A User's Guide to

The COPE Programming Environment

Richard Conway
Dan DeJohn
Steve Worona

TR 84-599
April 1984

Department of Computer Science
Cornell University
Ithaca, New York 14853
A User's Guide to

The COPE Programming Environment

Richard Conway, Dan DeJohn and Steve Worona
Cornell University

1. Introduction

COPE is an experimental prototype of a new approach to developing and testing programs. Systems like COPE are called "programming environments" because they provide an encompassing set of tools for the task -- all that is required in one integrated package.

Many modern programming environments include "smart" editors -- so-called because the rules of the programming language are built into the editor, permitting immediate response if the user violates those rules. COPE probably carries this idea further than any other environment.

Environments like COPE are called "complete" if, in addition to the editor that manages the writing of the program, there is provision for execution of the program under close supervision. Environments like COPE are called "integrated" and "incremental" if even an incomplete program can be executed (up to a point), and if a program can be changed in the middle of execution (while paused) without necessarily having to begin execution again from the beginning.

However, in two important respects there are no other environments like COPE. First, COPE assumes full responsibility for ensuring that the program is complete and correct at every stage in the development process. At the beginning of development of a program, COPE presents the user with a short, but complete and correct program. For example, if the user said that the name of the file was SAMPLE, COPE would automatically supply the following initial procedure in that file:

```
sample: proc;
  section COMMENT;
  trace(8);
  delay(1);
  end;
end sample;
```

Notice that even at this stage the procedure is complete, correct, and executable, although obviously its execution does not accomplish anything useful. Uppercase words, such as "COMMENT", are used as "prompts" to show the user those places where COPE was unwilling to invent a specific example and instead gives a description of the type of construction that the language requires. (Entries can be in either upper or
Now, starting with this initial procedure, suppose the user types the phrase "WHILE x||7<y y" into the line where the flashing cursor invites an entry. The screen would appear as follows:

```plaintext
sample: proc;
  WHILE x||7<y y
  section COMMENT;
  trace(8);
  delay(1);
  end;
end sample;
```

At this point, the program is momentarily incorrect, according to the rules of the PL/CS language. Perhaps the user made this particular entry because his recollection of the rules was somewhat hazy, or perhaps because, having used COPE previously, he knew that this was sufficient. In either event, whenever he gives COPE a chance to examine the program -- by pressing the ENTER (or RETURN) key, or by moving the cursor to another line -- COPE immediately restores correctness, yielding the following display:

```plaintext
sample: proc;
dcl (x) char(80) var;
dcl (y) char(80) var;
section COMMENT;
trace(8);
delay(1);
do while (x||7<y);
  put list (y);
end;
end sample;
```

An expanded form of the loop suggested by the user's entry has been generated and placed where it belongs (not where the entry was made), and the necessary declarations (with default attributes) have been placed where declarations belong. The program is again complete, correct, and executable, and in this case, probably even coincides with what the user had in mind.

At this point, you might wonder what sort of entries COPE will tolerate. You might experiment, trying some entries that are close to correct form, and some that are far afield. For example, suppose the text "WHEN ELSE THITHER TO FLY" were entered at the cursor position, as follows:

```plaintext
sample: proc;
dcl (x) char(80) var;
dcl (y) char(80) var;
section COMMENT;
```
trace(8);
delay(1);
do while (x||7 < y):
    put list (y);
    WHEN ELSE THITHER TO FLY
end;
end sample;

COPE's response is the following complete, correct, executable, albeit non-sensical PL/CS program:

sample: proc;
dcl (x) char(80) var;
dcl (y) char(80) var;
dcl (thither) float;
dcl (fly) float;
section COMMENT;
do while (x||7 < y):
    put list (y);
    select;
    when CONDITION if CONDITION
        then :
        else put list (thither);
        when CONDITION do ASSIGNMENT to fly by 1:
        _
        end;
    otherwise;
    end;
end;
end sample;

COPE should be able to restore correctness, no matter what the user does, although the current version does not repair expression (see Section 1.1).

While exploring the COPE response to abbreviated entries, you will soon discover the virtue of COPE's second distinctive characteristic -- a powerful recovery facility. In cases where your entry is ambiguous, or too cryptic even for COPE, the response may be different from what you expected and intended. When that happens, the UNDO command completely undoes the COPE response, restoring the screen to the state where you had completed the last entry. This can then be modified, and resubmitted.

This recovery facility is natural and convenient, and would be useful even in systems whose response is not as potentially surprising as that in COPE. However, in COPE the recovery and repair facilities are nicely complementary, and together they encourage one to be bold in abbreviation and rely heavily on COPE to generate a large fraction of the required text. Although the COPE response to a user entry may occasionally be surprising, it is never painful since recovery is always available.
Although most of the time, single-level recovery is adequate, in fact, the COPE UNDO command can be used repeatedly to return the procedure to any desired prior state -- even all the way back to its original state, if necessary.

One might view the development of a program under COPE as a politely adversarial process -- you may repeatedly "damage" the program; COPE patiently restores it. Alternatively, your task is to "prompt" an automaton who makes up in energy what is lacking in creativity. The hypothesis that COPE is meant to test is that this cooperative approach to program development enhances the your productivity.

In any event, the COPE editor is remarkably easy to use. There is a very small set of commands to learn (see Section 4) There are virtually no special rules -- at any time, you can move the cursor to any part of the program and make any arbitrary insertion, deletion or change. Whatever the change, COPE will immediately restore the entire procedure to complete correctness. (Unfortunately, there is no guarantee that the program is useful or that results are what you really wanted.)

There are some occasions when you must inhibit COPE's enthusiasm. For example, if a change must affect two or more lines it may be necessary to restrain COPE from intervening while the cursor is moved from one line to the other. This can be done with the SUSPEND command (see Section 4).

1.1. Expressions

Unfortunately, in the interest of making this COPE prototype accessible as soon as possible, we had to cut a few corners. The portion of the editor that deals with expressions was generated by an automatic process that precluded enthusiastic repair. Consequently, while COPE can cope with outrageous abbreviations in the high-level structure of statements, it is nearly helpless in the face of trivial errors and omissions of punctuation within the expressions in those statements -- faulty expressions are just discarded.

The potential for expansion of abbreviated expressions appears to be just as great as at the higher structural level, so eventually that inconsistently uncooperative component of COPE will be replaced.

1.2. The Position of Declarations

PL/CS requires that all variables be declared, and that all declarations be placed at the head of the procedure. COPE, of course, ensures that this is done, regardless of where you enter declarations.

COPE keeps track of the last place where a declaration was inserted, and the last place where a statement (a non-declaration) was inserted. If you enter a statement in the declaration area, it is automatically repositioned in the statement area, and vice-versa.

You can conveniently jump back and forth between the declaration and statements areas just by entering an empty line. That is, if the
cursor is in the statement area and you hit RETURN (ENTER) without entering or changing the current line, the cursor is immediately repositioned in the declaration area.

2. **Execution of Programs under COPE**

At any point in its development, a COPE program can be executed. The display during execution is distinctively different. Schematically, the arrangement is the following:

```
entry line
message line
input window

----------------------------------------
trace window => check window

----------------------------------------
output window
```

During execution, the four windows show a synchronized, simultaneous, real-time display of the program's action:

The **trace window** displays the text of the program, with a pointer to indicate the statement current being executed. The nesting depth of this display is controlled by the argument of the most recently executed TRACE statement.

The **check window** displays the current state of all local variables, with values changing as corresponding statements are executed.

The **output window** displays the result of the execution of PUT statements.

The **input window** displays the stream of values (moving from right to left) that is supplied for the execution of GET statements. It is also a port through which the user can enter data for execution.

The **rate of execution** is controlled by the argument of the most recently executed DELAY statement. The delay imposed is based on display, so statements not visible on the screen are executed without artificial delay.

Execution is paused by any of the following events:

1. The user presses any key on the keyboard.
2. The output window is filled up.
3. The input stream is exhausted.
4. A PAUSE statement in the program is executed.
5. An ASSERTION is not satisfied.
6. A runtime error occurs (for example, subscript-out-of-range, division by zero, type conversion errors).

Whenever a procedure is paused, for whatever reason, the user can
perform any of the following actions:

a. **RESUME** execution
b. **RESTART** execution (of the current or another procedure)
c. Supply data in the input window.
d. Execute any PL/CS statement "immediately", in the current environment, but without changing the text of the program (see Section 2.1).
e. Recover to some prior statement of execution (see Section 2.2).
f. Alter the text of any procedure -- including the one being executed.

In most cases, the user eventually **RESUMES** execution, although often after performing several of the other possible actions during the pause. If and when the program reaches its normal termination, it is terminally-paused, at which point all the above actions except **RESUME** are allowed.

### 2.1 Immediate Execution

During any pause in execution, any PL/CS construction can be executed immediately by entering it in the "entry window" and giving the EXECUTE ENTRY command. Usually the entry will be a single PL/CS statement, but in fact any arbitrary program fragment can be entered. The entry can employ the same abbreviated form allowed during the development of a program, since identically the same smart editor checks (and expands) these entries.

In most diagnostic execution systems, the most useful action during a pause in execution is the display of an intermediate result. For example, in COPE, to display the current value of the variable X, one need only type the name "x" in the entry window, and give the EXECUTE ENTRY command. The name "x" would be automatically expanded to "put list(x);", the execution of which would display the current value of x in the output window. Ironically, as convenient as this is, it is seldom necessary in COPE, since the current value of x is already automatically on display in the check window.

Consequently, under COPE, more useful statements for immediate execution are such things as:

1. assignment statements, to change the values of variables (particularly in response to "uninitialized variable" errors)
2. **TRACE** and **DELAY** statements, to change the depth of trace display and the rate of execution
3. **LEAVE** to abort a loop
4. **CALL** to invoke a subroutine
2.2. **Reverse Execution**

The COPE recovery facility is available at any time -- in execution, as well as editing. UNDO gives the impression of running the program backwards, although actually all that happens is recovery from the effect of individual RESUME and EXECUTE commands. The facility is entirely automatic, simple to understand and use, and spectacular in its effect on the way one can go about testing a program.

3. **The PL/CS Language**

COPE-PL/CS is a subset of the PL/CS language, which in turn is an almost compatible subset of PL/I. PL/CS was designed to be a "highly disciplined" programming language in which selected constructions from PL/I are required to be used in ways that represented "good programming practice". The only incompatibilities are minor extensions to support a highly interactive, diagnostic system.

The compatible PL/I statements in COPE-PL/CS are the following. The forms shown are the only ones allowed. Except for the restrictions noted, the usage and meaning is the same as for PL/I.

**Procedures**
1. only external procedures and functions
2. all procedures and functions recursive
   (no RECURSIVE attribute)
3. any procedure executable as "main"
   (no OPTIONS(MAIN))

**Declarations**
1. Explicit declaration of all variables is required
2. Declarations are positioned at head of procedure
3. The only attributes are FIXED, FLOAT, BIT, CHAR(n) VAR
4. The required form is
   DCL (variable-list) type-attribute;
5. In an array dcl, the lower bound must be given; for example DCL (X(1:10)) FIXED;

**Statements**

**Assignment:**
numeric variable = arithmetic expression;
string variable = string expression;
bit variable = (condition);
SUBSTR(...) = string expression;
array = array;
array = constant;

**Input/output:**
PUT LIST(expression list);
PUT SKIP LIST(expression list);
GET LIST(variable list);

**Repetition:**
DO WHILE (condition);
DO UNTIL (condition);
DO var = expr TO expr BY expr;
LEAVE;

Conditional:
IF (condition)
  THEN statement;
IF (condition)
  THEN statement;
ELSE statement;
SELECT:
  WHEN (condition) statement;
  WHEN (condition) statement;
  ...
  OTHERWISE statement;
END;
(In any conditional "statement" is either a simple statement or a compound DO; ... END;)

Procedure invocation:
  CALL proc-name(arguments);
  RETURN;

Function invocation:
  function-name(arguments)
  RETURN(expression);
HALT;

The following COPE-PL/CS statements are not compatible with PL/I:

TRACE(integer-expression);
Begin the tracing of the current procedure, with the trace window display limited to the nesting depth specified by the integer argument. TRACE(0) at the beginning of a CALled procedure suppresses tracing altogether, leaving the CALLing procedure on the screen.

DELAY(integer-expression);
The integer argument of DELAY specifies the minimum time (in tenths of a second) between screen changes. (Since the movement of the tracing pointer is a change, this effectively limits the speed of execution.) DELAY(0) permits full-speed execution.

SECTION title;
SECTION is simply a way of identifying different sections of the program. The section title is the only form of comment available in COPE. SECTIONs can be nested. SECTIONs cannot be used in place of DO; ... END; as compound statements in conditionals.

Assertions:
  ASSERT (condition);
  Halt execution if the condition is not true.

  ASSERT (condition) FOR SOME var = expr TO expr BY expr;
  Halt execution if the condition is not true for at
least one value of the specified values of the index variable.

ASSERT (condition) FOR ALL var = expr TO expr BY expr;
Halt execution if the condition is not true for one
or more of the specified bvalues of the index vari-
able.

4. The COPE Commands
The "special" commands of the COPE prototype are the following:

DISPLAY DEVELOPMENT
Display the development (editing) form of the procedure
named in the "entry line"; if none, then the current pro-
cedure.

SUSPEND CHECKING
Turn ON (or OFF) the normal automatic checking that
occurs whenever the cursor is moved off of a line that
has been changed.

EXECUTE PROGRAM
Begin execution, from the beginning, of the procedure
whose name is given in the "entry line"; if none, then
the current procedure.

RESUME EXECUTION
Resume execution of the current procedure from the point
at which it was paused.

EXECUTE ENTRY
Execute the construction given in the "entry line" as if
it had been inserted at the point at which the current
procedure is paused -- but without changing the text of
the procedure.

UNDO
Undo the effect of the most recent command. (UNDO is not
itself an undoable command, so repeated UNDOs move
further back in command history.)

CONDENSE DISPLAY
Condense the multi-line program unit indicated by the
cursor to single line form for display.

EXPAND DISPLAY
Expand the condensed program unit indicated by the cursor
to normal multi-line form for display.

QUIT SESSION
Terminate the COPE session. (Note that when the next
COPE session is initiated, it begins in precisely the state in which the previous session QUIT. For example, a program paused in execution when QUIT is given can be RESUMEd in the next session.)

In addition, there are the usual actions for editing and for cursor motion:

ENTER (or RETURN) to submit a line for checking;
also to open a new line for entry, and to
switch between declaration area and statement area
UP, DOWN, LEFT, RIGHT to move cursor
BACK PAGE, FORWARD PAGE to scroll screen
LEFT END, RIGHT END to skip cursor to end of line
DELETE to delete character
ERASE EOL to erase from cursor to right end of line
TAB to skip cursor rightward to beginning of next word
CHANGE WINDOW to move cursor between different windows

5. References
