DISPLAY CONDENSATION OF PROGRAM TEXT*

James Archer, Jr.
Richard Conway

TR 81-463
August 1981

Department of Computer Science
Cornell University
Ithaca, New York 14853

*This work supported in part by the National Science Foundation under grant MCS 77-08198, and in part by ARPA under grant 903-80-C-0102.

TR 81-463

Display Condensation of Program Text

James Archer, Jr.
Stanford University

Richard Conway
Cornell University

A problem of some general interest concerns the display of program text on a CRT screen. Usually program text is very large relative to the capacity of the screen, so the entire text cannot be displayed at once and some portion must be selected for display at any one time. Even the portion of text that is currently relevant to the user is usually larger than the screen capacity. Although screen capacity will certainly be increased in the future, the inherent disparity between screen and text size will remain. Hence the problem of determining what portion of text to display will continue to exist.

Although this problem is perhaps minor from the system designer's point-of-view, for the user it may have significant bearing on the overall utility of the system. Current screen capacities are very small relative to the information capacity of a desktop, and the interactive user must, in effect, learn to work while peeking through a keyhole at the text. This is not an entirely natural mode of operation, and in many situations the lines on the display screen should be considered one of the limiting resources of interactive systems.

Condensation is clearly related to the formatting issues of prettyprinting, but the two problems are not identical. Prettyprinting can be considered the choice of display format if space were of no concern. Condensation is the compromise that is necessary when space is limited.

The concept is not new. Certain editors -- particularly those dealing with hierarchically structured objects -- have provided various forms of condensation [refs 7,8,9]. The process has variously been called 'zooming' and 'holophrasting', and is generally described only in user's manuals. But the problem has now become more important with increasing interest in integrated, interactive program development environments and it merits separate identification and discussion.

In order to describe the problem precisely, suppose that the total text in question consists of n lines, and that the screen can display only k lines at a time, where $k \ll n$. The problem for each display is to select those k lines from n that will be the most helpful to the user -- or more realistically, at least the k lines that the user wants to see.

This work has been supported in part by the National Science Foundation under grant MCS 77-08198, and in part by ARPA under grant 903-80-C-0102.

The simplest and most obvious strategy is to display k contiguous lines, starting with some line number s, and offer the user various means for determining s. Typical facilities include absolute specification of s, specification relative to current s, and search facilities to find s such that the line contains a specified character string.

It is also not uncommon to allow the screen to be divided into multiple windows, so that several segments of text can be displayed simultaneously. That is, the k available lines can be partitioned into k1, k2, ... , kJ. But this neither solves nor eliminates the basic problem of selection. Even when windows are allowed to overlap there are still J problems essentially similar to the original problem with an unpartitioned screen -- choose ki lines to display in the ith section of the screen. The only difference is that since ki < k, even more severe degrees of condensation are required. Nevertheless, with little loss of generality, our discussion will ignore the divided screen option and concentrate on the basic problem of choosing k lines from n.

Our present purpose is to consider selection strategies in which the k lines are not contiguous. That is, we describe strategies that select k lines in such a way that they effectively span and represent a block of m contiguous lines, where k <= m <= n. A segment of m text lines is thereby 'condensed' into k display lines. The premise is that not all text lines are equally important, and that by omitting some less important lines a larger block of text can be usefully represented, although not fully displayed.

Basically condensation can be effected only by reformatting or elision. Reformatting is essentially a matter of using some alternative compact display format for less crucial lines so as to conserve screen space for the most important lines. For example, a prettyprinted display (with no concern for space) of a conditional statement might be the following:

```
IF (X < Y)
    THEN DO;
        GET LIST(X);
        C = C + 1;
        END;
```

An alternative, vertically compact form might be:

```
IF (X < Y) THEN DO; GET LIST(X); C = C + 1; END;
```

Although this latter form is presumably less effective in showing control structure, it is obviously more economical of display lines. Consequently one might sometimes choose this form simply to increase the size of the program segment that can be instantaneously displayed. However, the question becomes technically interesting only if one is not forced to choose between these extremes, but rather has the option of displaying some portion of a segment in prettyprinted form, and the balance in some condensed form.

Elision is just a variation or extention of reformatting. An ellipsis can be substituted for some portion of the text that is not displayed. For example:

```
IF (X < Y) ...
```

Some systems attempt to convey some information about the elided text -- for example, information such as how many statements are represented [ref 2], or whether the elided text contains incomplete or incorrect program segments [ref 10].

If elision is allowed, arbitrarily large segments of text can be condensed to be represented on a single line. Again, the question is interesting only if this can be applied selectively and dynamically so that a focus is displayed in well-formatted form and peripheral context is indicated in elided form.

1. Automatic Condensation

A basic question is whether effective condensation is possible without requiring any explicit effort on the part of the user. That is, can a display system be made clever enough to manage condensation entirely automatically, deriving clues as to what sections of text are currently relevant from other commands and actions of the user? Mikelsons and Wegman [ref 1] have designed and implemented such an automatic facility in the context of a program development environment for PL/I programs called PDE1L. Mikelsons has separately described the condensation facility [ref 2].

This system always displays an entire procedure -- m is always the total number of lines in the current procedure. This system automatically identifies one or more foci of interest, and displays text in the neighborhood of these foci in full, prettyprinted form. Text some distance from any foci is automatically reformatted and/or elided. The system always manages to elide enough text so that the entire procedure is represented in the display. Since the current implementation uses an IBM 3270 terminal where k is at most 24, the m/k ratio can be quite large.

The Mikelsons-Wegman algorithms are ingenious, although somewhat costly. Substantial amounts of computation are required, but it is likely that on modern personal systems such as PERQ, SUN, APOLLO, etc., adequate power is available.

2. User-Controlled Condensation

An alternative is to give the user explicit control over this process. User-controlled condensation is illustrated by two program development environments implemented at Cornell. The first of these (1978) was the Cornell Program Synthesizer [ref 4, 5, 6]. The second is COPE [ref 3]. Both these systems also support the development of PL/I programs, so the contrast with PDE1L is direct and instructive.

Sections 2.1 and 2.2 describe condensation in COPE and the Synthesizer, respectively. These facilities are similar in concept, and essentially illustrate two variations on the same theme.

2.1 Hierarchical Condensation

A condensation strategy keyed to the hierarchical control constructs of the language is used in COPE. The scheme uses two alternative formats for the display of any segment of program text. 'Expanded' form has one statement per line (using PL/I's definition of 'statement') and a strict indentation convention to indicate the control structure. For example, in expanded form, nested loops would appear as follows:

```
ROW_LOOP: DO I = 1 TO M BY 1;
   GET LIST(X(I));
   COL_LOOP: DO J = 1 TO N BY 1;
       A(I,J) = B(I,J) + X(I);
       END COL_LOOP;
   END ROW_LOOP;
```

COPE condensation is defined in terms of program 'units'. In expanded form a unit consists of any line, and all lines immediately following that are indented with respect to it. For example, there are four units in the nested loops shown above:

1. The assignment statement A(I,J) = B(I,J) + X(I);
2. The three lines of COL_LOOP.
3. The GET statement.
4. The six lines of ROW_LOOP.

A 'compound unit' is any unit which, in expanded form, consists of more than a single line -- COL_LOOP and ROW_LOOP in the example.

The 'condensed' form for a compound unit concatenates all of its statements on a single line, with all statements after the first being elided. Using the example shown above, if COL_LOOP was condensed, the display would appear as follows:

```
ROW_LOOP: DO I = 1 TO M BY 1;
   GET LIST(X(I));
   COL_LOOP: DO J = 1 TO N BY 1; ...
   END ROW_LOOP;
```

If ROW_LOOP was condensed the entire segment would appear as:

```
ROW_LOOP: DO I = 1 TO M BY 1; ...
```

The user can specify for each compound unit whether it is displayed in condensed or expanded form. This is accomplished by positioning the screen cursor and giving a CONDENSE or EXPAND command:

CONDENSE
: Condense the innermost expanded unit containing the cursor-line.

EXPAND
: Expand the outermost condensed unit identified by the cursor-line.

Commands in COPE are assigned to special function keys, so the format of a particular unit (denoted by the position of the cursor) can be altered by a single keystroke.

The format of each individual compound unit can be independently specified as either condensed or expanded. Condensation of an outer unit masks the state of its contained units but does not change the display state of those units.

An important characteristic (copied from the Synthesizer) is that the display state of units is relatively permanent -- that is, it is a property of the stored version of the text and not the displayed version.

By giving the CONDENSE command repeatedly without moving the cursor, the user can progressively condense higher levels of the program. In the example above, if the cursor is on the assignment statement the first CONDENSE condenses COL_LOOP and the cursor is on the COL_LOOP line. Another CONDENSE condenses ROW_LOOP and leaves the cursor on the ROW_LOOP line.

Conversely, repeated EXPANDs progressively display more detail of the program. EXPAND is an exact inverse of CONDENSE. From any initial positioning of the edit cursor, c consecutive CONDENSE commands can be followed by c consecutive EXPAND commands to restore the display to its original state.

For example, a section of the command processor of COPE might look like the following if the reader was primarily interested in studying implementation of the UNDO command:

```
EXECUTE_COMMAND: SELECT(CMD);
    WHEN (CC_INSERT) DO; ...
    WHEN (CC_REPLACE) DO; ...
    WHEN (CC_FETCH) DO; ...
    WHEN (CC_UNDO) DO;
        /** Restore file pages from checkpoint file */ ...
        CALL POP_CHECKPOINT;
        CALL REDRAW;
        CALL DISPLAY_MESSAGE(MSG_24);
        END;
    WHEN (CC_REDO) DO; ...
```

For example, suppose the cursor was positioned on the 'Restore file pages' line. Then the EXPAND command would redraw the screen from this point down, expanding this unit, displacing other lines to make room. On the other hand, CONDENSE would condense the 'WHEN (CC_UNDO)' unit that contains the cursor line, making room for five additional lines to move up onto the bottom of the display. A second CONDENSE would condense the entire EXECUTE_COMMAND unit and the display would be simply:

```
EXECUTE_COMMAND: SELECT(CMD); ...
```

Five further lines would be brought up onto the bottom of the screen. Continued CONDENSEs would eventually condense the complete procedure that contains EXECUTE_COMMAND to a single line.

2.1.1 Condensation During Program Execution

Modern integrated environments such as the Synthesizer and COPE are not just editors; they also have the ability to execute the program under development. Both of these systems can switch easily back and forth between editing and execution, can execute partially-completed programs, and can dynamically trace the progress of execution. The program is traced by displaying (on part of the screen) the program text, with a cursor that moves to show the current locus of execution. Although scrolling of this trace window is automatic, too frequent change in this window greatly diminishes its usefulness. Hence condensation is at least as important during such tracing as during editing.

COPE provides slightly stronger forms of condensation for execution than for editing. This is necessary because the trace window generally has fewer lines available than the edit window, and possible because the trace window need not display the same level of detail as required during editing. For example, END lines convey no useful information about the progress of execution, so they are automatically omitted from the trace display. Similarly, declarations are not really executable statements so they are also omitted.

The condense/expand alternative formats apply to tracing as well as the edit screen, but during execution the user has the further ability to limit the depth of tracing by means of a TRACE statement. For example, TRACE(3) limits the trace display to lines that would appear at the first and second indentation levels beneath the procedure heading. This does not change the condense/expand status of any unit; it is superimposed on that facility. For example, consider the following segment:

```
TRACE(5);
/** Set X = Y * Z */
    ROW: DO I = 1 TO N BY 1;
        COL: DO J = 1 TO N BY 1;
            X(I,J) = 0;
            SUM: DO K = 1 TO N BY 1;
                X(I,J) = X(I,J) + Y(I,K) * Z(K,J);
                END SUM;
            END COL;
        END ROW;
/** Transpose and normalize X */ ...
/** Display X in row major order */ ...
```

This would appear as follows in the trace window:

```
/** Set X = Y * Z */
    ROW: DO I = 1 TO N BY 1;
        COL: DO J = 1 TO N BY 1;
            X(I,J) = 0;
            SUM: DO I = 1 TO N BY 1;
/** Transpose and normalize X */ ...
/** Display X in row major order */ ...
```

TRACE(0) completely suppresses the tracing of a procedure. Note that TRACE is a statement, so it can be included in the text of a procedure which can then control its own execution display. (In COPE, any statement can also be executed

'immediately' from the terminal.)

There is also provision by which the speed of execution can be limited by the granularity of the trace display. That is, each line of the trace is considered an atomic unit for execution, and rate of execution can be limited so the time per unit will not be less than a specified number of milliseconds. Consequently, low-levels suppressed from the trace display can be executed at full speed, while the apparent rate of execution (as shown in the trace) can be readily controlled.

2.1.2 Condensation of Complete Programs

The value of condensation during initial program development is obvious. The user can condense sections that are completed, inactive, or momentarily unimportant for any other reason. This makes the screen more effective for the sections of immediate interest. Condensation would also be useful for completed programs. It might be a helpful practice to leave all but the top two or three levels of all large procedures in condensed form to facilitate 'hierarchical reading'. For example:

```
COPE: PROC OPTIONS(MAIN);
    /** System data structures and controls */ ...
    /** Initialize for new terminal session */
        /** Restore all working files as of end of last session */ ...
        /** Recreate final display of last session */ ...
        /** Prompt user for first command */ ...
    /** Process user comands */
        CMD_PROCESS: DO UNTIL(CMD = CCODE_QUIT);
            CALL GET_COMMAND(CMD);
            EXECUTE_COMMAND: SELECT(CMD); ...
            END CMD_PROCESS;
    /** Terminate session */
        /** Cleanup, close and save working files */ ...
        CALL DISPLAY_MESSAGE(MSG_TERM);
        RETURN;

    /** Definition of Processing Routines */
        GET_COMMAND: PROC(CMD); ...
        MODIFY_PROC_TEXT: PROC(PROC_NAME, POS_PTR, INPUT); ...
        EXECUTE_PROC_TEXT: PROC(PROC_NAME, ENV, EXEC_PTR); ...
        /** Secondary Routines */ ...
    END COPE;
```

Here a procedure of several thousand lines has been condensed so it can be displayed on a single screen, and yet both the control structure and the textual organization of the program are clearly presented. When it becomes necessary to read the program for maintenance work it is relatively easy to scan the text to find the appropriate section, and hierarchically expand that section to find some individual statement or segment.

2.2 Condensation Based on 'Statement Comments'

The essential ideas of this form of condensation were all present in the Cornell Program Synthesizer. That is, the user could choose between expanded and elided display formats; the format specification was relatively permament; and the condensation was used in both editing and execution tracing. However, condensation in the Synthesizer was deliberately limited solely to statement comments. That is, a comment used as a high-level specification of objective is the only type of compound structure whose content can be elided. For example, a segment in expanded form might look like the following:

```
/** Read N and load X(1:N) from data */
    GET LIST(N);
    DO I = 1 TO N BY 1;
        GET LIST(X(I));
        END;
```

In the Synthesizer this can be condensed/elided to the following:

```
/** Read N and load X(1:N) from data */
    ...
```

The Synthesizer has a single 'condense/expand' command that reverses the display state of the statement comment denoted by the screen cursor -- expands it if already condensed, condenses it if in full prettyprinted form.

Teitelbaum justifies the restriction to comments on two grounds. First, when text is elided the comment provides a clearer indication of the purpose of the omitted lines than just the first line of the elided unit. Second, since the Synthesizer was desisned to be an instructional system, coercion of the programmer toward increased use of high-level functional specification is in itself valuable. Realistically, this restriction of condensation to comments means that a certain amount of retroactive commenting is required when it becomes appropriate to condense the display. This is not difficult to do in the Synthesizer, but it probably means that condensation is sometimes not used because it is just too much trouble.

Other aspects of the condensation facility in the Synthesizer reflect the expectation that it will primarily be used for relatively small programs. For example, a separate line is used for the ellipsis in the condensed format. This perhaps indicates more clearly that text has been omitted, and can be tolerated when the m/k condensation ratios are not expected to be larse. Similarly, it was not believed necessary to take advantage of the opportunity for greater condensation during execution tracing, and the possible confusion of having two different displays of the same text could thereby be avoided.

The decision to use a single command in the Synthesizer to invert the format, rather than separate commands to condense and expand, makes it slightly more difficult to progressively expand or condense the display. With this scheme the cursor must be repositioned for each successive condense or expand.

3. Conclusions

These three systems illustrate a useful idea, as well as two significantly different approaches to its implementation. Of the three, only the Synthesizer is really completed and in production service. It has been used for more than two years at Cornell and several other universities by thousands of students. Its use todate has been entirely for introductory instruction, so the programs have tended to be small and the users have been unsophisticated. To some extent the design of the condensation feature was deliberately tutorial, rather than optimally convenient. Nevertheless, the condensation feature is used, so students presumably consider it valuable, and they may in the process have developed slightly better habits regarding program documentation than would otherwise have been the case.

The COPE condensation facilities are simply minor variations of the concepts illustrated by the Synthesizer. To what extent, and for what type of use, these might be improvements remains to be determined by actual user experience.

There is no question but that some form of condensation is a useful facility in any screen-oriented interactive system where n lines can only be viewed k at a time, and k is substantially smaller than n. The real question is how to achieve the condensation and how to control the process.

The choice between automatic and user-controlled condensation is only going to be resolved by extensive user experience, and it is likely that the conclusion will be that there are different circumstances under which each of these alternatives is advantageous. The issues will be simply how much effort is required for user control, and how well automatic condensation can infer the user's needs and intent. There is also, of course, the possibility of compromise. A semi-automatic system might allow the user to vary parameters and set modes so as to provide some guidance and exercise some control over condensation, without having to completely manage the process.

References

1. Mikelsons, M., and M. Wesman, 'PDE1L: The PL1L Program Development Environment', RC 8513 IBM Research Division

2. Mikelsons, M., 'Prettyprinting in an Interactive Programming Environment', RC 8756 IBM Research Division (also in SIGPLAN/SIGOA Symposium on Text Manipulation, June 1981)

3. Archer, J., and R. Conway, 'COPE: A Cooperative Programming Environment', TR81-459, Cornell Computer Science, May 1981

4. Teitelbaum, T., 'The Cornell Program Synthesizer: A Microcomputer Implementation of PL/CS', TR79-370, Cornell Computer Science, June 1979

5. Teitelbaum, T., and T. Reps, 'The Cornell Program Synthesizer: A Syntax-Directed Programming Environment', TR80-421, Cornell Computer Science, May 1980 (also to appear in Communications of ACM, Sept. 81)

6. Teitelbaum, T., T. Reps, and S. Horowitz, 'The Why and Wherefore of the Cornell Program Synthesizer', SIGPLAN/SIGOA Symposium on Text Manipulation, June 1981

7. Donzeau-Gouge, V., G. Huet, G. Kahn, B. Lang, and J. Levy, 'A Structure Oriented Program Editor', Technical Report IRIA-LABORIA, France 1975

8. Engelbart, D., and W. English, 'A Research Center for Augmenting Human Intellect', AFIPS Proc. V.33 (FJCC 1968)

9. Hansen, W., 'Creation of Hierachical Text with a Computer Display', PhD Thesis, Stanford Computer Science, 1971

10. Krafft, D., 'AVID: An Interactive Development and Verification System', PhD Thesis, Cornell Computer Science, 1981