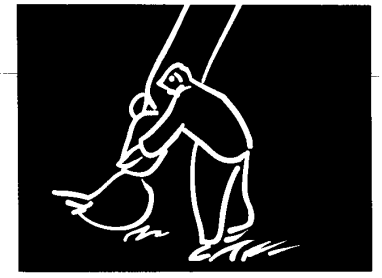

Error Reporting with Graduated Color

BRUCE OBERG, *Microsoft*
DAVID NOTKIN, *University of Washington*

◆ *A technique to avoid interruptions during programming uses color and information hiding to provide error reports on demand, rather than when you least want them.*

Although the usefulness of color displays is well accepted in applications like hardware design, many people are not convinced that color serves any special purpose in a programming environment. However, we found in investigating practical applications of color, that it can be used quite effectively to notify the programmer of errors without forcing him to correct the error in the middle of a task.

Color for nonintrusive error notification is not a new concept. Some spell checkers display misspelled words in a distinct color, for example, without requiring the user to correct the error immediately. In programming, however, simply using color to display an error is not enough. The programmer must have some way of knowing how old the error is and more detailed information about it in the form of an error report.

To satisfy those requirements, we combined graduated color with elision—the temporary hiding of information—and applied it to the Cornell Synthesizer Generator.[1] Our goal was only to demonstrate that color can be used in this way, not to produce a validation study. Although we have not tried to formally measure productivity, the resulting prototype seems pleasing and largely natural, and there should be at least some productivity increase because programmers are not interrupted in the middle of a task and error-navigation paths are not limited.

The amount of work was quite moderate, in large part because we enhanced a synthesizer generator that was written extensibly with good facilities for error reporting and display control. Nonetheless, the techniques we use are straightforward and enhancements might be easier for a tool than for a tool generator.

The prototype is just over 1,000 lines of uncommented C code. Although it is suitable only for X Window System, Version 11 Release 3 editors, we tried to make design decisions that would facilitate inte-

0740-7459/92/1100/0033/$03.00 © IEEE

gration with other display systems as well. Bruce Oberg, who designed, coded, and tested the implementation, had prior development experience, but he did not have to know anything about the generator or the specific display systems we eventually used.

GRADUATED COLOR AND ELISION

Graduated color and elision are a powerful combination. Graduated color provides the nonintrusiveness and indication of elision, while elision provides the full error report only when it is requested. The users can see errors and their age by the color and then, by clicking on a mouse, for example, can look at the associated error explanation when they are not busy. Interruption is kept to a minimum during notification, and the explanation is close at hand and complete when it is wanted. Also, combining graduated color and elision made it easy to divide error reporting into the smaller, easier jobs of notification and explanation.

Notification. The role of notification is to indicate only that an error has occurred, so it can be made less intrusive than error reporting. On the other hand, delaying full notification of an error merely because you don't want an interruption is not desirable either because the user may believe that some later, unrelated action was related to an earlier error. This is where the variance of color can be useful.

Graduated color preserves the age of the error; the older the error, the more intense the color. If the user fixes the problem very quickly, the slight coloring would be hardly noticeable. Program editors are particularly suited for this approach because even expert users can erroneously insert a new variable before its declaration. Slowly increasing the intensity of a notification color lets the user continue a train of thought. Making old— and thus possibly more critical— errors gradually more visible makes the user aware that the error is old and that it's probably about time to look into it. Although we do not persistently store information about the age of errors in our prototype, we see no practical difficulties in doing so.

Graduated color preserves the age of the error. Elision provides a complete error report when needed.

This use of color is different from most applications. Color is not representing an object in the real world, nor is it strictly ornamental. In graduated color, the color both indicates an object's status and distinguishes it from others.

Explanation. To handle explanation, it might be tempting to create new visual objects, like windows. However, if you then used a color to distinguish an explanation or to link it with the actual error, you could run into color overload— the human eye can distinguish only about seven colors at a time.[2,3] To avoid this, we handle error explanation through elision. Elision lets the user focus on the task at hand. An example is the outlining modes in some word-processing systems, which let the user focus on the structure of a document while temporarily disregarding the actual text.

Elision involves the reformatting of an existing display object, usually with some sort of abbreviation to indicate that elision has taken place. Most certainly, elision is only useful from an error-reporting standpoint if there is some indication that an object is elided; hiding too much error information is equivalent to not reporting at all.

CHANGES TO THE GENERATOR

Given a description of a language's syntax and static semantics, the Cornell Synthesizer Generator produces a syntax-directed editor that supports interactive, incremental error checking. Release 3.0 lets you use the color facilities of several window systems for ornamental coloring. The editors included with the generator report errors either immediately, usually as a comment, or not at all.

The generator provides a good base to implement graduated color. It has facilities for advanced visual displays, error handling, and simple elision, as well as fully functional editors for various languages. Because the mechanisms for error detection and reporting were already there, our only task was to separate them.

The changes we made were to the generator's Synthesizer Specification Language, described in the box on p. 35, and we did not have to change SSL to support color. We made the following changes:

- Expanded SSL style and style-file capabilities to let the generated editors use graduated (or plain distinctive) color and to allow different coloring schemes for different errors.
- Added the concept of time passage.
- Simplified elision.

Expanding style. We expanded style descriptions to define two attributes, foreground and background color:

```
foreground ::= "*"[color][":"[color]
        ["/"[speed][":"[delay]]]]
background ::= "#"[color][":"[color]
        ["/"[speed][":"[delay]]]]
  color ::=
 description | "*" | "#" | " "
  speed ::=
  delay ::=
```

The simplest way to use these attributes is to specify only the first color. Text with that style is then colored as designated. For example, substituting "blue" for the first occurrence of [color] in the foreground line would make the letters of the text blue; substituting "yellow" for the first occurrence of [color] in the background line would make the background yellow. In X11R3 (X Window System, Version 11, Release 3), the description part is a color name that the system translates into the appropriate low-level color description.

A more complex use of foreground and background is to specify two colors separated by a colon. This means that an object's color should move from the

first color to the second as the object gets older. That is, the object is initially drawn in the first (original) color, and as time passes, the color is changed to be more and more like the second (destination) color.

Eventually, the object will have the destination color, which will not change. Either (but not both) of the two color descriptions can be left blank, indicating that the current background or foreground color should be used; #:magenta, for example means that an object's background color should change from the current background to magenta. Specifying * or # for a description indicates that the default X11R3 foreground (or background) for the editor window should be used.

Adding time passage. How fast an object's color moves is specified by the speed part of the foreground and background colors. Speed is specified as an integer from 1 to

SYNTHESIZER SPECIFICATION LANGUAGE

The Cornell Synthesizer Generator uses the Synthesizer Specification Language to specify an editor for a particular language. All the examples here, for instance, are taken from the Pascal editor description that comes with the generator. Some are edited for clarity.

Program elements are described within SSL in terms of *phyla*, which can be thought of as prototype descriptions of the nodes in a language's syntax tree. Phyla are very closely related to productions In context-free grammars or Backus-Naur format descriptions. For example, a for statement phylum such as

```
statement: ForTo (identi-
  fier expr expr statement)
```

specifies that it is made up of an identifier, start and stop expressions, and a target statement. The identifier phylum specifies that it can be only an alphanumeric token; the expression phylum specifies that it can be a constant, an identifier, or a combination of two expressions with a binary operator between them. The actual program the user enters consists of instances of phyla, called *productions*.

A phylum's display format is described separately from its structure. Unparsing strings describe how the editor displays a production. A possible unparsing string for the for statement phylum is

```
ForTo[ ^ : "for" @ " :=
  "@" to "@" do %t%n" @
  "%b"]
```

When an instance of a for statement is to be displayed, the strings within double quotes are output (almost) verbatim. The @ signs are placeholders corresponding to the four components of a for statement in the phylum's definition. The unparsing string for the components is inserted at their respective placeholders when the for statement is displayed.

The % escapes within double quotes are similar in intent to formatting commands used by the printf() function in C. The %t, %n, and %b escapes direct the editor to increment the tab level, insert a new line, and decrement the tab level, respectively. These and other escapes let you fine tune the display of productions. The pair of escapes of concern to us in our modification is "%S(" and "%S)."

Within an unparsing string, "%S(" and "%S)" bracket text that will be displayed in a specific style. In this example, keyword parts of the for statement unparsing string are bracketed to be shown in a style called Keyword:

```
ForTo[ ^
"%S(Keyword:for%S)" @
  := "@"
    "%S(Keyword:to%S)"=@
    "%S(Keyword:do%S)
      %t%n" @ "%b"]
```

SSL requires that the Keyword style be declared among a list of possible styles, but the attributes associated with Keyword are not specified. Instead, the user defines them at runtime in a style file:

```
fonts {timr24, helvr18};
largest: timr24, +bold ;
Normal: timr24 ;
Keyword: +bold ;
Placeholder: +italic ;
Error: +bold, +italic ;
Comment: helvr18;
```

All fonts are listed first in the fonts section of the style file. The exact interpretation of font names depends on the display system. Subsequent entries associate attributes with particular styles. Specifying a font name tells the system to use that font once the style is invoked via %S. If no font is listed, the editor's current font is not changed when the style is invoked. The +bold specification indicates the use of the bold form of the style's font; +italic indicates the italic form. The –attribute form indicates that an attribute (such as italic) should be turned off; !attribute toggles an attribute.

SSL also provides a minimal elision capability. Any phylum can list two unparsing strings: a default (the first listed) and an alternate (the second):

```
ForTo [ ^ : "for" @ " := "
@ " to " @
    " do %t%n" @ "%b"
]
    [ ^ : "for" @ error " := " @
      to " @
    " do %t%n" @ "%b"
```

While using the editor, the user can select a production and request that it be displayed using a particular unparsing string or whichever unparsing string is not current. In this example, an alternative scheme can be used to elide a production's error reports. The error symbol is a local variable to each ForTo production and when such a production contains an error, the error variable for that production is set to a string describing the error. By toggling between unparsing strings, the user can turn error reporting on and off.

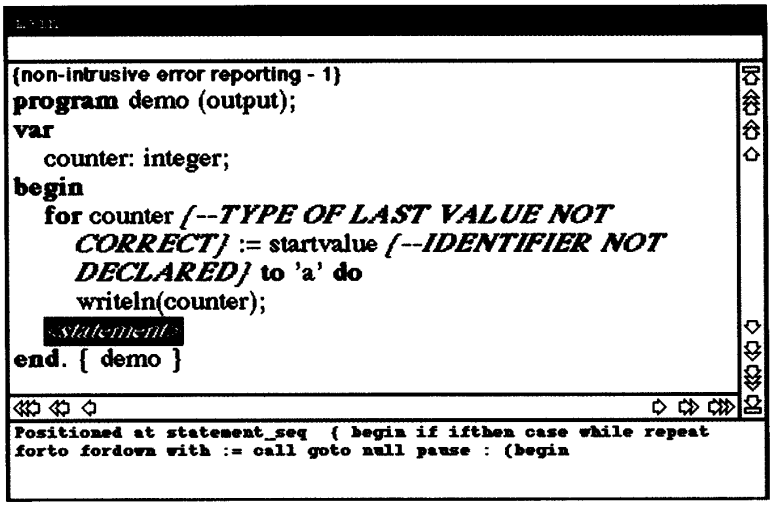

Figure 1. Intrusive error reporting.

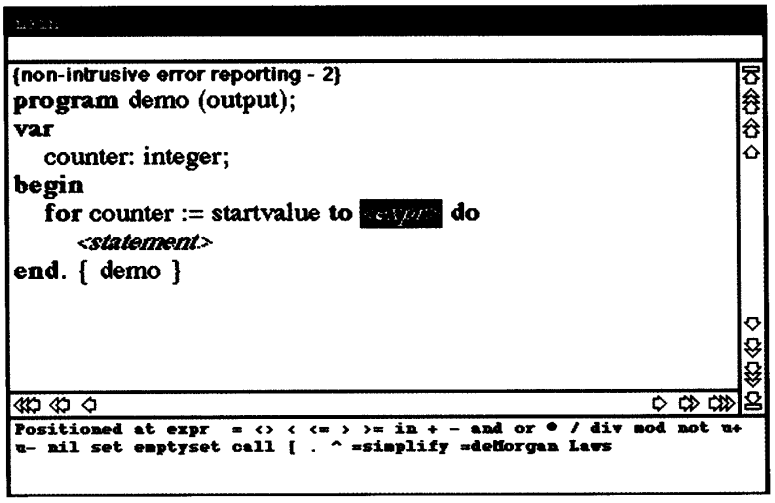

Figure 2. No initial error reporting.

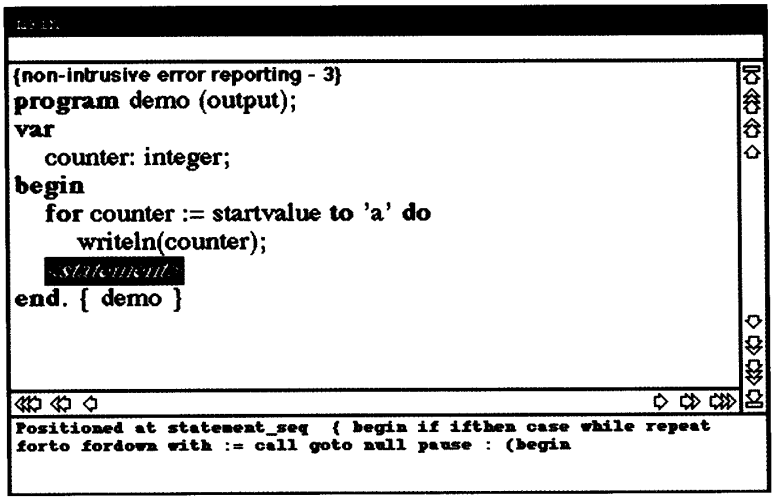

Figure 3. Multiple unreported errors.

100 and describes what percentage of the destination color should be mixed in after each unit of time, or tick, passes. A tick corresponds to a keystroke.

The keystroke convention seems reasonable, compared with a real-time definition of ticks. Not only is it cheaper (you avoid the cost of alarm-signal interrupt handlers), but keystrokes seem to be a more accurate measure of how much work the user is doing. While the user thinks about a passage of code without typing anything, real time is passing and a changing color might be unnecessarily intrusive. Timing through keystrokes provides some measure of assurance that moving color will be distanced from what the user is concentrating on. In addition, the use of keystrokes could easily be expanded to encompass all user events (such as mouse clicks).

The default speed is 1, which means that an object using a style that contains the attribute #yellow:red could initially have a yellow background, but after one tick (keystroke), the background will be changed to be a yellow-red ratio of 99:1. After 50 ticks, the ratio would be 50:50 (orange), and, after 100 ticks, the background would be entirely red and remain that way.

A color's motion may be postponed. Specifying the delay (again, an integer) part of the moving color attribute postpones adding the destination color into the mix until after the specified number of ticks. For example, the attribute *green:purple/10:20 means that text is originally displayed in green, and, after 20 ticks, purple is mixed in at the rate of 10 percent a tick.

Simplifying elision. When an editor is generated in the unmodified version of the generator, the middle mouse button has no function. The other two are used for location and selection and menu-based commands. To make elision of a selected area easier, we made the middle button act as the location and selection button adding the ability to switch unparsing strings for that area. When the user clicks on the middle button, whatever area is selected is given to the alternate-unparsing-toggle command. Alternate

unparsing strings are used to hide error messages while keeping coloring intact. Thus, the user clicks on a colored area to display the error message.

SAMPLE DISPLAYS

Figures 1 through 8 are screen images from a Pascal editor that we generated using the modified version of the generator. The following style file defines the policy used in the editor that produced the figures.

```
fonts {timr24, helvr18};
largest : timr24, +bold ;
Normal: timr24 ;
Keyword: +bold, *blue ;
Placeholder: +italic, #yellow ;
ErrorA: #:red/5:30 ;
ErrorC: #:magenta/:20 ;
ErrorMsg: +bold, +italic ;
Comment: helvr18;
```

This style uses graduated colors to distinguish categories of errors. Keywords are written in blue, while placeholders of unspecified parts of the syntax tree have a yellow background. Different background colors distinguish the two error types: undeclared variables (Error A) are displayed in red and type mismatches (Error C) are displayed in magenta. We describe more about this implementation elsewhere.[4]

Figure 1 shows how normal, intrusive error reporting works within the editor. When both errors are detected, the full error message is immediately inserted as a comment. Figure 2 shows how the errors appear in the modified editor. Even multiple errors will not trigger a reporting mechanism, as Figure 3 shows.

Figure 4 shows the errors after a small amount of time. Our policy specifies that the color of type mismatches must change more quickly than the coloring of undeclared variables. Thus, the coloring of the type mismatch has started, but coloring for the undeclared variable has not, even though the undeclared variable was entered first.

Error A's background moves faster and has the longer delay. Figures 5 and 6 show fully saturated instances of Error A. Error C has a slower changing background and shorter delay, as Figures 4 through 7

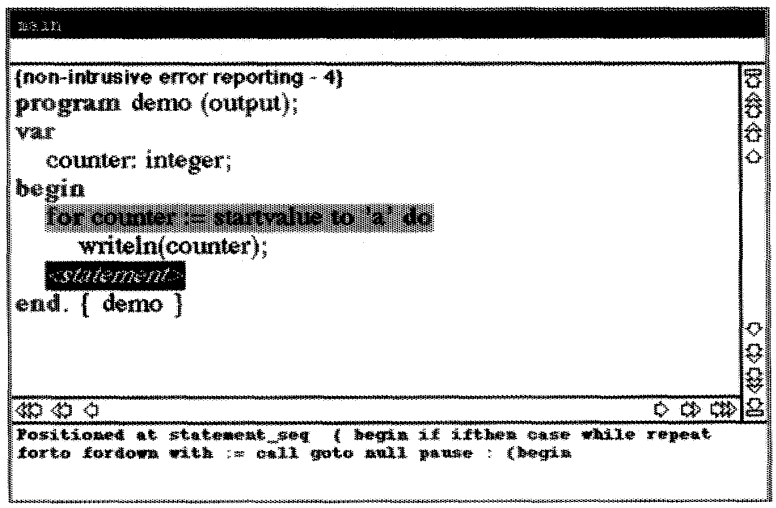

Figure 4. Partial, independent shading.

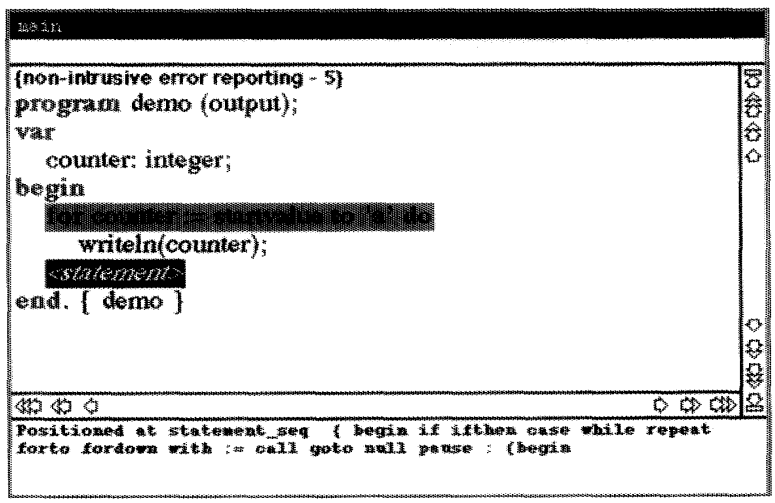

Figure 5. Full shading.

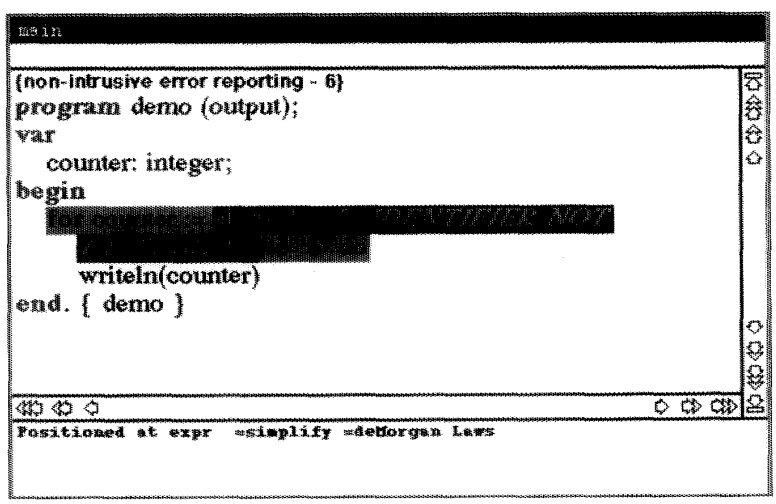

Figure 6. Elision.

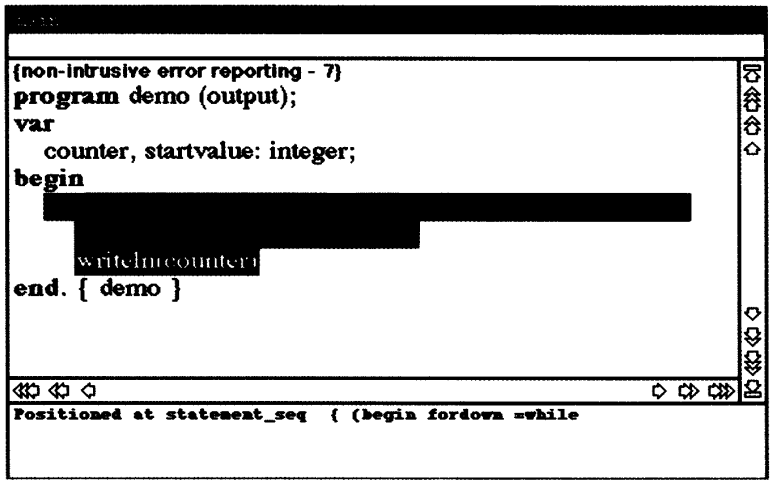

Figure 7. Correction and elision.

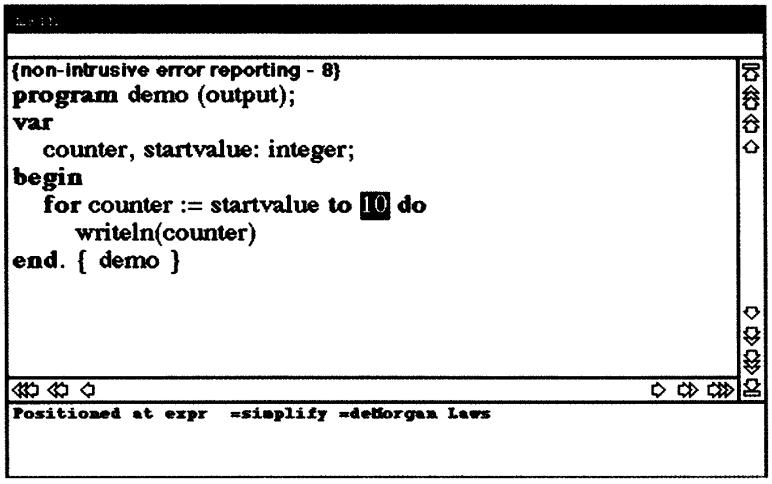

Figure 8. All errors corrected.

show. A curious race occurs if two errors of types Error A and Error C are created at the same time. Which will get to full saturation first? Error C will get the jump because of Error A's longer delay. After 10 ticks, though, Error A starts up and takes it to full saturation after 50 ticks, when Error C is still at only 30-percent saturation.

After enough time passes, both error sections become fully colored, as Figure 5 shows. Using elision, the user could expose the exact nature of either error. Figure 6 shows the result of eliding the undefined symbol error. The elision of regions with errors and the resolution of those errors could be done in any order, at the discretion of the user. In Figure 7, the undefined symbol has been corrected and the type mismatch has been elided. When all apparent errors have been corrected, as in Figure 8, no colors indicating errors remain. Note that a semantic error still exists within the program in Figure 8; the variable startvalue is used before it is set. This is the result of less-than-adequate error detection within the editor. If an error cannot be detected, any form of reporting mechanism becomes useless.

Is nonintrusive error reporting worth it? The answer depends on how useful you find it and how much effort is involved. Although our prototype is small and the resulting evidence is largely anecdotal, we believe it demonstrates the feasibility of pursuing such an approach either in larger, more widely used systems or in cognitive experiments that consider the effectiveness of using graduated color for error reporting.

We do not recommend any specific coloring policy— not even the one we used. Our purpose is merely to provide a way for system designers and interface experts to exploit and experiment with color in novel ways. ●

ACKNOWLEDGMENTS

This work was done while Oberg was an employee of AT&T Bell Laboratories. We thank Steve Mann, who helped with the original photographs for the figures.

Research for this article was supported in part by National Science Foundation grants CR-8858804 and CCR-9113367 and by grants from Digital Equipment Corp. and Xerox.

REFERENCES

1. T. Reps and T. Teitelbaum, *The Synthesizer Generator: A System for Constructing Language-Based Editors*, Springer Verlag, Berlin, 1988.
2. P. Robertson, "A Guide to Using Color on Alphanumeric Displays," Tech. Report TR.12.183, IBM UK Laboratories Ltd., Hursley Park, UK, 1979.
3. J. Rice, "Display Color Coding: 10 Rules of Thumb," *IEEE Software*, Jan. 1991, pp. 86-8
4. B. Oberg, "Nonintrusive Error Reporting: One Use of Interactive Color Displays Within Software Environments," masters thesis, Univ. of Washington, Seattle, 1989.

Bruce Oberg is a software-design engineer at Microsoft, where he is interested in software engineering, cryptography, and virtual reality.

Oberg received a BS in mathematics from the University of Nebraska, Lincoln, and an MS in computer science from the University of Washington.

David Notkin is an associate professor of computer science and engineering at the University of Washington. His research interests are software engineering, software evolution, software restructuring, programming environments, and parallel and distributed systems.

Notkin received an ScB from Brown University and a PhD from Carnegie Mellon University.

Direct questions about this article to Notkin at CS and Eng. Dept, University of Washington, FR-35, Seattle, WA 98195; Internet notkin@cs.washington.edu.