrams.* North Light Books, Cincinnati, OH 1988.

22. Schoen, D. A. *The Reflective Practitioner: How Professionals Think in Action.* Basic Books, New York, 1983.

23. Tufte, E. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire CT 1983.