

(Presented at STEP '97, International Conference on Software Technology and Engineering Practice, July 14-18, 1997, London, England)

Ahead-of-time Debugging, or Programming Not In the Dark

James L. Snell, Ph.D.
Snell Software Consultants, Inc.
524 N.E. 83rd Street
Seattle, WA 98115-4158
206-525-5440 voice/fax
jsnell@acm.org
www.snellsoftware.com

Abstract

The activity of composing code is highly error-prone because it places severe demands on the programmer's cognitive processes, e.g., mental simulation of the computer, recall of many changing states, and maintenance of complex goal structures. The programmer could benefit greatly from the machine's help during this process. Unfortunately, program development environments (PDEs) are essentially passive then; the assumption is that debugging is done after a whole routine has been entered, although by this time the programmer's memory of its details has significantly decayed.

We propose that coding efficiency could be improved by providing an environment in which, as soon as the programmer types a statement, it is executed, and the new program state is displayed for each of several test cases. The programmer can see the statement's consequences, verify that they are as intended, and if not, correct it, all while the thinking for that statement is still fresh in mind. We call this feature "ahead-of-time" (AOT) debugging.

We have implemented a prototype environment that provides AOT debugging, and have done informal testing with a few users, who are asked to develop short programs working from printed pseudocode. We observe that users detect and fix substantially more bugs during initial code entry with AOT than with a standard PDE, and propagation of logic errors is reduced. AOT users produce code faster and with fewer errors, and report higher satisfaction and lower stress and fatigue. Several report that, compared with AOT, using the standard PDE feels like "programming in the dark."

Keywords: debugging, programming environments, software reliability, programmer productivity, software ergonomics/human factors, program understanding.

Introduction and rationale

This project was motivated by two observations:

(1) It is widely recognized that the process of composing code places heavy demands on human cognition (see, e.g., Shneiderman 1980; Green 1990; Brooks 1977). In coding a single statement, the programmer¹ typically must recall several program states, decide what state changes are needed, choose code to produce them, mentally simulate those changes (often for several different cases), verify that the changes are the intended ones, and remember the new states. In addition, the programmer must maintain a complex goal structure that relates this statement to others, all ultimately back to the top-level programming goal. As any programmer knows, this finely detailed mental activity can be severely demanding, and can strain the limits of human cognition (related to mental workload and memory load (Hancock and Caird 1993), stress and fatigue (Galinsky et al. 1993), etc.). Given the highly error-prone nature of this process, it is not surprising that programmers write code containing bugs.

(2) Program development environments (PDEs) (and debuggers, which we include in this category for simplicity), provide powerful features for debugging after a program or routine has been entered (breakpoints, single-stepping, watching variables, etc.), but little help during the original entry of program code. This we conclude from an informal survey of some widely used PDEs, including: the Borland C++ and Turbo Pascal Integrated Development Environments; the Microsoft Visual C++ Visual Workbench and Visual Basic programming environment; the DBG debugger supplied with Prime computers; the UNIX debuggers adb, gdb, sdb; and dbx; and the common programming environments for APL, Prolog, and LISP. Some of these provide syntax checking, highlighting, and/or formatting during code entry, which is useful but is no help with logic errors. And some provide an “evaluate mode” that allows a single statement or expression to be typed in, and executes or evaluates it interactively; however, this feature is very limited: It does not display program states automatically, as with a watch window in “debug mode”; and it is completely separate from “entry/edit mode,” in which the programmer enters statements to build a new routine. There is apparently an implicit assumption that the programmer will finish composing a routine before starting to test and debug it.

As we juxtaposed these two observations, a new feature occurred to us that could benefit programmers during the initial composition process. This feature would combine “entry/edit mode,” “evaluate mode,” and “debug mode”: As the programmer entered the code for a new routine, as soon as each statement was entered, the

¹ The terms “programmer,” “coder,” “software engineer,” and the like are subject to a variety of definitions, sometimes controversial. Here we use the word “programmer” to mean someone who may, when the occasion demands, design programs, compose program code, and do debugging.

environment would execute it and display the new program state. In this way, the programmer would be able to compare the actual consequences of a statement with its intended consequences, and to modify it immediately if necessary. The choice of objects to be displayed as the program state would be specified interactively, as with a debugger watch list. Furthermore, the execution should be able to show the program states for several different test cases. To use the latter feature, the programmer would specify a list of test cases beforehand, and optionally a corresponding list of intended states. The point would be to allow the programmer to offload as much of the memory burden and mental work as possible onto the machine, thereby freeing his/her cognitive resources for more creative tasks. We decided to call this feature “ahead-of-time” (AOT) debugging, since it promotes debugging before a routine is complete, rather than afterwards.

Our main hope and expectation was that use of AOT debugging would reduce time to complete a given programming task, and bug frequency in the code produced. We also expected that discovering and fixing an error in a given statement should help to prevent errors in subsequent statements. Finally, we predicted some affective side-effects of the offloading of cognitive burden: that programmers would experience greater satisfaction and lower levels of stress and fatigue, and might also adopt a freer, more experimental style of programming. The purpose of this project is to test these various hypotheses.

AOT debugging and the evolution of programming environments

An analogy with the evolution of other classes of software suggests that the AOT feature is a natural progression from current PDE designs. Consider how text-processing software tools (or “document development environments,” by analogy) have evolved in the past four decades. To summarize: In the 60’s one typed text on keypunch machines, programmed the formatting using a batch-mode markup language (e.g., Script or SGML), and waited for paper printout. In the ‘70s video terminals and line editors became common (e.g., Wylbur, QED, ed), allowing the user to edit text interactively, but displaying the changes on only one line at a time. In the early ‘80s line editors were supplanted by full-screen editors (e.g., XEDIT, emacs), allowing the user to edit anywhere on the screen and immediately see the results, but still only with typewriter text and embedded formatting commands; one had to get printout to see the actual formatting. In the ‘90s WYSIWYG word processing has become widely available (starting with Apple’s MacWrite in 1984, and now with all the major products, e.g., Microsoft Word and WordPerfect); these allow the user to interactively view and edit text of any style, exactly as it will appear when printed—one mode for the entire document development process. Thus there has been a progression of levels of interactiveness, i.e., increasingly immediate visibility of the consequences of user actions, with fewer and fewer modes to switch among, allowing the user to offload more and more memory and cognitive burden onto the computer, and to concentrate on creative tasks.

Programming environments for high-level languages have evolved similarly. In the '60s, the only "programming environment" entailed punch-card input and line-printer output; in the '70's, interactive line-editing of source code became standard, with remote submission and retrieval of batch jobs for compilation and execution; in the 80's full-screen editing and interactive compilation, execution, and symbolic debugging became common, though typically with separate software for each; and in the 90's program development environments have become widely available, providing all four functions in a single software product. As with text processors, there has been a progression of levels of interactivity. But notice that in current PDEs the functions are not fully integrated (notwithstanding the word "integrated" in some product names): "Edit/entry mode" is still completely separate from "execution/ debugging mode," and from "evaluate mode." The AOT debugging feature removes these boundaries, providing a truly integrated program development environment—as with WYSIWYG text processors, one mode for the entire program development process.

A fictitious example of AOT debugging

Consider the following programming problem, as attempted by a fictitious programmer whom we will call Tom. We have chosen an artificially simple problem for the sake of simplicity, but we can partly compensate by supposing that Tom is a relatively inexperienced programmer. The problem is to write a Pascal function, `pow`, that takes two real arguments, `x` and `y`, and returns x^y , computing the result with a loop when the value of `y` is an integer, otherwise with logarithms. In case the value is undefined, the function should return the predeclared constant `undef_real`. The function must satisfy the following 14 test cases:

```
pow(2.5,3)=15.6250; pow(-2.5,3)=-15.6250; pow(2.5,-3)=0.0640; pow(-2.5,-3)=-0.0640;
pow(2,1.5)=2.8284; pow(-2,1.5)=undef; pow(2,-1.5)=0.3536; pow(-2,-1.5)=undef;
pow(0,0)=1; pow(2,0)=1; pow(-2,0)=1; pow(0,1.5)=0; pow(0,-1)=undef; pow(0,-1.5)=undef
```

Suppose Tom begins by writing the following pseudocode, which contains several bugs:

```
if y is an integer —      val=1; for n=1 to y, val=val*x
if y is not an integer —  if x<=0 then val is undefined, else val=ex*log(y)
return val
```

Working from this pseudocode, in a conventional PDE he might compose the following Pascal code (for simplicity, we will assume there are no syntax problems):

```
function pow(x,y:real):real; {returns x to the y power}
var val:      real;          {value to be returned}
    n:        integer;      {counter for loop}
begin
```

```

if y=trunc(y)                {if y is an integer...}
then begin
  val:=1;                    {init val to 1 before loop}
  for n:=1 to y do           {loop y times}
    val:=val*x               {accumulate product of val}
  end
else                          {if y isn't an integer...}
  if x<0                     {if x is negative}
  then val:=undef_real       {value is undefined}
  else val:=exp(x*log(y));   {otherwise use log}
pow:=val                      {return val}
end;

```

Notice that in the translation and entry steps, this code inherited all the bugs in the pseudocode, and gained more besides. Using the PDE, Tom adds a driver program around the function to test it, and tests $\text{pow}(2.5,3)$, which happily yields 15.625000. He proceeds to test further cases, but soon runs into cases where the function fails. He has trouble understanding which statements are causing which cases to fail, and is not quite sure which statements he intended to handle which cases. After many tests using the debugger's step-through and variable watch features, Tom discovers the bugs: the cases he missed originally (negative integer exponents, negative bases with non-integer exponents), the reversal of x and y and the use of \log rather than \ln in the original formula, and the miscopied \leq symbol. With persistence he fixes them, although while doing so he introduces a new bug 0 with a negative integer exponent), which he also has to fix.

Now let us return to the original pseudocode, this time with Tom starting out using the AOT debugging environment. He first enters the 14 triples of values for x , y , and the required value for $\text{pow}(x,y)$ in the test cases window, and specifies "all variables" for the program state variables. He then starts to enter the same code as before:

```

function pow(x,y:real):real; {returns x to the y power}
var val:      real;          {value to be returned}
    n:        integer;       {counter for loop}

```

In response to each of these three lines, the program state window shows first x and y , then val , and finally n , all as undefined for all 14 cases. Tom continues:

```

begin
if y=trunc(y)                {if y is an integer...}
then begin
  val:=1;                    {init val to 1 before loop}

```

The environment is passive until Tom types this statement, at which point the program state window updates the value of val to 1 for the eight cases in which y is an integer. Tom continues:

```

for n:=1 to y do
    val:=val*x
end

```

{loop y times}
 {accumulate product of val}

Here the program state window shows some new values for those eight cases: n becomes y+1 in each case, and val gets a new value (x^y) in the two cases where y is positive, pow(2.5,3) and pow(-2.5,3). Tom is surprised because he expected val to be updated for the negative y cases too (pow(2.5,-3) and pow(-2.5,-3)). But since he just wrote that statement, his intentions for the loop, the details of its logic, and the relevant variable states are all fresh in his mind, and he easily sees where the bug is, and changes the code to:

```

for n:=1 to abs(y) do
    val:=val*x;
if y<0 then val:=1/val
end

```

{loop abs(y) times}
 {accumulate product of val}
 {if y negative, use val's reciprocal}

Now, to Tom's surprise, the state of val for the case of pow(0,-1), changes to "overflow"! But again, since he has just composed the buggy statement, he quickly understands the cause of the problem and replaces the line with:

```

if y<0
then if x=0
    then val=undef_real
    else val:=1/val

```

{if y negative...}
 {if x=0, val=0}
 {0 to neg power is undefined}
 {if x<>0, use val's reciprocal}

Notice how in this situation, fixing one bug introduced another one. In a conventional PDE this second bug might have gone unnoticed for a while, and would likely be approached as completely unrelated to the one from which it propagated. In the AOT environment, that kind of propagation is likely to be caught immediately, while the logic associated with it is fresh in the programmer's mind. Tom verifies that all eight integer-exponent cases are correct, and continues with:

```

else
    if x<0
    then val:=undef_real

```

{if y isn't an integer...}
 {if x is negative}
 {value is undefined}

The states of val for pow(-2,1.5) and pow(-2,-1.5) change to undef_real as Tom expects, but he is surprised to see that the same didn't happen for the other two undefined cases. Because these are the x=0 cases, it is immediately obvious that he has copied \leq in the pseudocode as $<$, so he makes the change and the other two cases are now shown correctly. Next he enters:

```

else val:=exp(x*log(y));

```

{otherwise use log}

The states of `val` for the two remaining cases change, but to the wrong numbers—1.422168 for `pow(2,1.5)` instead of 2.828427, and “overflow” for `pow(2,-1.5)` instead of 0.353553. Referring to the pseudocode, Tom sees that `x` and `y` are backwards and switches them, but still the values are wrong! Using a conventional debugger here, one might assume the bug could be anywhere in the whole routine. But in the AOT environment, Tom knows that all the code down to this point in the routine is already debugged, so he has high confidence that it is worth investing a lot of thought in this one statement. Rethinking the mathematics from first principles, sure enough he realizes his error in the original pseudocode, and replaces the `log` function with `ln`. The states of `val` for the remaining two cases now become correct. He concludes the function with:

```
pow:=val          {return val}
end;
```

At the instant that Tom is done entering the code, he is also done debugging it.

This example is fictitious, although it is based on observed user interactions with the AOT debugger prototype. It is also unrealistic: Increasingly in modern software development, programmers work not with language primitives, but with high-level objects, some original but most from a standard library; the coding and debugging issues concern the behaviors and interfaces of individual objects, and the interactions among them. However, we suspect that the benefits of AOT debugging are largely independent of the level of programming; this is another hypothesis we intend to test.

Experience with a prototype AOT debugging environment

We have built a small prototype programming environment that implements the AOT debugging feature. It incorporates an interpreter/debugger for almost the full Pascal language (written in C) [Mak 91], and is driven by a standard graphical user interface running in Windows 95 (written in Delphi). As the user types each line of code into the edit window, the interpreter is invoked automatically and displays in another window the new program state for each of several test cases. The interface includes windows in which the user specifies lists of test data and variables for the program state, which the user can modify interactively at any time. It also includes controls and windows for loading and saving the source file, specifying its name, running the complete program, issuing commands to the debugger, displaying interpreter and debugger messages and program output, etc.

To date we have done only informal testing with a few users (students and colleagues), in the process of refining our hypotheses, factor definitions, methodology, and the software itself. Users are given printed specifications, pseudocode, and test data for short programs, and asked to code and validate the programs as quickly as possible, using either the prototype AOT debugging environment or a conventional PDE (the Borland Turbo Pascal Integrated Development Environment). Before a session each

user is asked to rate his/her levels of stress and fatigue, and afterwards the same ratings are requested, as well as level of satisfaction with the programming environment, and any other relevant comments. So far we have made the following general observations:

1. Users detect and fix a substantial number of bugs during initial code entry when using the AOT environment, compared with almost none using a standard PDE.
2. Propagation of logic errors is reduced by using the AOT environment.
3. The AOT users generally produce code faster and with fewer errors in the process, as well as fewer bugs in the end result.
4. AOT users report a higher degree of satisfaction and smaller increases in stress and fatigue than users of the standard PDE.

There are also some anecdotal observations that seem interesting. Some users, when first introduced to the AOT environment, have made remarks like “This makes programming fun” and “This is what I’ve always needed but didn’t know it.” Several have stated that they regret having to go back to using other PDEs, since AOT debugging is more productive. And several have said that, in comparison with the AOT environment, using the standard PDE feels like “programming in the dark.”

As the project continues, we plan to improve the prototype in several ways, including: (1) Enhancements to the Pascal interpreter, e.g., language extensions and additional debugger commands; (2) Enhancements to the AOT environment itself, in particular, a feature for graphically displaying state changes in linked data structures (Stasko and Badre 1993; Schreiweis 1993); and (3) Features for automatically collecting timings, session snapshots, and other data on user interaction with the system, to be used in its evaluation.

We are now planning formal trials with a large number of subjects to test the validity of our preliminary observations, as well as some other hypotheses. For example, we have not yet had enough experience to suggest whether AOT debugging promotes greater freedom and experimentation in programming style. Nor do we have evidence about whether the apparent benefits for low-level programming generalize to higher levels. We plan to use as subjects a wide range of programmers, from beginning students to seasoned professionals, to allow analyzing factors such as users’ previous experience and training in program design, coding, debugging, mathematics, etc. (Holt et al. 1987). We also intend to use state-of-the-art psychometric instruments to measure user satisfaction (Chin et al. 1988), stress (Galinsky et al. 1993; Hancock and Warm 1989), mental workload and fatigue (Hancock and Caird 1993; Hendy et al. 1993; Hill et al. 1992; Wierwille 1993), and perhaps other factors (Treu 1994).

In summary, based on preliminary experience with our prototype, the AOT debugging feature appears to significantly improve programmer productivity and software quality, and would be a beneficial enhancement for any program development environment.

References

- Barstow, David R., Howard E. Shrobe, and Erik Sandewall, eds. *Interactive Programming Environments*. McGraw-Hill, 1984.
- Boyle, T. and B. Drazkowski. "Exploiting natural intelligence: Towards the development of effective environments for learning to program." In A. Sutcliffe and L. Macaulay, eds., *People and Computers V* (British Informatics Society), Cambridge Press, 1989.
- Brooks, R. "Towards a theory of the cognitive processes in computer programming." *Int'l. Jnl. Of Man-Machine Studies*, Vol. 9 (1977), pp. 737-751.
- Brusilovsky, Peter. "Program visualization as a debugging tool for novices." *Adjunct Proc. INTERCHI '93, Human Factors in Computing Systems*, pp. 29-30. ACM, 1993.
- Card, Stuart K., Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.
- Chin, J.P., V.A. Diehl, and K.L. Norman. "Development of an instrument measuring user satisfaction of the human-computer interface," *Human Factors in Computing Systems, Proceedings of the CHI '88 Conference*, pp. 213-218. ACM Press, 1988.
- Dix, Alan, Janet Finlay, Gregory Abowd, and Russell Beale. *Human-Computer Interaction*. Prentice Hall, 1993.
- Eisenstadt, Marc. "Tales of debugging from the front lines." In C. Cook, J. Scholtz, and J. Spohrer, eds., *Empirical Studies of Programmers: Fifth Workshop*, pp. 86-112. Ablex, 1993.
- Fritzson, Peter, Tibor Gyimothy, Mariam Kamkar, and Hanid Shahmehri. "Generalized algorithmic debugging and testing." *Proceedings of ACM SIGPLAN '91, Conf. on Programming Language Design & Implementation*, pp. 317-326. ACM Press, 1991.
- Galinsky, Traci L., Roger R. Rosa, Joel S. Warm, and William N. Dember. "Psychophysical determinants of stress in sustained attention." *Human Factors*, Vol. 35, No. 4 (Dec. 1993), pp. 603-614.
- Gould, J.D. "Some psychological evidence on how people debug computer programs." *Int'l. Jnl. Of Man-Machine Studies*, Vol. 7 (1975), pp. 151-182.
- Green, Thomas R.G. "The nature of programming." In J.-M. Hoc et al., eds., *Psychology of Programming*. Academic Press, 1990.
- Gugerty, Leo and Gary M. Olson. "Comprehension differences in debugging by skilled and novice programmers." In Elliot Soloway and Sitharama Iyengar, eds., *Empirical Studies of Programmers: First Workshop*, pp. 13-27. Ablex, 1986.
- Hancock, P.A. and J.K. Caird. "Experimental evaluation of a model of mental workload." *Human Factors*, Vol. 35, No. 3 (Sept. 1993), pp. 413-430.
- Hancock, P.A. and Joel S. Warm. "A dynamic model of stress and sustained attention." *Human Factors*, Vol. 31, No. 5 (Oct. 1989), pp. 519-538.
- Hendy, Keith C., Kevin M. Hamilton, and Lois N. Landry. "Measuring subjective workload: When is one scale better than many?" *Human Factors*, Vol. 35, No. 4 (Dec. 1993), pp. 579-602.

- Hill, Susan G., Helene P. Iavecchia, James C. Byers, Alvah C. Bittner, Allen L. Zaklad, and Richard E. Christ. "Comparison of four subjective workload rating scales." *Human Factors*, Vol. 32, No. 4 (Aug. 1992), pp. 429-439.
- Holt, Robert W., Deborah A. Boehm-Davis, and Alan C. Schultz. "Mental representations of programs for student and professional programmers." In G. Olson, S. Sheppard, and E. Soloway, eds., *Empirical Studies of Programmers: Second Workshop*, pp. 33-46. Ablex, 1987.
- Katz, Irvin. R. and John R. Anderson. "Debugging: An analysis of bug-location strategies." *Human-Computer Interaction*, Vol. 3, No. 4 (1987-1988), pp. 351-399.
- Kline, Paul. *The Handbook of Psychological Testing*. Routledge, 1993.
- Klint, Paul, Thomas Reps, and Gregor Snelting, eds. "Programming environments / Report on an international workshop at Dagstuhl Castle." *ACM SIGPLAN Notices*, Vol. 27, No. 11 (Nov. 1992), pp. 90-96.
- Koenemann, Jürgen and Scott P. Robertson. "Expert problem-solving strategies for program comprehension." *Proceedings of INTERCHI '91, Conference on Human Factors in Computing Systems*, pp. 125-130. ACM Press, 1991.
- Mackie, R.R., C.D. Wylie, and M.J. Smith. "Comparative effect of 19 stressors on task performance: Critical literature review (what we appear to know, don't know, and should know)." *Proceedings of the Human Factors Society 29th Annual Meeting*, pp. 462-469. Santa Monica, CA: HFES, 1985.
- Mak, Ronald. *Writing Compilers and Interpreters / An Applied Approach*. Wiley, 1991.
- Nanja, Murthy and Curtis R. Cook. "An analysis of the on-line debugging process." In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, eds., *Empirical Studies of Programmers: Second Workshop*, pp. 172-184. Ablex, 1987.
- Moray, Neville, ed. *Mental Workload: Its Theory and Measurement*. Plenum, 1979.
- Pennington, Nancy. "Comprehension strategies in programming." In Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, eds., *Empirical Studies of Programmers: Second Workshop*, pp. 100-113. Ablex, 1987.
- Schreiweis, U., A. Keune, and H. Langedörfer. "An integrated Prolog programming environment." *ACM SIGPLAN Notices*, Vol. 28, No. 2 (Feb. 1993), pp. 53-60.
- Shneiderman, Ben. *Designing the User Interface / Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 1987.
- Shneiderman, Ben. *Software Psychology / Human Factors in Computer and Information Systems*. Little, Brown, 1980.
- Stasko, John and Albert Badre. "Do algorithm animations assist learning? An empirical study and analysis." *Proceedings of INTERCHI '93, Conference on Human Factors in Computing Systems*, pp. 61-66. ACM Press, 1993.
- Treu, Siegfried. *User Interface Evaluation / A Structured Approach*. Plenum, 1994.
- Weinberg, Gerald M. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.
- Wickens, Christopher D. *Engineering Psychology and Human Performance*. Harper Collins, 1992.

Wierwille, W.W. and F. Thomas Eggemeier. "Recommendations for mental workload measurement in a test and evaluation environment." *Human Factors*, Vol. 35, No. 2 (June, 1993), pp. 263-282.