AUTOMATIC PROGRAM ANALYSIS AND EVALUATION

Marvin V. Zelkowitz

Department of Computer Science
University of Maryland
College Park, Maryland 20742

and

Institute for Computer Sciences
and Technology
National Bureau of Standards
Washington, D. C. 20234

ABSTRACT

There is currently considerable interest in the computing community in the evaluation of computer programming. However, in order to objectively evaluate such concepts, it is necessary to undertake a thorough evaluation of the programming process itself. Most previous studies of this type have analyzed, by hand usually, a few instances of programs. This has led to some general conjectures; however, the amount of information that must be processed precludes any large scale analysis. In order to avoid this problem, an automatic data collection facility has been implemented as part of a PL/1 compiler at the University of Maryland. This system automatically collects information on each program that has been compiled - at almost no additional cost to the user of the compiler. This paper will describe the system and will evaluate some of the characteristics of some of the 25,000 programs that have been run since July, 1975.

KEYWORDS: Evaluation, Errors, Program complexity, Program measurement, Static and dynamic analysis

1. Introduction

The ability to write reliable computer programs is becoming of prime importance in the industry today. While hardware costs are rapidly dropping, software costs have frequently risen to fill the gap. Approaches towards solving this problem have generally centered in two different areas.

At one end of the scale, various development techniques, such as structured programming, chief programmer team, or top-down design, have been proposed. The problem, however, is that there is little objective data that can be used to evaluate the effectiveness of such techniques. At the other end of the spectrum are the various automated tools that have been developed [8]. These tools allow the programmer to better evaluate and debug a given program. With such a proliferation of tools, each with its own control language, however, the tendency may be to ignore them all.

At the University of Maryland, the development of the PLUM PL/1 compiler has developed into a system where both of the above aspects of program design could coexist in one system. Users use the automated tool support of PLUM for program checkout while data is being collected to analyze the actual programming process. PLUM is basically a diagnostic PL/1 compiler for the Univac 1108, much as PL/C is for the IBM 360/370 [3] by generating diagnostic code to check for all error conditions [9]. However, over the past two years, PLUM has been extended to collect data on program characteristics.

2. The PLUM System

The PLUM system consists of four phases.

Translation Phase

This phase is related to the standard compilation process of most translation systems. Source PL/1 programs are converted into Univac machine language. The compiler generates the usual assortment of error messages typical of such systems.

In addition, program structure information is also generated. Several PL/1 constructs that are not errors but are error-prone also generate warnings. For example:

1. BEGIN; A=0; END; will generate a warning that DO should be used instead of BEGIN for effficiency even though BEGIN is correct. BEGIN implies a new stack area; DO does not.

2. END X; will generate a warning if the END is used to end several blocks. This may be due to a forgotten END earlier in the program. Thus, the program will compile and not execute correctly.

3. The sequence X=40; Y= X/2.0; will generate a warning since the correct PL/1 answer is 16, not the expected 20!

4. Program structure is used to optionally produce an indented listing, based upon the procedure and the DO group nesting depth.

Execution Phase

The execution phase follows compilation. Generated code checks out all error conditions and if any error occurs, control is turned over to an interactive diagnostic system. Using it, the programmer may display and alter any program variables. The programmer may also turn on and off tracing facilities, set breakpoints, and step through execution - all at the source language level.

Post-Execution Phase

In this phase, collected statistics about the program just executed are printed. This includes statement frequency profiles, dynamic statement counts, post-mortem dumps, trace tables, and other forms of collected data.

These three phases allow the programmer to efficiently debug a program and to feed back information on its structure. The fourth phase of PLUM permits evaluation of the programming process. It is this phase that provides data for evaluating the program development cycle.

Language Evaluation Phase

Each of the three previous phases saves information about the program in a mass storage file. Currently 84 words of information (3 sectors in the Univac file system) are saved for each run of a program. The data collection facility is automatic, inexpensive (only 1/2 second additional compilation time), and it was easy to add. A standard file on the Univac system can contain about 2500 such entries, thus a program to backup this information onto magnetic tape need be run only once a month or so (at current usage at Maryland).

Initially the data to be collected was that easily obtainable from the compiler (figure 1). The data consists of four general classifications:

1. System accounting information
2. Program characteristics (e.g. size)
3. Error data
4. System load at time of run

The goal of this facility is to evaluate the programming process. Given only 84 words per program, there was no hope of duplicating the detailed analysis on individual runs of a program as was done by Gannon [5]. However, his analysis was a careful hand analysis of each listing, which necessitated a rather small sample size. The PLUM implementation permits rapid processing of large amounts of data so that statistically significant results can be obtained over a large class of users and programs.

The goal is to keep the data collection facility operational indefinitely. This will allow the "capture" of information from a large community of users, rather than the typically small set of users in previous implementations [1, 5, 7].

3. Preliminary Evaluation

The data collecton facility was turned on during July, 1975, and by May, 1976 over 25,000 programs have been saved. As the next section will describe, certain deficiencies in the system have been discovered, therefore, it is difficult to say that any of the following statements are proven facts. However, the results are interesting, relevant conjectures for further research, and more importantly, give a flavor of the types of analyses that such a data collection facility can provide.

Data Analysis

Beween July 17,1975 and December 28, 1975, a total of 16,027 runs were tapped for data. Figure 2 gives a histogram of program size of 4,583 such programs

PROGRAM STATISTICS	ACCOUNTING INFORMATION
Number of Statements	Account Number
Number of Comments	Source Program Name
Number of Program Tokens	Source Program Name
Program Size (internal form)	Compiler Options
Object Code Size	SYSTEM LOAD
Symbol Table Size	Active Batch Jobs
Run Time Stack Size	Active Interactive Jobs
Compile Time	Time and Date
Execution Time	ERROR STATISTICS
Static Language Analysis	Error Messages
Dynamic Analysis-if user generated	10 Messages by Phase
Blocks Activated	Last 6 statement numbers of messages

Figure 1. Data Collected by PLUM

```
STMTS COUNT    1%  2%  3%  4%  5%  6%  7%  8%  9%
 000-9 203 ********************
       89 *********
       56 ******
      330 *****************************
      366 *********************************
 050-9 478 ********************************************
      202 ******************
      175 ***************
      146 *************
      145 *************
 100-9 123 ***********
      128 ***********
      225 ********************
      139 ************
       78 *******
 150-9 110 **********
      165 ***************
       85 *******
      112 **********
      122 ***********
 200-9 137 ************
       71 ******
       98 *********
       87 ********
       49 ****
 250-9 122 ***********
       74 *******
       79 *******
       81 *******
       52 *****
 300-9  36 ***
       48 ****
       32 ***
       19 **
        5 *
 350-9  22 **
       11 *
       24 **
        0
        9 *
400-889 29 ***
```

Figure 2. Distribution of program size on sample of 4583 programs

collected over a period of 6 weeks. Average program size was approximately 100 statements, with the maximum being 884. Figure 3 gives a breakdown, by statement type, of one subfile of these programs consisting of 1294 programs and 100,776 statements.

In order to evaluate the collected data, it was decided to restrict the evaluation to the 5672 runs that represent all runs by students in two sections of a computer science programming language course. This course was an upper division undergraduate course (open to graduate students) that assumed that the students knew how to program. The author did not teach either section of this course, and so no obvious biasing of the data occurred. In addition, both instructors of the course assured us that students were free to program as they pleased. Therefore, no overt control of programming style was enforced as a class standard. For instance, "No GOTO's" was not enforced as a structured programming restriction. Furthermore, since both sections had the same projects, it was interesting to see whether the characteristics of the classes differed.

An auxiliary goal was to determine the effect of structured programming on the characteristic methods of programming by students. In order to obtain this information, each class was divided into 3 groups (Figure 4). Those students who had attended Maryland's Intermediate Computer Programming Course (structured programming was a major topic) were in one file while those who did not have the course were in a second. (Students seem to ignore prerequisites quite freely!)

A third file was for students whose background was unknown (chiefly graduate students from other schools).

One factor, which we wanted to filter out, was the effect of multiple runs by the same student. Thus, each of the 6 available files was processed for unique programs. This was not easy since students could run the same source program with various names or have two different programs with the same name. The algorithm used was that two programs were considered identical if they were run under the same account number, same program name, and differed in total statements by no more than 20%. While not perfect, it did seem to filter out about 90% of all runs. This action led to 6 more files which were processed.

One effect, which was unexpected, became apparent by this uniqueness process (Figure 3). The percentages of each statement type in the file did not change significantly between counting all runs and counting only unique runs. This was true in every subfile processed. No one user, in this university environment, seems to dominate any one file. However, if different sets of users are measured, percentages do vary (Figure 5). Thus, it may not be statistically significant to actually find all unique programs - the total data may be of equal value.

Figure 5 also demonstrates that care must be taken in evaluating these percentages by statement type. Even with a well defined set of projects, the percentages vary, so that gross generalizations about the structure of PL/1 programs must be avoided.

From this data, the following general observations can be made:

	TOTAL IN FILE	%	UNIQUE RUNS	%	DIFF
Runs	1294		268		
Statements	100776		12498		
Comments	24048	23.9	3398	27.3	-3.4
END	17266	17.1	2135	17.1	0
PUT	13112	13.0	1282	10.3	2.7
DECLARE	10488	10.4	1301	10.5	-0.1
ASG (1 oper.)	7750	7.7	916	7.4	0.3
IF	7519	7.5	971	7.8	-0.3
Simple ASG	6539	6.5	782	6.3	0.2
General ASG	6484	6.4	919	7.4	-1.0
DO (iter.)	5671	5.6	663	5.3	0.3
Simple DO	4751	4.7	582	4.7	0
CALL	4321	4.3	665	5.3	-1.0
PROCEDURE	3770	3.7	517	4.2	-0.5
RETURN	3758	3.7	415	3.3	0.4
GET	2804	2.8	355	2.9	-0.1
DO WHILE	2213	2.2	287	2.3	-0.1
GOTO	1866	1.9	280	2.2	-0.3
BEGIN	760	0.8	85	0.7	0.1
STOP	635	0.6	64	0.5	0.1
Null stmt	433	0.4	80	0.6	-0.2
Leave (added)	214	0.2	22	0.2	0
DO Case (added)	101	0.1	19	0.2	-0.1
OPEN	88	0.1	26	0.2	-0.1
CLOSE	87	0.1	24	0.2	-0.1
FORMAT	43	0	7	0.1	-0.1
READ	31	0	26	0.3	-0.2
EXIT	28	0	5	0	0
Debugging	21	0	2	0	0
Deleted	15	0	8	0	0
WRITE	7	0	8	0.1	-0.1
SIGNAL	1	0	1	0	0

Figure 3. Statement profiles from 1294 programs collected during the period Oct. 17, 1975 to Nov. 15, 1975 from the research oriented Univac 1108. (Comments do not count in computing percentages.)

1. The students in each class who had structured programming used 6% to 12% more comments that those who did not. In addition 17.03% (355 out of 2084) of all listings in the structured programming group generated the warning "PROGRAM INSUFFICIENTLY COMMENTED" while 29.52% (447 out of 1514) of the non-structured programming group generated such a message.

2. Each non-structured programming group used more GOTOs than its corresponding structured programming group; however, both groups in class 2 used more GOTOs than either group from class 1.

3. The structured programming groups used more PUT (output) statements than the non-structured programming groups. Does that mean that they have more debugging I/O in the programs?

4. An interesting result was the use of the CASE statement (which was added to PLUM). The non-structured programming groups used more CASE statements than the structured programming groups (although neither group used very many of them) in spite of the fact that the structured programming group used previously a language called SIMPL [2] which has a CASE and thus were more familiar with the construct.

Clustering

Reflecting on these results, two questions came to mind. Can the statement profiles be used to determine whether a student had the structured programming course and could it be determined whether a student was in class 1 or in class 2? In order to test these hypotheses, a simple clustering algorithm was programmed.

	TOTAL RUNS	UNIQUE RUNS	USERS
Class 1	3,087	348	40
Had Str. Prog.	1,474	166	23
No Str. Prog.	1,222	126	11
Unknown	391	56	6
Class 2	2,577	320	25
Had Str. Prog.	610	83	7
No Str. Prog.	292	42	4
Unknown	1,683	195	14

Figure 4. Data from two classes which was processed

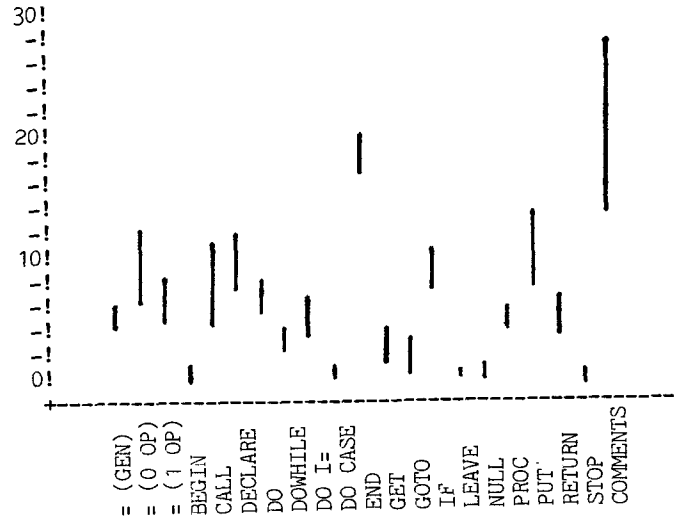

Figure 5. Range in statement distribution across several files of programs

1. Clustering by Structured Programming Background	2. Clustering by Current Instructor
Cl 1. S. P.	Cl 1. S. P.
Cl 1. S. P. (unique)	Cl 1. No S. P.
Cl 2. S. P.	Cl 1. S. P. (unique)
Cl 2. S. P. (unique)	Cl 1. No S. P. (unique)
Cl 1. No S. P.	Cl 2. S. P. (unique)
Cl 1. No S. P. (unique)	Cl 2. Unknown
Cl 2. No S. P.	Cl 2. No S. P.
Cl 2. No S. P. (unique)	
Cl 2. Unknown	Cl 1. Unknown
	Cl 1. Unknown (unique)
Cl 1. Unknown	
Cl 1. Unknown (unique)	Cl 2. S. P.
Cl 2. Unknown (unique)	Cl 2. No S. P. (unique)
	Cl 2. Unknown (unique)

Figure 6. Clustering based upon background and current instructor

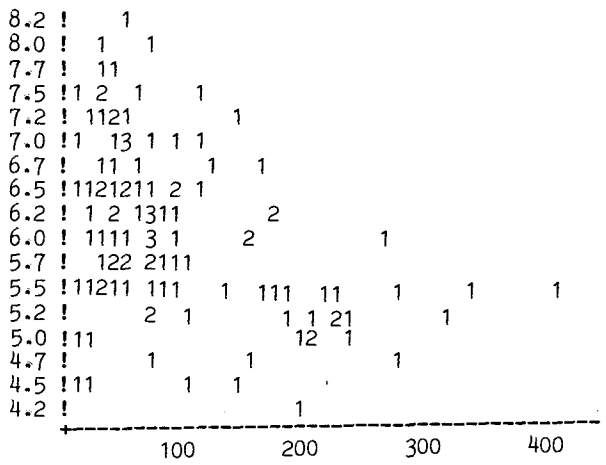

Figure 7. Plot of statement complexity versus program size. (Numbers represent number of programs at each point).

The clustering was performed two ways (figure 6). In the first case "centroids" were based upon average values for the points which represented students which had a structured programming background, and those that didn't. The results clearly showed that all four groups having structured programming were near one centroid, all four (with one additional) were near the other centroid, with the other three points scattered.

Similar, but not identical, results were obtained using the current instructor as a criteria. Class 1 points seem to cluster, as well as class 2 points.

The results of this seem to indicate that it is possible to see differences in programming habits of groups that have structured programming backgrounds and in groups that do not have such a background. In addition, the current instructor also has a strong influence in programming style - not a totally unexpected result.

The data also seems to indicate that the unknown groups and the class 2 programs have a wide variance. One possible explanation for this is class 1 students were mainly undergraduates, while class 2 had many more graduate and part-time students (14 in class 2 as opposed to 6 in class 1). These part-time students had varied backgrounds and more professional experience, and seemed to be less influenced by the instructor than those of class 1. Class 1 students seemed more consistent in style.

Program Complexity

Work by Halstead [6] and others has been investigating the physical structure of programs. While Halstead's measures have not as yet been implemented in PLUM, it is intended to be added in the near future.

However, given the data that is available, some work on program complexity has been done. Statement complexity has been plotted against program size (figure 7). In this case several different measures of complexity have been considered: number of tokens per source statement, object code generated per source statement and parsed program per source statement. In all six files of programs, a least squares fit had a negative slope.

In figure 7, a breakpoint seems to appear at approximately 200 statements. While the curve has an obvious negative slope for small programs, it does seem to hover around 5.5 tokens per statement for larger programs. In order to study this further, statement profiles were produced for these small and larger programs. Figure 8 presents statement profiles for small and for large programs. A similar analysis was performed on a set of 336 general (non-computer science) programs.

The large general programs generally are not C.S.-student written. Note that very few comments are used (3.66%). Also, the DO WHILE is almost unheard of (.19%) while C.S. students use them about 2.2% of the time. GOTO's are an astonishing 13% while C. S. students use them only .61%. While the general data is quite similar to the data exhibited by Elshoff [4], the student data is more in line with good structured programming practices. This student data is also similar to the figures exhibited by Wortman in his study of student XPL programs [1]. We must be doing something right (at least in programs students have to turn in for grades).

A final comment about figure 8 is in complexity of assignment statements. The general assignment line is for assignments with at least 2 operators. The general assignment appears 8.55% of the time in the general programs and only 2.66% for C. S. student programs. This may be due to students using simpler statements, but is probably due to the general file being more number-crunching oriented, while the student programs were more systems programming. This conjecture must still be checked.

Interactive Usage

After investing much effort in a powerful interactive processor, we were curious to see if the facilities provided were ever used. Several files of 1400 runs over a three month period were scanned, and the results displayed as successive 200 runs in order to see if there was any learning effect (Figure 9).

In general students ran about 80% of all programs interactively. In addition, about 77% of these interactive runs do not use any interactive commands. Also, about 15% use only the STOP or EXECUTE commands. Thus only 8% of all interactive runs use any of the features such as displaying or altering variables, displaying stack contents, setting break points or turning on tracing facilities - a most discouraging statistic. In general it seems as if students are not taught proper ways to debug programs and do not use interactive systems effectively.

The one facility which did seem to be used is the autoindenter feature on source listings. This facility

produces an indented listing based upon the block structure of the program. Initially it was used about 10% of the time, but by the end of the semester usage seemed to be leveling off at one-third of all listings. This feature, which took only two hours to add one evening, has been a most worthwhile investment.

Comparing this data with similar data for all 5200 runs on one of Maryland's two computers over a five month period shows some similarities and differences. Interactive usage was about 95% of all runs. Also, programs which used the interactive features hovered around 6% of the interactive runs. Indented listings varied wildly (probably because the users who used them executed programs in spurts over the five month period), but generally rose from 10 to 15 per 200 runs to 25 to 30 by the end of the period. Due to the varied backgrounds and experiences of all users, it is not surprising that the data is not as consistent as the student data.

Error Analysis

The two classes generated a total of 26,671 compilation and 32,194 execution messages while a general file of 5287 runs generated 27,315 compilation and 10,646 execution messages. In considering the number of messages. it is interesting to note that both files have approximately the same number of compilation messages but radically different execution messages. However, if programs with more than 10 messages in any one phase are counted as 10 (to ingore the effects of a few programs with many errors) then the student data reduces to 18,005 compilation messages (or 3.17 per compilation) and 9532 execution messages (1.85 per execution) while the general file reduces to 15,935 compilation messages (3.01 per compilation) and 5245 execution messages (1.06 per execution). As figure 10 will show, the number of 1.06 is probably quite low since that file has a large number of runs terminated by the operating system and hence most execution data was lost. If only runs where the termination message is present (3186 out of 5287 runs) then the average rises to 1.64. These figures of 3 to 3.2 and 1.6 to 1.9 seem reasonably consistent and deserve further study.

In considering the data of figure 9, we were interested in determining how programs terminated. This data is presented in figure 10 as a range of percentages of the three files: the structured programming group, non-structured programming group and the unknown group.

Three aspects of this data must be clarified. Data is only collected for the first 10 messages per phase thus "Data Missing" represents that fraction of the runs where the final termination message is unknown. While we don't know the termination message, it can probably be argued that after 10 messages, what difference does it make.

The entry "Unaccounted" refers to runs where the final message has not been collected. This could happen in two ways. The user could have replied "STOP" to some non-fatal error (about 3% to 6% of all runs), or else the user could have used the operating system command language to instantly stop execution. While this seemed to be used freely in the general file, most students seemed to use the facilities of PLUM itself to terminate execution.

Finally, the list of fatal errors is quite short since PLUM is a diagnostic system that executes as long as possible. Terminal compilation errors are quite rare, and most execution errors will not terminate execution.

	C. S. Class		General Progs	
	small	large	small	large
Programs	545	14	292	44
Statements	44320	5516	18092	37279
Comments	18.87%	20.25%	18.83%	3.66%
Gen Asgn	4.30	2.66	10.50	8.55
Asgn (0 op)	7.06	10.18	9.27	5.20
Asgn (1 op)	6.87	8.32	5.52	3.14
Begin	.64	.05	.80	.01
Call	8.35	10.49	9.18	22.51
Close	0	0	.16	.21
Declare	8.89	7.74	10.46	4.96
Do	6.81	10.04	3.54	4.51
Do while	2.43	2.21	1.19	.19
Do iter.	3.80	.94	3.85	2.25
Do case	.43	.65	.27	.01
End	18.88	17.83	13.94	9.53
Format	0	0	.01	.03
Get	2.14	.56	1.75	1.57
Goto	.73	.61	3.57	13.21
If	7.95	9.49	7.89	14.03
Null	.39	.03	.38	.01
On	.01	0	.64	.01
Open	.01	0	.43	.28
Procedure	4.74	3.93	4.41	2.53
Put	10.01	6.39	9.10	3.05
Return	4.78	7.12	1.92	3.95
Stop	.30	.54	.33	.17

Figure 8. Frequency profiles for small (less than 300 statements) and large programs.

INTERACTIVE (%)	BATCH (%)	NONE (%)	S&E (%)	OTHER (%)	LIST	IND	%
75	25	84.6	9.3	6.0	115	11	9.5
85	15	74.1	18.8	7.1	107	16	14.9
73.5	26.5	71.4	22.4	6.1	129	31	24.0
82	18	75	21.3	3.6	103	35	33.9
90	10	71.6	9.4	18.9	131	51	38.9
77.5	22.5	83.8	10.3	5.8	135	44	32.5
74.5	25.5	71.1	15.4	13.4	117	30	25.6

NONE - NO INTERACTIVE COMMANDS
S&E - STOP AND EXECUTE ONLY
OTHER- ALL OTHER COMMANDS

LIST - NUMBER OF LISTINGS IN 200 RUNS
IND - NUMBER OF INDENTED LISTINGS

Figure 9. Interactive/batch usage per 200 runs on a file of 1400 programs over a 3 month period.

	C. S. CLASS	GENERAL
PROGRAMS WITH ERRORS (%)		
Compile time	39.87-57.39	47.15
Execute time only	16.71-20.97	16.16
No errors	25.89-39.15	36.18
CAUSE OF TERMINATION(%)		
Terminal compilation error	0-0.24	0.16
Normal exit	52.17-66.46	38.62
Read past endfile	6.76-12.61	10.45
Maximum output	0.67-2.31	0.34
Stack overflow	3.30-5.37	2.36
Max execution time	0.39-1.35	0.62
User interrupt	1.06-3.30	1.02
File error	0.43-1.05	0.68
Other	7.27-11.70	5.60
Data Missing	2.17-2.79	0.88
Unaccounted	7.53-11.36	32.57

Figure 10. Causes of program termination. CS class gives max and min value of files of 1514, 2074 and 2084 runs. General is file of 5287 runs.

From figure 10 it can be seen that almost half of all runs contain at least compile time errors. It if hasn't been said enough before, this strongly argues for good diagnostics in all compilers. In addition 15% to 20% of all runs contain execution errors only. Only one third of all runs do not contain any messages. (Since PLUM is a diagnostic system, these numbers are probably high when compared to a production compiler.)

About one half of all runs terminate normally with another 10% reading past end of file. Running out of stack space terminates another 5% while the remaining messages take only a few percent. Note that half of all runs terminate in a non-normal error (another thing the compiler writer should be keenly aware of.)

Further details on PL/1 error messages will appear in a paper now in preparation.

4. Disclaimers

In the previous section several conjectures about programming style were made; however, the data itself is open to some interpretation. An evaluation of the data collection facility itself leads to the following considerations:

1. The universe of users of PLUM is relatively small. Most runs were classroom jobs of Computer Science students. Since Maryland does not have Univac's production PL/1 compiler, many users are reluctant to use a system like PLUM; although usage is growing and may be more representative over a longer time period.

2. The problem of identifying successive runs of a given program must be addressed if accurate error analysis of the data is to be performed. While students frequently use names like PROJECT1, PROGRAMA, etc., they also use names like MAIN and JUNK (to name just a few). In order to prevent this, two possibilities are being considered. One would make PLUM a closed system, much like BASIC or APL systems. A user would enter PLUM and then edit, compile and execute programs. It would be more difficult (but not impossible) to change program names. A second consideration would be for the data collection facility to know about certain classes, and enforce naming standards on such programs.

5. Conclusions

Over the past year, data has been collected on over 25,000 executions of PL/1 programs. This facility has been inexpensive to use, and lends itself to processing large amounts of data about programs quite quickly. The initial implementation has been quite useful. Certain deficiencies in the data collection facility have been noted, and will be fixed shortly.

The next step in the process is the further evaluation of error data. Individual programs will be traced from execution to execution in order to evaluate how programs are corrected.

The important aspects of this research is that the data collection process is inexpensive, transparent to the user and a continuing process. Maintaining the data files requires minimal work on our part, and the building of a large data base of usage statistics should lead to some significant results in the future.

This research is an ongoing operation. The collection of this data, along with similar data for other languages is an important step in evaluating the programming process and in deciding upon the direction to be taken in future language development.

REFERENCES

[1] W. G. Alexander and D. B. Wortman, Static and Dynamic Characteristics of XPL programs, Computer, November, 1975, pp. 41-46.

[2] V. R. Basili and A. J. Turner, A Transportable Extendable Compiler, Software Practice and Experience 5, 1975, pp. 269-278.

[3] R. W. Conway and T. R. Wilcox, The Design and Implementation of a Diagnostic Compiler for PL/1, Communications of the ACM 16, no. 3, March, 1973, pp. 169-179.

[4] J. L. Elshoff, An Analysis of Some Commerical PL/1 Programs, IEEE Transactions on Software Engineering SE-2, No. 2, June, 1976, 113-120.

[5] J. D. Gannon and J. J. Horning, Language Design for Programming Reliability, IEEE Transactions on Software Engineering SE-1, No. 2, June, 1975, pp 179-191.

[6] M. Halstead, Algorithm Dynamics, ACM National Conference, 1974.

[7] P. G. Moulton and M. E. Muller, DITRAN - A Compiler Emphasizing Diagnostics, Communications of the ACM 10, no. 1, January, 1967, pp 45-52.

[8] C. V. Ramamoorthy and S. B. F. Ho, Testing Large Software with Automated Software Evaluation Systems, IEEE Transactions on Software Engineering SE-1, no. 1, March, 1975, pp. 46-58.

[9] M. V. Zelkowitz, Third Generation Compiler Design, ACM National Conference, Minneapolis, Minn., October, 1975, pp. 253-258.