Embedding languages within design environments

Michael Eisenberg

Large-scale computer applications reflect numerous sources of complexity: complexity originating in the constructs of the application domain, complexity of feature-rich interfaces, and (in many cases) complexity arising from the inclusion of end-user programming environments. The paper discusses a software design strategy for the creation of *programmable design environments* (PDEs); such environments are geared toward alleviating, for users, the negative effects of various sources of complexity. The paper illustrates this notion of PDEs with a description of a charting/graphing application named SchemeChart.

Keywords: programmable design environments, applications, complexity

The past decade has witnessed a noticeable blossoming in application software — in its variety, aesthetic appeal, reliability, speed, and affordability. For numerous professionals (including video editors, graphics designers, organic chemists, electrical engineers, librarians, and architects), the very definition of professional activity has become bound up with the use and mastery of various sets of large-scale applications. This change has come at a price, however: while the advent of these applications affords new media of expression to users, the complexity of the applications presents users with something of a dilemma. Such applications often require a long time to learn, and thus represent a major investment in 'startup time' for new users; even when learned, the applications represent a moving target, as new versions and upgrades often appear at a rate that exceeds the users' capacity to master previous instantiations. Also, because the applications are so large and feature-rich, they often compel even experienced users to spend large amounts of time searching for the presence (or, frustratingly, confirming the absence) of some particular feature [Eisenberg and Fischer, 1994].

Why must applications be such complicated artifacts? Is the complexity avoidable, or is it simply an inevitable result of creative or economic forces? In considering such questions, it is worthwhile to begin with a tentative classification of the sources of complexity in application software:

- *Domain complexity:* Application software may deal with inherently complex domains, such as the simulation of ecosystems, atmospheric reaction chemistry, or macroeconomics.
- *Interface complexity:* The software may present a large, perhaps an incomprehensibly large, feature set to the user. The collection of features may be (or appear to the user as) an *ad hoc* mass of unrelated functionality, in which certain features are included in the software while others, equally plausible or useful, are not, and thus the interface may prove difficult not only in terms of quantity but in terms of arrangement, organization, or predictability.
- *Language complexity:* In certain instances, applications may include extension languages of various sorts: macros, 'scripting' tools, application-specific programming languages, or even fully fledged programming environments based on an enriched dialect of some general-purpose language. These linguistic additions to applications create new sources of complexity.

The reason why a classification such as this one must be deemed tentative is because the relationship between these various sources of complexity is itself complex. One might argue, for example, that (in the case of the first category) complexity is not necessarily inherent in some given domain, but is rather derived from the increasing power of applications. For example, where, in the past, the visual arrangement of a typewritten document might not have seemed a particularly compli-

Department of Computer Science and Institute of Cognitive Science, University of Colorado, Boulder, CO 80309–0430, USA

Paper accepted 15 April 1994

cated task, it has become so with the advent of powerful word processors and their profusion of associated fonts, text styles, and layout tools. Thus, the purported complexity of the domain may itself be a function of (say) the provision of a feature-rich interface. Moreover, the development of application languages, the third source of complexity listed above, may be viewed in part as a response to cope with the phenomenon of feature expansion. By providing users with a medium in which to create their own customization and extensions to software, the designer is relieved of the task of providing specific tools for each anticipated task (cf. the discussion in Nardi [1993]).

We will return to this last point shortly, but, before this, it is worth looking once more at the taxonomy ventured above. Plausibly, one might argue that the first (domain) category of complexity represents an unavoidable source of difficulty in the learnability and use of applications: if one wishes to design an application for the simulation of (for example) international economics, such an application can hardly avoid the inclusion of numerous tools for the setting of a wide range of simulation parameters. Moreover, such an application will probably either include tools for (say) statistical analysis, graphing and charting, and numerical integration, or will need to communicate with a variety of other stand-alone applications designed for these (again complex) tasks. Thus, inherent domain complexity (assuming, of course, that we accept such a notion to begin with) may be viewed as the 'constraining' source of complexity, affecting the design of the application as a whole. From the software designer's standpoint, the remaining two sources of complexity, interface features and embedded languages, must then be created to reflect (but ideally not amplify) the basic complexity of the subject matter itself.

Conversely, if we accept the taxonomy above, we look to interface features and languages as the areas in which software complexity might be alleviated. As noted above, end-user programming languages might themselves be seen as a partial (and risky) step toward the alleviation of feature explosion.

To summarize the argument thus far, if we wish to design applications that alleviate the problems of software complexity for users, we must not seek to eliminate the phenomenon (or perception) of complexity entirely. Rather, the various sources of complexity must be considered in the design of expressive software applications; those techniques that mitigate complexity must do so while respecting the origins of the phenomena to which they embody a response.

- Our software should include tools that enable users to represent (or that enable students to learn) the domain concepts that underlie the application.
- Our software should seek to identify those features best represented in graphical (or perhaps 'non-linguistic') interface features, and those best represented in the form of programming constructs.
- Given that we decide to include a programming environment in an application, we must seek means of alleviating the special types of complexity (learning vocabulary, learning special application-relevant programming constructs, and so forth) that accompany this decision and often render it problematic in the eyes of software designers and users.

One additional, if deceptively self-evident, observation is in order in this summary, in preparation for the discussion to follow: learning and working with complex software is a long-term process, often demanding months or years of a user's time. As designers, we may seek to make our application software more learnable, and to make the period of 'novice' status shorter for our users, but we should not expect that even the most 'expert' users will (or need) ever feel that there is nothing left to learn about the application. The issues involved in the design of such software will most likely differ from those in the design of 'walk-up-and-use' software systems such as automatic teller machines. Walk-up-and-use software designers are concerned with anticipating all possible user actions and rendering the user's decision process swift, stereotyped, and error-free; the designers of complex applications are concerned with providing avenues of exploration, learning, and patient creative mastery over a new medium. We should not, indeed, expect it to be otherwise. The very point of professional applications is that they reward and give personalized expression to expertise (while, ideally, remaining accessible for beginning users).

The remainder of this paper elaborates upon the themes of this introduction through the notion of programmable design environments (PDEs), applications composed of elements designed specifically to accommodate the various types of complexity listed above. The following section describes the notion of PDEs in greater detail; the third section illustrates this notion by discussing SchemeChart, a prototype graphing/charting application developed in our laboratory at the University of Colorado, USA. We conclude with a discussion of related and future work.

PROGRAMMABLE DESIGN ENVIRONMENTS

Programmable design environments are perhaps best described by recourse to their genealogy: they are an attempt to produce a creative blending of the notions of *programmable applications* on the one hand, and *domain-enriched design environments* on the other. Programmable applications, as described in Eisenberg [1991], are applications that seek to integrate programming languages with direct manipulation interfaces, distinguishing those activities best performed 'by hand' (e.g. drawing a portrait of a friend, choosing an attractive pattern for an interior design task, identifying a familiar color combination) with those best performed by the use of language and abstraction (e.g. drawing a geometric pattern, modeling a complex system). The SchemePaint system [Eisenberg, 1992] is an example of a programmable graphics application. It includes a skeletal direct-manipulation paint-program interface, as well as a 'graphics-enriched' Scheme interpreter. Collectively, these permit the user to create drawings that weave together patterns produced by programs (turtle-graphics figures, tiling patterns, and views of three-dimensional solids, to name a few) with patterns produced by hand.

Domain-enriched design environments are eloquently described in a series of papers by Fischer and his colleagues [Fischer, 1991; Fischer *et al.* 1991; 1992]. These are environments that include 'scaffolding' elements allowing users to cope with potentially complex domains. For example, software *critiquing mechanisms* are employed in design environments as background 'demons' that monitor the user's work and alert the user to potential problems in his or her designs. *Catalogs*, another element typical of design environments, are browsable collections of illustrative or exemplary work; these provide both starting examples for the user's design, and samples that provide a foundational database for systems that maintain records of design rationale. The best-known example of such a design environment is the Janus system for the domain of kitchen design [Fischer *et al.* 1991]. Janus includes, among other elements, a 'construction kit' with which to create new kitchen designs from a palette of sample objects, domain-specific critics (e.g. for tasks such as the placement of appliances), and a catalog of preexisting designs (illustrating 'typical' kitchen layouts).

In creating PDEs, the motivation for blending the two 'ancestor' design paradigms is to overcome the perceived weaknesses in each. Programmable applications exhibit characteristic problems in learnability. While artists have been able to produce interesting work with the SchemePaint program, there is nothing in the program that assists the artist in learning about the Scheme language, the specific capabilities of the system, or, for that matter, the domain of graphics itself. Thus, an artist new to SchemePaint would have no clear pathway for learning about (say) the system's color-related procedures, or what kinds of interesting programmed effects those procedures might be able to produce. While design environments are strong at providing precisely this type of domain-specific help and information to the user, they too exhibit weaknesses characterized by the absence of a rich programming environment with which to create new extensions and abstractions. Programmable design environments, then, represent an attempt to augment programmable systems with techniques for providing domain-oriented assistance. Such an environment should include:

- A domain-enriched programming environment, in which the primitive 'building blocks' (procedures and data objects) reflect important concepts of the application domain. The programming language may be a brand new application-specific language, or it may be a domain-enriched variation on a general-purpose language (this latter path is the one taken in the SchemeChart system to be described shortly); reasonable arguments may be advanced in support of either decision. Whatever 'foundational' language is chosen for a given PDE, however, the primitives of that language should be written to allow users to write meaningful programs, within the application domain, in the space of a few lines of text.
- A direct manipulation interface supporting those tasks best performed by 'extralinguistic' means (such as drawing lines or shapes by hand).
- 'Scaffolding' elements, such as the critiquing mechanisms and catalogs characteristic of design environments. These should permit the user to learn aspects of both the application domain and the embedded language environment.

The relationship between this strategy of software design and the themes of 'alleviating complexity' described in the first section are worth drawing out. The purpose of the programming environment in a PDE is to provide a means by which feature explosion may be avoided. The user is able to build a personalized vocabulary of new abstractions over time, thus avoiding the frustrating scenario in which software designers toss in numerous *ad hoc* features in the (in any event hopeless) attempt to anticipate every possible task (cf. Eisenberg and Fischer [1994]; Nardi [1993]). The direct manipulation interface is thus limited to those tasks that seem to resist representation within the constraints of a formalized programming language, while the scaffolding elements are provided to alleviate the complexity of the language (by providing, for example, catalogs of modifiable examples) and, to some degree, the complexity of the domain (by providing critiquing mechanisms that alert the user to specific domain-related issues of which he or she may not be aware).

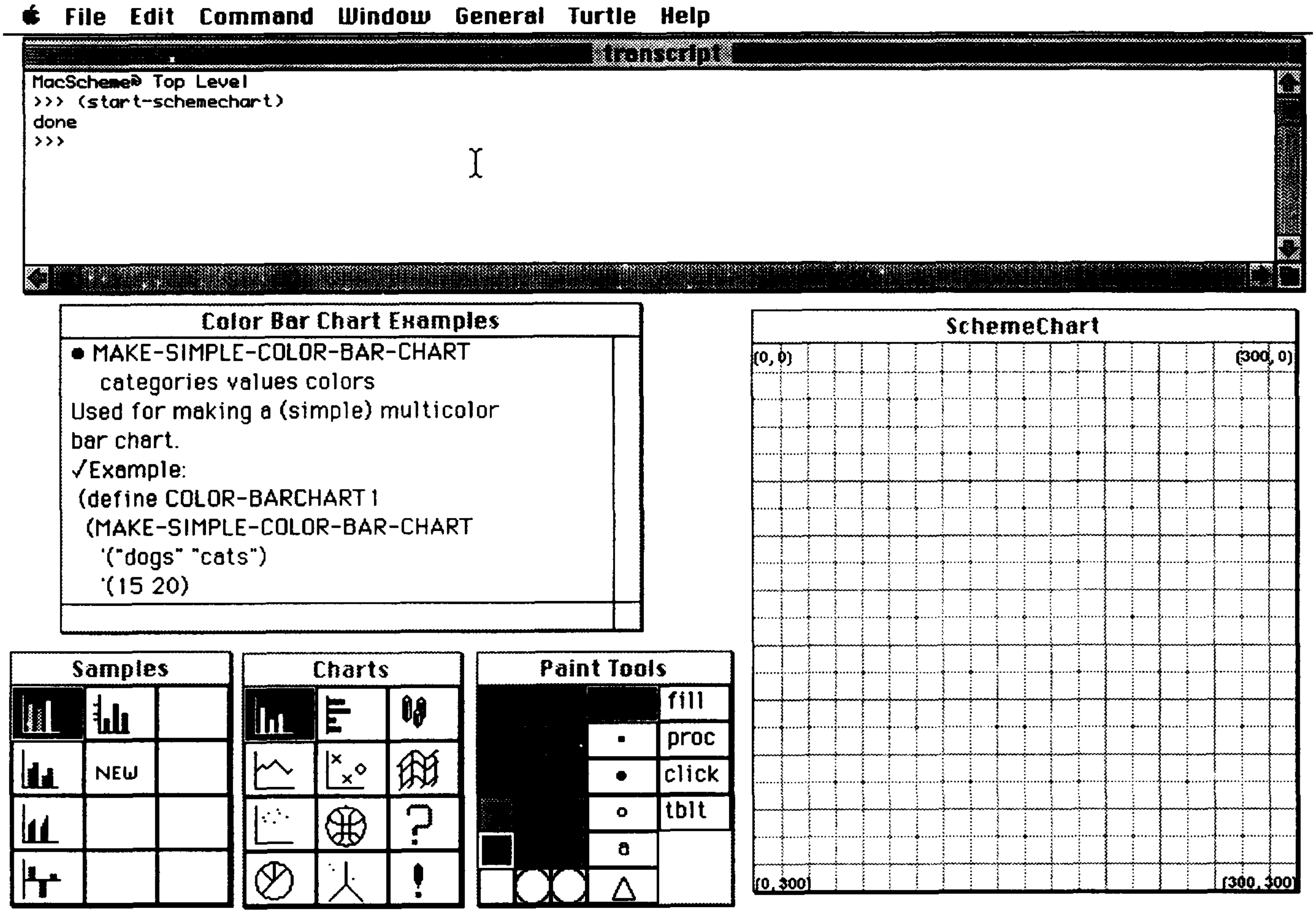
In the following section, these aspects of PDE design are illustrated through the presentation of a working prototype.

SCHEMECHART: PDE FOR CREATING CHARTS AND GRAPHS

SchemeChart, an application for the creation of charts and graphs, is a programmable design environment prototyped in the Human–Computer Communication Laboratory at the University of Colorado. The system is built in the MacScheme dialect of Lisp and it runs on all color Macintosh computers.

Figure 1 presents a view of the SchemeChart screen. When the application is run, the user is presented with a set of windows: the SchemeChart window (in which new charts will be created), the Paint Tools window permitting the user to select, among other parameters, the default pen color and pen width, the Charts window, presenting a variety of chart types from which to select (e.g. barcharts and line charts), and the Samples window, which is used to present specific varieties of the chart types selected through the Charts window. Finally, the transcript window presents an application-enriched Scheme interpreter.

Figure 1 actually shows a scenario in progress. Here, the user has selected the barcharts icon in the Charts window, and the system has responded by presenting a variety of specific barcharts in the Samples window: there are choices for multicolor barcharts, grouped barcharts, 'trapezoidal' charts and several others. The user has selected the multicolored chart icon (at the upper left in the Samples window). She is then, by a menu selection, able to bring up yet another window labeled Color Bar Chart Examples. This window contains a variety of SchemeChart sample procedures, with documentation, relevant to the creation of multicolored barcharts. For instance, the first procedure presented in the window (and visible in *Figure 1*) is the *make-simple-color-bar-chart* procedure. The user may now evaluate the sample expression directly and view the sample chart thus created, or she may edit the text in place and create a barchart based on the sample presented.



```
MacScheme® Top Level
>>> (start-schemechart)
done
>>>
```

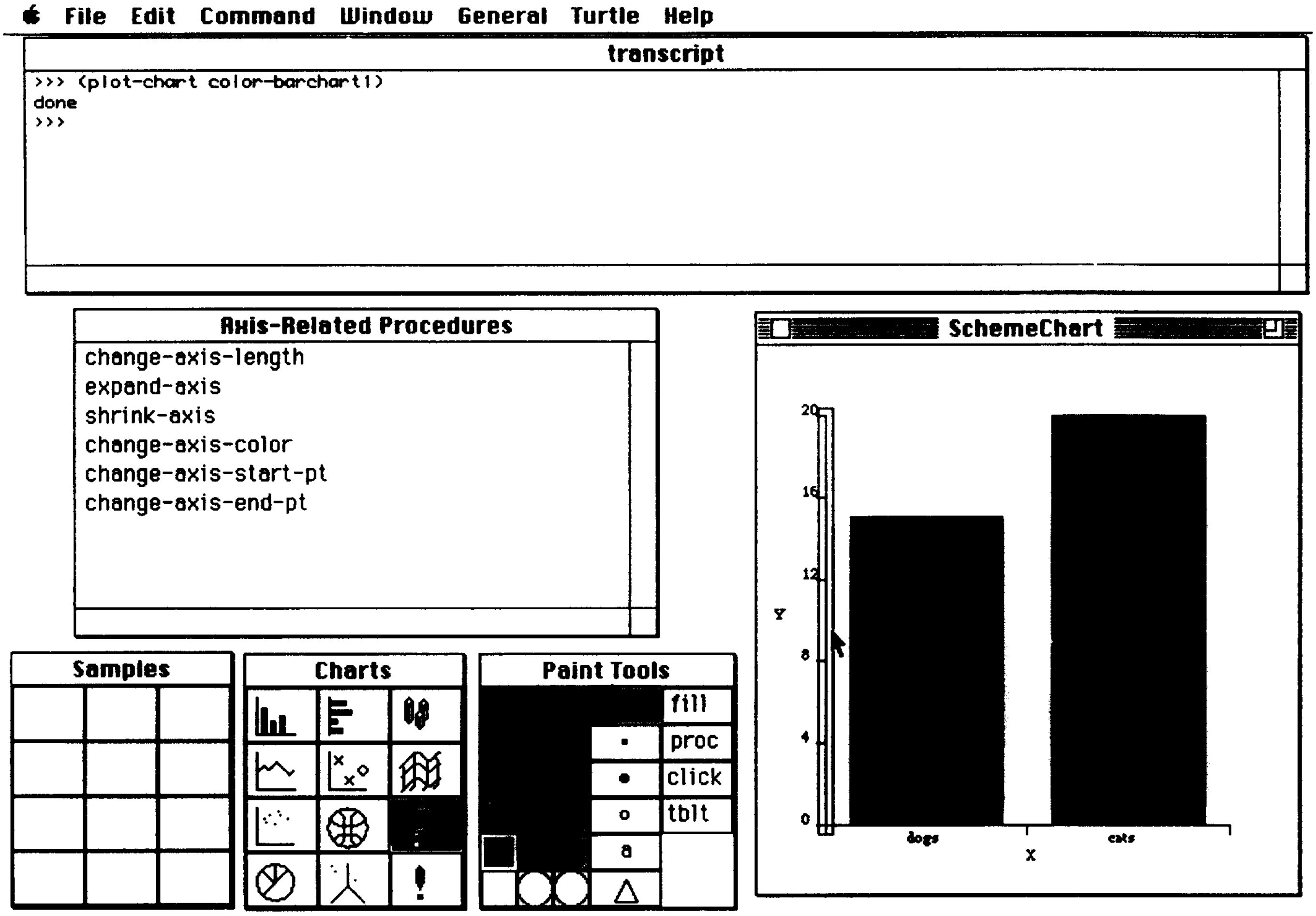
```
• MAKE-SIMPLE-COLOR-BAR-CHART
   categories values colors
Used for making a (simple) multicolor
bar chart.
✓Example:
 (define COLOR-BARCHART 1
  (MAKE-SIMPLE-COLOR-BAR-CHART
   '("dogs" "cats")
   '(15 20)
```

Figure 1 Screen view of SchemeChart application
[The windows are described in the text. Not shown is an optional window permitting the user to enter data elements in the familiar spreadsheet format.]

It is worth pausing at this juncture to call attention to several aspects of the scenario presented thus far. First, the system provides a means by which selected language examples may be located through the use of an iconic catalog. Thus, the presentation of the language is organized according to the kinds of tasks typically attempted in the application domain. The user is able to create new charts by simple variations of the procedures presented. This is similar in principle to the notion of 'learning programming by example modification' advocated by Lewis and Olson [1987] (see also the discussion in Nardi [1993], pp 69–71).

Another mechanism by which the SchemeChart user may become acquainted with the system's language constructs is shown in *Figure 2*. Here the user has selected 'query mode' (the question mark icon in the Charts window), which allows her to locate graph elements within an already-created graph, and to access procedures relevant to the creation or change of those graph elements. In *Figure 2*, the user has 'queried' the vertical axis of a newly created graph, and the system responds with a (still rather skeletal) list of procedures that are especially useful in altering chart axes. (This feature is very similar in design to the example-driven software-reuse system of Redmiles [1993].)

Because SchemeChart includes a fully fledged Scheme interpreter, the user is able to employ the 'domain-enriched' programming environment to write new chart-creation procedures. In fact, the SchemeChart system includes mechanisms by which such new procedures may be accessed in approximately the same fashion as the system-provided examples. *Figure 3* illustrates the idea. Here, the user has selected the box labeled 'New' in the Samples window. Having done this, she can bring up a new window with already-existing user-defined procedures (much like the sample-procedure window shown in *Figure 1*). The user now writes a brand-new barcharting procedure which automatically plots the charted values in colors that range from red (corresponding to the highest value) to blue (corresponding to the lowest); the intermediate values are plotted in colors 'between' red and blue (the RGB values of these colors are found by interpolation between those of red and blue). Once the user has created and debugged her new chart-creation procedure, she can save out the altered contents of the window; the next time she accesses the box labeled 'New' in the Samples window, her new barchart creation procedure will be among the sample procedures provided. This aspect of the SchemeChart system permits users to augment the 'browsable catalog' with their own creations. (Admittedly, many improvements to this process are still desirable. For example, the user should be able to quickly add new iconic elements to the Samples window as well.) For large-scale programs, this technique of adding elements to the catalog would become unwieldy; in such a case, the user can of course avail herself of the underlying Scheme programming



```
>>> (plot-chart color-barchart1)
done
>>>
```

change-axis-length
expand-axis
shrink-axis
change-axis-color
change-axis-start-pt
change-axis-end-pt

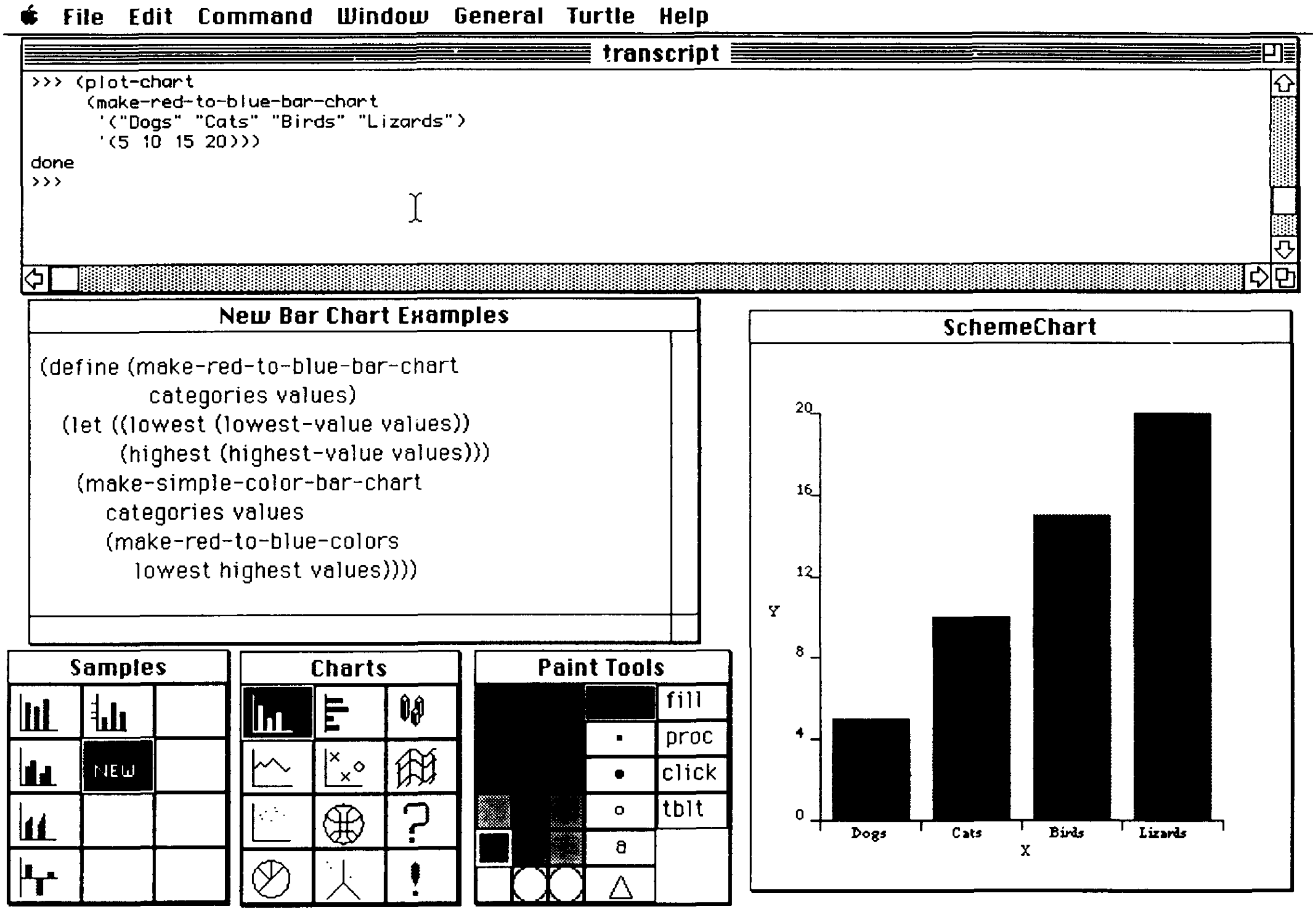
Figure 2 User has selected 'query mode' (question mark icon) from Charts window
[By dragging the mouse over a particular element of the newly created graph (in this case, the *y* axis), the user can access a list of language procedures relevant to the manipulation of this element.]

environment's file-management facilities (for saving, restoring, and editing files).

Figure 4 illustrates still another technique by which the SchemeChart system is able to provide a gentle introduction to its language component. The program simply presents, at random, a series of graphs (chosen from the available set built into the program) along with the language forms that generate those graphs. The effect is similar in spirit to a video game 'attractor mode', and it allows users to get a sense of the system's expressive range. This manner of presenting language primitives paired with the types of graphical effects that they create is similar in intent to educational software systems that present new (natural language) vocabulary items in 'contextualized' fashion, pairing words with scenarios in which those words are employed (cf. Sternberg *et al.* [1983]), and it is also similar, in its presentation of 'serendipitous' information, to 'did you know' facilities as described in Owen [1986].

Finally, the SchemeChart system includes a number of software 'critics' that work in the manner described in the previous section, i.e. as 'monitor' routines that alert the user to potential problems in her creations of specific charts, and that are intended to reflect 'standard' principles of chart design such as those presented in Kosslyn [1994], Tufte [1983] and MacGregor [1979]. In SchemeChart, each individual critic is linked to a particular set of chart-creation procedures. *Figure 5* depicts the critiquing mechanism at work. Here the user has created a barchart in which the values being graphed happen to be too distant to distinguish in standard barchart format (such charts are best at presenting moderate differences between data elements). In this case, once the user has created the new chart, the program responds by flashing the exclamation point 'critic alert' signal in the Charts window. The user may ignore the signal if she wishes, or, if curious, she may use a menu choice to bring up the particular critic message that the system has created for this instance. In *Figure 5*, the system presents a window in which the potential problem with the newly created barchart is cited. The user may now take the system's advice and change the graph so that it uses logarithmic values, or she may decide to reject the system's advice and simply proceed with the chart that she has created.

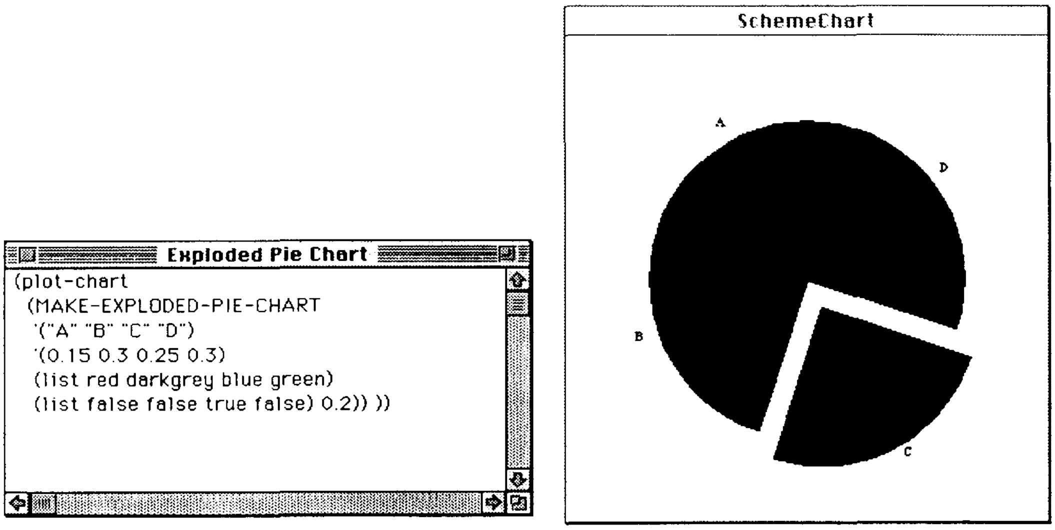
In sum, then, the SchemeChart system illustrates a number of experimental methods, typical of the PDE approach, that seek to alleviate problems of complexity along several dimensions. It includes an iconic catalog through which the user can access editable programming examples, a 'query mechanism' that permits the user to select elements of certain completed graphs and find language procedures relevant to those particular elements, an 'attractor mode' to help users encounter new vocabulary items by serendipity, and critiquing mechanisms that allow the user to encounter potentially unfamiliar domain-specific design concepts.



```
>>> (plot-chart
     (make-red-to-blue-bar-chart
      '("Dogs" "Cats" "Birds" "Lizards")
      '(5 10 15 20)))
done
>>>
```

```
(define (make-red-to-blue-bar-chart
          categories values)
  (let ((lowest (lowest-value values))
        (highest (highest-value values)))
    (make-simple-color-bar-chart
      categories values
      (make-red-to-blue-colors
        lowest highest values))))
```

Figure 3 User has written new procedure to create barcharts for which bar color is determined by interpolating between red (for the highest value) and blue (for the lowest)
[This new procedure may now be stored so that the next time the user accesses the New Bar Chart Examples window, she will retrieve the *make-red-to-blue-bar-chart* procedure (along with all the earlier-created user-defined procedures).]

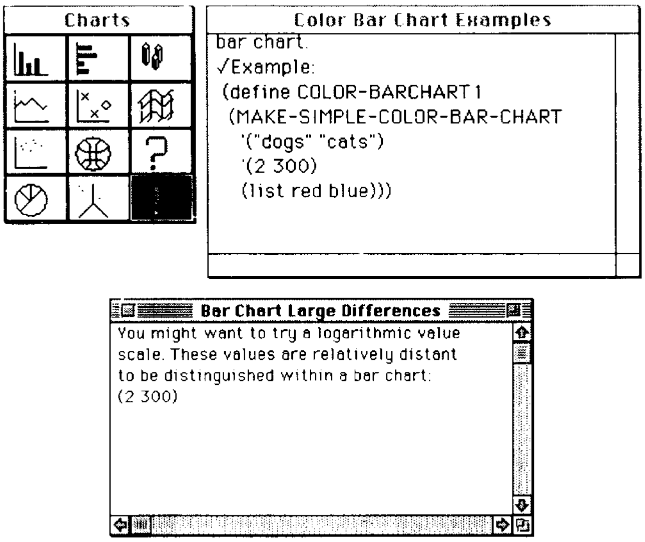


```
(plot-chart
  (MAKE-EXPLODED-PIE-CHART
   '("A" "B" "C" "D")
   '(0.15 0.3 0.25 0.3)
   (list red darkgrey blue green)
   (list false false true false) 0.2)) ))
```

Figure 4 ‘Attractor mode’ in SchemeChart
[The system presents a (randomly chosen) sample procedure from a preexisting set — this one for an ‘exploded’ pie chart. Left in this free-running mode, the system simply displays the available vocabulary and functionality.]

ONGOING AND RELATED WORK

The SchemeChart program may be fairly described as relatively early work in progress. Certainly, many quantitative improvements are needed in the program (more chart types, more (and more interesting) chart critics, better textual online documentation). We also intend to perform more extensive user testing with the system during the coming year (thus far, the system has been tested only briefly and informally with a set of five student volunteers). Additionally, portions of the SchemeChart system are currently being incorporated



```
bar chart.
√Example:
(define COLOR-BARCHART1
 (MAKE-SIMPLE-COLOR-BAR-CHART
  '("dogs" "cats")
  '(2 300)
  (list red blue)))
```

```
You might want to try a logarithmic value
scale. These values are relatively distant
to be distinguished within a bar chart:
(2 300)
```

Figure 5 Critiquing mechanism in SchemeChart
[The user has created a barchart representing widely disparate numerical values. The program signals the user (with a flashing exclamation point) that a potential problem has been spotted with this chart; the user may then bring up the critic window shown at the bottom of the figure.]

into a new version of SchemePaint to be used as the basis of an introductory computer science course to be taught for cognitive scientists during the coming year.

The SchemeChart development effort (more broadly, the entire notion of PDEs) shares numerous concerns with research into end-user programming. Certainly, the choice of Scheme as a 'base language' for such a system is a potentially controversial one. Much of the work in end-user programming focuses on providing alternative (and arguably more easily learned) language enviroments such as the Boxer system [diSessa and Abelson, 1986], or on relieving the user of the need to deal with programming syntax altogether through such means as visual programming or programming by demonstration (see the recent book by Nardi [1993], as well as compilations edited by Myers [1992] and Cypher [1993], for excellent discussions of many of these issues). In this context it is worth noting several points. First, there are good arguments for employing a domain-enriched general purpose language for applications, as opposed to a brand-new (e.g. visual) language. For instance, users familiar with the basic syntax of the general-purpose language can employ that familiarity across a range of applications, rather than learn a new language structure for each new application (see Eisenberg [1991] for further discussion along these lines). Second, we deliberately make no attempt to shield users from what we believe are powerful programming constructs such as iteration, procedure definition, and recursion, whereas such notions are often difficult to realize in visual or demonstrational systems. In this sense, the choice of Scheme reflects a desire to provide users with the full functionality of a complete programming environment. Additionally, the choice of Scheme likewise reflects the influence of the discussions in Abelson and Sussman with Sussman [1985], in which Scheme is used as a rich foundation for the construction of 'embedded languages' such as the one developed for SchemeChart.

Finally, it should be noted that it is unlikely that SchemeChart, as it stands, could ever act as a particularly good framework for introducing basic Scheme programming to users: in this sense, an ideal SchemeChart 'novice' is a person who has some familiarity with the core concepts of the Scheme interpreter, and who employs the system's special support features to become acquainted with the application-specific vocabulary of the system. In the terminology employed by diSessa [1991], the system thus assumes that users initially have a 'structural model' of Scheme, and that what they need for this particular application is a 'functional model' of the language extensions provided for charting and graphing. In order to use a PDE such as SchemeChart for introducing basic language concepts such as procedural definition, recursion, and data abstraction, additional online tutorial material would be required. For instance, specific chart examples might be suggested to the user for their tutorial value (e.g. the function-plotting procedures might be used to introduce the notion of procedural arguments). We are currently working on incorporating such tutorial material (using modules developed for SchemeChart) into the SchemePaint system, to render that system closer to a true 'instructional design environment'.

ACKNOWLEDGEMENTS

Chris DiGiano and Ed Post made important contributions to the work described in this paper. The author would also like to thank Hal Abelson, Andy diSessa, Wally Feurzeig, Gerhard Fischer, Paul Horwitz, and Gerald Sussman for especially helpful discussions and insight. The research described in this paper is partially supported by the US National Science Foundation (grant RED-9253425) and by an NSF Young Investigator grant (IRI-9258684).

REFERENCES

Abelson, H and Sussman, G with Sussman, J *Structure and Interpretation of Computer Programs* McGraw-Hill, 1985

Cypher, A (ed.) *Watch What I Do* MIT Press, USA, 1993

diSessa, A 'Local sciences: viewing the design of human-computer systems as cognitive science' *in* J M Carroll (ed.) *Designing Interaction: Psychology at the Human-Computer Interface* Cambridge University Press, UK, 1991

diSessa, A and Abelson, H 'Boxer: a reconstructible computational medium' *Communications ACM* 29:9 (1986) pp 859-868

Eisenberg, M 'Programmable applications: interpreter meets interface' *Technical Report 1325* Artificial Intelligence Laboratory, Massachusetts Institute of Technology, USA, 1991

Eisenberg, M 'SchemePaint: a programmable application for graphics' *Technical Report CU-CS-587-92* University of Colorado, USA, 1992

Eisenberg, M and Fischer, G 'Programmable design environments: integrating end-user programming with domain-oriented assistance' *CHI '94 Conference Proceedings* 1994, pp 431-437

Fischer, G 'Supporting learning on demand with design environments' *Proceedings International Conference Learning Sciences* Lawrence Erlbaum. 1991, pp 165-172

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* 7:3 (1992) pp 281-314

Fischer, G, Lemke, A C, Mastaglio, T and Morch, A 'The role of critiquing in cooperative problem solving' *ACM Transactions Information Systems* 9:2 (1991) pp 123-151

Kosslyn, S *Elements of Graph Design* W H Freeman, USA, 1994

Lewis, C and Olson, G 'Can principles of cognition lower the barriers to programming?' *Empirical Studies of Programmers: Second Workshop* Ablex, USA (1987) pp 248-263

MacGregor, A J *Graphics Simplified*, University of Toronto Press, Canada, 1979

Myers, B (ed.) *Languages for Developing User Interfaces* Jones and Bartlett, USA, 1992

Nardi, B *A Small Matter of Programming* MIT Press, USA, 1993

Owen, D 'Answers first, then questions' *in* D A Norman and S W Draper (eds.) *User Centered System Design, New Perspectives on Human–Computer Interaction* Lawrence Erlbaum, USA, 1986, pp 361–375 (Chapter 17)

Redmiles, D F 'Reducing the variability of programmers' performance through explained examples' *INTERCHI '93 Conference Proceedings* 1993, 67–73

Sternberg, R, Powell, J S and Kaye, D B 'Teaching vocabulary-building skills: a contextual approach' *in* A C Wilkinson (ed.) *Classroom Computers and Cognitive Science* Academic Press, USA, 1983

Tufte, E *The Visual Display of Quantitative Information* Graphics Press, USA, 1983