OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

DESIGN AND CODE TRACEABILITY USING A PDL METRICS TOOL

Dr. Paul W. Oman
Department of Computer Science
University of Idaho
Moscow, Idaho 83843

Dr. Curtis R. Cook
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3902

90-60-11

DESIGN AND CODE TRACEABILITY USING A PDL METRICS TOOL

Dr. Paul W. Oman
Department of Computer Science
University of Idaho
Moscow, Idaho 83843
(208) 885-6589
oman@ted.cs.uidaho.edu

Dr. Curtis R. Cook
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331-3902
(503) 737-3273
cook@mist.cs.orst.edu

Abstract

This paper describes an analysis tool that extracts complexity metrics from Program Design Language (PDL) as contained in design specification documents. The tool analyzes pseudocode and computes token-count metrics of PDL complexity on a module by module basis. We used the tool to measure consistency within and across modules and for assessing traceability between PDL and the corresponding source code. In applications of the tool we were able to identify PDL descriptions that were too detailed, those lacking sufficient detail, identify inconsistent PDL descriptions, and measure traceability between the PDL description and the corresponding source code.

INTRODUCTION

Software design decisions have considerable impact on later phases of the software life cycle. For instance, studies have shown that most errors discovered and corrected during the testing phase were introduced during the design phase [Basi84]. Maintenance, the most costly phase, is significantly impacted by design decisions. A typical program undergoes numerous changes during maintenance; for this reason, Parnas [Parn79] believes that ease of change should be a major design criterion.

The importance of early defect removal and the utilization of software complexity metrics have long been established in numerous software studies [Basi84, Boeh76, Boeh84, Jone79]. It is many times more costly to remove an error late in the software life cycle than early in the software life cycle. Software complexity metrics have been shown to be useful in identifying error-prone or difficult to test modules [Grem84, Shen85, Taka85]. However, these metrics are derived from the actual program code – late in the software life cycle.

Some recent studies have described the use of design-time software metrics to aid later phases of software development and maintenance. For instance, Henry and Selig [Henr90] found correlations between the structural complexity of Ada-like design descriptions and the corresponding source code complexity. In

another study, Rombach [Romb90] found correlations between structural complexity measures of high level designs and the cost of changes during maintenance. Rombach points out that the difficulty in assessing design complexity is partially due to the creativity of the design process. Because design documentation ranges from informal English descriptions to formal graphic notations, the computation of design metrics must be tailored to a specific design method and notation. In this paper we demonstrate the ability to extract complexity metrics from Program Design Language (PDL), as contained in design specification documents. We show the utility of these metrics in providing a mechanism for measuring consistency within and across modules and for assessing traceability between PDL and source code.

We implemented a prototype PDL metric extractor that analyzes pseudocode from design specification documents and computes simple measures of PDL complexity. Specifically, Halstead's software science metrics [Hals77], the number of lines of pseudocode, and the number of tokens are calculated on a module by module basis. When used in conjunction with a software tool for identifying metric "outliers," these measures provide a basis for (i) rank ordering the complexity of modules, (ii) identifying inadequate PDL descriptions of modules, (iii) identifying inconsistencies within the design document, and (iv) targeting potential problems prior to coding. When used in conjunction with source code complexity metrics obtained after coding, the comparison of PDL metrics to code metrics provides a verification mechanism to check for consistency and traceability.

In the next section we review the literature on applying complexity metrics to source code, and then discuss how metrics can be applied to software designs. The third section describes our prototype PDL metrics extractor and the fourth section describes our experiments applying the tool to software designs. The final section reviews our findings and discusses how these tools need to be verified in empirical tests with ongoing software development efforts.

APPLYING COMPLEXITY METRICS TO SOFTWARE DESIGN

Software complexity metrics are objective measures of how complex a piece of code is and how difficult it may be for a programmer to test, maintain, or understand [Cook84]. Software complexity metrics do not measure the complexity itself, but instead measure the degree to which those characteristics thought to contribute to complexity exist within the source code. For example, a program may be considered complex if it has complicated control flow and many different execution paths. In this instance, a suitable software complexity metric would be the number of decision statements.

Numerous studies have demonstrated the utility of software complexity metrics [Bern84, Elsh85, Gord79, Grem84, Kafu85, Shen85, Taka85]. Experiments have shown a strong relation between programs with high complexity metric values and the difficulty of performing programming tasks such as program comprehension, debugging, and maintenance. Software complexity metrics have been used to identify error-prone program modules and have provided reasonable predictions of the number of errors in modules. For instance, in a study of three IBM systems, Shen, et. al., found that modules most likely to contain errors could be identified through Halstead's n2 metric (distinct operands) and the number of decision statements [Shen85].

Software complexity metrics can aid in the allocation of resources for testing and maintenance [Harr86]. Modules with high metric values are likely to be more difficult to test and maintain. They contain most of the errors and, hence, should be allocated more resources. Program errors are the major cost associated with software development. Over half of the program development effort is spent on program testing and debugging. The cost of finding and correcting errors is related to the software life cycle phase in which the error is found: The earlier an error is found, the less expensive it will be to correct. But most surprising is the dramatic increase in costs. It is over ten times as expensive to find and fix an error discovered during the test phase as during the design phase. Although errors can be introduced in any phase of the life cycle, most (well over 50 percent) are introduced during the design stage [Basi84, Zelk79].

Since software complexity metrics have been shown to identify modules likely to be error-prone, it seems natural to extend complexity metrics to program design documents. Two reasons for choosing the program design phase are (1) most errors are introduced in this development phase and (2) the cost of finding and correcting an error increases the longer the error is undetected. Hence, the objective behind design complexity metrics is to identify sections of a program's design that are likely to be hard to test and to contain errors.

The major problem in extending software complexity metrics to the analysis of software design is that program designs range from diagrams to natural language descriptions. Examples illustrating the range of program design languages are flow diagrams, pseudocode, and Nassi-Shneiderman charts [Pfle87]. A flow diagram is like a flowchart with enforced structure. The basic building blocks (process, decision, collection point, and expansion point) have distinct geometric shapes, but design structures can only be built using sequence, selection (if-then-else, case), and repetition (do-while, repeat-until) structures.

Pseudocode is similar to a procedural programming language. It has similar control structures (if-then-else, case, repeat-until, while-do), but English-like phrases or sentences exist in place of the formally defined statements and conditions. There are many varieties of pseudocode depending on the allowable control structure constructs and allowed forms of the English-like phrases.

Nassi-Shneiderman charts are like pseudocode with program design statements enclosed in rectangles. The interior of the rectangle is partitioned into sections corresponding to processes of the design. Control constructs are indicated by partitioning the rectangles to clearly indicate the conditions (if-then-else, do-while, repeat-until, case) and the processes executed for each condition.

Most program design languages are highly visual and allow considerable freedom in the types of natural language statements which can be placed inside the various geometrical shapes and boxes. This leads to the second major problem in extending software complexity metrics to program designs: variation in the level of detail in the design documentation. Some designs can be very detailed with an almost one-to-one relation between the design and program statements. In others, this relation is one-to-many. Not only is there a difference in level of detail between designs, but individual parts within the same design may differ considerably in level of detail.

Several extensions of software complexity metrics for software designs have been proposed. Szulewski, et. al., [Szul81] defined Halstead's metrics for the elements of a graphical representation of design. Troy and Zweben [Troy81] defined 21 metrics for structure charts, and Brandl [Bran90] recently described applying structure chart metrics to Hughes Aircraft Company designs. Hall and Preiser [Hall84] applied McCabe's cyclomatic complexity to hierarchically structured design graphs. Later this was extended by McCabe and Butler [McCa89], who defined three levels of complexity: Module complexity, the cyclomatic complexity of a reduced graph; Design complexity, the sum of the module complexities; and Integration complexity, the design complexity minus the number of modules plus one. All of these attempts at applying complexity metrics to software design show promise, but further empirical tests are necessary to determine how they will affect the development process.

In this paper we have restricted ourselves to extending complexity metrics to pseudocode design descriptions. The next section describes the prototype PDL metrics tool that analyzes pseudocode and computes simple token-count metrics such as Halstead's Software Science measures and the number of lines of pseudocode. The tool allows the user to interactively define operands and operators or to use the program's defined values. Our work is similar to that of Reynolds [Reyn87], who developed a pseudocode complexity metrics analyzer as part of his automatic programming Partial Metrics System. His system uses a database and reasoning system to "hypothesize" which words in the pseudocode are operators and which are operands. Ours is simpler because we use external state tables and binary search trees to classify operators and operands.

A PROTOTYPE PDL METRICS EXTRACTION TOOL

The initial purpose in building our PDL metrics extractor was simply to establish the feasibility of doing so. It is an experimental tool to assist us in determining what metrics can be calculated from pseudocode descriptions, and how those metrics can be used to influence software development. The primary design consideration in building the tool was that it had to be able to process any and all pseudocode descriptions. This demand for flexibility excluded the use of syntax directed parsing and restricted the tool to simple token scanning and analysis.

The metrics extractor is a window oriented TurboPascal program written for PC compatible microcomputers. It uses an external state table to drive the scanning of the input file containing pseudocode. Tokens extracted from the pseudocode are identified as being either Operators, Operands, Skipped (for extraneous prepositions), or control tokens for directing the calculation of the metrics. Two modes of identification are possible: Automatic, which automatically classifies all tokens, and Query, which asks the user for direction whenever the current token has not been previously classified. External default lists of operators, operands, and tokens to be skipped are used to initialize binary search trees which guide the token classification. These binary search trees are updated with every token and token counts are then used to compute the metrics.

Figure 1(a) shows the metric extractor in Query mode. The tool is querying the user for instructions on how to process the "4.15" token. The input pseudocode scrolls through the upper window while classified tokens are added to the three scrolling windows for operators, operands, and skipped tokens. Queries to the user appear in the middle and bottom lines of the display. Figure 1(b) shows the metrics extractor upon reaching the end of a module. The binary search trees are scanned and dumped into the three scrolling windows at the bottom of the display. The frequency of occurrence is listed next to each token.

The external state table and external token lists permit the metric extractor to be tailored to specific applications (like processing the reserved words in a programming language). The default lists are based on the counting strategies first enumerated by Salt [Salt82] and later adopted by Conte, Dunsmore, and Shen [Cont86]. Thus, when using its default tables, the metric extractor is capable of processing any text file, but is most accurate with Pascal-like pseudocode. Figure 2(a) shows the metric extractor processing pseudocode for a program that calculates the dates of Easter (taken from [Rohl83]). Figure 2(b) shows the metric extractor processing the Pascal code resulting from that design.

Output from the metrics extractor includes: (1) a screen display of the total number of lines of pseudocode processed (LOC), the total number of tokens processed (TOK), and totals for Halstead's four baseline metrics (n1, N1, n2 and N2), and (2) an output file containing these metrics on a line by line basis for each module processed by the tool. This format is consistent with the output from our source code metrics extractor (designed and implemented for earlier studies). Table 1 shows an example PDL metrics output file; Table 2 shows the corresponding source code metrics as calculated by our source code metrics extractor. Both files are compatible with our metric outlier identification program (described in the next section) and commonly used statistical packages.

TESTING THE PDL METRICS TOOL

Initial testing of the PDL metrics extractor was conducted with small pseudocode samples taken from published sources [Nann85, Rohl83, Shel86]. These tests showed the metrics extractor was reliable and useful for calculating token-based metrics such as Halstead's Software Science measures. Harder to calculate structure and hybrid metrics [Kafu85] could not be approximated reliably. Also, input was limited to the "polished" pseudocode as typically found in academic texts, so except for demonstrating the feasibility of extracting metrics from clean pseudocode, these tests did little to establish the relationship between real-world designs and the code derived from them.

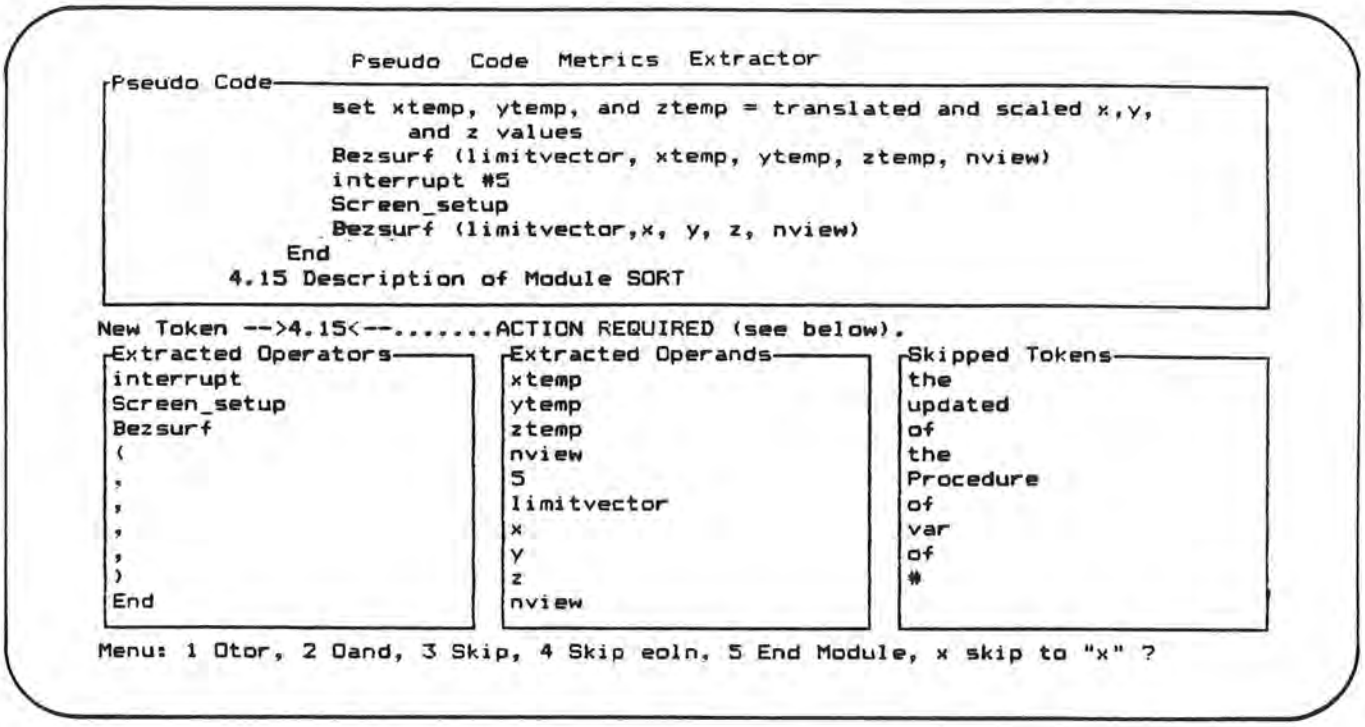

Figure 1(a). Query Mode.

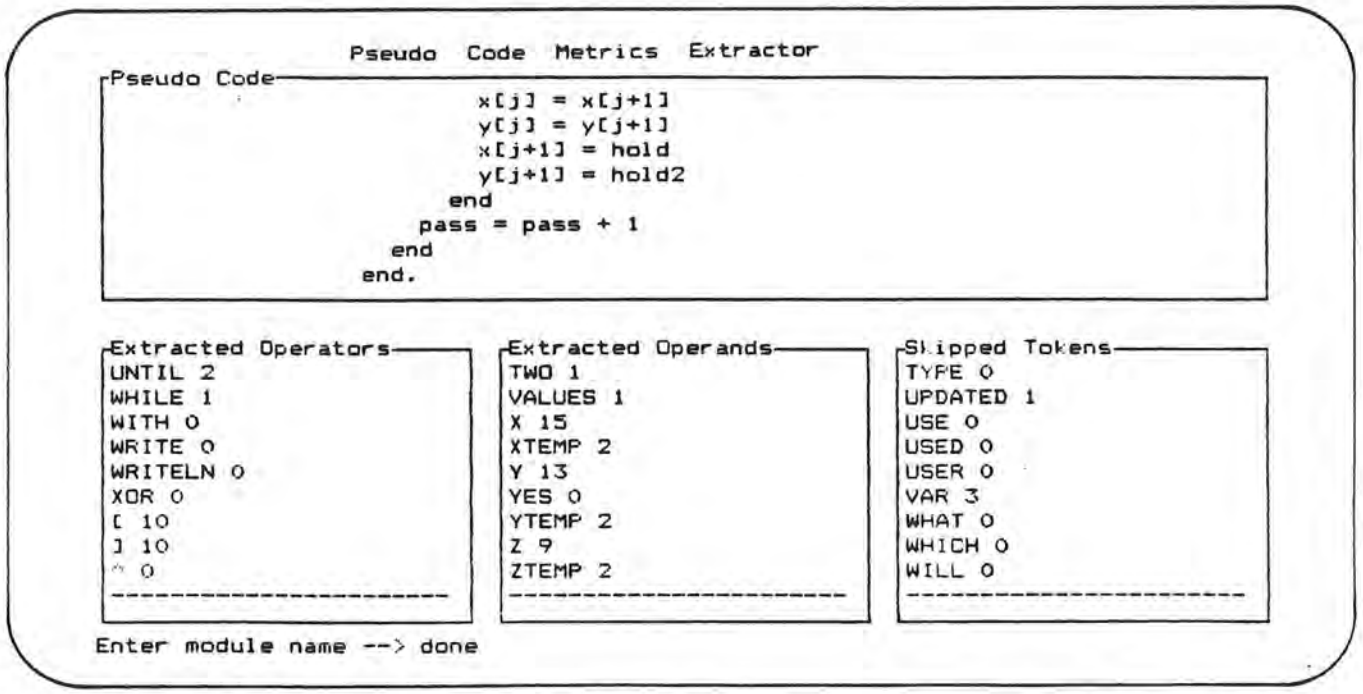

Figure 1(b). At the End of a Module.

Figure 1. Screen Displays of the PDL Metrics Extractor.

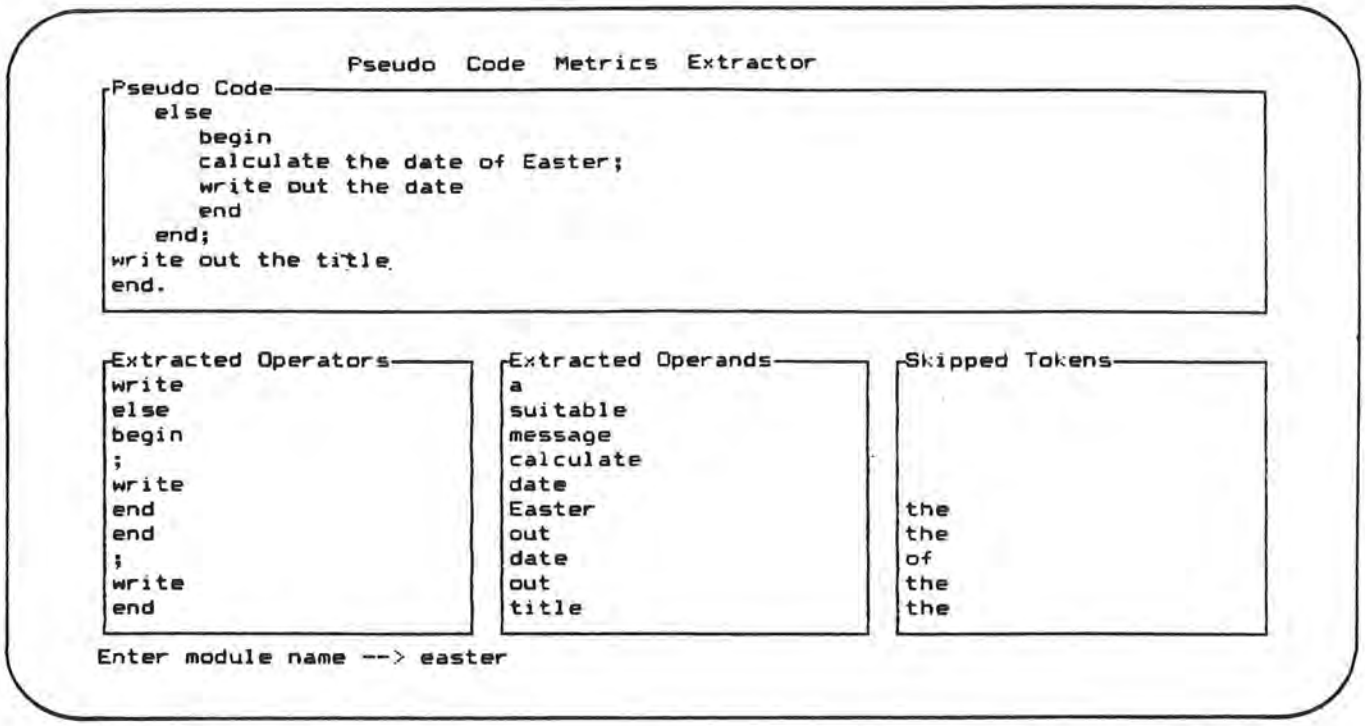

Figure 2(a). Processing Pseudocode.

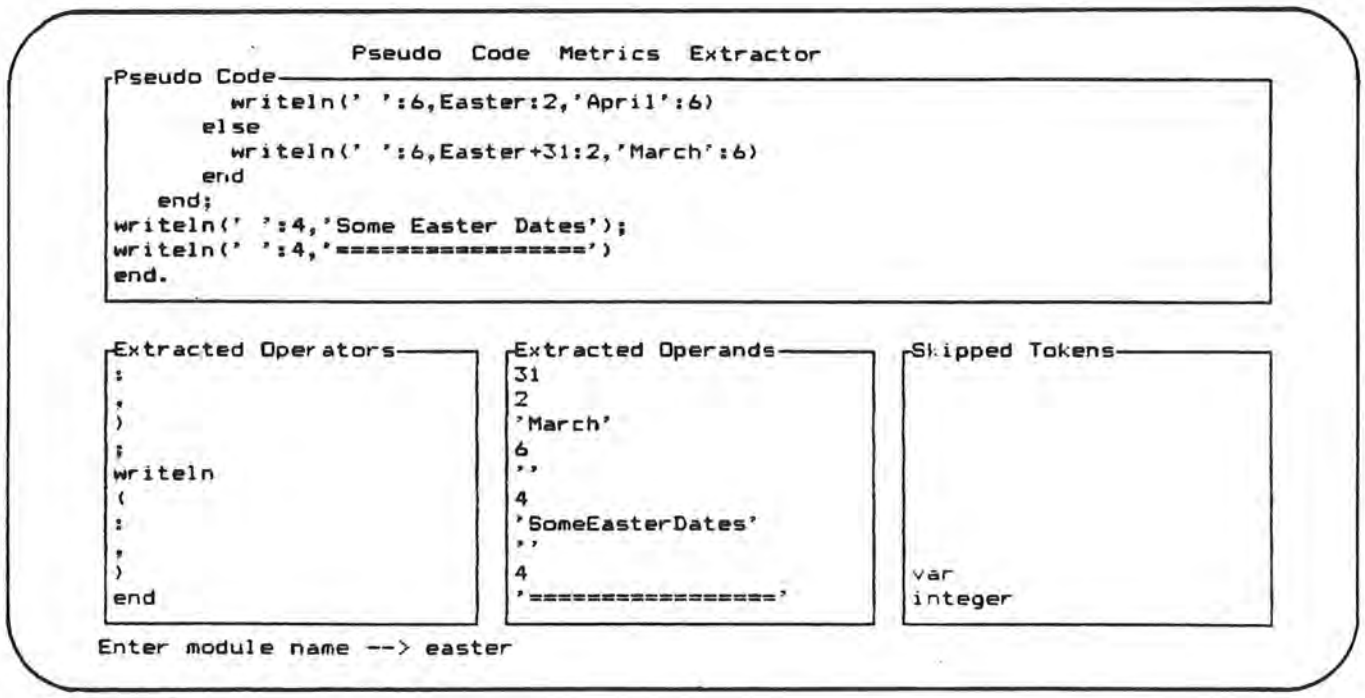

Figure 2(b). Processing Source Code.

Figure 2. Processing Pseudocode and Code.

Table 1. PDL Metrics Extractor Output File.

	name	tok	loc	n1	N1	n2	N2
1	Init	21	10	3	7	11	14
2	DrawMenu	89	22	3	15	22	74
3	Control	43	16	7	16	24	26
4	MoveCursor	124	25	7	24	33	99
5	LoadData	190	39	17	96	33	92
6	SaveData	79	17	13	40	21	38
7	PrintData	16	10	6	6	7	9
8	Simulate	37	13	9	17	11	19
9	DrawEdge	131	45	16	63	31	67
10	ClearScree	35	15	9	14	12	20
11	DrawNode	67	22	15	29	21	38
12	DeleteNode	81	22	18	38	23	43
13	DeleteEdge	37	15	10	17	14	20
14	QuitRGS	32	14	9	16	8	16
15	PrintTable	95	26	8	60	12	34
16	PrintSimul	47	9	12	17	18	24
17	GetNextCha	6	4	3	3	3	3
18	CalcPath	24	8	6	11	11	12
19	SimPath	52	10	9	18	12	32
20	BuildNodeT	15	4	4	7	4	8
21	UpdateNode	40	7	10	21	11	19
22	LabelEdge	14	10	4	5	7	9
23	BuildEdgeT	118	16	15	50	29	64
24	UpdateEdge	65	14	13	30	21	33
25	BezierCurv	153	29	15	76	28	77
26	UpdateStat	25	5	9	10	10	13
27	CalcFact	20	6	7	10	5	10
28	DrawCursor	5	4	1	1	3	3
29	CheckCurso	20	5	6	8	7	12
30	StatusLine	6	4	1	1	5	5
31	BuildState	26	4	8	9	13	15

Table 2. Code Metrics Extractor Output File.

	name	dsl	loc	tok	com	Vg	Nst	n1	N1	n2	N2
1	CHECK_CU	6	5	47	18	1	0	7	16	8	14
2	DRAW_CUR	45	42	390	19	16	5	14	176	22	151
3	MOVE_CUR	276	273	1553	46	37	28	21	781	74	531
4	CALC_FAC	10	6	43	16	2	1	6	12	4	9
5	BEZIER_C	48	41	396	23	5	2	19	156	37	116
6	LABEL_ED	131	122	981	46	16	10	26	433	64	349
7	BUILD_ED	22	19	112	24	2	1	5	34	30	30
8	BUILD_ST	23	18	231	19	8	5	19	92	21	60
9	ATAN2	24	19	133	15	16	7	16	55	6	40
10	DRAWARRO	49	38	360	37	12	3	23	146	35	108
11	DRAW_EDG	309	291	2165	86	58	16	37	1043	82	720
12	UPDATE_E	20	19	76	16	2	1	5	34	18	30
13	UPDATE_S	10	7	65	16	3	2	10	21	11	19
14	DELETE_E	112	104	806	39	20	8	30	361	67	282
15	BUILD_NO	5	4	29	14	1	0	5	10	4	8
16	DRAW_NOD	82	75	539	35	15	8	33	261	40	161
17	UPDATE_N	5	4	29	15	1	0	5	10	5	8
18	DELETE_N	77	70	534	35	18	8	34	258	44	160
19	PRINT_ST	152	140	1014	18	52	13	18	474	36	294
20	PRINT_SI	26	24	211	15	8	4	23	95	23	71
21	PRINT_DA	23	21	143	15	6	2	22	70	17	39
22	STATUSLI	9	8	71	18	1	0	4	24	17	22
23	INITIALI	47	42	231	23	6	2	12	101	53	92
24	LOAD_DAT	150	137	1014	39	28	10	38	444	76	360
25	SAVE_DAT	58	51	388	30	13	6	29	172	50	134
26	GET_NEXT	16	15	97	16	2	1	15	47	19	31
27	CALCULAT	9	6	53	18	3	1	11	16	8	11
28	SIM_PATH	17	16	106	19	4	1	9	41	12	33
29	SIMULATE	237	221	1501	65	53	11	36	738	68	471
30	QUIT_RGS	34	31	194	20	8	3	23	102	26	54
31	CLEAR_SC	35	32	203	22	8	3	24	108	25	56
32	DRAW_MEN	87	45	850	35	1	0	5	142	41	140
33	CONTROL_	22	19	83	26	3	2	20	47	17	21
34	RGS	73	12	344	38	2	1	11	27	11	15

As a more rigorous and practical test, we applied the metrics extractor to ongoing software development projects at the University of Idaho. In their senior year of study, all computer science majors at the University of Idaho must complete two software engineering practicums where they are required to find a customer with a problem requiring software design and implement a solution in one semester. In the first semester students work individually; in the second, they work in groups of four or five. At the time of this study, both practicums called for a specifying approach to software development using abridged versions of the IEEE standards for requirements and design specifications (IEEE Std. 830-1984 and IEEE Std. 1016-1987, respectively) [IEEE87]. Details of the U.I. software engineering practicums can be found in [Oman86].

The PDL metrics extractor was tested on pseudocode taken from the design documents of three second-semester group projects: (1) A Bezier curve fitting program, (2) A materials and resource planning system, and (3) An interactive DFA simulator. In all cases, the metric analysis was conducted unbeknown to the students working on the project. This was done deliberately to avoid interfering with the ongoing development and to exclude Hawthorne effects. All three projects were implemented successfully (each exceeding 3000 lines of Pascal code) and passed the customer's implementation sign-off. The data shown in Tables 1 and 2 are from the DFA simulator project.

The PDL metrics were analyzed statistically to determine intrasystem characteristics, and they were compared to metrics calculated from the final source code. These results showed:

1. PDL metrics follow many of the same statistical patterns exhibited in code metrics. Specifically, there are high correlations ($r>.75$) between size related metrics such as LOC, Halstead's Vocabulary (V), Halstead's Volume (N) and estimated Volume ($\hat{N}$).

2. Metrics for detailed pseudocode modules correlated very high with corresponding source code metrics ($r>.8$), while metrics for loosely written pseudocode showed almost no correlation to the corresponding code ($r<.3$).

3. A generic linear regression model predicting source code metrics from PDL metrics could not be constructed because of variations within a system (as in #2 above) and across systems.

4. Pseudocode modules could be rank ordered by complexity as estimated from the PDL metrics. However, this rank ordering did not always follow the complexity ordering of source code modules. Differences can be attributed to variations mentioned above.

5. Unusual PDL module descriptions could be identified by calculating metric outliers (i.e., values greater than 2 standard deviations away from the average) in precisely the same manner as is done for source code metrics. Furthermore, these PDL outliers frequently matched source code outliers. That is, modules flagged as unusual in the pseudocode descriptions were frequently unusual in the source code implementations.

This last result is the most promising because it provides a mechanism for identifying error-prone designs. Table 3 shows the output from our metric-outlier program as it processed the data from Table 1. Likewise, Table 4 shows the metric-outliers for the source code metrics in Table 2. As shown in these tables, the metric- outlier program calculates average metric values for each column of metrics and then prints, on a module-by-module basis, flags indicating where metric outliers exist. For instance, the value "+2+" indicates the value for a metric (column) of a module (row) is greater than two, but less than three, standard deviations above the average.

Using the calculation of metric outliers we were able to identify anomalies in the PDL. Outliers with

Table 3. PDL Metrics Outlier Table.

	name	tok	loc	n1	N1	n2	N2
1	Init	.	.	-1-	.	.	.
2	DrawMenu	.	.	-1-	.	.	+1+
3	Control	.	.	.	.	.	.
4	MoveCursor	+1+	+1+	.	.	+1+	+2+
5	LoadData	+2+	+2+	+1+	+3+	+1+	+2+
6	SaveData	.	.	.	.	.	.
7	PrintData	.	.	.	.	.	.
8	Simulate	.	.	.	.	.	.
9	DrawEdge	+1+	+2+	+1+	+1+	+1+	+1+
10	ClearScree	.	.	.	.	.	.
11	DrawNode	.	.	+1+	.	.	.
12	DeleteNode	.	.	+1+	.	.	.
13	DeleteEdge	.	.	.	.	.	.
14	QuitRGS	.	.	.	.	.	.
15	PrintTable	.	+1+	.	+1+	.	.
16	PrintSimul	.	.	.	.	.	.
17	GetNextCha	-1-	-1-	-1-	.	-1-	-1-
18	CalcPath	.	.	.	.	.	.
19	SimPath	.	.	.	.	.	.
20	BuildNodeT	.	-1-	-1-	.	-1-	.
21	UpdateNode	.	.	.	.	.	.
22	LabelEdge	.	.	-1-	.	.	.
23	BuildEdgeT	+1+	.	+1+	+1+	+1+	+1+
24	UpdateEdge	.	.	.	.	.	.
25	BezierCurv	+2+	+1+	+1+	+2+	+1+	+1+
26	UpdateStat	.	.	.	.	.	.
27	CalcFact	.	.	.	.	-1-	.
28	DrawCursor	-1-	-1-	-1-	.	-1-	-1-
29	CheckCurso	.	.	.	.	.	.
30	StatusLine	-1-	-1-	-1-	.	-1-	.
31	BuildState	.	-1-	.	.	.	.

Table 4. Code Metrics Outlier Table.

	name	dsl	loc	tok	com	Vg	Nst	n1	N1	n2	N2
1	CHECK_CU	.	.	.	.	.	.	-1-	.	-1-	.
2	DRAW_CUR	.	.	.	.	.	.	.	.	.	.
3	MOVE_CUR	+2+	+2+	+2+	+1+	+1+	+3+	.	+2+	+1+	+2+
4	CALC_FAC	.	.	.	.	.	.	-1-	.	-1-	.
5	BEZIER_C	.	.	.	.	.	.	.	.	.	.
6	LABEL_ED	.	.	+1+	+1+	.	.	.	.	+1+	+1+
7	BUILD_ED	.	.	.	.	.	.	-1-	.	.	.
8	BUILD_ST	.	.	.	.	.	.	.	.	.	.
9	ATAN2	.	.	.	.	.	.	.	.	-1-	.
10	DRAWARRO	.	.	.	.	.	.	.	.	.	.
11	DRAW_EDG	+3+	+3+	+3+	+3+	+2+	+1+	+1+	+3+	+2+	+3+
12	UPDATE_E	.	.	.	.	.	.	-1-	.	.	.
13	UPDATE_S	.	.	.	.	.	.	.	.	.	.
14	DELETE_E	.	.	.	.	.	.	+1+	.	+1+	.
15	BUILD_NO	.	.	.	.	.	.	-1-	.	-1-	.
16	DRAW_NOD	.	.	.	.	.	.	+1+	.	.	.
17	UPDATE_N	.	.	.	.	.	.	-1-	.	-1-	.
18	DELETE_N	.	.	.	.	.	.	+1+	.	.	.
19	PRINT_ST	+1+	+1+	+1+	.	+2+	+1+	.	+1+	.	.
20	PRINT_SI	.	.	.	.	.	.	.	.	.	.
21	PRINT_DA	.	.	.	.	.	.	.	.	.	.
22	STATUSLI	.	.	.	.	.	.	-1-	.	.	.
23	INITIALI	.	.	.	.	.	.	.	.	.	.
24	LOAD_DAT	+1+	+1+	+1+	.	.	.	+1+	+1+	+1+	+1+
25	SAVE_DAT	.	.	.	.	.	.	+1+	.	.	.
26	GET_NEXT	.	.	.	.	.	.	.	.	.	.
27	CALCULAT	.	.	.	.	.	.	.	.	-1-	.
28	SIM_PATH	.	.	.	.	.	.	.	.	.	.
29	SIMULATE	+2+	+2+	+2+	+2+	+2+	+1+	+1+	+2+	+1+	+1+
30	QUIT_RGS	.	.	.	.	.	.	.	.	.	.
31	CLEAR_SC	.	.	.	.	.	.	.	.	.	.
32	DRAW_MEN	.	.	.	.	.	.	-1-	.	.	.
33	CONTROL_	.	.	.	.	.	.	.	.	.	.
34	RGS	.	.	.	.	.	.	.	.	.	.

extremely low values pointed to pseudocode that was either too vague or described small simple modules. On the other hand, outliers with extremely high values pointed to pseudocode that was either too detailed or highly complex. In this fashion we were able to identify "trouble spots" in the PDL. For example, in the Bezier curve fitting project our PDL analysis identified modules Load, Profile, and Sort as three outliers. Subsequent analysis of the source code identified Change, Profile, and Sort as the three most complicated code modules. Further examination showed that both Load and Change had been written by the same student and in both cases the pseudocode was vaque and inadequate. While the implementation of Load proceeded within normal bounds, the implementation of Change contained many instances of redundant code causing abnormally high code metric values. Not unexpectedly, the student designing and implementing these modules was the least experienced of the five working on the project.

We obtained similar results from our analyses of the other two senior projects. For instance, the PDL outliers for the DFA simulator (shown in Table 3) include: MoveCursor, LoadData, DrawEdge, and BezierCurve. Source code outliers (shown in Table 4) include: MoveCursor, DrawEdge, PrintStateTable, and Simulate. MoveCursor and DrawEdge are examples of overly complex modules in both pseudocode and code. BezierCurve and LoadData are examples of false hits: Targeting trouble spots that don't exist. Although the pseudocode showed excessive complexity, the source code showed no unusual characteristics. That is, the pseudocode complexity was either overstated (as in the case of LoadData) or the coding process dispersed the complexity into other code modules (which was the case for BezierCurve). On the other hand, the pseudocode for Simulate and PrintStateTable showed nothing unusual, but the source code was quite complex. PrintStateTable is not overly complex, but its printing conditions inflate the V(g) metric. Simulate, however, is an example of a clear miss: Not identifying a trouble spot. The code for Simulate is quite complex and the pseudocode description does not reflect the true nature of the module.

CONCLUSIONS

We have implemented and tested a prototype PDL metrics extractor that calculates token-based metrics from design pseudocode. The metrics extractor is flexible, but inherently primative because of its context-free classification of tokens. Further, the metrics extractor cannot process entire design specifications documents (such as those outlined in the IEEE 1016-1987 Standard). The pseudocode must be extracted from the design specification prior to processing. Future enhancements of the PDL metrics extractor will include:

1. Context sensitive classification of tokens.

2. Guided processing of whole standard specification documents.

3. Automatic post-processing identification of metric outliers.

To test our metrics extractor we analyzed the PDL sections of design documents from published sources and from systems being developed in a software engineering lab at the University of Idaho. We extracted the PDL complexity metrics and then conducted comparative analysis with respect to intramodule and intermodule relationships and with respect to metrics derived from the corresponding source code. Intramodule analysis suggests that PDL metrics follow the known patterns of code metrics; i.e., the relationships observed and documented across code metrics from a single module are also found in PDL metrics from a single module. Deviations from these known patterns can be used to identify internal inconsistency.

Intermodule relationships are useful in isolating PDL descriptions that are inconsistent with respect to the other modules within the design specification; i.e., identification of metric values that are statistical outliers with respect to the average metric value for all modules within the system. In this study we were able to:

1. Identify PDL descriptions that were inadequate because of lack of detail.

2. Identify PDL descriptions that were too detailed.

3. Identify inconsistent PDL descriptions.

4. Demonstrate a high degree of traceability between detailed PDL descriptions and corresponding source code.

5. Demonstrate a low degree of traceability between inadequate PDL descriptions and corresponding source code.

We believe that PDL metrics can be utilized in manners similar to code complexity metrics; namely, rank ordering module complexity for purposes of resource allocation. Studies are presently underway to determine how PDL metrics can be used for early fault identification and defect removal.

References

[Basi84] V. Basili & B. Perricone, "Software Errors and Complexity: An Empirical Investigation", *Communications of the ACM*, vol. 27(1), Jan. 1984, pp. 42-52.

[Bern84] G. Berns, "Assessing Software Maintainability", *Communications of the ACM*, vol. 27(1), Jan. 1984, pp. 14-23.

[Boeh76] B. Boehm, "Software Engineering," *IEEE Transactions on Computers*, vol. C-25(12), Dec. 1976, pp. 1226-1241.

[Boeh84] B. Boehm, "Verifying and Validating Software Requirements and Design Specifications", *IEEE Software*, vol. 1(1), Jan. 1984, pp. 75-88.

[Bran90] D. Brandl, "Quality Measures in Design," *ACM SIGSOFT Software Engineering Notes*, vol. 15(1), Jan. 1990, pp. 68-72.

[Cont86] S. Conte, H. Dunsmore, & V. Shen, *Software Engineering Metrics and Models*, Benjamin/Cummings, Menlo Park, Ca., 1986.

[Cook84] C. Cook, "Software Complexity Measures," *Proceedings of the 1984 Pacific Northwest Software Quality Conference*, Portland, Oregon, pp. 343-363, 1984.

[Elsh84] J. Elshoff, "Characteristic Program Complexity Measures," *Proceedings of the Seventh International Conference on Software Engineering*, IEEE, Florida, pp. 288-293, 1984.

[Gord79] R. Gordon, "Measuring Improvements in Program Clarity", *IEEE Transactions on Software Engineering*, vol. SE-5(2), Mar. 1979, pp. 79-90.

[Grem84] L. Gremillion, "Determinants of Program Repair Maintenance Requirements", *Communications of the ACM*, vol. 27(8), Aug. 1984, pp. 826-832.

[Hall84] N. Hall & S. Preiser, "Combined network complexity measures," *IBM Journal of Research and*

Development, Jan. 1984, pp. 15-27.

[Hals77] M. Halstead, *Elements of Software Science*, Elsevier, New York, N/Y., 1977.

[Harr86] W. Harrison & C. Cook, "A Micro/Macro Measure of Software Complexity," *Journal of Software Systems*, vol. 7(3), Aug. 1987, pp. 213-219

[Henr90] S. Henry & C. Selig, "Predicting Source Code Complexity at the Design Stage," *IEEE Software*, vol. 7(2), Mar. 1990, pp. 37-44.

[IEEE87] IEEE, *Software Engineering Standards*, John Wiley, New York, N.Y., 1987.

[Jone79] C. Jones, "A Survey of Programming Design and Specification Techniques", *Proceedings, Specifications of Reliable Software*, Apr. 1979, pp. 91-103.

[Kafu85] D. Kafura, "A Survey of Software Metrics," *Proceedings of the 1985 ACM Annual Conference*, Oct. 1985, pp. 502-506.

[McCa89] T. McCabe & C. Butler, "Design Complexity Measurement and Testing," *Communications of the ACM*, vol. 32(12), Dec. 1989, pp. 1415-1425.

[Nann85] T. Nanney, *Computing and Problem-Solving with Pascal*, Prentice Hall, Englewood Cliffs, N.J., 1985.

[Oman86] P. Oman, "Software Engineering Practicums: Case Study of a Senior Capstone Sequence," *ACM SIGCSE Bulletin*, vol. 18(2), June 1986, pp.53-57.

[Parn79] D. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, vol. SE-5(2), Mar. 1979, pp. 128-138.

[Pfle87] S. Pfleeger, *Software Engineering: The Production of Quality Software*, Macmillan, New York, N.Y., 1987.

[Reyn87] R. Reynolds, "The Partial Metrics System: Modeling the stepwise refinement process using partial metrics," *Communications of the ACM*, vol. 30(11), Nov. 1987, pp. 956-963.

[Rohl83] J. Rohl, *Writing Pascal Programs*, Cambridge University Press, London, England, 1983.

[Romb90] D. Rombach, "Design Measurement: Some Lessons Learned," *IEEE Software*, vol. 7(2), Mar. 1990, pp. 17-25.

[Salt82] N. Salt, "Defining software science counting strategies," *ACM Sigplan Notices*, Mar. 1982, pp. 17-26.

[Shel86] G. Shelly, T. Cashman, & S. Forsythe, *Turbo Pascal Programming*, Boyd & Fraser, Boston, MA., 1986.

[Shen85] V. Shen, T. Yu, S. Thebaut, & L. Paulsen, "Identifying Error Prone Software - An Empirical Study," *IEEE Transactions on Software Engineering*, vol. SE-11(4), pp. 317-324, April, 1985.

[Szul81] P. Szulewski, P. Bucher, S. DeWolf, & M. Whiteworth, "The measurement of Software Science parameters in software design," *Performance Evaluation Review* (ACM SIGMETRICS), vol. 10(1), 1981, pp. 89-94.

[Taka85] M. Takahashi & Y. Kamayachi, "An Empirical Study of a Model for Error Prediction", *Proceedings of the Eight International Conference on Software Engineering*, 1985, pp. 330-336.

[Troy81] D. Troy & S. Zweben, "Measuring the quality of structured designs," *Journal of Software and Systems*, vol. 2, 1981, pp. 113-120.

[Zelk79] M. Zelkowitz, A. Shaw, & J. Gannon, *Principles of Software Engineering and Design*, Prentice-Hall, Englewood Cliffs, N.J., 1979.