Combining Programming Languages and Direct Manipulation in Environments for Computational Science

Eric Blough
Michael Eisenberg
Department of Computer Science and Institute of Cognitive Science
University of Colorado at Boulder, Campus Box 430
Boulder, Colorado 80309-0430
{blough, duck}@cs.colorado.edu
(303) 492-8136, (303) 492-8091

ABSTRACT
Creating computational environments for scientists presents an unusual challenge to software designers. Computational scientists have the skills and motivation to explore models via programming, yet also have highly-developed qualitative visual skills (e.g., interpretation of plots). Unfortunately, software designers have traditionally considered programming and point-and-click interfaces to be mutually exclusive. We propose instead that the most expressive computational environments for scientists are those in which programming and direct manipulation are both present, each supplementing the other. We present several broad themes of interface-language integration, illustrating them with three prototype applications that we are developing to support specific research areas of computational science; and we extend these themes into promising paths for future exploration.

KEYWORDS: interactive programming environments, computational science, programmable applications, direct manipulation.

INTRODUCTION
That computers have revolutionized scientific practice has become something of a truism. Chemical synthesis is now routinely performed with the aid of powerful graphics workstations; complex simulations are used to investigate ecosystems, animal behavior, galaxy formation, molecular collisions, and indeed an almost uncountable number of phenomena of interest to scientists. Perhaps in no other broad field of human endeavor can it be said that the advent of computers has been so pervasive, or so triumphantly productive.

Nonetheless, in creating computational tools for scientists, software designers face a variety of provocative challenges. In the past decade, much of the most visible work in human-computer interaction has focused on the provision of interfaces geared either toward novices ("walk-up-and-use" systems), or on professional systems aimed at confirmed nonprogrammers. In this tradition of work, the provision of a direct manipulation interface [8,14] is typically seen as a corollary to the elimination of a programming environment; indeed, the former is often presented as a means of subsuming the functionality of the latter.

DIS 95 Ann Arbor MI USA © 1995 ACM 0-89791-673-5/95/08..$3.50

Our own belief — argued at greater length in [5] — is that direct manipulation and programming need not be viewed as disparate design strategies, locked in an inevitable struggle for the hearts and minds of users. Rather, direct manipulation techniques and elements of interactive programming environments are capable of symbiotic combination, the strengths of each supporting the lacunae in the other. Briefly, direct manipulation techniques excel at supporting those activities most easily achieved by hand-eye coordination, and associated with those commonsense or qualitative judgments that appear most difficult to formalize: searching for interesting patterns among data, arranging a screenful of simulable objects in "pleasing" fashion, identifying a region of interest in a graph, tuning a simulation parameter until the numerical model behaves in a "novel" way. Conversely, programming languages are best at building repeatable, editable, formal representations of complex phenomena: specifying the myriad parameters of a customized random walk, representing the behavior of a nonlinear oscillator, performing graph-theoretic analysis of a chemical mechanism.

The examples mentioned in the previous paragraph focus on scientific practice for a good reason: scientific computation is, we believe, an arena in which the combination of direct manipulation and programming techniques should prove especially fruitful. First, unlike many other domains in which users are often presumed (we believe unfairly) to adopt a posture of computerphobia, the scientist is typically portrayed as a person willing to use programming to create and investigate formal models. Thus, whereas for many domains direct manipulation is implicitly presented as the next-best-thing to programming, software designers cannot provide scientists with less powerful media simply by arguing that such media will relieve the scientist of the burden of working with formal notation. On the other hand, we needn't move to the other extreme of providing scientists only with programming environments; scientific practice is rife with just the sort of visual, qualitative, highly contextualized judgments for which direct manipulation provides the ideal support.

This paper addresses the design of software environments for computational science — environments that interweave techniques of direct manipulation with tools provided by interactive programming languages. We will address several themes of "interface-language integration" that we feel will become recurrent (if not central) issues in the ongoing design of computational systems for scientists. In presenting these themes, we will use several working prototype systems developed in our laboratory as a source of examples: these systems, all based on the Scheme dialect of Lisp, are scientific

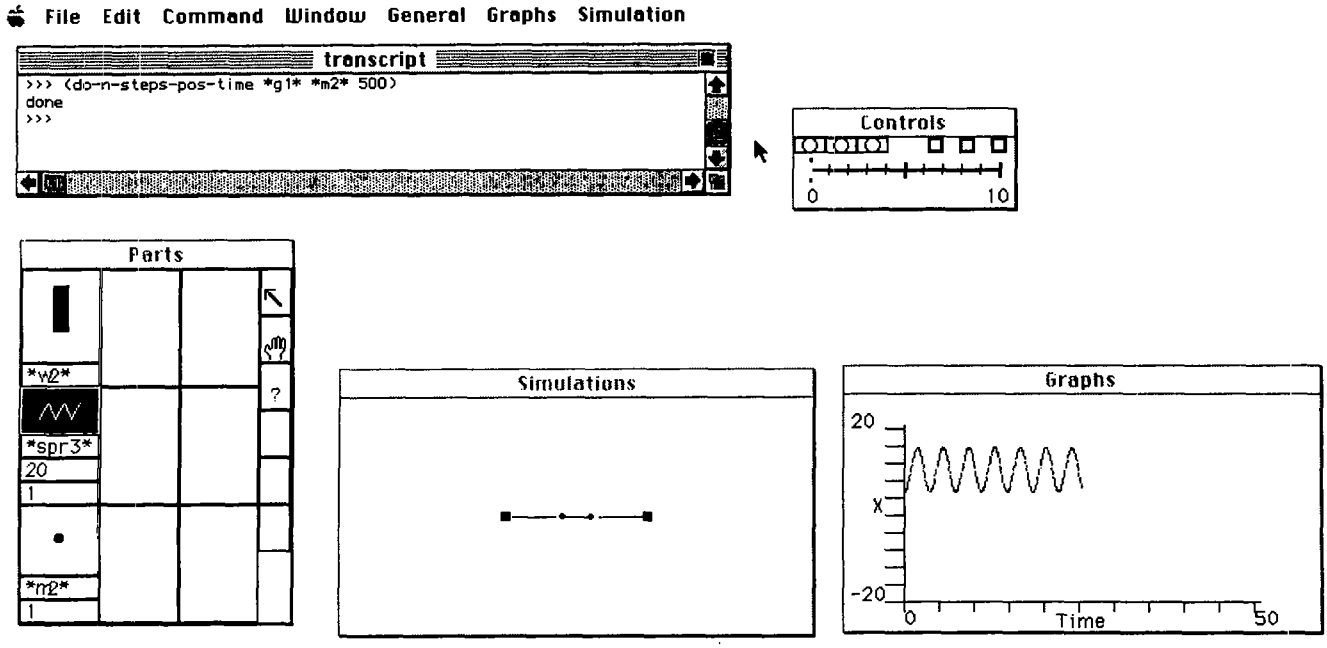

Figure 1. A view of the SchemeSprings screen in the course of a simulation of a two-mass coupled oscillator system. The various windows are explained in the text; the three menu titles visible at right at the top of the screen are specific to SchemeSprings, while the four textual titles at left are standard within the MacScheme system.

applications geared toward the domains of oscillators, chemical kinetics, and diffusion-limited aggregation.

The remainder of this paper is organized as follows: in the second section we briefly present the three experimental systems and their core functionality, while the third section describes major themes of interface-language integration, illustrating these themes with specific examples drawn from our applications. The fourth section discusses related work in scientific computation, and briefly contrasts the themes of this paper with those issues generally associated with "scientific visualization". We conclude by outlining those avenues that we feel are especially promising for future work in the creation of truly expressive environments for scientific computation.

LANGUAGE-BASED APPLICATIONS FOR SCIENTIFIC COMPUTATION: THREE PROTOTYPE SYSTEMS

This section describes, in outline form, our three prototype applications. In each case, the description is necessarily abbreviated, focusing on overall structure; in the remainder of the paper we will elaborate upon these descriptions with additional examples of the various systems' behaviors.

SchemeSprings

SchemeSprings is an application for creating and simulating complex systems of coupled oscillators. The application is written in MacScheme [S4] and runs on all color Macintosh computers. Figure 1 shows the SchemeSprings screen in the course of a scenario. The Simulations window is the window in which simulations are constructed by placing icons for "walls," "masses," and "springs" at appropriate positions; the Parts window is a palette from which these objects are chosen (it also includes, at right, icons for selecting, moving, and querying particular objects); the Graphs window displays numerical graphs of running simulations; and the Controls window offers programmable buttons and sliders that allow users to interact with running simulations. Finally, the transcript window allows the user access to a MacScheme interpreter that has been "enriched" with a variety of special-purpose functions and objects for working with oscillator systems.

Figure 1 depicts a typical SchemeSprings scenario. Here, the user has placed two "wall objects" on either end of a system of two "mass objects" linked by three springs. In this manner, relatively straightforward simulations of linear oscillator systems may be constructed simply by choosing objects from the Parts window (stationary walls, linear springs, and constant masses); these simulations may then be run (using a fourth-order Runge Kutta integrator) and numerical results of various types plotted in the Graphs window. On the other hand, by using the SchemeSprings language, it is relatively straightforward to create more elaborate customized simulations. For example, suppose that we wish to create a spring whose restoring force is nonlinear: in this instance we can use a built-in SchemeSprings procedure named make-pure-difference-spring. This procedure takes as arguments a "natural length" for the spring and a procedure which, when called on the difference between the spring's current and natural length returns a value for the magnitude of the spring's restoring force. Thus, to create a spring object whose natural length is 10 and whose restoring force is dictated by a fifth-order polynomial, we could write the following SchemeSprings expression:

```
(define *fifth-order-spring*
  (make-pure-difference-spring 10
    (lambda (difference)
      (+ (* -0.0005 (expt difference 5))
         (* 0.008 (expt difference 3))
         (* 2 difference)))))
```

This spring object could now be incorporated (via language expressions) into simulations of the sort shown in Figure 1; by similar means it is possible to make springs whose restoring

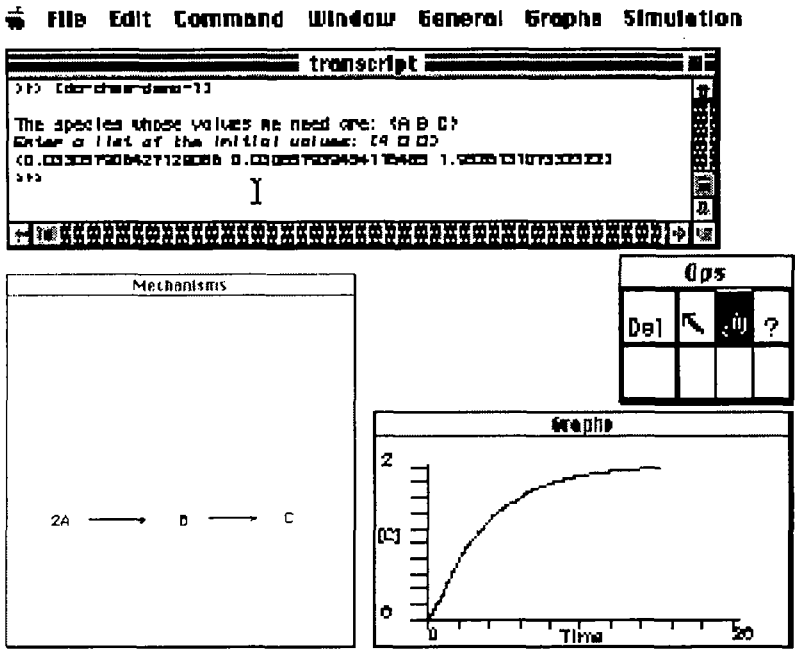

Figure 2. The SchemeReactions system. Here the user has entered a simple two-step reaction sequence as the mechanism to simulate; this is shown graphically in the Mechanisms window at bottom left (the mechanism has been arranged on the screen by employing the "hand" icon in the Ops window). Subsequently, the mechanism has been simulated and the growing concentration of species C plotted in the Graphs window.

force depends on time, or varies randomly over a specified range, among innumerable other possibilities. Likewise, it is possible to create masses whose mass varies as a function of time or position; global forces (such as gravity) that vary in some systematic way (e.g., one could create a sinusoidally varying gravitational force); and many other customizations.

SchemeReactions

SchemeReactions is an application for simulation of chemical reaction mechanisms. The fundamental activity in this sort of simulation is the creation of systems of so-called "elementary steps" (usually assumed to represent molecular collisions or rearrangements) whose combination gives rise to larger-scale observed chemical reactions. Thus, to take a particular example, the system of elementary reactions

2A --> B
B --> C

might be proposed as a model for the overall observed laboratory reaction 2A --> C. In general, modeling of elementary steps employs what is known as "mass action kinetics," meaning that each elementary reaction corresponds to an ordinary differential equation whose terms are derived in a straightforward fashion from the reactants and products of the individual step; thus, the first of the two reactions above would be modeled by an equation of the form:

$$d[B]/dt = k\,[A][A]$$

Here, the rate of production of species B (due to the first reaction) is a constant (the so-called rate constant) k times the square of the concentration of A; in physical terms, we are stipulating that the first reaction will produce new molecules of B for some constant proportion of collision events between two molecules of A [10].

It is not hard to imagine that the behavior of large systems of such ordinary nonlinear differential equations can become, in principle, quite complex. Indeed, models of this form — augmented by additions such as external "sources" of reactants — have been used to explain the behavior of a variety of oscillating and even chaotic chemical reactions [11,13]. Thus, the SchemeReactions user is capable of creating and simulating extremely complex mechanisms.

Predla

Diffusion-limited aggregation (DLA) is a technique for the exploration of aggregation processes, and is used to model such phenomena as electrodeposition and crystal growth [12, 15, 17]. The basic model is simple in concept: particles diffuse randomly through a medium and stick irreversibly to a growing cluster. Computationally, this is represented by placing a seed particle at the origin and then allowing single particles, released one at a time, to perform a random walk until they are adjacent to the cluster and thus stick to it. Numerous variations on this model exist — in types of

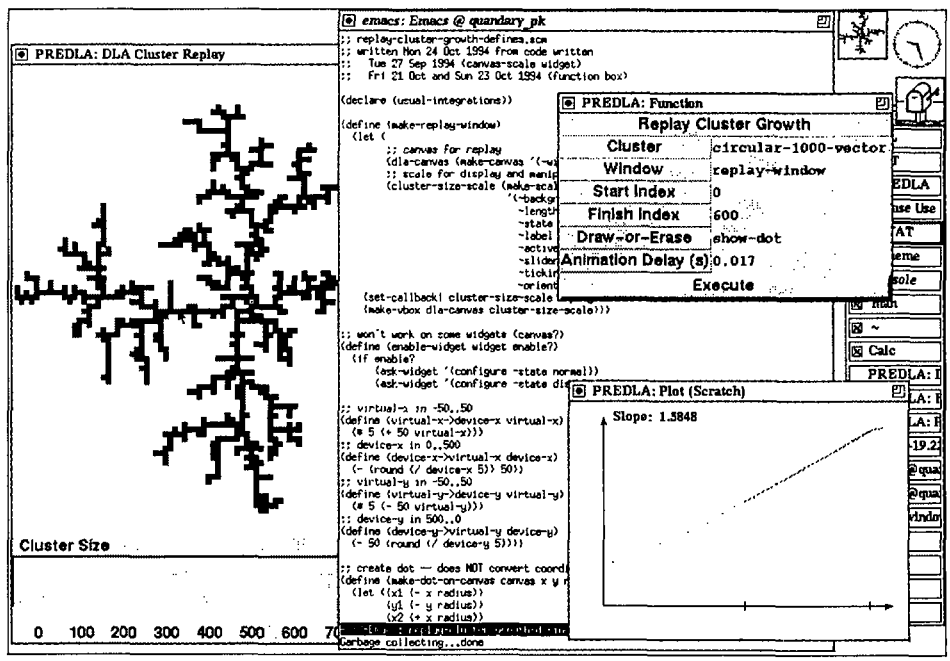

Figure 3. A view of the PREDLA screen, illustrating selection of part of a cluster, a plot that changes with the size of the cluster, and a dialog box for "re-playing" the growth of the cluster.

particles, constraints on walking and sticking, and so on.

PREDLA (Programmable Research Environment for Diffusion-Limited Aggregation, shown in Figure 3), provides support for the interactive construction and execution of such DLA simulations, and analysis of the resulting clusters. Simulations can be constructed from pre-existing sample data structures (e.g., particles, lattices) and functions (e.g., random walks, sticking functions); alternatively the user can modify these elements or add new ones. The clusters produced by running simulations can be displayed (either in their final states or earlier ones); and interesting numerical properties of the clusters may be plotted.

INTEGRATING DIRECT MANIPULATION AND PROGRAMMING IN SCIENTIFIC COMPUTATION: THEMES FOR DESIGN

In this section, we explore several major themes that we believe are important for the development of language-based applications for scientists, using our three prototype applications as sources of examples.

Linking Language Elements to Direct Manipulation Operations

Perhaps the most straightforward means of integrating direct manipulation and programming is to endow direct manipulation tools — buttons, sliders, and so forth — with the capability of executing user-written procedures. This sort of functionality is a staple of interface-building tools such as HyperCard [S1]; but for our purposes, the crucial point is that in order to be of use to computational scientists, such interface-building tools must be accompanied by languages rich enough to accommodate powerful domain-specific constructs. For example, in the SchemeSprings system, it is possible to create a cubic nonlinear spring for which the cubic term is determined by the value of a global variable `*cubic-constant*`:

```
(define *cubic-spring*
  (make-pure-difference-spring
    10
    (lambda (diff)
      (+ (* *cubic-constant* diff diff diff)
         (* 2 diff)))))
```

Here, the linear term in the spring is unchanging, while the cubic term depends on the particular value of the global variable. Now, by linking the value of the slider in the Controls window in SchemeSprings to the value of `*cubic-constant*`, we can create a simulation in which the nonlinearity of the spring being simulated is directly under the user's control. The expression for performing this linkage would be written as follows:

```
(set-slider-procedures!
  (lambda (val)
    (set! *cubic-constant* (* 0.04 val)))
  (lambda (val) '()))
```

Briefly, the intent of this expression is to link a procedure to the movement of the Controls panel slider; when the slider is moved, the value of `*cubic-constant*` is set to a value determined by the product of 0.04 and the slider value (since the slider value ranges from 0 to 10, the cubic coefficient of the spring will thus vary from 0 to 0.4).[1] Figure 4 illustrates the continuation of this scenario. Once the simulation is run, the user can now move the slider back and forth, effectively "tuning" the spring's nonlinear term. Again, the point of this example is not merely to suggest that interface-building kits are useful to scientists; rather, it is only by combining those kits with constructs such as `make-pure-difference-spring` (i.e., language constructs geared toward particular scientific domains) that the power of such kits can be realized in full.

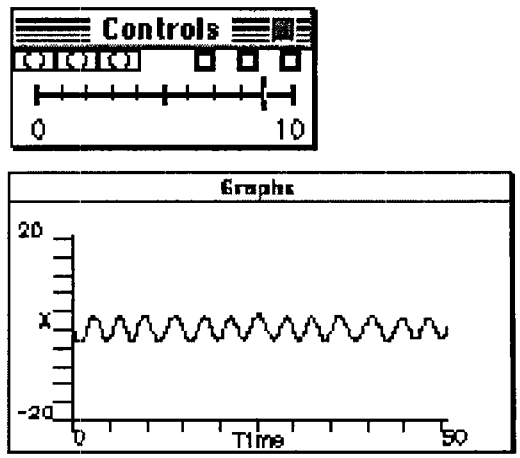

Figure 4. A simulation is performed while the slider is used to "tune" the value of a spring's cubic coefficient (the irregular oscillation of the mass at the end of this spring, shown in the Graphs window at bottom, is a result of varying the coefficient as the simulation proceeds).

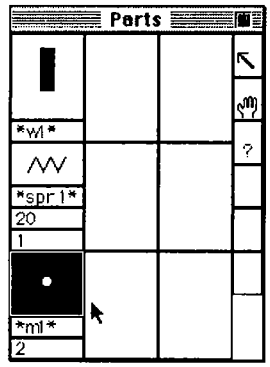

Figure 5. The user has selected the mass icon. When placed the object will have a mass of 2 units, and will be accessible by the name `*ml*` in the interpreter.

Going a bit further with this notion, we can in fact augment standard direct manipulation constructs with richer language-related behavior than is usually seen in interface-building kits. The SchemeSprings Parts window (see Figure 5), for instance, includes elements for placing new walls, springs, and masses; and the icons for these elements are accompanied by boxes in which the user can type in not only important values for these

[1] The second argument in the Scheme expression is a procedure to be called when the slider is released; in this case, that second procedure simply returns an empty list and thus has no effect on the simulation.

parts (e.g., the natural length of a spring) but additionally can type in the name by which the system's language will henceforth refer to the object. Thus, Figure 5 shows the user selecting the mass icon: she may now place this mass on the screen (using the mouse to dictate position) at which point it will have a mass of 2 and a name `*m1*`. The key point of this example is not so much technical as attitudinal: here, a direct manipulation construct is seen not as a device for hiding programming concepts as it is for accompanying those concepts. The name `*m1*` in our example is of use only if the scientist chooses to employ the system's language to refer to the newly-placed object; as designers of scientific environments, we therefore assume that the interface should be tightly coupled to the user's employment of (and familiarity with) the system's language elements.

Visual Interaction with Models, Simulations, and Graphs

A second broad theme by which language and interface may interact involves the visual specification, by the user, of "interesting" elements of simulations: elements chosen from the pictorial representation of a model to be simulated, from the pictorial results of the simulation itself, or from a numerical plot.

In both the SchemeSprings and SchemeReactions systems, straightforward means are available for interacting visually with structural models — the arrangement of masses and springs in the former case, the graphical representation of the mechanism in the latter. A "selection" facility (represented by an arrow icon in both applications) may be used to select individual simulation elements (e.g., a mass or wall in SchemeSprings; a set of reactants or products in SchemeReactions); performing such a selection automatically assigns a default name to the selected entity (e.g., `*last-selected-reaction-object*` in the case of SchemeReactions).

A somewhat more interesting example of integrating programming constructs with visual interaction is the notion of adding "advice" to regions of graphs, implemented in the SchemeSprings system. This facility allows the user to sketch out, by eye, "regions of special attention" on graphs: the idea is that the user might intend a region of the simulation graph to be associated with a particular program behavior (that is, if the simulation ever causes points to be plotted within that region, then some action should be taken by the program at that time). Suppose, for instance, that the user has set up a simulation of a spring/mass system, and feels that if the mass ever attains an x-coordinate less than -10 or so, the program should signal an alarm condition (in the simplest case, by printing the word "ALARM" on the transcript window). In this case the user can add an "advice procedure" to the graph by using the following expression (here, the graph and the "advice procedure" are arguments to the primitive `add-read-in-advice-polygon-to-graph!`):

```
(add-read-in-advice-polygon-to-graph!
  *g1* (lambda (val g) (display "ALARM!")))
```

The user is now prompted to draw a polygon on the Graphs window; he responds by producing the trapezoidal region seen in Figure 6; note that as the simulation proceeds, the mass does indeed enter the "alarm region".

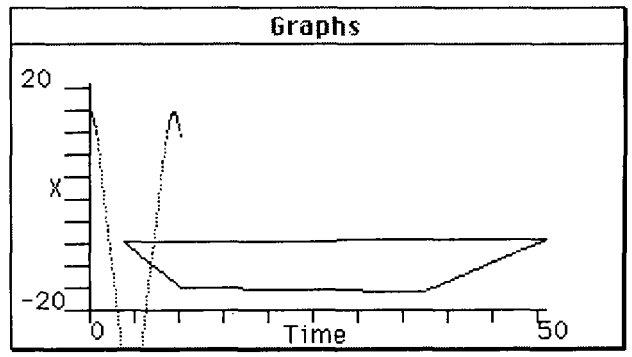

Figure 6. A simulation in which a trapezoidal "alarm polygon" has been drawn by hand on the graph. When the simulation plot enters the polygon, some user-defined action is taken.

In the course of the simulation, every point plotted within the alarm region causes the system to print out the stipulated alarm message:

```
>>> (do-n-steps-pos-time *g1* *m1* 200)
ALARM!
ALARM!
ALARM!
ALARM!
ALARM!
ALARM!
done
```

In addition to "visual" interaction with graphs and structural models, it is potentially useful to interact with the actual results of simulations. In PREDLA, the cluster resulting from a simulation is not simply a bitmap, but is rather a graphical depiction of an underlying (and user-accessible) data structure. Selecting a particle or group of particles with the mouse is thus equivalent to selecting part of this data structure, and the selected element may be passed as an argument to a user-specified function. Figure 7 illustrates the idea: here, a researcher selects a branch of a DLA cluster by clicking on a particle on the trunk of that branch. This mouse action in turn is linked to a user-specified function which in this example traces the cluster out to the tips. In the figure, the effect of this function is to highlight the selected particles, but other straightforward effects could include (e.g.) storing the particles in a data structure and analyzing the resulting "subtree."

Qualitative/Symbolic Programming Constructs

In the previous examples, the particular programming constructs employed were in some sense "standard" numerical elements of scientific computation — e.g., procedures for constructing nonlinear springs, for plotting numerical results, and so forth. Scientific computation, however, may also profitably incorporate symbolic constructs; and these too may interact in creative ways with direct manipulation operations. As a brief example of this notion, we might consider the following simple reaction mechanism as constructed in the SchemeReactions system:

2A <--> B <--> C

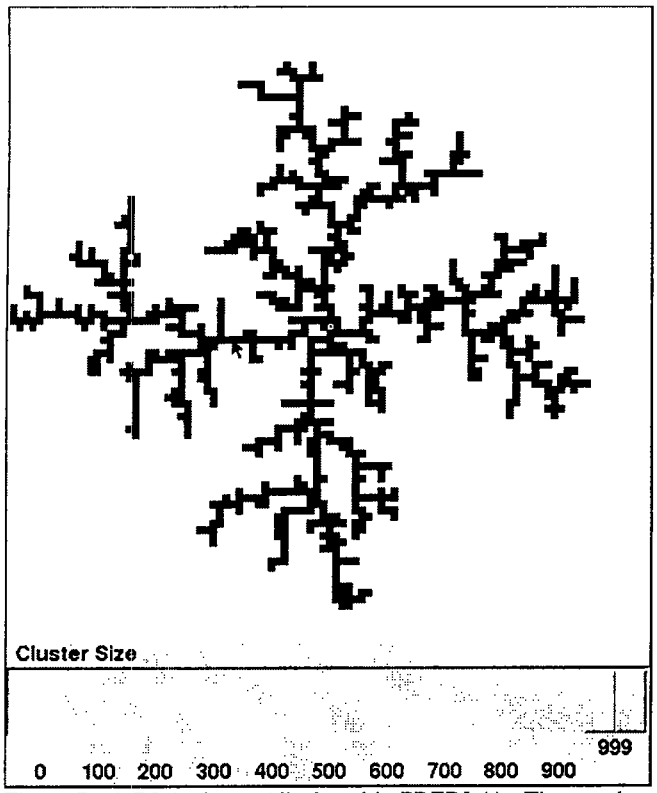

Figure 7. A DLA cluster (displayed in PREDLA). The user has clicked on a particle in the cluster. In this example, she has linked mouse selection to a function which traces out to the tips of the cluster, and thus all the particles in the branch are highlighted.

It is not hard for a chemist to look at such a reaction system and determine, merely by the qualitative structure of the mechanism, that (under normal assumptions of kinetics) this system must reach a unique equilibrium state determined by the initial concentrations of A, B, and C. The SchemeReactions system in fact incorporates a variety of graph-theoretic constructs that are useful for determining whether certain mechanisms — such as the one above — will necessarily reach an equilibrium state. (These constructs incorporate ideas articulated by Feinberg and his colleagues [6]; cf. also the description of similar constructs in the Kineticist's Workbench program [3,4].) Thus, once the user has constructed the mechanism above, he may perform a graphical analysis upon it; a portion of the program's response is shown below:

```
Deficiency: 0
Weakly Reversible? :#t

Rule 2.1: If the deficiency of the mechanism
          is 0 and the mechanism is weakly
          reversible, then the mechanism must
          reach equilibrium at positive (non-
          zero) concentrations of all species.
```

Loosely, the "deficiency" of the mechanism is a graph-theoretic quantity indicating, in certain cases, whether this mechanism is a "simple" or "typical" one (i.e., a "zero-deficiency" mechanism, as in the present case). The program's analysis has thus informed the scientist that this mechanism must necessarily reach equilibrium with positive concentrations of all species. The scientist might now wish to explore what would happen if a particular reaction were dropped from the current mechanism; this can be accomplished via direct manipulation (using the "Del" choice, for "delete", from the Ops window shown in Figure 2). The scientist selects the tip of the reaction arrow that he wishes to delete in the Mechanisms window, and the new (smaller) mechanism is redisplayed. Thus, if the deleted reaction arrow corresponds to the reaction B--> C, then the new mechanism would be written as:

2A <--> B <-- C

Now, the program's analysis is different than before:

```
Deficiency: 0
Weakly Reversible? :#f

Rule 2.3: If the deficiency of the mechanism
          is 0, and the mechanism is not weakly
          reversible, then the mechanism cannot
          reach equilibrium at positive
          concentrations of all species.
```

RELATED WORK; FUTURE DIRECTIONS

There are a number of systems available to computational scientists for simulation and data analysis. Programming environments such as Mathematica [S5], MATLAB [S6], and IDL (Interactive Data Language) [S2] provide powerful built-in functionality and sophisticated graphical presentations; while visualization tools such as LinkWinds [9], AVS [16], and apE [2] are geared specifically for image display, and moreover have significant features for composing data analysis functions.

All of these popular systems are both useful and powerful; but arguably they have not proceeded very far towards the sort of techniques discussed in the previous section. Mathematica, perhaps the most widely known, supports the construction of "transcripts," which can contain executable Mathematica statements as well as pictures and text. While these transcripts are quite useful (especially for presentations and tutorials), they do not support an especially tight integration between direct manipulation techniques and the Mathematica language. In contrast, IDL has roughly the same support for interaction as the versions of Scheme we are using in our prototypes — namely, the ability to construct interface tools and handle interaction events. But IDL, like Mathematica, is conceived as a general-purpose environment. Although it would be possible to create domain-specific applications within IDL (much as we have done in Scheme), realizing the techniques of language/interface integration for specific domains would pose much the same challenge in IDL as it does in Scheme.

It should also be mentioned that the techniques that we have focused on in this paper are distinct from those often associated with the field of "scientific visualization." Typically, "visualization" implies creative or dynamic rendering of data produced within (relatively standard) programming environments. In contrast, our concerns focus upon the enhancement of the computational environments themselves; and our systems focus on supporting interaction with data both during and after the simulation process.

As to future work, we see two important directions for development: the first involves the use of qualitative or "visual" primitives to catalog and retrieve scientific data; the

second involves the integration of textual and visual elements within scientific programming languages. In the remainder of this section, we outline each of these directions in turn.

Qualitative databases

The discussion in Section 3.3 above suggests means by which symbolic and numerical computation may be linked in environments for scientific computation — and likewise emphasized the notion that this integration was further encouraged by the philosophy of linking direct manipulation and programming. That particular example was, however, rather limited in scope, focusing only on the manner in which computation over graphical data structures could enhance the interpretation of numerical results. A more ambitious (and perhaps more promising) prospect would take into account the often unacknowledged styles of qualitative reasoning that pervade scientific research.

Consider, for instance, the case of a scientist who happens to produce a DLA cluster like that shown in Figure 7 earlier. It would not be improbable for the scientist to compare this cluster, mentally, with the results of earlier simulations; and the nature of this comparison might well proceed in terms that intermingle both numerical results (the fractal dimension of the cluster, the statistical distribution of types of branches, and so forth) with qualitative or visual aspects of the simulation (how "spread out" the cluster appears, patterns of distinct colors that appear among the cluster particles, or even perhaps how unexpected or unusual the results of the simulation happened to be). It would be desirable, then, to enable computational environments for scientists to employ both numerical and qualitative concepts in the process of analyzing, annotating, and retrieving results; in effect, the argument is that we would like to maintain "qualitative/quantitative databases" within our applications. As with standard database systems, it should be possible for users to write procedures that access the databases according to possibly complex patterns — but in this case, the relevant patterns would employ both qualitative and numerical constructs. In principle, then, it should be possible for the scientist to ask the system to retrieve (e.g.) all DLA clusters that "remind" the program of some particular target, and for the program to employ a variety of dimensions along which the "reminding" can take place.

Conceivably, qualitative information can be derived by a program using techniques similar to those employed in machine vision (cf. the fascinating work of Yip, whose KAM program [18] makes use of visual primitives to direct the autonomous exploration of Hamiltonian systems); thus, the program might in fact translate notions such as "the spread out nature of a DLA cluster" into algorithmic or numerical terms (cf. [1]). The program would then use these internal representations of qualitative concepts to direct its database search. Quite plausibly, the system might also make use of symbolic annotations directly requested of the scientist (e.g., how unusual or interesting this example might be). In either event, this type of extended database usage could make effective use of both direct manipulation and programming constructs: to take a hypothetical scenario, we might imagine that the scientist could select a region of a DLA cluster by eye and ask the system to retrieve all previously generated DLA clusters within a certain class that (in effect) "look like" the selected region.

Integrating Visual and Textual Language Elements

Our current environments employ "domain-enriched" languages; but these languages (based upon "standard" Scheme dialects) are, admittedly, text-based. It would be worthwhile to explore the use of visual, manipulable objects as program elements. For example, we could imagine a new version of PREDLA in which some of the data structures and function names are pictures; these pictures would be but reduced-yet-recognizable versions of (e.g.) portions of the growing cluster visible on the screen, and would be linked to the same data structures. Pictorial elements of clusters, in this scenario, might be selected by mouse on the screen, and "dropped" into the textual body of a written program.

Even in this scenario, the basic structure of the program is still text-based; we could carry this idea further by representing (say) the control flow of a program via spatial or pictorial means, as is often done in visual programming language environments [7] (cf. also LabVIEW [S3]). Our own feeling is that graphical representations should not be employed to the exclusion of text; rather, pictures and text — like direct manipulation and programming — have distinct strengths that are capable of complementing one another.

ACKNOWLEDGMENTS

This work has benefited from the guidance, scholarship, and support of numerous individuals. Liz Bradley and Scott Peckham have been gracious collaborators, working on projects involving our environments. We would also like to thank Hal Abelson, Gerhard Fischer, Paul Horwitz, Gerald Sussman, the MIT Scheme Team, and the members of the Center for LifeLong Learning and Design at the University of Colorado for their conversation and creative ideas. This work has been supported by the National Science Foundation under grant no. IRI-9210324. Additionally, the second author is supported by an NSF Young Investigator award IRI-9258684. Finally, we would like to thank Apple Computer Corporation and Digital Equipment Corporation for their generous donations of equipment.

SOFTWARE

S1. HyperCard. Apple Computer, Inc., Cupertino, CA.

S2. IDL. Research Systems, Inc., Boulder, CO.

S3. LabVIEW. National Instruments, Inc. Austin, TX.

S4. MacScheme. Lightship Software, Inc., Palo Alto, CA.

S5. Mathematica. Wolfram Research, Inc., Champaign, IL.

S6. MATLAB. The MathWorks, Natick, MA.

S7. MIT Scheme. Massachusetts Institute of Technology, Cambridge, MA.

REFERENCES

1. Abelson, H., Eisenberg, M., Halfant, M., Katzenelson, J., Sacks, E., Sussman, G.J., Wisdom, J., and Yip, K. Intelligence in Scientific Computation. Communications of the ACM 32,5 (1989), pp. 546-562.

2 Dyer, D.S. A Dataflow Toolkit for Visualization. IEEE Computer Graphics and Applications 10,4 (July 1990), pp. 60-69.

3 Eisenberg, M. Combining Qualitative and Quantitative Techniques in the Simulation of Chemical Reaction Mechanisms, in W. Webster and R. Uttamsingh (eds.), AI and Simulation: Theory and Applications (Simulation Series Vol. 22, No. 3.). Society for Computer Simulation, San Diego, CA, 1990, pp. 233-242.

4 Eisenberg, M. The Kineticist's Workbench: Combining Symbolic and Numerical Methods in the Simulation of Chemical Reaction Mechanisms. MIT AI Laboratory Technical Report No. AI-TR 1306, June 1991.

5 Eisenberg, M. Programmable Applications: Interpreter Meets Interface. SIGCHI Bulletin 27, 2 (April 1995).

6 Feinberg, M. Chemical Oscillations, Multiple Equilibria, and Reaction Network Structure, in W.E. Stewart, W.H. Ray, and C.C. Conley (eds.), Dynamics and Modeling of Reactive Systems. Academic Press, New York, 1980.

7 Glinert, E. Visual Programming Environments: Paradigms and Systems. IEEE Computer Society Press, 1990.

8 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.

9 Jacobson, A.S., Berkin, A.L., and Orton, M.N. Linkwinds: Interactive Scientific Data Analysis and Visualization. Communications of the ACM 37, 4 (April 1994), pp. 42-52.

10 Laidler, K.J. Chemical Kinetics (third edition). Harper and Row, New York, 1987.

11 Nicolis, G. and Prigogine I. Self-Organization in Nonequilibrium Systems. John Wiley and Sons, New York, 1977.

12 Paranjpe, A.S., Bhakay-Tamhane, S., and Vasan, M.B. Two-Dimensional Fractal Growth by Diffusion-Limited Aggregation of Copper. Physics Letters A 140, 4 (18 Septmber 1989), pp.193-196.

13 Rössler, O. E. Chaos and Strange Attractors in Chemical Kinetics, in A. Pacault and C. Vidal (eds.), Synergetics: Far from Equilibrium. Springer-Verlag, Berlin,1979.

14 Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages. IEEE Computer 16, 8 (1983), pp. 57-69.

15 Stanley, H.E., Bunde, A., Havlin, S., Lee, J., Roman, E., and Schwarzer, S. Dynamic Mechanisms of Disorderly Growth: Recent Approaches to Understanding Diffusion-Limited Aggregation. Physica A 168 (1990), pp. 23-48.

16 Upson, C., Faulhaber, T., Jr., Kamins, D., Laidlaw, D., Schlegel, D., Vroom, J., Gurwitz, R., and van Dam, A. The Application Visualization System: A Computational Environment for Scientific Visualization. IEEE Computer Graphics and Applications 9, 4 (July 1989), pp. 30-41.

17 Witten, T., and Sander, L. Diffusion-Limited Aggregation, a Kinetic Critical Phenomenon. Physical Review Letters 47, 19 (1981), p. 1400.

18 Yip, K. (1991) KAM: a system for intelligently guiding numerical experimentation by computer. MIT Press, Cambridge, MA, 1991.