# 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
Phone: +1-503-737-3273
E-mail: {atwoodj,burnett,walpolr,wilcoxer,yang}@research.cs.orst.edu

**ABSTRACT**
Despite years of research into human computer interaction (HCI), the environments programmers must use for problem-solving today—with separate modes and tools for writing, compiling, testing, visualizing, and debugging—derive their basic structure from historical accident, and take little advantage of HCI research into the cognitive issues of programming. Neglecting these issues is an impediment to the programmers' ability to produce reliable, maintainable software. In this paper, we describe a system 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 programmers need for problem-solving.

**KEYWORDS:** Programming environments, Development tools, Debugging, Psychology of programming, Visual programming, Steering

## INTRODUCTION

In the popular "Back to the Future" movies, a young man named Marty travels backward and forward in time. When Marty changes the past, he often gets immediate feedback on the consequences of those changes (photos from the present morph to reflect the changes). He also sees the effects of those changes on other times as he continues to travel through time. We sought to provide programmers this kind of flexible environment for problem-solving, with strong support for immediate visual feedback and exploration.

Today's programming environments are still designed around implementation strategies. While cognitive issues of programming have been studied and published by HCI experts, they have not been incorporated into the design of programming systems. As a result, instead of event-driven systems structured around the notion of "working with programs", programmers must work in highly modal environments, memorizing results and repeating steps as they switch among separate tools for editing, compiling, testing, debugging, and visualizing. Although integrated approaches improve on this to some degree, they still integrate only a few of these modes. 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 leave the functionality separated but allow the mechanisms to be invoked via menu selection and to access shared information. Even many of today's visual programming languages are highly modal, and retain much of this functional separation.

This paper shows an approach to programming environments that follows the direction pointed out by part of Thomas Green's research into cognitive dimensions [11, 12]. 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 tracking and 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 [13], 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 go to in order to evaluate an unfinished program.

---

*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.

*Viscosity* is programmer effort required to make a change to the program. As Green and Petre point out [12], studies show that programmers iteratively create their programs, making change after change throughout the entire process, from specification though design to coding. If the environment 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, the ability for the programmer to interactively modify the visualization at any time, and the ability to modify *any* aspect—not just input parameters—of a program at any time and immediately see the effects without restarting the computation [17].

Forms/3 supports steering through an extension to the spreadsheet paradigm that includes an explicit notion of time and time travel. Our approach to steering supports problem-solving as a flexible, modeless process, removing barriers among traditionally separated programming tasks. For example:

- A programmer specifies program behavior (code) in the same way that data is entered (as formulas of cells); this is the same way that visualizations and all other kinds of programming is specified.

- The programmer can use time travel to explore causes and effects of a program's behavior, using tools such as high-level and low-level visualizations, and examining program specifications at any time; the environment keeps all output synchronized and consistent.

- If the programmer alters the behavior of a program or visualization, either at the current moment in time or at some point in past history, the change is reflected not only in the present and future computations but also in all past computations.

- The programmer has tools such as visualizations and animations to aid in understanding a program. Low level visualizations are automatically produced whenever a new snippet of program is entered. Facilities for higher level visualizations and animations are an integral part of the programming language and environment.

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.

## RELATED WORK

Our ideas about steering were inspired in part by the work on steering from the field of scientific visualization, as described by the NSF Panel on Graphics, Image Processing and Workstations [17] and surveyed in [5]. 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 [26], Vista [25], VASE [14], and SCENE [27]. The primary difference between scientific steering systems and ours is that our environment 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.

Highly interactive visual programming environments provide some of the features of steering. The visual object-oriented language Prograph [8], the visual dataflow language VPL [15], the by-demonstration language KidSim [9], and most spreadsheets are examples. Visibility of the data in these environments 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. However, even in these environments, there is little support for efficiently exploring previous states in a program that has gone mysteriously awry.

Several debugging systems have supported a form of time travel and visualization for the purposes of error detection. PROVIDE [19] 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 [6] provides graphical visualizations of Prolog queries that can be viewed at variable speeds forward and (if viewed post-mortem) in reverse. ZStep 94 [16], 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 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 [21] in that both systems support algorithm animation as a problem-solving technique for programmers. Other algorithm animation systems such as Balsa [1], Zeus [2], Trip [18, 24], and

Animus [10] are oriented more toward instruction and do not support algorithm animation for incremental problem-solving.

## STEERING

To show concretely how steering can be used to maximize progressive evaluation and minimize viscosity, we introduce the Forms/3 approach to steering by example. A complete description of the language part of Forms/3 and its evaluation model is given in [3].

### Specifying a program

Following the spreadsheet paradigm, the programmer creates cells and gives each cell a formula. 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. Each addition or change to a program is immediately reflected on the screen. The immediate feedback as to whether her intentions were accurately specified is the way Forms/3 provides progressive evaluation. Important aspects of this spreadsheet-like approach are that the feedback is immediate, incremental, and automatic, imposing no effort on the programmer. In Forms/3, however, unlike spreadsheets, the source code (formulas) and accompanying values can be shown together.

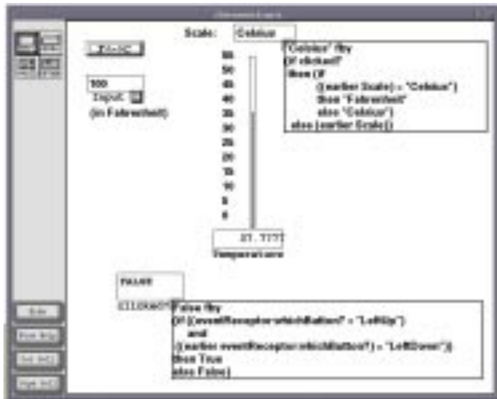

*Figure 1: The thermometer application after the programmer has finished specifying it. In the user's view, no formulas would be shown, and the* clicked? *cell would be hidden (using formatting attributes).*

### 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 *clicked?*, and that both *clicked?* and the button reference an eventReceptor, shown in Figure 2. 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.

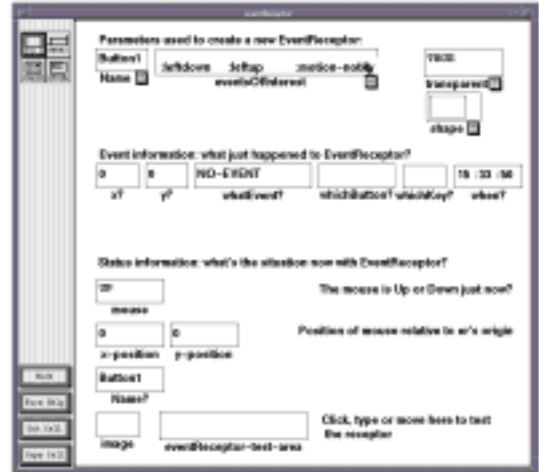

*Figure 2: A programmer has access to mouse and keyboard events via this event receptor form. The formula tab below* eventsOfInterest *indicates that this is a parameter whose formula can be modified.*

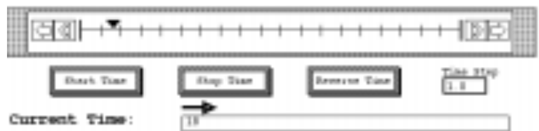


*Figure 3: The slider used for time travel in Forms/3. Programmers can navigate using the stepper arrows, dragging the time indicator triangle, or by clicking directly at the desired point along the time slider.*

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

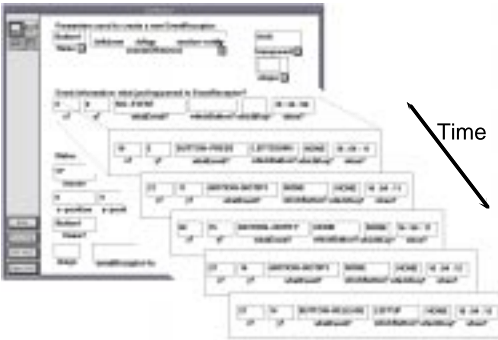


*Figure 4: A sketch (not a screen shot) of the sequence of values in time for cells on the eventReceptor form. The programmer travels through time by manipulating the time slider.*

**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 possible for the program's 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 onscreen values are automatically re-computed and redisplayed. This allows the programmer to explore the program, reviewing how values changed, and determining 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. 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 re-compiling. Furthermore, the programmer's context is preserved and the programmer can even re-create a previous context by traveling backward in time.

**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 the Forms/3 environment, 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 in the environment, 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. He in turn could then re-create the bug by loading the saved System form into his environment. Loading the saved System form restores the complete context, 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, but rather can explore them systematically.

**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 thinks for a moment about how such a picture would look, and 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 begins creating cells and formulas, and soon has the visualization shown in Figure 5. This allows her to see in a graphic way why the clicks are being missed.

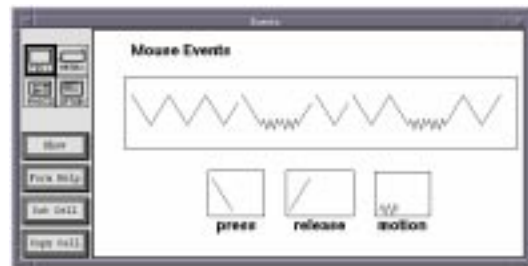


*Figure 5: A visualization of mouse events.*

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 point out "The mental representation of a program is at a higher level than pure code... Spohrer and Soloway [23] 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 mechanisms to aid the programmer in comprehending the program. The first derives from the spreadsheet paradigm on which the language is based. 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, the appearance of the data will help communicate its current state.

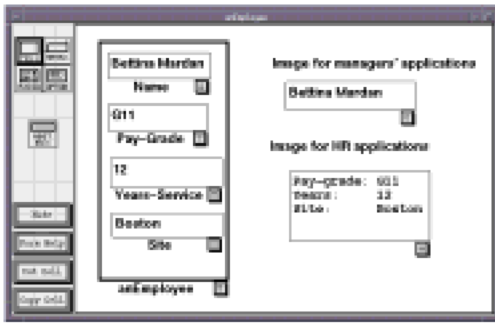


*Figure 6: An example of programming the appearance of data to enhance progressive evaluation.*

Thirdly, Forms/3 allows algorithm animation, the ability to animate the abstract operations of a program [7]. 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, outside both the input and output, through which the elements travel. The animation is shown in Figure 7.

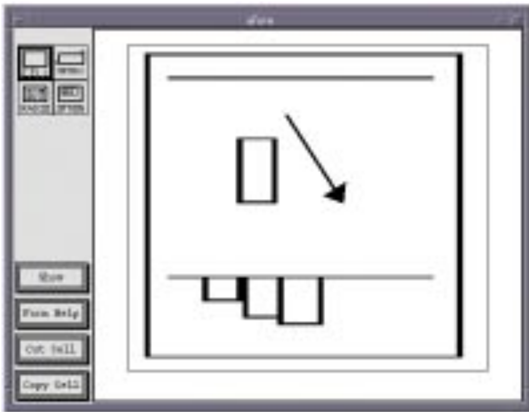

*Figure 7: A sort animation shows the elements of the unsorted group being moved one at a time to the sorted group.*

Such a mechanism can be important in programming because 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 sync with the program, even when logical time is moved backward. Animations can be run backward, and steered just like any other Forms/3 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 animation and program visualization an integral part of the programming language, 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. The programming environment can't eliminate the hard work of designing a visualization or animation, but can lessen the effort required by providing a low-viscosity environment with easy access to visualization and animation operators.

## WHAT MAKES THIS APPROACH WORK

Forms/3's approach to steering rests upon its support for time travel. The three features that are particularly important to making time travel work are the way logical time is used to synchronize values and define history, the unified approach to events and values, and the strategies used for the implementation's efficiency.

### 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. For example, a user event advances time forward 1 step, even though many values change on the ER form (refer back to Figure 2). A button click moves time forward 2 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*.

### Events as Values

If the handling of user events had to be programmed through event loops and polling devices, as is true in most 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 of the spreadsheet-like evaluator, which 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-handling 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-handling 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 keyboard event of pressing a "q" causes values to be defined (at a new logical time) for the cells on the eventReceptor form handling keyboard events for the active area of the screen.

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.

### Time Travel and Efficiency

A disadvantage to some other systems that allow review of past program states is inefficiency, both space and time. If our approach were time-inefficient, it would be an especially significant problem, because our direct-manipulation approach to time travel demands a responsive environment. However, our approach is tunable; it can be optimized for time or space efficiency, or a balance of the two.

We 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 environment, the sequence of a cell's values may be stored in part, in total, or not at all—the number actually stored is simply an optimization parameter. This is possible because Forms/3 (like other spreadsheet-oriented systems) is a declarative environment, and 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 described in the section on re-creating a bug after-the-fact.

But, although the memory required to store the histories is small, saving portions of them can increase execution speed. 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 computations by using the well-known techniques of lazy evaluation and lazy memoization. It also adds a new technique called lazy marking [4] 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 environment is that 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. This incremental approach avoids traversal of dependency information except that needed to produce the value itself, keeping the cost below other approaches commonly used to keep displayed values up-to-date in most circumstances.

## CURRENT STATUS

We have implemented the approach to steering in our research prototype, which runs on Sun and Hewlett-Packard color workstations using Lucid Common Lisp and the Garnet user interface development system [20].

Forms/3 has been evolving since 1991, and some of its features have been present in the implementation for quite some time. However, the implementation has included full time-travel support of steering for only a short time. Because of this fact, so far we can report only one empirical study that relates to the effectiveness of the approach. In that study, we evaluated the effectiveness of Forms/3's spreadsheet-oriented style of programming through the use of formulas. The study showed that, when applied to matrix-oriented programming problems, this style allows programmers to construct programs more reliably than when they use traditional languages [22]. We are now in the second stage of a new empirical study in which we are evaluating how our support for progressive evaluation affects a programmer's ability to find and correct bugs.

## CONCLUSION

We have presented an environment supporting problem solving for programmers through an extension of the spreadsheet paradigm. Forms/3 provides steering via time travel to maximize progressive evaluation and minimize viscosity in programming.

Progressive evaluation provides immediate feedback about the impact of each code fragment, large or small, as soon as each new fragment is entered. Programmers can explore a program ad hoc—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.

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, because it eliminates the traditional multiple mode-switching and context rebuilding required to serially make changes, compile them, test them, and debug them. 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.

The Forms/3 programming environment dissolves the traditional demarcations of programming tools to give the programmer a productive problem-solving, task-oriented development environment. Rather than attempting to glue yesterday's approaches together with interactive trappings, we believe it is time to start afresh, creating new approaches to programming that are designed around HCI principles.

## ACKNOWLEDGMENTS

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

## REFERENCES

1. Brown, M. Perspectives on Algorithm Animation. Proc. CHI'88: Human Factors in Computing Systems, Washington, DC, (May 15-19, 1988), 33-38.

2. Brown, M. and Najork, M. Algorithm Animation Using 3D Interactive Graphics. UIST'93, Proc. ACM Symposium on User Interface Software and Technology, Atlanta, Georgia, (Nov. 3-5, 1993), 93-100.

3. Burnett, M. and Ambler, A. Interactive Visual Data Abstraction in a Declarative Visual Programming Language. J. Vis. Lang. Computing, (Mar. 1994), 29-60.

4. Burnett, M. and Atwood, J. Lazy Marking: A Lazier Implementation of Functional I/O for Graphical User Interfaces. Technical Report 94-60-9, Oregon State University, Department of Computer Science, Dec. 1994.

5. 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.

6. Brayshaw, M. and Eisenstadt, M.. A Practical Graphical Tracer for Prolog. Int. J. of Man-Machine Studies, 35(5), (1991), 597-631.

7. Carlson, P. and Burnett, M. A Seamless Integration of Algorithm Animation into a Visual Programming Language with One-Way Constraints. Proc. International Workshop on Constraints for Graphics and Visualization, Cassis, France, (Sept.1995).

8. 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 Publications, 1995.

9. 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.

10. 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.

11. 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.

12. Green, T. and Petre, M. Usability Analysis of Visual Programming Environments: a 'Cognitive Dimensions' Framework, Technical Report, MRC Applied Psychology Unit, 1995.

13. 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.

14. Haber, R., Bliss, B., Jablonowski, D., and Jog, C. A Distributed Environment for Run-Time Visualization and Application Steering in Computational Mechanics. Symposium on High-Performance Computing for Flight Vehicles, Washington, DC, (Dec. 7-9, 1992).

15. 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.

16. 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.

17. McCormick, B. H., DeFanti, T. A., and Brown, M. D. eds., Visualization in Scientific Computing, Computer Graphics 21(6), (Nov. 1987).

18. Miyashita, K., Matsuoka, S., Takahashi, S., and Yonezawa, A. Declarative Programming of Graphical Interfaces by Visual Examples, Proceedings of the ACM Symposium on User Interface Software and Technology, Monterey, CA, (Nov. 15-18, 1992), 107-116.

19. Moher, T., PROVIDE: A Process Visualization and Debugging Environment, IEEE Transactions on Software Engineering. 14(6), (June 1988).

20. Myers, B. et al. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces, Computer, (Nov. 1990), 71-85.

21. 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.

22. 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.

23. 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.

24. 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.

25. Tuchman, A., Jablonowski, D., and Cybenko, G. Run-time Visualization of Program Data, Proceedings of Visualization '91, San Diego, CA, (Oct. 22-25, 1991).

26. Upson, C., Faulhaver, 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.

27. Walther, S. and Peskin, R. Object-oriented Visualization of Scientific Data. Journal of Visual Languages and Computing (March 1991), 43-56.