OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

PROGRAMMING STYLE AUTHORSHIP ANALYSIS

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

Curtis R. Cook
Computer Science Department
Oregon State University
Corvallis, Oregon 97331

87-60-11

PROGRAMMING STYLE AUTHORSHIP ANALYSIS

Paul W. Oman
Computer Science Dept.
University of Idaho

Moscow, Idaho 83843
(208) 882-6589

Curtis R. Cook
Computer Science Dept.
Oregon State University

Corvallis, Oregon 97331
(503) 754-3273

Abstract

Detecting instances of software theft and plagiarism is a difficult problem. The statistical analysis of peculiar words or phrases known to be used by an author is a common method of settling authorship disputes in English literature. This paper presents a similar method for identifying authorship of programs. The method is based on typographic or layout style program characteristics. Our experiments show that these characteristics can be used to determine authorship. The major benefits of the method are that it is simple and easy to automate.

CR Categories and Subject Descriptors: D.2.3 [Coding], D.2.3 [Metrics], D.2.2 [Tools and Techniques]

Keywords: Programming style, coding style, style analysis, typographic style, authorship identification, plagiarism detection.

1. INTRODUCTION

Identifying program authorship is important in detecting instances of software theft and plagiarism. Methods for "fingerprinting" programs to determine instances of software theft and plagiarism have centered on: (1) the comparison of the structural decomposition of the systems or programs under investigation, and (2) the application of a battery of software complexity metrics to identify suspect programs.

Dakin and Higgins [Daki82] describe a method of comparing the logical similarity of two programs based on a Warnier-Orr decomposition of the program structures. They suggest that analysis and comparison of three components -- the logical output structure, the logical data structure, and the logical process structure -- across the two systems in question, will provide strong proof of similarity or dissimilarity.

An analogous method has been proposed by Glass [Glas85], who suggests that two levels of fingerprinting need to be conducted. The first is a comparison of the "external" features of the programs being compared -- the inputs, outputs, and a black-box view of the logical processes. Then, if there remains any questions as to the origin of one of the programs, an internal investigation of the program structure is required. He advocates modular decomposition resulting in a calling structure chart showing the possible control-flow through the system. His methodology is similar to that proposed by Dakin and Higgins.

While these methodologies may be useful in determining the functional equivalency of two programs, it is questionable if they are satisfactory solutions to the authorship identification

problem. Both methods suffer from the same inaccuracies and limitations; neither have been implemented in an automated fashion.

Several articles [Grie81, Otte76, Dona81, Robi80] describe programs that apply a battery of software complexity metrics to detect plagiarism in student programs. Metrics commonly included in the battery are Halstead software science parameters, McCabe's cyclomatic complexity, and counts of the number of lines of code, statements, and/or subprograms. Groups of programs with a high correlation for the battery of metrics are identified as suspects requiring further scrutiny.

However, a study by Berghel and Sallach [Berg84] concluded that software metrics were of limited use in detecting plagiarism. Their study investigated 15 common complexity metrics. After conducting a factor analysis of the metrics representing student programs, they concluded that there was nothing unique about the program features isolated by the metrics. That is, the metrics identified similarities that didn't exist and missed other obvious instances of style similarity. Hence, the best use of complexity metrics based plagiarism detection programs seems to be as a deterrent rather than in actually detecting cases of plagiarism.

The problem of authorship disputes also occurs in English literature. The Federalist Papers [Most64] and Shakespearean writings [Efro86, Kola76] are two well known examples. A widely accepted method for resolving authorship disputes is the statistical analysis of the occurrences of certain "marker"

characteristics (peculiar words or phrases) that occur in the writing, where the marker characteristics are gleaned from other known writings by the author.

In this paper we describe the application of marker characteristics to determining authorship identification of programs. These markers represent unique features of a programmer's programming style. We found that software complexity metrics did not yield markers, but the typographic or layout style characteristics (e.g. indentation, line length, comment format, blank lines, spacing) provide a simple and rich set of markers. We easily discovered a set of typographic style markers for each programmer that allowed us to successfully group programs by programmer.

2. COMPLEXITY METRICS AND PROGRAMMING STYLE

The motivation for our study was an attempt to use traditional software complexity metrics to group a collection of programs by algorithm and/or author. We thought the metrics would uncover the markers that would allow us to group the programs. Twelve complexity metrics (delivered source lines, lines of code, lines of declaration, lines of comments, number of tokens, number of arguments, cyclomatic complexity, Halstead's operand and operator counts, and level of nesting) were calculated for the same three algorithms taken from six different data structure textbooks. Each vector of measurements represented a specific algorithm implemented in Pascal by a known author.

Even though the data contained multiple instances of the

same algorithm written by different authors, repeated statistical analysis for appearance of clustering trends and principal components failed to find any relationship between authorship or algorithm domain and the code complexity measures. We concluded that code complexity metrics are inadequate measures of stylistic factors and domain attributes.

Our conclusion is supported by two previous studies. As mentioned earlier, the Berghel and Sallach [Berg84] study showed that software complexity metrics identified some program similarities that did not exist and missed some obvious instances of similarities. Evangelist's [Evan84] analysis of complexity metrics with respect to style rules provides a plausible explanation for the loose relationship between style and complexity. He demonstrated how application of 26 rules from Kernighan and Plauger's style guide had differing effects on five software complexity metrics (Halstead's effort, McCabe's cyclomatic complexity, Henry and Kafura's information flow, level of nesting, and number of lines of code). Some rules increased complexity (as measured by the metrics) while others decreased complexity and others had inconsistent effects across the different metrics. He concludes, "current complexity metrics are improper indices of program quality, as measured by style."

After our failure to find markers using complexity metrics, we investigated ways in which authors could be identified through markers in their programming style. In a study on programming style, Oman and Cook [Oman87] demonstrated the benefits of distinguishing between classes of style factors and studying the affects and utility of the factors in each style class.

Specifically, they found it helpful to distinguish between the typographic and the structural style classes.

Typographic characteristics represent the physical layout of the code and do not, in any way, affect the performance of the code, although they may affect the maintainability of the code. The typographic category includes factors such as indentation, line length, comment formats, blank lines, spacing, and other layout characteristics. They showed that the typographic factors are more than cosmetic and can have a significant affect on program comprehension.

Structural characteristics impact both efficiency and maintainability. Included in the structural category are the characteristics pertaining to modularity, looping and branching constructs, methods of type and data declarations, level of nesting, control flow, information flow, operator and operand usage, and other factors related to program complexity. The structural decomposition methods of Dakin and Higgins [Daki82] and Glass [Glas85] fall into this category.

Talks with programmers uncovered the belief that programmers can identify authorship from simple typographic characteristics. For example, indentation, commenting, and character usage. Casual observation seemed to support this belief; mainly, that simply by looking at the typographic characteristics, we where able to group code by authorship. In the next section we describe experiments that show typographic style factors do provide unique programmer markers which can be used for programmer identification.

3. AUTOMATED AUTHORSHIP ANALYSIS

This section describes experiments that show typographic characteristics provide a rich source of markers and that a simple statistical analysis of the markers permits grouping by author.

3.1 Identifying authorship -- a protocol study.

Simple protocol studies were conducted to see if authorship could be determined from analysis of the typographic characteristics of the source code.

To test this hypothesis we took Pascal source code for three algorithms from each of six computer science textbooks. The code segments were a bubble sort, quicksort and a set of tree traversal algorithms (preorder, inorder, and postorder). Each code segment was copied verbatim onto a microcomputer and printed one per page to eliminate all publisher differences (i.e. typesetting). The 18 pages were then shuffled and given to eleven programmers with instructions to group the code by author. Each author's collection would contain one bubble sort, one quicksort, and one set of tree traversals. All but one of the subjects grouped the code perfectly, the other subject made one mistake by switching the tree traversals on the two most inconsistent authors.

An informal protocol analysis was conducted while the subjects were grouping the code listings. The subjects recognized that some authors are very consistent across code segments while others are much less so. It was interesting to note that all subjects easily grouped the most consistent

authors' works, and did so first, leaving the harder task (i.e. grouping the inconsistent authors) for last. The subjects proceeded by identifying certain characteristics or peculiarities about the authors' style (markers) and then used these markers to distinguish between authors. An example marker would be placing multiple assignment statements on one line; another would be always differentiating keywords and identifiers by case.

The protocol analysis and subsequent post-test discussions led to the following mechanisms by which authorship could be identified:

1. Whether comments are inline, blocked, bordered, and/or occur after keywords.
2. The consistency of indentation, number of spaces used, and how certain constructs are aligned (e.g. the IF-THEN-ELSE statement).
3. The use of upper and lower case, and the underscore character, to differentiate between keywords and identifiers.
4. The placement of statements, how statements are placed in conjuction with others (especially nested statements) and whether or not there are more than one per line.
5. The presence or absence of blank lines and how they are used to separate chunks or blocks of code.
6. The choice and length of identifiers (e.g. meaningful names versus single letter identifiers).

3.2 A typographic style checker.

To verify the results of our protocol studies we designed and implemented a typographic style analyzer based on the mechanisms identified by the protocol subjects. Specifically, the analyzer processes Pascal source code and generates a boolean value for each of the following conditions:

a. Inline comments on the same line as source code.

b. Blocked comments (two or more comments occurring together).
c. Bordered comments (set off by repetitive characters).
d. Keywords followed by comments.
e. 1 or 2 space indentation most frequently occurring.
f. 3 or 4 space indentation most frequently occurring.
g. 5 or greater indentation most frequently occurring.
h. Lower case characters only (all source code).
i. Upper case characters only (all source code).
j. Case used to distinguish between keywords and identifiers.
k. Underscore used in identifiers.
l. BEGIN followed by a statement on the same line.
m. THEN followed by a statement on the same line.
n. Multiple statements per line.
o. Blank lines in the declaration area.
p. Blank lines in the program body.

For each condition the boolean value is true if the characteristic is present in the code under analysis, and false if the characteristic is absent. The analysis proceeds on a module-by-module basis with the output being a boolean matrix with each row representing a module (program, procedure, or function) and each column representing one of the above conditions. To obtain a typographic style bit vector for algorithms with embedded modules, the bit vector for each embedded module was OR-ed together with the main module. This is functionally equivalent to processing the entire algorithm as one block (except that multiple indentation methods may appear).

The above measures are highly consistent across modules written by an author with a consistent style and not so with inconsistent authors. The measures are easily quantifiable and reflect the consistency of the author. Furthermore, these typographic factors are generally invariant with respect to problem requirements.

Table 1 shows the boolean typographic style measures for the data used in the protocol study. A simple index of typographic

Table 1. Typographic Style Vectors for Textbook Data

	INL	BLK	BOR	KEY	I2	I4	I5	LCO	UCO	<C>	U_S	BGN	THN	;;;	BLD	BLB
Text A (IR = 0)																
Bubblesort	1	1	0	1	1	0	0	0	1	0	0	0	0	0	1	1
Quicksort	1	1	0	1	1	0	0	0	1	0	0	0	0	0	1	1
Tree Trav.	1	1	0	1	1	0	0	0	1	0	0	0	0	0	1	1
Text B (IR = 1)																
Bubblesort	1	1	1	1	1	0	0	0	0	1	0	0	1	0	0	0
Quicksort	1	1	1	1	1	0	0	0	0	1	0	0	1	0	0	1
Tree Trav.	1	1	1	1	1	0	0	0	0	1	0	0	1	0	0	0
Text C (IR = 2)																
Bubblesort	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
Quicksort	0	0	0	0	0	1	0	0	0	1	0	0	0	0	1	1
Tree Trav.	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0	0
Text D (IR = 3)																
Bubblesort	1	1	0	1	0	0	1	1	0	0	0	0	1	0	0	0
Quicksort	1	0	0	1	0	0	1	1	0	0	0	0	1	0	1	1
Tree Trav.	1	0	0	1	0	0	1	1	0	0	0	0	1	0	0	0
Text E (IR = 6)																
Bubblesort	1	0	0	0	1	0	0	1	0	0	0	0	1	1	1	1
Quicksort	1	1	0	1	1	0	0	1	0	0	0	0	1	1	1	1
Tree Trav.	0	0	0	0	1	0	0	1	0	0	0	0	1	0	0	0
Text F (IR = 6)																
Bubblesort	1	0	0	1	0	0	1	1	0	0	0	1	0	1	0	0
Quicksort	1	0	0	1	0	1	0	1	0	0	0	1	0	1	0	0
Tree Trav.	0	0	0	0	0	0	1	0	0	0	0	1	0	0	0	0

INL - inline comments
BLK - block of comments
BOR - bordered comments
KEY - comments after keywords
I2 - 1 & 2 space indentation
I4 - 3 & 4 space indentation
I5 - 5 & greater indentation
LCO - lower case only
UCO - upper case only
<C> - case distinguishes keywords
U_S - underscore used
BGN - BEGIN & statement on 1 line
THN - THEN & statement on 1 line
;;; - multiple statements per line
BLD - blank lines in declarations
BLB - blank lines in the body

IR - Inconsistency Rating

style consistency can be derived by counting the number of typographic style factors showing any instance of inconsistency. In our study of 16 factors, the best consistency rating would be zero for perfect consistency; while a rating of 16 would represent complete inconsistency (in all factors). Inconsistency Ratings (IR) for each textbook are also shown in Table 1. Note the differences in consistency between Textbook A and Textbook F.

Early versions of the typographic style analyzer also computed identifier variety and length, keyword variety, and frequency of certain control structures (e.g. repeat loops). All of these measures proved to be too inconsistent within suites of modules written by the same programmer to be of value here.

3.3 Clustering by authorship.

To test the utility of the boolean typographic style vectors we conducted a clustering analysis to determine if distinct typographic styles could be grouped by author. Using the textbook data, the proximity of each algorithms's style measurements to all other algorithms was computed by taking the Hamming distance between the typographic style bit vectors for each algorithm.

The result is a symmetrical N x N distance matrix with zeros along the main diagonal. The lower (or upper) triangle of this distance matrix was then analyzed using the SPSS-X Cluster procedure with a minimum distance clustering criteria [Noru85]. The results, shown in Figure 1, follow those of our protocol study. Specifically, algorithms written in a consistent style are clustered perfectly, while those written by less consistent

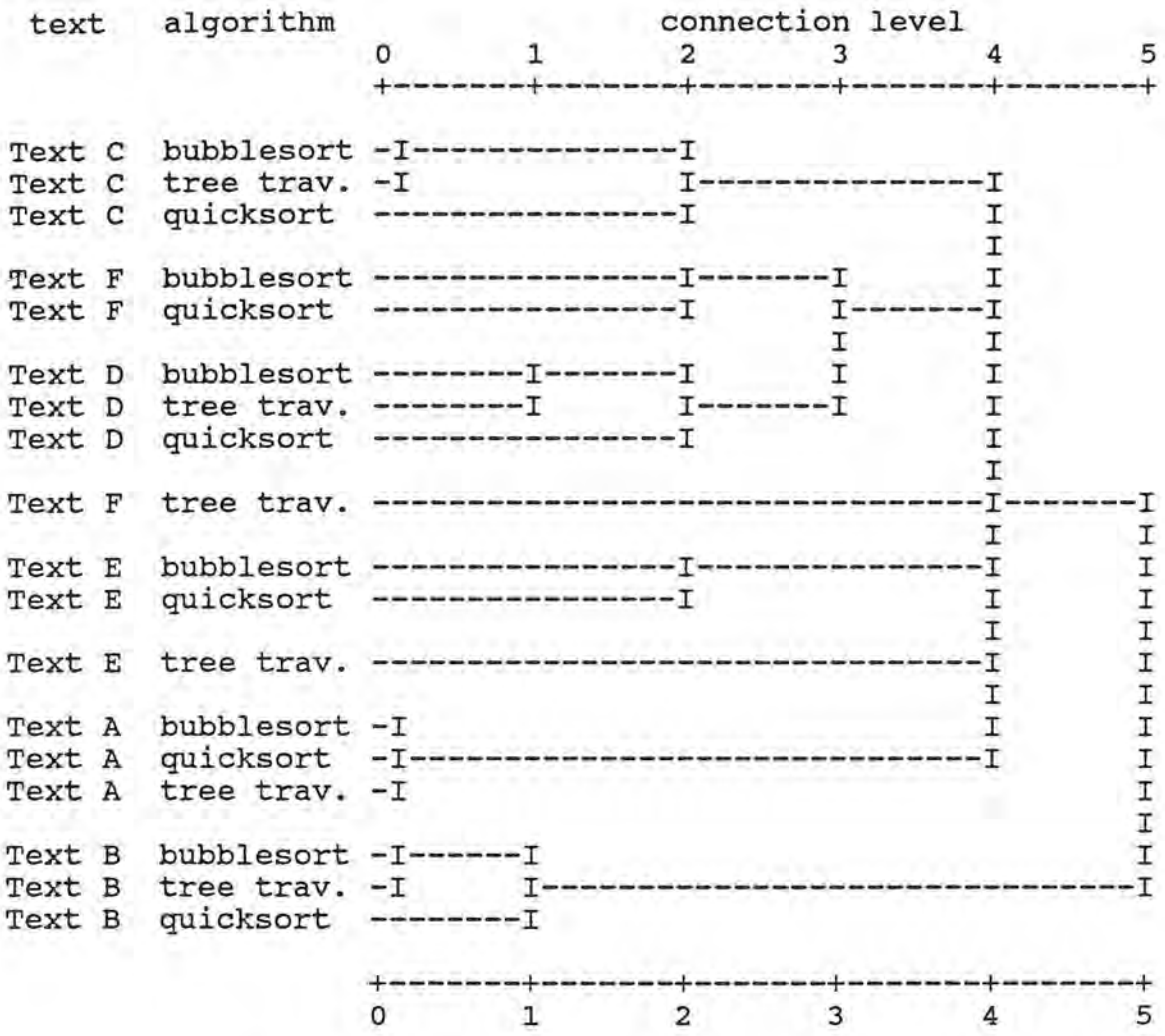

Figure 1. Cluster Analysis of Textbook Authors

authors are somewhat scattered.

We then repeated this clustering analysis on industrial data. Pascal source code was obtained from two international computer manufacturers and one microelectronics research laboratory. The source code from each firm exemplified a program written and used by that organization. The length of the three programs (including blank lines and comments) was 6024, 1445, and 2711 lines of source code.

The typographic style checker was useful in measuring the internal consistency of the code style, identifying anomalies amoung the modules contained within a firm's program, and comparing the styles across companies. Figure 2 demonstrates the ability to cluster code styles by authorship. Six modules were selected at random from each of the three programs, run through the style checker, reduced to a distance matrix, and then clustered using the minimum distance criteria. As shown in Figure 2, this methodology identifies and clusters the modules from Company A and Company B perfectly. The Company C code is least consistent; joining at higher connection levels and containing anomalies like Module #15.

We have applied this system to other industrial code, individual student projects, and code from teams of students working in software engineering practicums. It is far from infallible, but usually provides a convenient means of measuring style consistency and grouping code by authorship.

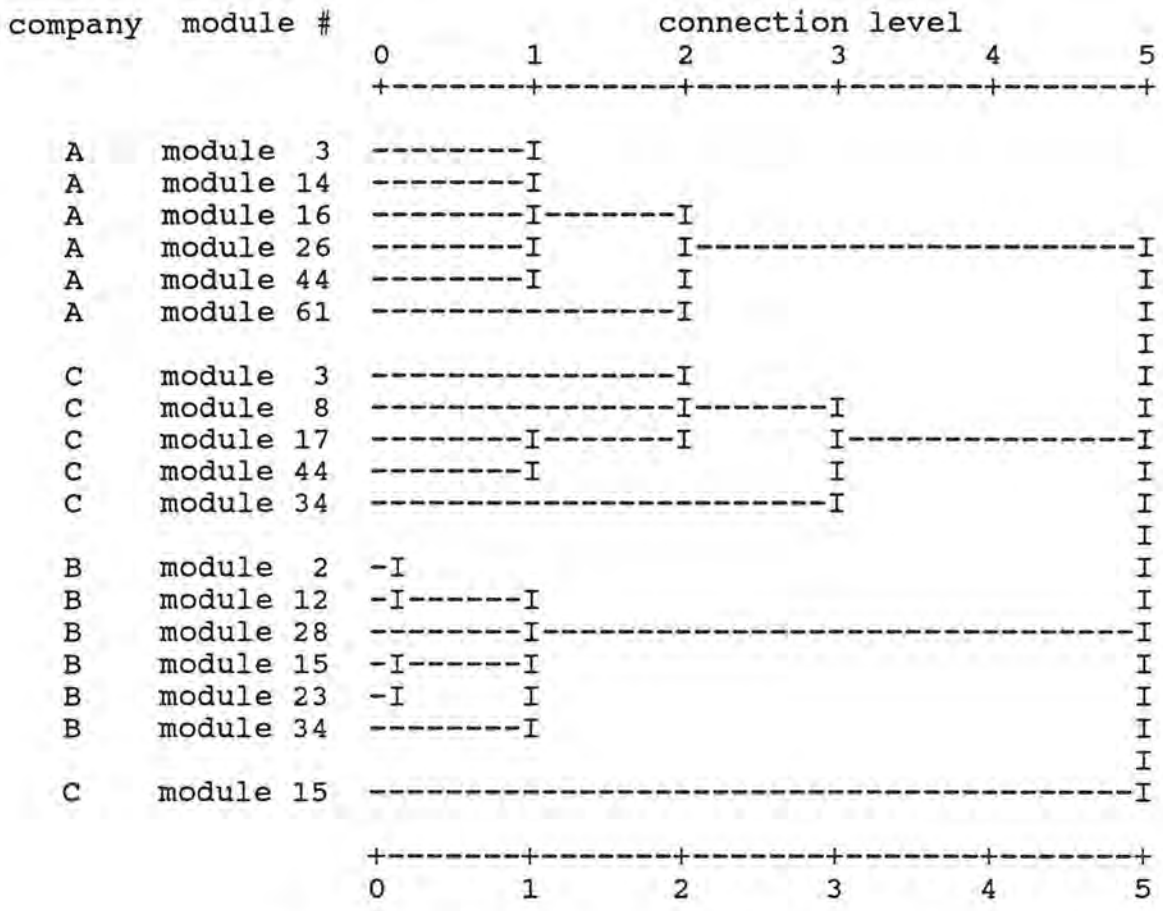

Figure 2. Cluster Analysis of Industrial Code

4. CONCLUSIONS

The statistical analysis of markers is a common and widely used method for settling authorship disputes in English writing. We have proposed a similar method for identifying authorship of programs. Typographic style characteristics provide a rich and easily automated source of markers for use in statistical analysis. This method can assist in detecting software theft and plagiarism.

A prototype author identification system has been developed wherein the typographic bit vector from a sample program is compared to a database of bit vectors to determine the closest match. To date, this has only been tested with student data, but preliminary results are surprisingly accurate. This type of system may someday prove useful as a plagarism detection mechanism when used in conjuction with other methodologies.

Measuring code consistency is another advantage of this method. A program with a consistent typographical style is easier to maintain than one with an inconsistent style. A tool that automatically checks for typographic style consistency would aid the maintainer and would also check adherence to style standards.

We do not claim that the typographic and structural classification of programming style and corresponding typographic style analyzer completely captures programming style. We suggest that automated authorship identification and plagiarism detection systems should be sensitive to both typographic and structural concerns, as well as other factors contributing to programming style. A programming style analyzer that takes into account

several classes of stylistic characteristics would be a powerful tool for consistency checking, standards enforcement, and maintainability assessment as well as authorship identification and plagiarism detection.

References

[Berg84] H. L. Berghel and D. L. Sallach, "Measurements of Program Similarity in Identical Task Environments", ACM SIGPLAN Notices, vol. 19(8), Aug. 1984, pp. 65-76.

[Daki82] K. J. Dakin and D. A. Higgins, "Fingerprinting a Program", Datamation, Aug. 1982, pp. 133-144.

[Dona81] J. L. Donaldson, A. Lancaster, and P. H. Sposato, "A Plagiarism Detection System", ACM SIGCSE Bulletin, Feb. 1981, pp. 31-40.

[Efro76] B. Efron and R. Thisted, "Estimating the Number of Unseen Species: How many words did Shakespeare know?", Biometrika, Vol. 63, 1976, p. 435.

[Evan84] M. Evangelist, "Program Complexity and Programming Style", Proceedings of the International Conference on Data Engineering (Los Angeles CA, Apr. 24-27). IEEE, Silver Springs MD, 1984, pp. 534-541.

[Glas85] R. L. Glass, "Software Theft", IEEE Software, vol. 2(4), July 1985, pp. 82-85.

[Grie81] S. Grier, "A Tool that Detects Plagiarism in Pascal Programs", ACM SIGCSE Bulletin, Feb. 1981, pp. 15-20.

[Kola86] G. Kolata, "Shakespeare's New Poem: An Ode to Statistics", Science, Vol. 24, Jan. 1986, pp 335-336.

[Most64] F. Mosteller and D. L. Wallace, Inference and Disputed Authorship: The Federalist Papers, Addison-Wesley Publishing Co., Reading, MA, 1964.

[Noru85] M. Norusis, SPSS-X: Advanced Statistics Guide, McGraw-Hill Book Co., New York NY, 1985.

[Oman87] P. Oman and C. Cook, "A Paradigm for Programming Style Research", (Submitted for publication).

[Otte76] K. J. Ottenstein, "An Algorithmic Approach to the Detection and Prevention of Plagiarism", ACM SIGCSE Bulletin, Dec. 1976, pp. 30-41.

[Robi80] S. Robinson & M. Soffa, "An Instructional Aid for Student Programs", ACM SIGCSE Bulletin, Feb. 1980, pp. 118-127.