

# STEERING PROGRAMS VIA TIME TRAVEL

J. W. Atwood, Jr., M. M. Burnett\*, R. A. Walpole, E. M. Wilcox, and S. Yang

Department of Computer Science, Oregon State University  
Corvallis, Oregon 97331-3202 USA

E-mail: {atwoodj,burnett,walpolr,wilcoxe,yang}@research.cs.orst.edu

## Abstract

*The environments programmers traditionally use for problem-solving—with separate modes and tools for writing, compiling, testing, visualizing, and debugging—derive their basic structure from historical accident, and take little advantage of Human Computer Interaction (HCI) research into the cognitive issues of programming. We believe that neglect of these issues impedes programmers' ability to produce reliable, maintainable software. Visual programming languages (VPLs) have begun to address this problem by creating more flexible, less modal programming environments, and we have taken a step further in this direction. In this paper, we describe a VPL in which programmers can modelessly steer as they specify, visualize, explore, and alter the behavior of a program while traveling through the program's logical time. This approach supports two often-neglected cognitive principles that HCI research shows can help programmers in their problem-solving.*

## 1. Introduction

Historically, programming environments were structured around the evolution of programming tools. Programmers in these environments worked in highly modal fashion, memorizing results and repeating steps as they switched among separate tools for editing, compiling, testing, debugging, and visualizing. Today's integrated approaches improve on this to some degree, but they still integrate only a few of these tools. For example, syntax-directed editors integrate part of compilation with editing; visual debuggers integrate part of visualization with debugging; interpreters integrate compiling with testing; and integrated programming environments retain the modality but allow the functionalities to be invoked via windows and menus and to access shared information. Recent VPL research is leading the effort to change the way programming environments are structured, and in this paper we take a step further in this direction.

This paper shows an approach to VPLs that follows the direction pointed out by part of Thomas Green's research into cognitive dimensions [Green 1991, Green and Petre 1996]. Cognitive dimensions are a set of terms describing the structure of a programming language's

components as they relate to cognitive issues in programming. They provide a framework for assessing the cognitive attributes of a programming system and for understanding a programming device's cognitive benefits and difficulties to programmers. Two of the dimensions, progressive evaluation and viscosity, are of particular relevance in the realm of problem solving.

*Progressive evaluation* is the ability for a programmer to execute a portion of a program immediately, even before the program is complete. In a study comparing the comprehension differences in debugging between novice and expert programmers [Gugerty and Olson 1986], it was shown that evaluating their progress frequently was essential for novice programmers and that, while it was not essential for experts, the experts actually use evaluation of partially-completed programs even more frequently while debugging than novices do. To maximize the availability of progressive evaluation is thus to reduce the amount of effort a programmer must exert in order to evaluate an unfinished program.

*Viscosity* is programmer effort required to make a change to the program. As Green and Petre point out [Green and Petre 1996], studies show that programmers iteratively create their programs, making change after change throughout the entire process. If the VPL does not allow these changes to be easily inserted, the programmer must exert considerable extra effort devoted solely to the mechanics of change. Minimizing viscosity will thus minimize this extra effort.

Our goal was to address these two cognitive issues, maximizing both the availability and the quality of progressive evaluation and feedback, and minimizing viscosity. Our strategy for doing so is termed *steering*.

The term *steering* has not been used consistently in the literature. Our use of the term comes from the scientific visualization community, which describes steering as the ability to receive a continuous visualization of data as the program executes, coupled with the ability for the programmer to interactively modify *any* aspect—not just the visualization or the input parameters—of a program at

---

\*This work is supported in part by Hewlett-Packard Corporation and by the National Science Foundation under grant CCR-9308649 and a Young Investigator Award.

any time and immediately see the effects without restarting the computation [McCormick et al. 1987].

Essential to our support for steering is an explicit notion of time and *time travel*. We use the term *time travel* to mean the ability of a programmer to return to a previous step of a computation. Our approach of combining time travel with steering supports problem-solving as a flexible, modeless process, removing barriers among traditionally separated programming tasks. For example:

- Specification: A programmer specifies program behavior (code) in the same way that data is entered; this is the same way that visualizations and all other kinds of programming are specified.
- Exploration: The programmer can use time travel to explore causes and effects of a program's behavior; the VPL keeps all output synchronized and consistent.
- Alteration: If the programmer alters the behavior of a program or visualization, the change is reflected not only in the present and future computations but also in all past computations.
- Visualization: The programmer has capabilities for visualization to aid in understanding a program. Low level visualizations are automatically produced whenever a new snippet of program is entered. Facilities for higher level algorithm animations are also an integral part of the VPL.

These features allow the programmer to review the past to understand behavior and find problems, attempt to fix the problem, and immediately see if the changes solve the problem or introduce any new problems.

## 2. Related Work

Our ideas about steering were inspired in part by the work on steering from the field of scientific visualization, as defined by the NSF Panel on Graphics, Image Processing and Workstations [McCormick et al. 1987] and surveyed in [Burnett et al. 1994]. Researchers in scientific visualization have achieved some steering capabilities through command-driven interfaces or special-purpose GUI visualization tools that are used in combination with traditional programming languages such as C, FORTRAN, and Smalltalk. In such tools, the scientist instruments the application and adds visualization and graphics routines to achieve the desired visual feedback and steerability. Examples of these works include AVS [Upson et al. 1989], Vista [Tuchman et al. 1991], VASE [Haber et al. 1992], and SCENE [Walther and Peskin 1991]. The primary differences between scientific steering systems and ours are that our approach is aimed toward understanding and correcting program behavior without requiring the programmer to insert instrumentation, pre-plan how and where steering can be done, or use different sets of mechanisms for steering and for programming.

The most highly interactive VPLs provide some of the features of steering. The visual object-oriented language Prograph [Cox et al. 1995], the visual dataflow language VPL [Lau-Kee et al. 1991], the by-demonstration language KidSim [Cypher and Smith 1995], and most spreadsheets are examples. Visibility of the data in these interactive VPLs is higher than in traditional programming systems, and allows the programmer to spot some kinds of programming errors as soon as they are entered and to inspect values one at a time during program execution. Even in these VPLs, there is little support for efficiently exploring previous states in a program that has gone mysteriously awry. However, there is currently emerging work from Yale University [Freeman et al. 1995] and by Lehrenfeld et al. at the University of Paderborn to begin to support such exploration in VPLs.

Research into reversible execution has mostly been concerned with imperative languages, but our approach is more closely related to work in the declarative and functional communities. The debugger for Tolmach and Appel's concurrent extension of Standard ML has reversible "program time" [Tolmach and Appel 1991]. Baker introduced a reversible Lisp [Baker 1992]. Systems such as these are focused toward reversible time itself and do not address the issues of progressive evaluation and viscosity.

Several visual debugging systems have supported both time travel and visualization for the purposes of error detection. PROVIDE [Moher 1988] was a pioneering visual debugging and visualization environment for a simplified C-like language. PROVIDE supported a number of capabilities for programmers to observe and control program execution and to interactively create data visualizations. The Transparent Prolog Machine [Brayshaw et al. 1991] provides graphical visualizations of Prolog queries that can be viewed at variable speeds forward and (if viewed post-mortem) in reverse. ZStep 94 [Lieberman and Fry 1995], a visual debugger for a subset of Common Lisp, provides support for time travel, for viewing how values and code are related, and for live graphical stepping. Debuggers such as these provide for visualization of program execution and location of errors, but they do not address the issue of viscosity, because many kinds of program changes require a restart of the entire computation.

Our incorporation of the high-level form of visualization known as algorithm animation during problem-solving is similar in philosophy to the Lens system [Mukherjea and Stasko 1993] in that both systems support algorithm animation as a problem-solving technique for programmers. Other algorithm animation systems such as Balsa [Brown 1988], Zeus [Brown and Najork 1993], Trip [Miyashita et al. 1992, Takahashi et al. 1994], and Animus [Duisberg 1986] are oriented more

toward instruction and do not support algorithm animation for incremental problem-solving.

### 3. A Programmer's View of Steering

To show concretely how steering can be used to maximize progressive evaluation and minimize viscosity, we introduce the Forms/3 [Burnett and Ambler 1994, Pandey and Burnett 1993] approach to steering from the programmer's view.

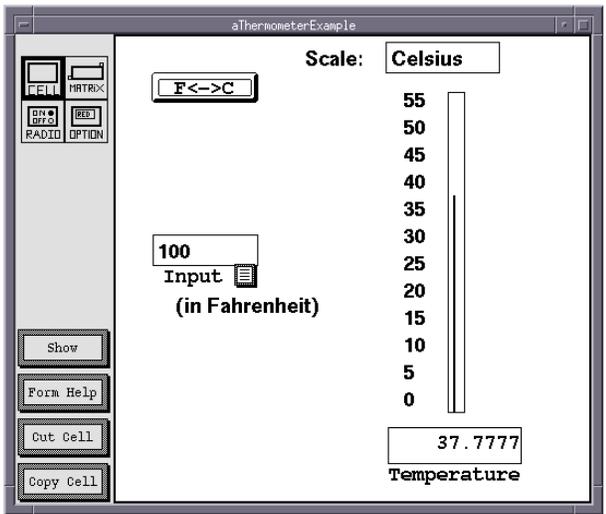


Figure 1a: The thermometer application, shown from the user's point of view. A tab indicates where the user is expected to provide an input formula.

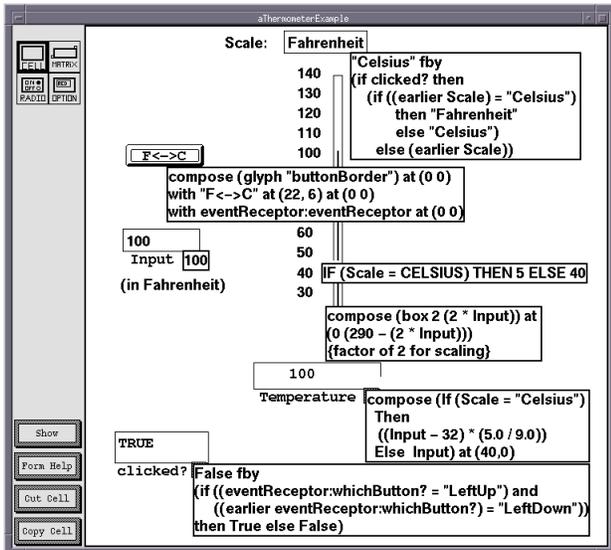


Figure 1b: The thermometer application shown from the programmer's point of view. The F <-> C button references eventReceptor, shown in Figure 2.

### 3.1. Specifying a Program

Specifying a program in Forms/3 is very similar to specifying a program in a spreadsheet—the programmer creates cells and gives each cell a formula. In Forms/3, however, unlike spreadsheets, the source code (formulas) and accompanying values can be shown together. For example, in Figure 1, the programmer has placed some cells on the screen and given them formulas to specify a program to display a thermometer, toggling between Celsius and Fahrenheit at the press of a button. As soon as she enters a formula, it is immediately evaluated and the result displayed, as in a spreadsheet. There is no compile phase, no need to mouse (i.e., to click on or point at) individual cells to see their values. The effects of each addition or change to a program are immediately reflected on the screen. The immediate feedback about the effects of her changes is the way Forms/3 provides progressive evaluation. Important aspects of this approach are that the feedback is immediate, incremental, and automatic, imposing no effort on the programmer.

### 3.2. Exploring and Time Travel

Now suppose that there is a bug. The button sometimes works, but sometimes doesn't: some mouse clicks don't cause the *Scale* value to toggle. The programmer decides to explore this strange behavior.

The formulas for *Scale* and the *F<->C* button are hidden from end users, but the programmer can shift-click to unhide the formulas. She examines the two formulas, and sees that the *Scale* cell depends on a hidden cell, named

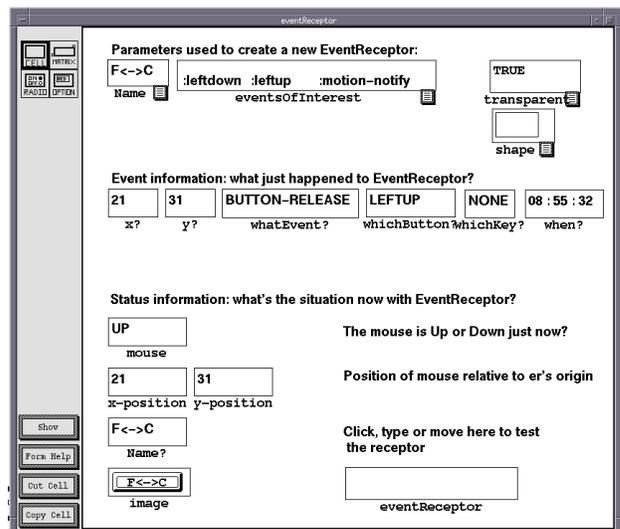


Figure 2: A programmer has access to mouse and keyboard events via this event receptor form. Tabs indicate modifiable cells.

clicked?, and that both *clicked?* and the button reference an eventReceptor, shown in Figure 2. Adding this form to the screen, the programmer travels backward and forward through time using the slider shown in Figure 3 to explore how the behavior of the eventReceptor might be affecting the *Scale* cell. She looks at the various cells on the eventReceptor form along with the *clicked?* and *Scale* values, and eventually notices that the bug occurs whenever there is an unusual sequence of values for the *whatEvent?* and *whichButton?* cells.

Hmmm... a click is defined in the formula of *clicked?* as the *whichButton?* cell having *leftUp* and an earlier *leftDown*, but the sequence of values she sees is *leftDown*, *None*, *leftUp*, as shown in Figure 4. This sequence seems to be where the clicks are being missed. When the *whichButton?* value is *None*, the *whatEvent?* value is *motionNotify*. Looking at the *eventsOfInterest* cell, the programmer sees that an irrelevant event type—*motionNotify*—is being attended to by this button, separating *leftDown*, the first half of a click, from *leftUp*, the other half. Here's the bug! It seems that the programmer didn't remove this event type from the default

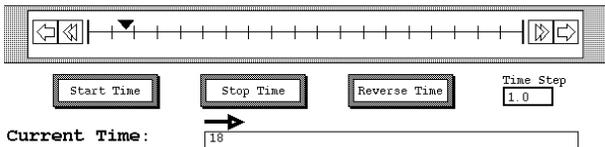


Figure 3: The slider used for time travel in Forms/3. Programmers can navigate using the stepper arrows, dragging the time indicator triangle, or by clicking directly at the desired point along the time slider.

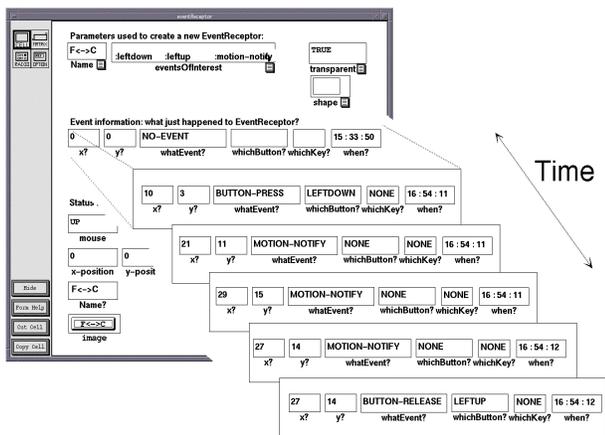


Figure 4: An annotated sequence of screen shots depicting the sequence of values in time for cells on the eventReceptor form. The programmer travels through time by manipulating the time slider.

*eventsOfInterest* specifications. She edits the formula of *eventsOfInterest* to remove *motionNotify*.

### 3.3. Altering Behavior Redefines History

Does the change the programmer just made actually fix the bug? To find out, the programmer explores the now-redefined history via time travel. It is inherent for the program's entire history to be redefined according to this change because cells' histories are defined solely by their formulas. This is another way progressive evaluation is used—as soon as a change is made, all affected histories are automatically redefined and all affected on-screen values are automatically recomputed and redisplayed. This allows the programmer to explore the program, reviewing how values changed, to determine whether the values changed as expected. In the example, the programmer sees that the clicks are now all recognized, and the bug is fixed.

The viscosity level of this approach is lower than modal approaches because, with the ability to time travel, the programmer is spared the usual effort of mode switching: re-running the program repeatedly, instrumenting the program with breakpoints or diagnostic statements, and recompiling. Furthermore, the programmer's context is preserved and the programmer can even return to a previous context by traveling backward in time.

### 3.4. Re-creating a Bug

Now suppose that the programmer who experiences the buggy behavior is not the program's author and doesn't have access to the source code. Thus, if she wants the problem fixed, she must seek help from the technical support programmer at the company that created the program. The first task of the technical support programmer will be to re-create the buggy behavior. Unfortunately, often a bug proves elusive; it happens only sporadically under a poorly understood combination of events, and cannot be demonstrated at will. When this situation arises, it adds difficulty to the process of finding and fixing bugs.

In Forms/3, any situation can easily be re-created. This is possible because most values are defined declaratively, and can therefore be recalculated to produce exactly the same history. For the only non-declarative values—user events—Forms/3 has a mechanism to save the relevant mouse and keyboard events to a file. These events are located in one place, the System form. The programmer reporting the bug would save the System form's values to a disk, and send the data to the technical support programmer. In turn, he could then re-create the bug by loading the saved System form into his environment. Loading the saved System form restores the current context *and the complete history*, because the events are restored and all other values can be recomputed.

He can then explore the program using the same approach described in the previous section. Thus, he does not have to use trial and error in an attempt to re-create sequences that led to bugs.

### 3.5. Visualization and Animation

Consider another scenario where the first programmer decides to investigate the bug, but wants to see the behavior better by creating a visualization. She decides that a good representation would be a line graph with a *buttonDown* event as a line down, and a *buttonUp* event as a line going up, and mouse motion as a jagged line. She creates a form to produce the visualization shown in Figure 5. This allows her to see in a graphical way why the clicks are being missed.

A critical part of a programmer’s job is understanding the program under scrutiny. A programmer can be easily overwhelmed by the low-level complexity of a program and not see the big picture. Green and Petre [1996] point out “The mental representation of a program is at a higher level than pure code... Spohrer and Soloway [1989] report that... [for novice programmers] ‘many bugs arise as a result of plan composition problems—difficulties in putting the pieces of a program together.’”

Forms/3 has several visualization mechanisms to aid the programmer in comprehending the program. Firstly, a formula’s current value is displayed when the formula is entered. Secondly, abstract data types have a default appearance, which is defined in the formula of a cell called an image cell. The programmer can alter this formula and thus specify the appearance of an abstract data type as desired. For example, an employee record could show the name in one application, and the pay grade, work site, and number of years of service in another application, as shown in Figure 6. By including aspects of the components of a data type in the image formula during testing and debugging, the appearance of the cell will help to visually communicate details of its current value.

Thirdly, Forms/3 supports algorithm animation, the ability to animate the abstract operations of a program [Carlson et al. 1996]. For example, the programmer may wish to highlight the “move” portion of a selection sort so each element steps across the screen to its new location. In the initial, unaugmented program, the elements simply appear in their new positions. To specify an animation, the programmer uses a built-in form to define intermediate positions through which the elements travel. The animation is shown in Figure 7.

Such a mechanism can be important in programming because algorithm animation can aid in understanding the program. In Forms/3, animations are done entirely within the language via animation primitive operators. The programmer need not learn a separate language or tool.

The language’s evaluation engine guarantees that an animation remains in synchrony with the program, even when logical time is moved backward. Animations can be run backward and steered just like any other Forms/3

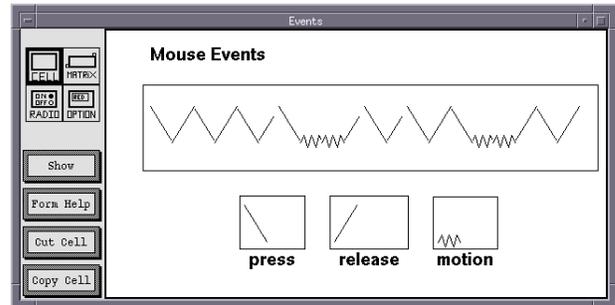


Figure 5: A visualization of mouse events.

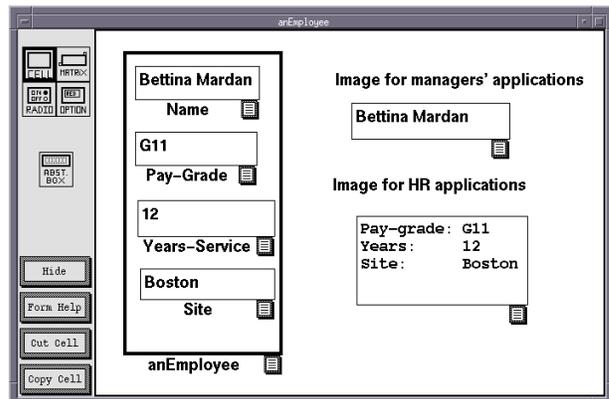


Figure 6: An example of programming the appearance of data to enhance progressive evaluation.

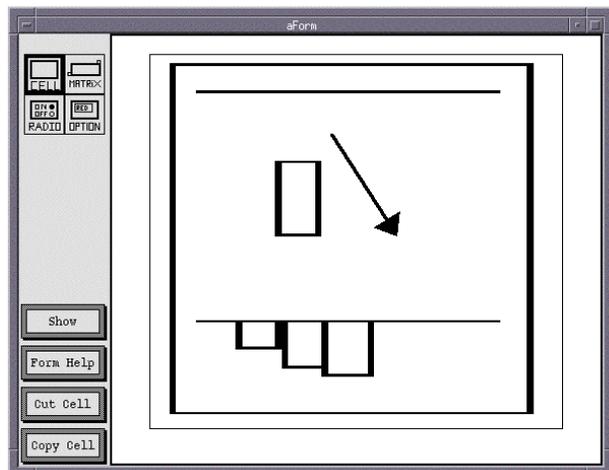


Figure 7: A sort animation shows the elements of the unsorted group being moved one at a time to the sorted group.

program. Also, the animation is programmed in a non-invasive manner using references to the program to be animated. The original program is unaltered, so there is no danger of inadvertently modifying the original algorithm.

By making algorithm animation an integral part of the VPL, mode switching is removed. This is another example of low viscosity because the programmer can change the algorithm animation at any time without the effort of switching to a different tool and rebuilding context.

## 4. What Makes This Approach Work

While our examples are shown in Forms/3, the approach is applicable to any declarative VPL that contains 3 key features: temporal assignment, declarative event handling, and responsiveness. In Forms/3, these features are implemented by logical time, events as values, and lazy marking, respectively.

### 4.1. Logical Time

Forms/3's concept of time is based on a notion of *logical* time. In Forms/3, logical time is viewed as a dimension, and each value in the environment has a fixed, permanent position along that dimension's axis. Thus, a cell does not have a single value, but rather a sequence of values positioned along that axis. Even a constant such as a text string is formally defined as a one-element sequence first defined at logical time 1, although the programmer uses such constants in the conventional way.

Each sequence's value is defined to start at the first moment in logical time at which all its components' values are defined, and to expire at the earliest time its components' values expire. For example, if  $X=Y+Z$ , then  $X$ 's first value starts at the first position in logical time at which both  $Y$  and  $Z$  have values defined, and expires as soon as either  $Y$  or  $Z$ 's first value expires. Through this global notion of relationships in time, all values in the environment are automatically synchronized, and any moment in time can be constructed, reconstructed, and redefined in a straightforward manner. Because a value might not expire for a long time, sequences may be sparse—there is no repetition of the same value over and over in a sequence just to reflect the fact that a value has not expired. For example, if  $Y$ 's formula is the constant "3", then  $Y$ 's first value is defined at time 1 and never expires (because it has no dependencies).

As this shows, logical time is about the progression of sequences, not about how fast the clock on the wall is ticking. Thus, logical time progresses much more slowly than clock-on-the-wall time (unless clock-on-the-wall time is one of the values referenced by the program). For example, a user event advances time forward one step,

even though many values change on the eventReceptor form (refer back to Figure 2). A button click moves time forward two logical time steps—no matter how much time elapses between the button press and the button release, and no matter how many cells change as a result of the click—because a click is not a low level event, but is synthesized (in the formula of the *clicked?* cell) from a sequence of two events, *leftDown* and *leftUp*.

### 4.2. Events as Values

If the handling of user events had to be programmed through event loops and polling devices, as is true in traditional programming languages, then key features in our support of steering, such as automatic synchronization among related values and events as the programmer travels through time, would be lost. However, in Forms/3, the same formula-based programming style is used for event handling as for value-based computations. Events are reported in cells, and other cells' formulas can refer to them. Low-level polling is not needed because the evaluation mechanism makes sure that all the formulas that refer to events (or any other value) are kept up-to-date when new data arrives. The programmer simply specifies via formulas what, if anything, is to be done when events arrive.

The details are as follows. Event detection is done by instances of an abstract data type called an eventReceptor, shown earlier in Figure 2. The programmer places the desired specifications of event detection on a copy of the eventReceptor form. The specifications include which events are to be recognized by this eventReceptor and which area of the screen is active for this eventReceptor. The environment automatically adds event information into the value sequences for cells on that form, such as the *whatEvent?* cell. For example, the event of pressing the left mouse button over the  $F \leftrightarrow C$  cell causes values to be defined (at a new logical time) for the cells on the eventReceptor form referenced by the  $F \leftrightarrow C$  cell.

This unified approach to events and values allows the same mechanism that supports value-related programming to fully support event-related programming, and thus the time travel supported for ordinary sequences of values works equally as well on event sequences. This use of a single mechanism, when combined with the approach to logical time described in the previous section, allows automatic synchronization of user events with the values they affect, facilitating debugging by allowing the determination of exactly just what events led to the values currently displayed by the program.

### 4.3. Time Travel and Efficiency

A disadvantage to some other systems that allow review of past program states is inefficiency, both space

and time. In particular, it would be a significant problem if our approach were time-inefficient, because our direct-manipulation approach to time travel demands a responsive VPL. However, our approach is tunable; it can be optimized for time or space efficiency, or a balance of the two.

First consider space efficiency. Most other systems that support review of a program's history require extensive amounts of space to do so because all prior values must be stored. However, in our declarative VPL, values need not be stored. All of a cell's sequence (history) is completely defined via its formula, making the storage of the actual values superfluous. The only information in addition to a cell's formula that absolutely must be stored are user events (mouse clicks, etc.) This formula-based approach means that the history of a program is much more compact than imperative systems, as was illustrated in the example on re-creating a bug after-the-fact.

Now consider time efficiency. Although the system need not save histories, response time can be improved if it does store some of them. When a programmer is traveling through time, she may be forcing the program to re-display values many times, generating many duplicate computations. Our approach allows trading off as much space as desired to reduce the number of duplicated computations by using the well-known techniques of lazy evaluation and lazy memoization (memoing is the saving and reuse of computed values [Michie 1968]).

To further enhance time efficiency, our system adds a new technique called lazy marking [Atwood and Burnett 1996] to efficiently ensure that all values on the screen are automatically kept up-to-date as the program progresses through time. Lazy marking's improvement to the time efficiency of the VPL comes from the way it "marks" values with expiration times. By employing a lazy, incremental approach to marking, it is able to mark a value with a conservative view of its expiration time as soon as the value is computed. When the next value in the same sequence is computed, the first value's expiration time can be revised if it was too conservative. By avoiding the marking of out-of-date dependent values, this incremental approach improves response time and often reduces total time spent.

## 5. Current Status

The approach to steering described in this paper is implemented in our research prototype, which runs on Sun and Hewlett-Packard color workstations, using Lucid Common Lisp and the Garnet user interface development system [Myers et al. 1990].

## 6. Conclusion

Forms/3 provides steering via time travel to maximize progressive evaluation and minimize viscosity in programming. Progressive evaluation in Forms/3 provides immediate feedback about the impact of each code fragment, large or small, as soon as each new fragment is entered. Further, programmers can explore a program's behavior over time with unusual flexibility—there are no breakpoints, no re-compilations with debugging options, and no switching from "running" to using a specialized debugger. This kind of progressive evaluation through time travel can be done on demand, simply by manipulating the time slider bar.

Forms/3 programmers can make changes to the program at any time. Doing so automatically adjusts the past, present, and future, which programmers can explore to see if the change had the desired effects. This flexible ability to alter a program at any point results in low viscosity and context preservation. Low-level visualizations are automatically provided by the environment, and programmers can modify them and add high-level visualizations if desired, without switching to another mode or tool. The visualizations are automatically synchronized with the rest of the program, and can be explored and altered along with the rest of the program because there is no distinction between steering visualizations and steering programs. Using these mechanisms, the Forms/3 VPL dissolves the traditional demarcations of programming tools to give the programmer a task-oriented development environment supporting the HCI principles of progressive evaluation and viscosity.

## Acknowledgments

We thank Jonathan Cadiz, Paul Carlson, Maureen Chesire, Herkimer Gottfried, Judith Hays, Rick Wodtli, and Pieter van Zee for their comments, their help with the implementation, and their testing of our environment.

## References

- [Atwood and Burnett 1996] Atwood, J. and Burnett, M. Culprit Tracking: Improved Lazy Marking for Better GUI Performance. Technical Report 96-60-1, Oregon State University, Department of Computer Science, Dec. 1996.
- [Baker 1992] Baker, H., NReversal of Fortune -- The Thermodynamics of Garbage Collection, 1991 *Int'l Workshop on Memory Management*, St. Malo, France, Sept. 1992, 507-524.
- [Brown 1988] Brown, M. Perspectives on Algorithm Animation. *Proc. CHI'88: Human Factors in Computing Systems*, Washington, DC, May 15-19, 1988, 33-38.
- [Brown and Najork 1993] Brown, M. and Najork, M. Algorithm Animation Using 3D Interactive Graphics.

- UIST'93, *Proc. ACM Symp. on User Interface Software and Technology*, Atlanta, GA, Nov. 3-5, 1993, 93-100.
- [Burnett and Ambler 1994] Burnett, M. and Ambler, A. Interactive Visual Data Abstraction in a Declarative Visual Programming Language. *J. of Visual Languages and Computing*, Mar. 1994, 29-60.
- [Burnett et al. 1994] Burnett, M., Hossli, R., Pulliam, T., VanVoorst, B., and Yang, X. Toward Visual Programming Languages for Steering in Scientific Visualization: a Taxonomy. *IEEE Computational Science and Engineering* 1(4), Winter 1994, 44-62.
- [Brayshaw and Eisenstadt 1991] Brayshaw, M. and Eisenstadt, M.. A Practical Graphical Tracer for Prolog. *Int. J. of Man-Machine Studies*, 35(5), 1991, 597-631.
- [Carlson et al. 1996] Carlson, P., Burnett, M., and Cadiz, J.J. A Seamless Integration of Algorithm Animation into a Visual Programming Language. *Advanced Visual Interfaces (AVI '96)*, Gubbio, Italy, May 1996, (to appear).
- [Cox et al. 1995] Cox, P. T., Giles, F. R., and Pietrzykowski, T. Prograph, in *Visual Object-Oriented Programming: Concepts and Environments*, (M. Burnett, A. Goldberg, T. Lewis, eds.), Prentice-Hall/Manning Pubs., 1995.
- [Cypher and Smith 1995] Cypher, A. and Smith, D. KidSim: End User Programming of Simulations, *Proc. CHI'95: Human Factors in Computing Systems*, Denver, CO, May 7-11, 1995, 27-34.
- [Duisberg 1986] Duisberg, R. A., Animated Graphical Interfaces using Temporal Constraints, *Proc. CHI'86: Human Factors in Computing Systems*, Boston, MA, April 13-17, 1986, 131-136.
- [Freeman et al. 1995] Freeman, E., Gelernter, D., and Jagannathan, S., In Search of a Simple Visual Vocabulary, *1995 IEEE Symposium on Visual Languages*, Darmstadt, Germany, Sept. 5-9, 1995, 302-309.
- [Green 1991] Green, T. Describing information artifacts with cognitive dimensions and structure maps, in *People and Computers VI*, (D. Diaper and N. Hammond, eds.), Cambridge University Press, 1991.
- [Green and Petre 1996] Green, T. and Petre, M. Usability Analysis of Visual Programming Environments: a 'Cognitive Dimensions' Framework, *Journal of Visual Languages and Computing*, June 1996, (to appear).
- [Gugerty and Olson 1986] Gugerty, L. and Olson, G. M., Comprehension Differences in Debugging by Skilled and Novice Programmers. In *Empirical Studies of Programmers*, (E. Soloway and S. Iyengar, eds.), Ablex, Norwood, NJ, 1986.
- [Haber et al. 1992] Haber, R., Bliss, B., Jablonowski, D., and Jog, C. A Distributed Environment for Run-Time Visualization and Application Steering in Computational Mechanics. *1992 Symp. on High-Performance Computing for Flight Vehicles*, Washington, DC, Dec. 7-9, 1992.
- [Lau-Kee et al. 1991] Lau-Kee, D., Billyard, A., Faichney, R., Kozato, Y., Otto, P., Smith, M., and Wilkinson, I. VPL: An Active, Declarative Visual Programming System. *1991 IEEE Workshop on Visual Languages*, Kobe, Japan, Aug. 1991, 40-46.
- [Lieberman and Fry 1995] Lieberman, H. and Fry, C. Bridging the Gulf Between Code and Behavior in Programming. *Proc. CHI'95: Human Factors in Computing Systems*, Denver, CO, May 7-11, 1995, 480-486.
- [McCormick et al. 1987] McCormick, B. H., DeFanti, T. A., and Brown, M. D. eds., Visualization in Scientific Computing, *Computer Graphics* 21(6), Nov. 1987.
- [Michie 1968] Michie, D., 'Memo' Functions and Machine Learning, *Nature* 218, 19-22.
- [Miyashita et al. 1992] Miyashita, K., Matsuoka, S., Takahashi, S., and Yonezawa, A. Declarative Programming of Graphical Interfaces by Visual Examples, *Proc. of the ACM Symp. on User Interface Software and Technology*, Monterey, CA, Nov. 15-18, 1992, 107-116.
- [Moher 1988] Moher, T., PROVIDE: A Process Visualization and Debugging Environment, *IEEE Transactions on Software Engineering*, 14(6), June 1988.
- [Myers et al. 1990] Myers, B. et al. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces, *Computer*, Nov. 1990, 71-85.
- [Mukherjea and Stasko 1993] Mukherjea, S. and Stasko, J. Applying Algorithm Animation Techniques for Program Tracing, Debugging, and Understanding. *Proc. 15th Int. Conf. on Software Eng.*, May 17-21, 1993, 456-465.
- [Pandey and Burnett 1993] Pandey, R. and Burnett, M. Is It Easier to Write Matrix Manipulation Programs Visually or Textually? An Empirical Study. *1993 IEEE Symp. on Visual Languages*, Bergen, Norway, Aug. 24-27, 1993, 344-351.
- [Spohrer and Soloway 1989] Spohrer, J. C. and Soloway, E., Novice Mistakes: Are the Folk Wisdoms Correct? In *Studying the Novice Programmer*, (E. Soloway and J. C. Spohrer, eds.), Erlbaum, Hillsdale, NJ, 1989.
- [Takahashi et al. 1994] Takahashi, S., Miyashita, K., Matsuoka, S., and Yonezawa, A. A Framework for Constructing Animations via Declarative Mapping Rules. *1994 IEEE Symposium on Visual Languages*, St. Louis, MO, Oct. 4-7, 1994, 352-357.
- [Tolmach and Appel 1991] Tolmach, A., and Appel, A., Debuggable Concurrency Extensions for Standard ML, *1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 20-21, 1991, 120-131.
- [Tuchman et al. 1991] Tuchman, A., Jablonowski, D., and Cybenko, G. Run-time Visualization of Program Data, *Proc. of Visualization '91*, San Diego, CA, Oct. 22-25, 1991, 255-261.
- [Upson et al. 1989] Upson, C., Faulhaber, T., 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(7), July 1989, 30-42.
- [Walther and Peskin 1991] Walther, S. and Peskin, R. Object-oriented Visualization of Scientific Data. *Journal of Visual Languages and Computing*, March 1991, 43-56.