PAST: Viewing the programming process

RICHARD T. REDMOND and JEAN B. GASEN
Virginia Commonwealth University, Richmond, Virginia

A software package called PAST (Programmer Assessment Software Tools) is described. PAST was designed as a research and diagnostic tool to analyze how programmers develop and test program code. PAST uses line changes between successive compiles to estimate a programmer's cognitive processing. Line changes are highlighted and graphically presented between any two versions or compiles. Summary graphs and statistics across all versions also are provided. Applications of PAST for expanding our understanding of programming and applying it to research, teaching, and learning are discussed.

Understanding how an individual develops a computer program is elusive, at best. Although a significant body of research has been undertaken to understand the cognitive skills and abilities that are part of the process (Adelson & Soloway, 1984; Brooks, 1977; Mayer, 1985; McKeithen, Reitman, Rueter, & Hirtle, 1981; Pennington, 1985; Tracz, 1979), there is still much we do not understand about how an individual actually thinks through the program development process. Empirical research often must make a tradeoff between the internal validity of the design and the external validity of the results. Thus, some of the better controlled studies of programmers and the programming process (Pennington, 1987; Sheil, 1981; Soloway & Ehrlich, 1984) may lack generalizability, whereas those that focus on the complexity of the programming process itself (Mayer, 1981; Perkins, Hancock, Hobbs, Martin, & Simmons, 1986; Vessey, 1986; Weidenbeck, 1985) must rely on somewhat less reliable measurement strategies and on research paradigms that are more idiographic in nature. We believe that PAST represents a dynamic middle ground between the more quantitative, a posteriori metrics, and the less reliable, qualitative, self-report measures described in the research literature.

The software program PAST (Programmer Assessment Software Tools) was designed to develop quantitative estimates of the program changes that take place during an individual's program development process. In analyzing program changes, we assumed that (1) a correspondence must exist between the thought processes of programmers and the code that they generate; (2) one line of code in a program must be linked to at least one cognitive process; and (3) an attempted program compile represents one measurable point of program development, in which the programmer's cognitive predictions about the program are being tested (Redmond & Gasen, in press).

An Overview of PAST

PAST attempts to capture a programmer's cognitive processes by systematically saving the changes made to source code between each compile attempted by the programmer. For each compile attempt, PAST compares the line change differences between the current and the preceding compile. In this fashion, the computer equivalent of a videotape of the programmer's activities is created. The changes between compiles are of interest because they often represent work performed by a programmer from one test point to another. The user of PAST can then visually examine the modifications made by a programmer to draw conclusions about the nature of change from a semantic perspective. A color graphic representation of the changes over time also is available to assist the user in focusing on the change data. In addition, both syntactic and semantic changes can be differentiated through analysis of PAST output statistics. These include the total number of line changes overall and per compile, the total number of compiles, the time and date for each compile, the time between compiles, and the total connect times. A graphic representation of these changes across time and changes across versions is also presented by the PAST program.

Major Characteristics of PAST

As in almost all software development situations, certain specifications are made that are hardware dependent. PAST was written and tested on a microcomputer running MS-DOS. The language used was C. The only specifications that are specific to MS-DOS environments are found in the "include" files; this enables portability to new environments without many modifications. The major functions of the software include the following:

- Read and store the final copy of a program.
- Display any version of a file as it goes through development.
- Display the additions and deletions made to a program between versions. This can illustrate specific changes and display changes as a summary over the entire version.

Address correspondence to Richard T. Redmond, Department of Information Systems, Virginia Commonwealth University, 1015 Floyd Avenue, Richmond, VA 23284.

- Generate data across all versions of a file including statistical data.

A major feature of the software is the layout of the input files. In particular, the first input file is the final version of the program that the user wishes to examine. It is simply a text file with variable record lengths. PAST reads in this file and stores it as a dynamic doubly linked list. This allows for easy insertion and deletion of lines as changes are posted. The second input file contains change data represented as lines to be changed, added, and/or deleted. A collection of these lines delimited by a date mark is used as input to create a particular version. For the most part, these change data are created by another program, called "diff." The purpose of diff is to show the differences between files. It is general in nature and normally available under a variety of operating systems. We used the diff program found under UNIX, which normally generates as output the minimum number of line changes necessary to change one file into another. The assumption is that these changes represent the activities a programmer performed between versions. Figure 1 illustrates the output of the diff program added to the second input file after a compile.

By appending the diff program output to the end of the change file, it then contains all the changes made between each compile throughout the programming process. For performance reasons, the PAST program generates an index of entry points into this file, which then provides a method of random access. The index is maintained in main memory throughout the run of the PAST program.

Both overlapping window and tiled window approaches are used by the software and are supported by a windowing package. Some overlapping windows are screen size, giving the impression of paging. Other windows, such as help and overview, appear as a subset of the screen to allow partial viewing of the main screen window, which is normally a version of a program. Tiled windows are used as headers and footers for the display of summary version data.

The PAST program first displays the final version of the program. By pressing the left cursor once, the user sees the program displayed with changes designated. A red background line indicates a line that was deleted to create the next version. A green background line marks a line added to create the next version. Hitting the left cursor a second time posts the changes resulting in the previous version of the file. A later version of the program can be viewed similarly by using the right cursor key. The up and down cursor keys allow the user to move through the displayed file. Figures 2 and 3 give examples of the visible interfaces.

The function keys 1 through 5 provide single keystroke interface with the program:

- F1—toggles a help window.
- F2—toggles a window to display an overview of the current file.
- F3—allows the user to specify a version of the program for display.
- F4—toggles a full screen window of statistical data and bar graphs.
- F5—toggles a full screen window of additional statistical data.

A user can exit the PAST program by pressing the ESCAPE key.

Applications of PAST

Currently, PAST is being used as a research tool to generate hypotheses concerning the kinds of changes that take place when individuals develop and test programs. Assessing changes at the individual level isolates patterns of change that are often masked when data are presented in aggregate form. Ultimately, however, we are interested in being able to generate theories concerning the nature of the program development process across different groups of individuals with similar aptitude profiles. This should be useful in both assessing and predicting programmer performance in a wide number of settings.

```

logic Mon Nov 30 10:14:10 EST 1987
10c10,12
<   char to_array[40],header_array[12];
—
>   char to_array[40][3],header[12][5];
>   char line[80];
>   int to_node,distance,j;
20a23,28
>   while (fgets(line,80,fpdata) != NULL) {
>       fseek(fpdata,81l * j + 4,0);
>       fscanf(fpdata,"%04d%04d",to_node,distance);
>       j++;
>       printf("%080d",line);
>   }

```

Figure 1. An example of the output of the diff program.

The screenshot shows the main display of the PAST program. At the top, it displays 'FILE: obs3.c' and 'DATE: Feb 20 10:46'. On the right, it says 'Version 61 of 94'. A 'COMMANDS' menu is visible on the left, listing various functions and their corresponding keyboard shortcuts (F1-F6, HOME, END, PgDN, PgUP, ESC). The main text area shows a portion of a C program, including a loop that converts lowercase characters to uppercase. A status bar at the bottom provides shortcuts for HELP, Overview, Jump to Version, and Exit.

```

FILE: obs3.c
DATE: Feb 20 10:46
Version 61 of 94

COMMANDS
HELP MENU      F1
Overview       F2
Jump Versions  F3
Stats/Time     F4
Stats/Version  F5
Save Version   F6
Top of File    HOME
End of File    END
Next page      PgDN
Prev page      PgUP
Move up line
Move dn line
Prior Version
Next Version
Quit           ESC

>
>
0
nge first char from lowercase to uppercase */
LINE];
n("../bin/data1", "n")==NULL)
le not found");
ine, MAXLINE, fp)!=NULL)

F1 --> HELP    F2 --> Overview    F3 --> Jump to Version    ESC --> Exit

```

Figure 2. Main display screen of PAST with pop-up overview window.

This screenshot shows the PAST main display with a pop-up help window. The main text area displays a C program snippet with line numbers 18 through 37. The help window on the right, titled 'HELP', shows a list of line ranges from 1-3 to 40-41. The status bar at the bottom includes 'Elapsed Time 2.60 minutes' and 'COMPILED: Feb 20 10:46', along with shortcuts for HELP, Overview, Jump to Version, and Exit.

```

FILE: obs3.c    Elapsed Time 2.60 minutes
COMPILED: Feb 20 10:46    Changes to version 60 of 94

18.  )
19.
20.  while (fgets(line, MAXLINE, fp) != NULL)
21.      for (i=0; line[i] != '\0';)
22.      {
23.          if (line[i] == ' ') {
24.              i++;
25.              line[i] = toupper(line[i]);
26.              printf("%s", line[i]);
27.              printf("%s", line[i]);
28.          }
29.          if (isupper(line[i])) {
30.              i++;
31.          }
32.          if (islower(line[i])) {
33.              i++;
34.          }
35.          printf("%s", line[i]);
36.          fclose(fp);
37.      }

HELP
1 - 3
4 - 6
7 - 9
10 - 12
13 - 15
16 - 18
19 - 21
22 - 24
25 - 27
28 - 30
31 - 33
34 - 36
37 - 39
40 - 41

F1 --> HELP    F2 --> Overview    F3 --> Jump to Version    ESC --> Exit

```

Figure 3. Main display screen of PAST with pop-up help window.

In addition, we hope to use PAST as a teaching tool by isolating the points at which students develop syntactic and/or logical difficulties in completing programming assignments. For example, frequent patterns of syntax errors can be isolated by looking at line changes that involve syntax correction only. Lines can also be identified that represent difficult logical concepts (e.g., the beginning of an iterative loop) so that future teaching efforts can focus on such typical problems.

We also hope to develop the use of PAST as a self-diagnostic tool for programmers/students. The ability to see previous compile attempts would assist programmers in recalling strategies already attempted. This may prove

useful to those who tend to return to previous mistakes out of confusion or lack of recall, or those who inadvertently destroyed previous code and wish to reuse the code in a later compile attempt.

Other applications include applying PAST to programming in other languages, writing, designing, game playing, communications, and so forth. The only requirement to adapt PAST for these other uses is the ability to generate the change data. The PAST program does not require modification. The generation of the change data used by PAST is handled by another program, thus providing independence between the tool and the application of the tool. In our research, the change data was gathered on

a Pyramid minicomputer running UNIX. The change data and final versions were ported to an IBM-compatible for use with the PAST program.

The PAST program produces one interesting measure of changes per time (Figure 4), namely an average across all observed compile points. It is similar to the lines-of-code measure used in many studies, but the bar graph representation of the data illustrates the variability of this measure. Over time, this data seems to be similar to a sine function where the programmer is going from peak to valley often with no sustained level of effort. This is referred to as the "eureka" phenomenon, which is characteristic of many of the programmers whose work has been

reviewed by the PAST program. In contrast, a measure generated by PAST of changes per version (Figure 5) appears to follow a more uniform distribution over time, indicating that these programmers set test points based on number of changes to the file.

Although PAST is still in the early stages of refinement, its potential applicability warrants further research and development. The PAST program contains no knowledge base relative to the language of the program it is processing. The interpretation of the changes is derived through user expertise. The possibility of providing context knowledge is being considered through the use of an expert system interface. In this way, the user could specify

Figure 4. An example of changes/time displayed as a bar graph.

Figure 5. A bar graph illustrating number of changes made per compile.

which type of text is being examined by PAST in order for the program to generate semantic information through its own knowledge base.

Another planned refinement of PAST includes the development of other measures, particularly an efficiency measure based on minimum changes to total changes (Redmond and Gasen, in press). Minimum change is defined as the minimum necessary to create the final program from the original program. The contention here is that all other changes are unnecessary from the perspective of set theory. We also have found that a programmer often may delete a line only to insert the same line later. This may be indicative of traveling down a dead end street and later backing up to try a new direction. We are interested in discovering how many lines in the final program were added only once to the file. These measures can be derived, but they require sophisticated processing of both input files beyond the current capabilities of PAST. The main concern is that these data be reliable; thus reasonably complex solutions are required.

The PAST software is written in the C language and compiled using a Microsoft C 5.00 compiler. It runs under MS-DOS on IBM and compatible machines with at least 256K of memory. It uses a screen processing library of virtual windows similar to the library curses available on many UNIX-based machines. For the most part, the base elements of the program are portable to any environment supporting C. Individuals interested in using PAST are encouraged to contact the authors for more information. Both the source and a demonstration version are available for a nominal fee of \$5. Please specify diskette type (i.e., 3.5 or 5.25). We will provide PAST free of charge if a blank disk and a return mailer are provided.

REFERENCES

- ADELSON, B., & SOLOWAY, E. (1984). Methodology revisited: The cognitive underpinnings of programmers' behavior. In G. Salvendy (Ed.), *Human-computer interaction* (pp. 181-184). Amsterdam: Elsevier.
- BROOKS, R. E. (1977). Towards a theory of the cognitive processes in computer programming. *International Journal of Man-Machine Studies*, **9**, 737-751.
- MAYER, R. E. (1981). The psychology of learning computer programming by novices. *Computing Surveys*, **13**, 121-141.
- MAYER, R. E. (1985). Learning in complex domains: An analysis of computer programming. *Psychology of Learning & Motivation*, **15**, 89-130.
- McKEITHEN, K. B., REITMAN, J. S., RUETER, H. H., & HIRTLE, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, **13**, 307-325.
- PENNINGTON, N. (1985). Cognitive components of expertise in computer programming: A review of the literature. *Psychological Documents*, **15**, No. 2702.
- PENNINGTON, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, **19**, 295-341.
- PERKINS, D. N., HANCOCK, C., HOBBS, R., MARTIN, F., & SIMMONS, R. (1986). Conditions of learning in novice programmers. *Journal of Educational Computing Research*, **1**, 37-56.
- REDMOND, R. T., & GASEN, J. G. (in press). Measuring change in the programming process. *International Journal of Man-Machine Studies*.
- SHEIL, B. A. (1981). The psychological study of programming. *Computing Surveys*, **13**, 101-120.
- SOLOWAY, E., & EHRLICH, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, **SE-10**, 595-609.
- TRACZ, W. J. (1979). Computer programming and the human thought process. *Software—Practice & Experience*, **9**, 127-137.
- VESSEY, I. (1986). Expertise in debugging computer programs: An analysis of the content of verbal protocols. *IEEE Transactions on Systems, Man & Cybernetics*, **16**, 621-637.
- WEIDENBECK, S. (1985). Novice/expert differences in programming skills. *International Journal of Man-Machine Studies*, **23**, 383-390.