Measuring Software Design Complexity

D. N. Card and W. W. Agresti

Computer Sciences Corporation, Silver Spring, Maryland

Architectural design complexity derives from two sources: structural (or intermodule) complexity and local (or intramodule) complexity. These complexity attributes can be defined in terms of functions of the number of I/O variables and fanout of the modules comprising the design. A complexity indicator based on these measures showed good agreement with a subjective assessment of design quality but even better agreement with an objective measure of software error rate. Although based on a study of only eight medium-scale scientific projects, the data strongly support the value of the proposed complexity measure in this context. Furthermore, graphic representations of the software designs demonstrate structural differences corresponding to the results of the numerical complexity analysis. The proposed complexity indicator seems likely to be a useful tool for evaluating design quality before committing the design to code.

1. INTRODUCTION

Typically, design is the earliest stage of software development at which the pending software system is fully specified and in which the system structure is clearly defined. Design usually proceeds in two steps—architectural, then detailed design. This study only considers the former. Throughout the following discussion, "design" will refer to architectural design unless otherwise indicated. Assessment of the quality of a software design rates high in the priorities of software developers and managers. However, the multitude of potentially conflicting design objectives, methods, and representations, as well as a lack of appropriate data, have hindered the development of effective measures of software design quality.

One quality attribute, complexity, has been studied extensively. Early investigations [1, 2] focused on the internal organization of individual programs or subprograms rather than on the structure of software systems composed of large numbers of subprograms (or modules). More recently, complexity studies have attempted to consider software systems [3, 4]. Many of these approaches require extensive analysis (usually special tools) to compute values of the complexity measures proposed. Moreover, few of these measures can be computed at design time. The objective of this study was to define some "simple" complexity measures that could easily be derived during early design.

The initial investigation considered many existing models of software complexity but did not find any of them suitable for this application because 1) necessary data were difficult to extract or compute, 2) required information was not available during architectural design, and/or 3) our data data did not support the model. For example, all of these reservations apply to software science [1]; see Card and Agresti [28].

This paper explains a new approach to measuring software design complexity that considers the structure of the overall system as well as the complexity incorporated in individual components. The measures derive from a simple model of the software design process. Analysis of data from eight medium-scale scientific software projects showed that the complexity measures defined in this report provide a good estimate of the overall development error rate, as well as agreeing with a subjective assessment of design quality. Furthermore, differences in design complexity indicated by the complexity measures also demonstrated themselves in design profile graphs.

This analysis relied on data collected by the Software Engineering Laboratory (SEL) from eight spacecraft flight dynamics projects. The SEL is a research program sponsored by the National Aeronautics and Space Administration [5]. It is supported by Computer Sciences Corporation and the University of Maryland. The objectives of the SEL are to measure the process of software development in the flight dynamics environment at Goddard Space Flight Center, identify technology improvements, and transfer this technology to flight dynamics software practitioners.

Address correspondence to David N. Card, Computer Sciences Corporation, 8728 Colesville Road, Silver Spring, MD 20910.

0164-1212/88/$3.50

2. NATURE OF DESIGN COMPLEXITY

Architectural design is the process of partitioning the required functionality and data of a software system into parts that work together to achieve the full mission of the system. Thus, architectural design complexity can be viewed as having two components: 1) the complexity contained within each part (or module) defined by the design, and 2) the complexity of the relationships among the parts (modules). In the following discussion, we will refer to design parts as modules, in the sense that a module is the smallest independently compliable unit of code [6]. Each design part will eventually be implemented as a software module. In the FORTRAN environment of the SEL, modules correspond to subroutines.

Many different approaches or methods achieve the same design result: a high-level architectural design and an integrated set of individual module designs. The detailed design (e.g., PDL) developed to implement the work assigned to a module provides another source of complexity that is not analyzed here. It is not the intent of this paper to address whether specific design methods result in lower-complexity (or better) design products. Rather, its objective is to demonstrate a complexity measurement approach that can be applied to a wide range of such products, regardless of how they were produced. The authors recognize that correct design practice is essential to achieving good designs. Generally, this report shows that the conditions that result in lower values of the complexity measures are consistent with accepted design practices.

Of course, any complete design must include nonmodules such as files and COMMON blocks (in FORTRAN). Furthermore, partitioning is not the only design process. This proposed model only attempts to capture a subset of all the possible factors in complexity. As Curtis [7] points out, complexity depends on the perspective from which an object or system is viewed. This paper examines software complexity with respect to the difficulty of producing the designed system (for example, the difficulty of changing the implemented system is not considered). The following discussion is intended to illustrate the line of reasoning followed in developing the model and measures. It should not be construed as a mathematical proof that this model is a necessary and sufficient explanation of complexity.

2.1 A Design Model

One common approach to design is functional decomposition (the basis of structured design [6]). It results in a hierarchical network of units (or modules). For any module, workload consists of input and output items

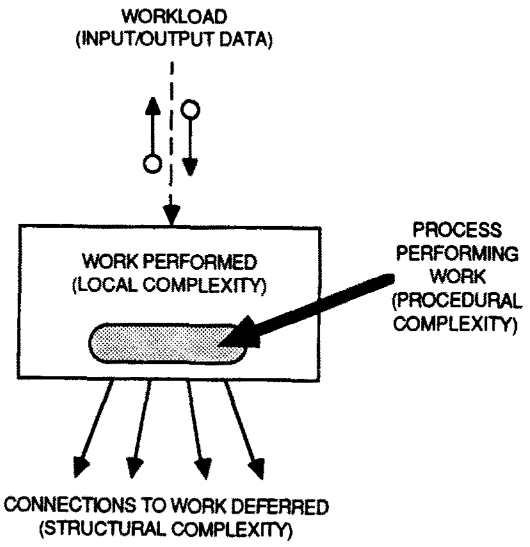

Figure. 1. Decomposition model of software design.

(data couples) to be processed. At each level of decomposition, the designer must decide whether to implement the indicated functionality (perform the work) in the current module or defer some of it to a lower level by invoking one or more other modules (via control couples). Deferring functionality decreases the *local* (intramodule) complexity but increases the *structural* (intermodule) complexity (see Fig. 1). Similar decisions also must be made when following other design approaches (e.g., object oriented [8]).

The internal design of a module (how the work is performed) may contribute *procedural* complexity, but that is outside the scope of this paper. Of course, many early studies of software complexity (e.g., [2]) focused on process construction. The distinction made here between local and procedural complexity parallels the distinction between the specification and the body of an Ada* package.

Thus, architectural complexity is a function of the work performed (within modules) as well as the connections among the work parts (modules). Effective design minimizes work as well as connections. This argument leads to the following formulation for the total complexity of a software design:

$$C\sim = S\sim + L\sim \qquad (1)$$

where

$C\sim$ = total design complexity

$S\sim$ = structural (intermodule) complexity

$L\sim$ = local (intramodule) complexity

* Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

That is, the total complexity of a design of given complexity $C\sim$ can be defined as the sum of intermodule plus intramodule complexity. In this simple model, all complexity resides in one or the other of these two components; hence, they are additive. These complexity components correspond to the structured design concepts of module strength (or cohesion) and coupling defined by Stephens et al. [6].

2.2 Relative Complexity

Because projects (and designs) vary greatly in terms of magnitude, a measure of relative complexity ultimately may prove more useful than total complexity. Dividing by the number of modules defined in the design normalizes these complexity measures for size so that designs of different magnitudes may be compared:

$$C = S + L \tag{2}$$

where

C = $C\sim/n$ (relative design complexity)

S = $S\sim/n$

L = $L\sim/n$

n = number of modules in system

Although individual modules may vary greatly in size in terms of lines of source code, the module, as it is used here, is the unit of design. Hence it is the appropriate normalization factor. The rest of this discussion will concern relative complexity.

3. DEFINITION OF COMPLEXITY MEASURES

The next sections define measures for each of the two components of relative complexity just identified in Equation 2. The measures incorporate counts in the design characteristics (calls, variables, and modules) identified in the model. (Table 1 summarizes some design measures from the modules studied in this analysis). The following sections also discuss methods and consequences of minimizing complexity as defined by this model.

Table 1. Design Measures Summary

	Minimum	Mean	Maximum
Module size	1	66	603
Fanin	1	1.3	16
Fanout	0	2.8	27
I/O variables	1	24	237
Level	2	6.1	11

Note: Based on 1,142 newly developed modules.

3.1 Structural Complexity

Structural complexity derives from the relationships among the modules of a system. The most basic relationship is that a module may call or be called by another module. The structurally simplest system consists of a single module. For more complex systems, structural complexity is the sum of the contributions of the component modules to structural complexity. These potential contributions are occurrences of fanin and fanout as noted by Henry and Kafura [9], as well as by Belady and Evangelisti [3]. (Fanin is the count of calls to a given module. Fanout is the count of calls from a given module.)

In the SEL data analyzed (see Table 1), multiple fanin generally confined itself to modules that were simple mathematical functions reused throughout the system. Consequently, fanin did not prove to be an important complexity discriminator. On the other hand, fanout proved to be highly sensitive, as indicated in a previous study [10]. Counting fanout only also ensures that each connection is counted exactly once. Note that lower fanout indicates less coupling in the sense that there are fewer couples (without regard to their strength [11] or type [6]).

According to this model, a module with a fanout of zero contributes nothing to structural complexity. However, the distribution of fanout within a system also affects complexity. The interconnection matrix representation of partitioning used by Belady and Evangelisti [3] suggests that complexity increases as the square of connections (fanout). All descendents of a given module are connected to each other by their common parent. Then, for a fixed total fanout, a system in which invocations are concentrated in a few modules is more complex than one in which invocations are more evenly distributed. These considerations lead to the following formulation for structural complexity:

$$S = \frac{\Sigma f_i^2}{n} \tag{3}$$

where

S = structural (intermodule) complexity

f_i = fanout of module "i"

n = number of modules in system

This quantity is the average squared deviation of actual fanout from the simplest structure (zero fanout). Henry and Kafura's term "(fanin * fanout) ** 2" [9] reduces to fanout-squared when fanin is assumed equal to one (the nominal case). Similarly Belady and Evangelisti's measure of complexity [3] is a function of the number of nodes (modules) and edges (fanout) in a system or cluster (partition).

The fanout count defined here does not include calls to system or standard utility routines, but does include calls to modules reused from other application programs. A reused module must be examined by the designer to determine its appropriateness—as opposed to standard utilities that are well understood by developers.

3.2 Local Complexity

The internal complexity of a module is a function of the amount of work it must perform. The workload consists of data items that are input to or output from higher or parallel modules. This definition is consistent with Halstead's concept [1] of the minimal representation of a program as a function (single operator) with an associated set of I/O variables (operands). This workload measure parallels the idea of actual data bindings as used by Hutchens and Basili [11].

Then, to the extent that functionality (work) is deferred to lower levels, the internal complexity of a module is reduced. Averaging the internal complexities of a systems's component modules produces its local complexity. Most guidelines for decomposition suggest decomposing into units of equal functionality. Assuming, for simplicity, that the workload of a module is evenly divided among itself and subordinate modules leads to the following formulation of complexity:

$$L=\frac{\Sigma \frac{v_i}{f_i+1}}{n} \qquad (4)$$

where

L = local (intramodule) complexity

v_i = I/O variables in module "i"

f_i = fanout of module "i"

n = number of new modules in system

The "+1" term represents the subject module's share of the workload (incidentally, it prevents the divide-by-zero condition from arising when a module has no fanout). I/O variables include distinct arguments in the calling sequence (an array counts as one variable) as well as referenced COMMON variables. An earlier study [10] indicates that the presence of unreferenced COMMON variables does not affect module quality. Data item complexity is not considered here (only newly developed modules enter into this computation).

Henry and Kafura [9] used the count of source lines of code to represent intramodule complexity. However, as used in Henry and Kafura [9], no matter how large the module, its complexity would be zero if it had no fanout. Basili et al. [12] showed source lines of code (size) to be highly correlated (r = 0.79) with the number of I/O variables (operands). Another earlier study [13] shows that high-strength modules [6] tend to be small. Consequently, the local complexity measure may be an indicator of average moaule strength (or cohesion [6]).

3.3 Minimizing Complexity

Design complexity, as defined in the preceding sections, can be minimized by minimizing its structural and local components. However, these components are not independent. Both measures include fanout. Minimizing structural complexity requires minimizing the fanout from each module. For a given number of both modules and total fanout, structural complexity is minimized when fanout is evenly distributed across all modules (except terminal nodes, of course). On the other hand, local complexity can be minimized by maximizing fanout or minimizing variable repetition.

Repetition occurs whenever a data item appears in more than one module as a calling sequence argument or referenced common variable. Internal uses (including CALLs to other modules) do not count as repetition. In general, minimizing local complexity will produce smaller modules (in terms of executable statements), but is also may increase structural complexity disproportionately. For a given module with a fixed number of I/O variables, the fanout that contributes minimum complexity can be determined as follows:

$$c=f^2+v/(f+1)$$

where

c = contribution of given module to total complexity per Equations 2, 3, and 4

then

$$dc/df=2f-v/(f+1)^2$$

at minimum

$$0=2f-v/(f+1)^2$$

then

$$v=2f(f+1)^2 \qquad (5)$$

Figure 2 shows a plot of Equation 5 as a step function (to reflect the discrete natures of v and f). It identifies the fanout that minimizes complexity for possible counts of I/O variables. For example, in the range from about 100 to 200 I/O variables, complexity is minimized with a fanout of 3. Since very few modules include as many as 200 I/O variables, the plot indicates that the commonly accepted range of values for fanout (up to 7 ± 2) is much too large. Curtis [7] suggests that the popularity of this bound derives from a misunderstanding of certain psychological studies. This implication is consistent with

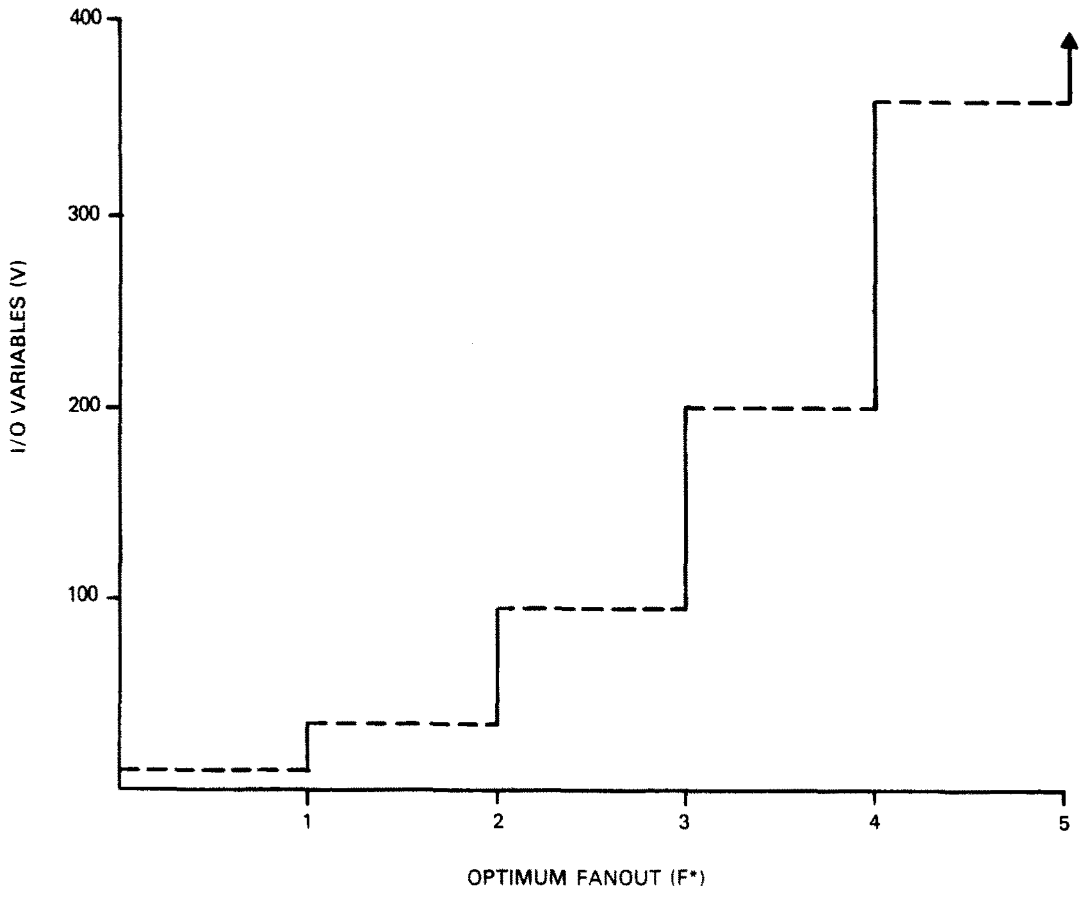

Figure 2. Selecting fanout to minimize complexity.

an earlier study [10]. Furthermore, Constantine [6] observes that most programs can be decomposed effectively into a common structure of three parts: input, process, and output. Larger fanouts may indicate too rapid decomposition. This result suggests than a fanout of one is a reasonable value for modules with few I/O variables.

In addition to the selection of an appropriate fanout, design complexity can also be minimized by reducing variable repetition, i.e., by not including variables where they are not needed. Rigorous application of the principle of information hiding [14] should reduce variable repetition and, hence, local complexity.

Figure 3 shows two design segments of equal structural complexity: The number and distribution of fanouts are identical. Each data couple represents a repetition of the variable "X". Figure 3a traces this variable through a design following strict topdown decomposition rules. "X" appears in the higher level modules (A, B, D) as well as in the lower level modules (C, E). Figure 3b shows an alternative design with a horizontal transfer of data that bypasses the higher level modules (for the case in which modules A, B, and D do not actually use "X"). The local complexity of the intermediate modules (B, D) in the strict top-down configuration (Figure 3a) exceeds their counterparts in the alternative design (Figure 3b) because their counts of I/O variables are larger.

Parameter transfer between hierarchically adjacent modules (e.g., from B to A) produces a lower complexity than transfer via a global area when that is as far as the data item goes. For a triplet connection (e.g., from B to A to D), the two approaches have the same complexity ("X" counts twice in each). This implication is consistent with the results of an earlier study [10]. Because this model emphasizes the number of data couples rather than the nature of the coupling mechanism, it penalizes "tramp data" (data passed through but not referenced by a module).

Rotenstreich and Howden [15] argue that both horizontal and vertical data flow are essential to good design. Appropriate use of horizontal transfers prevents data flows from violating levels of abstraction. COMMON blocks provide the only mechanism for horizontal data transfers in FORTRAN. Figure 3 shows that horizontal flows can reduce the magnitude of the local complexity measure in some situations.

Of course, a less complex design might also be

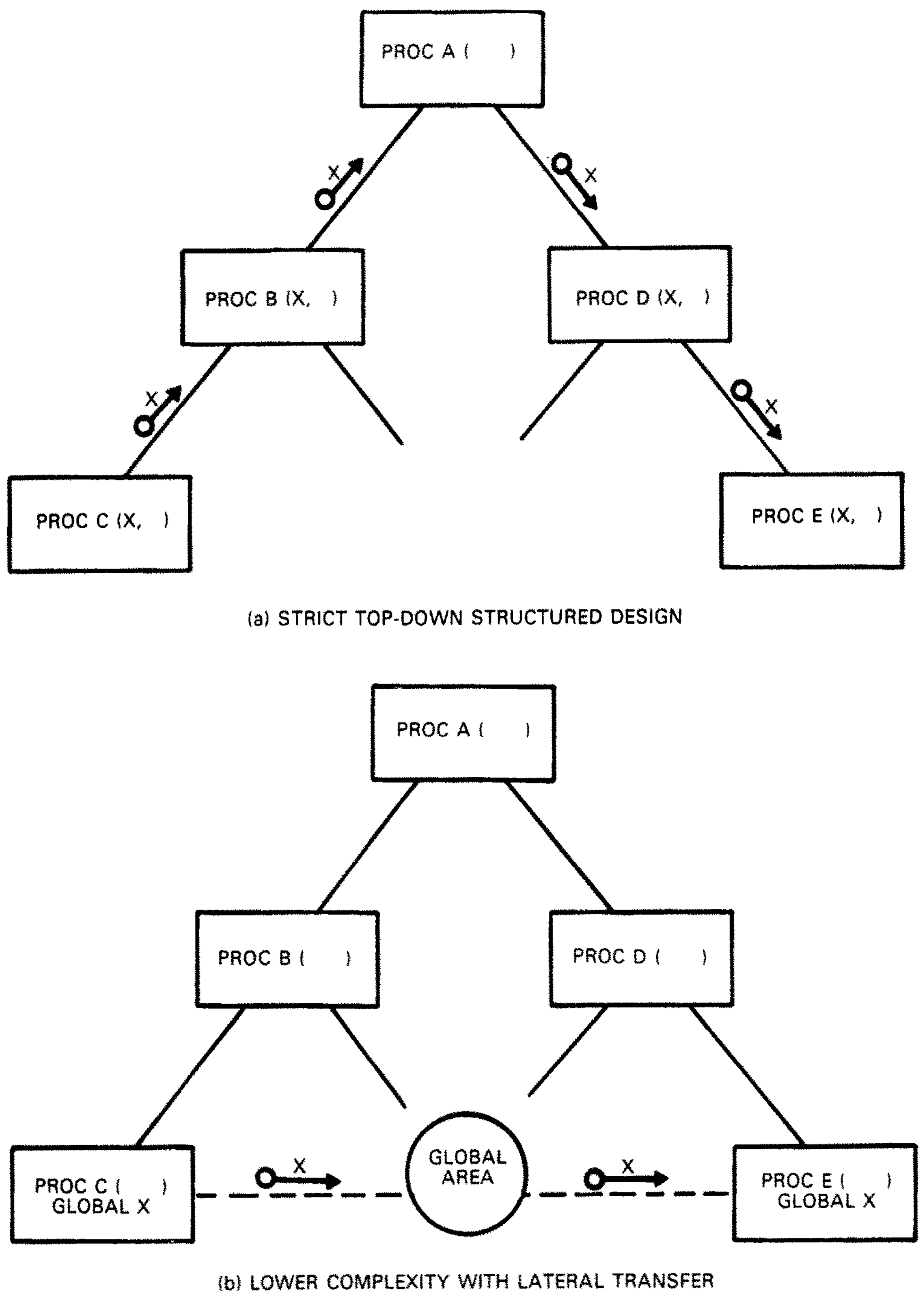

Figure 3. Reducing variable repetition to minimize complexity.

produced by partitioning the work differently and restructuring this design. For example, PROC C could be invoked directly by PROC E (if the nature of the problem permitted). This simpler structure would also be reflected in lower values of the complexity measures defined by this model. (PROCs B and C would each have fanout of one instead of PROC B having fanout of two. Thus, structural complexity diminishes.)

4. EVALUATION OF COMPLEXITY MEASURES

The value of the complexity measures defined in the preceding sections was evaluated in two ways. First the complexity scores for the eight projects were compared with a subjective rating of design quality using a nonparametric statistical technique. Then the complexity scores were compared with objective measures of development productivity and error rate. This section presents the results of the two evaluation approaches. Productivity and error rates were computed using the developed lines of code (DLOC) measure as defined by Basili and Freburger [16].

Data for this analysis were extracted from the source code of eight projects by a specially developed analysis tool. However, software developers can easily extract at design time the counts of modules, fanout, and I/O variables necessary to compute these complexity measures. The eight projects studied were ground-based flight dynamics systems for spacecraft in near-earth orbit. Table 2 summarizes some general characteristics of these software systems. The most recent project studied was completed in 1981.

All of these systems were designed and implemented to run under the Graphics Executive Support System (GESS), an interactive graphics interface [17]. Consequently, GESS occupies Level 1 of each design hierarchy. GESS manages most external data interfaces for these systems. It is not included in the complexity calculations.

4.1 Subjective Quality

The eight projects were subjectively ranked in order from best to worst, in terms of design quality, by a senior manager who participated in the development of all eight projects. Then, the four best-rated designs were classified as "good' while the other four were classified as "poor." Table 3 shows the results of that procedure. The table also includes the computed complexity measures. Note that the four designs subjectively rated as "good" also demonstrated the lowest relative complexity. The expert was not provided with specific criteria for "quality," but later reported that perceived "complexity" played a major role in assigning scores.

Table 2. Project Characteristics

Project	Total Modules	Percent Reused[a]	Size (KDLOC[b])	Error Rate[c]	Productivity[d]
A	158	11	50	8.7	3.5
B	203	34	49	8.0	2.9
C	338	32	106	4.5	4.7
D	259	84	37	4.0	4.7
E	327	24	83	4.5	4.8
F	393	47	79	7.1	4.1
G	199	49	57	7.2	2.3
H	245	43	56	6.6	2.4

[a] Percent of total modules.
[b] Thousands of developed lines of code.
[c] Errors per KDLOC.
[d] Developed lines of code per hour.

Table 3. Design Complexity and Quality

Project	Complexity			Design Rating[b]	Quality Class
	S	L	C[a]		
A	24.6	8.2	32.8	5	Poor
B	15.8	9.5	25.3	2	Good
C	11.8	12.1	23.9	3	Good
D	18.4	4.9	23.3	1	Good
E	12.6	10.0	22.6	4	Good
F	22.3	7.3	29.6	6	Poor
G	18.3	10.8	29.1	8	Poor
H	19.2	7.3	26.5	7	Poor

[a] $C = S + L$ as previously defined (Equation 2).
[b] Subjective evaluation (1 = best, 8 = worst).

Although the correspondence between subjective design rating and numerical design complexity is not one-for-one, if the data are viewed as quality classes, they provide persuasive evidence for a relationship. (If one uses the Wilcoxon rank sum statistic the probability is less than 0.02 that the observed good/poor grouping could occur by chance alone.) The objective complexity measure appears to capture much of the information that a human observer includes in a subjective evaluation of design quality.

4.2 Performance Prediction

The other test of the value of these complexity measures is their ability to predict software development performance in terms of the productivity and error rate ultimately realized by the development team. A more complex design will be more difficult to develop into an acceptable system. However, let us first define a few relevant quantities:

Developed lines of code—all newly developed source lines of code plus 20% of reused source lines of code [16].

Errors—conceptual mistakes in design or implementation. An error may result in one or more faults (code changes). These were detected during integration and system testing (after unit testing).

Effort—hours of work by programmers, managers, and support personnel directly attributable to a project.

Productivity—developed lines of code divided by effort (in hours).

Error rate—total errors divided by developed lines of code.

The developed lines of code metric attempts to account for the lower cost and error rate attributable to reused code. Table 2 shows the developed lines of code,

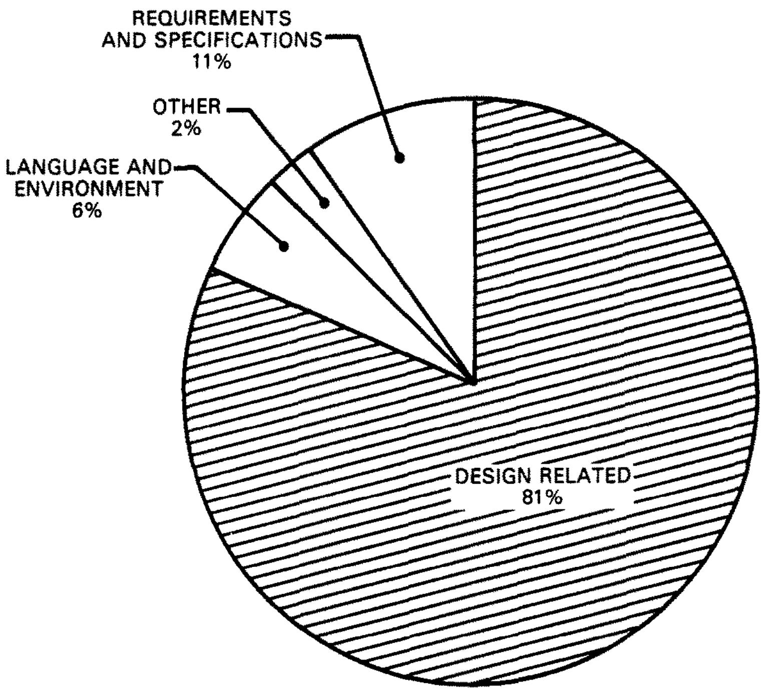

Figure 4. Source of software errors (from Weiss and Basili [19]). Note: Excludes clerical/transcription errors.

productivity, and error rate for the eight projects. Note that together these projects represent more than, 1,000 individual new modules produced by about 50 different programmers.

Designers and researchers commonly assume that higher complexity increases the propensity for error. Potier et al. [18] observe that the implementation process consists largely of translating design specifications into a programming language. It usually does not add complexity to a system. Weiss and Basili [19] show that the bulk (74–82%) of all nonclerical errors reported in three of these projects were related to design, although sometimes at very detailed levels. Figure 4 shows the median distribution of errors for the projects studied by Weiss and Basili [19]. Very few of these errors are true programming errors. Of course, many detailed design and implementation errors are detected during code reading and unit testing (not counted here). In this context, clerical/transcription errors can be regarded as random.

Figure 5 illustrates the relationship between design complexity and error rate. It shows that design complexity effectively predicts the total error rate for development projects. Complexity (as measured here) accounts for fully 60% of the variation in error rate. As seen in Figure 5, all but one of the points lie very close to the regression line. In that case, Project B, the implementation team consisted of an unusually large proportion of junior personnel (although its design team was comparable to those of the other projects). Consequently, it seems reasonable to find a higher error rate than would be indicated by design complexity alone.

Figure 6 illustrates the relationship between design complexity and productivity. No clear relationship emerges. However, as noted elsewhere [20], many important factors external to the development process (such as computer use and programmer expertise) strongly affect productivity. In this case (consistent with [20]), computer-hours-per-thousand-developed-lines of code correlates strongly with the residuals from the Figure 6 relationship ($r = -0.79$). Computer support was only provided to these projects for detailed design, coding, and testing, so it does measure a different set of activities. However, the small sample size (at the project level) inhibits evaluation of a more complex model incorporating both complexity and computer use.

In this organization, the design team forms the nucleus of the implementation and test teams. Additional personnel join as they are needed. Thus, the complexity measure provides an early indication of the performance of the development team as well as of the quality of the design. A good design team is likely to be a good implementation and testing team, although productivity may be difficult to predict.

5. REPRESENTATION OF DESIGN STRUCTURE

The numerical quantities defining these complexity measures are the number of modules, fanout, and I/O variables. Table 4 shows the distribution of these

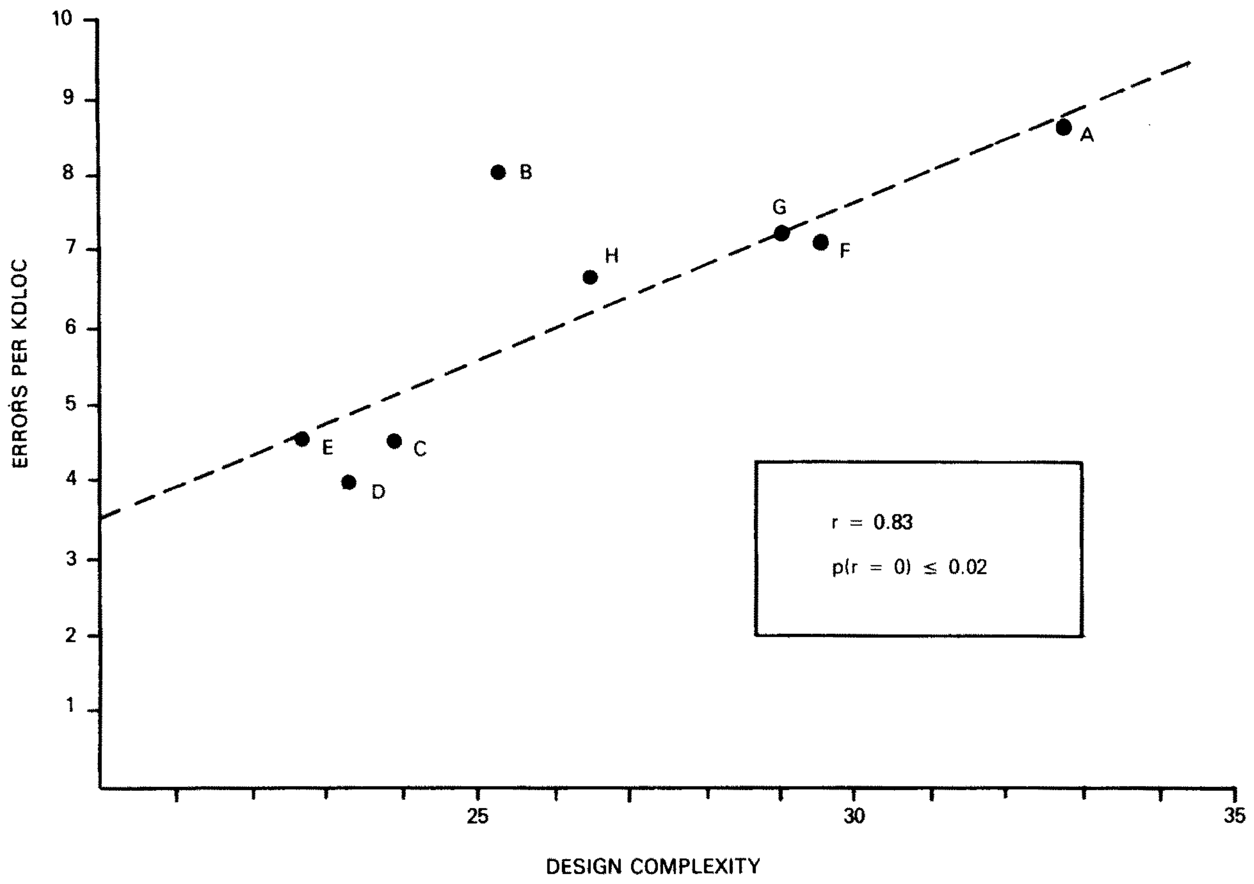

Figure 5. Relationship to error rate.

measures by hierarchical level for one project. This design structure can be represented graphically, as shown in Figure 7, by plotting the cumulative percentage of these quantities obtained at each level. Kafura and Henry [21] employed a similar technique to show the effect of design changes on complexity.

In this and subsequent plots, the design structure (or profile) is simplified by combining all utility modules, regardless of where they are invoked, into a single deepest level of the design. That point is not plotted (utility refers to new or reused modules that are invoked from several different points within a design but not to system or standard utilities). Levels greater than or equal to 10 also are combined into a single level to facilitate plotting.

As discussed earlier, the conditions that minimize structural complexity result in an even distribution of fanout. This produces an increasing growth rate in the cumulative percentage of total fanout in the initial levels of the design, followed by a gradual decrease in growth rate as subtrees terminate. The percentage of modules is driven by the fanout at the preceding level (minus calls to utilities). Uneven use of utilities causes the module line to fail to track fanout. Equation 5 showed that I/O variables should be proportional to fanout in order to minimize local complexity. Together, these conditions define the shape of a good (low relative complexity) design.

Figure 7 illustrates Project E, the design with the lowest relative complexity. It shows three closely fitted "S" shaped curves. Figure 8 illustrates Project A, the design with the highest relative complexity. It shows three separate and irregular lines. Profiles of the other six projects fall in-between these two extremes in correspondence to their measured complexity.

6. CONCLUSIONS

The complexity measures proposed in this report are supported by substantial empirical evidence. The structural complexity component is similar to measures used successfully by Belady and Evangelisti [3] and Henry and Kafura [9] for other languages and application areas. However, neither of these models, as originally formulated, fit the SEL data very well. The new model

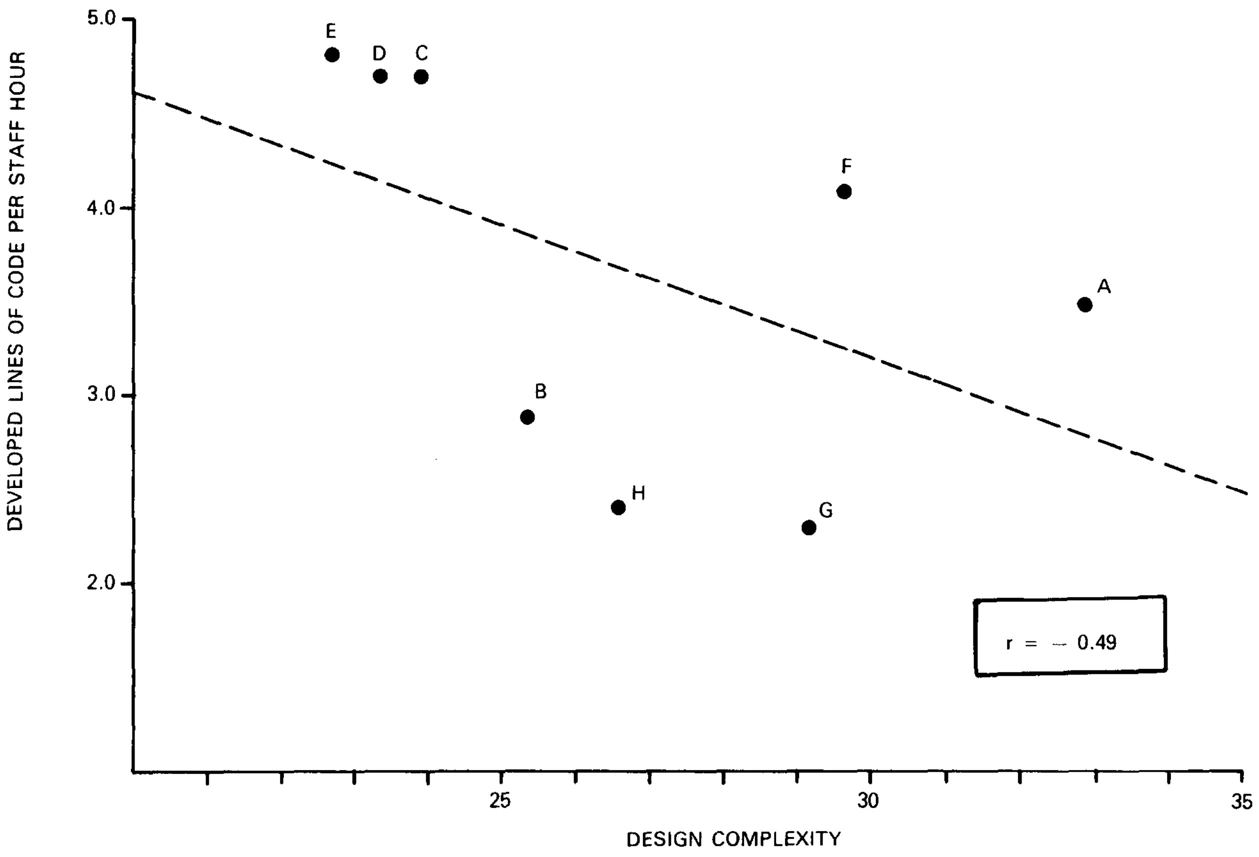

Figure 6. Relationship to productivity.

demonstrated good agreement with subjective assessments of design quality as well as a numerical measure of error rate. Moreover, all relevant measures can be extracted at design time; the Henry and Kafura model includes a code measure.

Table 4. Detailed Design Structure for Project E

		Module Average		
Level	Modules	Executable statements	Fanout	Input/output variables
2	2	91	6.5	45
3	4	37	4.8	9
4	19	59	5.6	29
5	93	67	2.2	26
6	62	59	2.0	24
7	54	59	1.8	20
8	33	37	1.4	14
9	7	19	0.7	8
≥10	2	8	0.0	5
Utility	51	90	2.4	21

Many software development methods, e.g., [22], encourage trying design alternatives. Because software developers can easily compute values for these complexity measures at design time, they seem likely to be useful for assessing design quality and comparing design alternatives before committing a design to code. Overall high-complexity designs, as well as individual high-complexity modules, can be identified. These measures could be adapted to support a measures-guided methodology such as that proposed by Ramamoorthy et al. [23].

Of course, complexity is not the only important attribute of software designs. The minimum complexity that can be achieved depends on the nature of the application and the presence of design constraints. Furthermore, design is not a deterministic process. The same design approach or method applied by different individuals can result in different designs. These complexity measures help us to answer the question, "Which is better?" However, it is not enough to produce a design that shows low complexity scores. Following a sensible and well-defined design method ensures that the design problem is responded to while minimizing complexity. Measures play a supporting role in the design process.

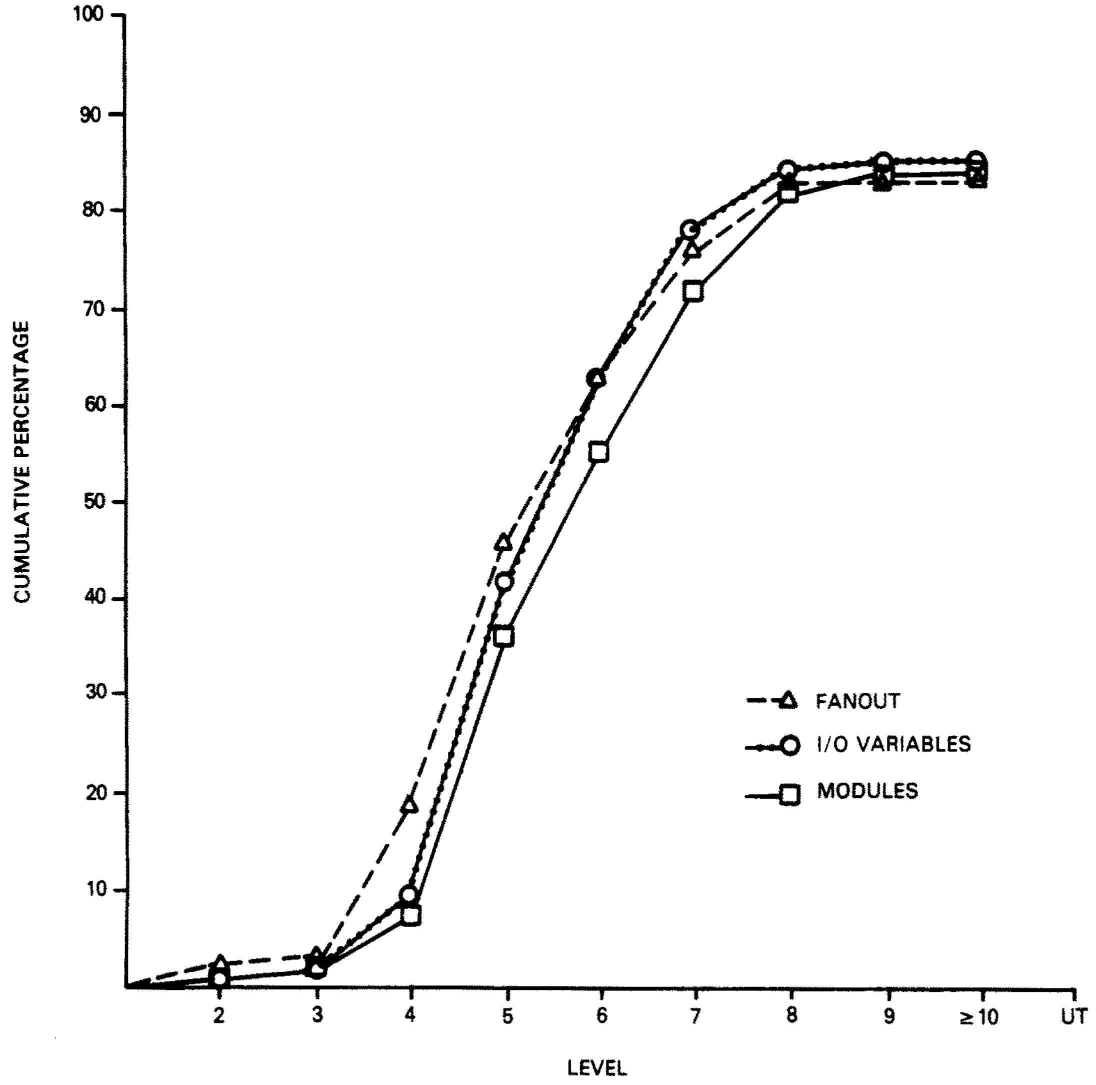

Figure 7. Design profile of Project E (lowest complexity).

As Kearney et al. pointed out [24], ill-founded reliance on complexity measures can degrade the software development process by rewarding poor programming practices. The approach to complexity measurement presented here satisfies the requirements of Kearney et al. [24] for effective complexity measures by clearly identifying the attributes measured, deriving them from a model of the design process, suggesting how they can be used in practice, and empirically testing their validity. Nevertheless, more work remains to be done.

Three aspects of this current complexity measurement approach require additional research. First, methods of incorporating external I/O (e.g., files) into the complexity measures must be developed. In the systems studied, much of the external I/O is handled by the GESS standard interface. Second, the application of the measures should be extended to designs using different formalisms intended for different implementation languages. "Modules" corresponding to FORTRAN subroutines are not a universal design structure. The SEL has begun to study the application of these measures to Ada design [25]. Third, the existence of two design complexity components suggests that two different types and distributions of the design errors (in addition to programming errors) also exist, as proposed by Basili and Perricone [26]. That needs to be verified empirically.

Finally, Kafura and Reddy [27] showed that similar complexity measures appeared to related to software maintainability. This suggests another new area of investigation.

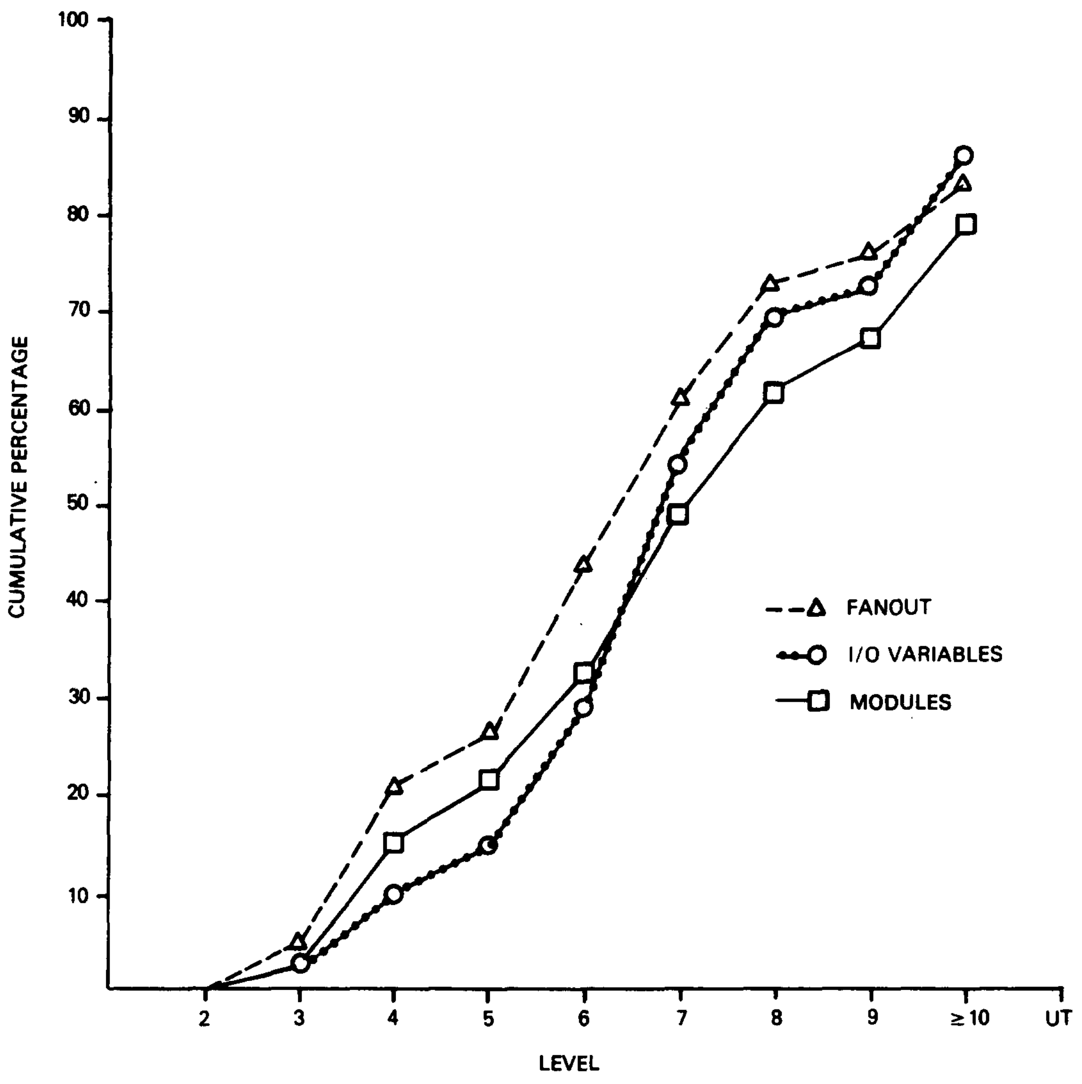

Figure 8. Design profile of Project A (highest complexity).

ACKNOWLEDGMENTS

The authors would like to recognize the significant contributions made to this work by F. E. McGarry (NASA Goddard Space Flight Center), G. T. Page (Computer Sciences Corporation), and V. R. Basili (University of Maryland). We would also like to thank the referees for their useful suggestions for discussion and clarification.

REFERENCES

1. M. H. Halstead, *Elements of Software Science*, Elsevier Science Publishing, New York, 1977.
2. T. J. McCabe, A Complexity Measure, *IEEE Trans. Software Engineering* 2, 308-320 (1976).
3. L. A. Belady and C. J. Evangelisti, System Partitioning and Its Measure, *J. Systems Software* 2, 23-39 (1981).
4. D. A. Troy and S. H. Zweben, Measuring the Quality of Structured Designs, *J. Systems Software* 2, 113-120 (1981).
5. D. N. Card, F. E. McGarry, G. T. Page, et al., *The Software Engineering Laboratory*, NASA/GSFC, SEL-81-104, 1982.
6. W. P. Stevens, G. J. Myers, and L. L. Constantine, Structured Design, *IBM Systems J.* 2, 115-139 (1974).
7. B. Curtis, In search of software complexity, in *Proceedings, IEEE Workshop on Quantitative Software Models*. Computer Society Press, New York, 1979, pp. 95-106.
8. G. Booch, Object Oriented Design, *IEEE Trans. Software Engineering* 12, 211-221 (1986).
9. S. M. Henry and D. G. Kafura, Software Structure Metrics Based on Information Flow, *IEEE Trans. Software Engineering* 7, 510-518 (1981).
10. D. N. Card, V. E. Church, and W. W. Agresti, An Empirical Study of Software Design Practices, *IEEE Trans. Software Engineering* 12, 264-271 (1986).
11. D. H. Hutchens and V. R. Basili, System Structure

Analysis: Clustering with Data Bindings, *IEEE Trans. Software Engineering* 11, 749-757 (1985).
12. V. R. Basili, R. W. Selby, and T. Phillips, Metric Analysis and Data Validation Across FORTRAN Projects, *IEEE Trans. on Software Engineering* 9, 652-663 (1983).
13. D. N. Card, G. T. Page, and F. E. McGarry, Criteria for software modularization, in *Proceedings, IEEE Eighth International Conference on Software Engineering.* Computer Society Press, New York, 1985, pp. 372-377.
14. D. L. Parnas, On the Criteria to be Used in Decomposing Systems into Modules," *Commun. ACM* 15, 1053-1058 (1972).
15. S. Rotenstreich and W. E. Howden, Two-Dimensional Program Design, *IEEE Trans. Software Engineering* 12, 377-384 (1986).
16. V. R. Basili and K. Freburger, Programming Measurement and Estimation in the Software Engineering Laboratory, *J. Systems Software* 2, 47-57 (1981).
17. Computer Sciences Corporation, CSC/SD-75/6057, *Graphics Executive Support System User's Guide,* 1975.
18. D. Potier, J. L. Albin, R. Ferreol, and A. Bilodeau, Experiments with computer software complexity and reliability, in *Proceedings, IEEE Sixth International Conference on Software Engineering.* Computer Society Press, New York, 1982, pp. 94-101.
19. D. M. Weiss and V. R. Basili, Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory, *IEEE Trans. Software Engineering* 11, 157-168 (1985).
20. D. N. Card, F. E. McGarry, and G. T. Page, Evaluating Software Engineering Technology, *IEEE Trans. Software Engineering,* 13, 845-851 (1987).
21. D. G. Kafura and S. M. Henry, Software Quality Metrics Based on Interconnectivity, *J. Systems Software* 3, 121—131 (1982).
22. S. Steppel, T. L. Clark, et al., *Digital Systems Development Methodology,* Computer Sciences Corporation, 1984.
23. C. V. Ramamoorthy, W. Tsai, T. Yamaura, and A. Bhide, Metrics guided methodology, in *Proceedings, IEEE Ninth International Conference on Software and Applications.* Computer Society Press, New York, 1985, pp. 111-120.
24. J. K. Kearney, R. L. Sedlmeyer, W. B. Thompson, et al., Software Complexity Measurement, *Commun. ACM* 29, 1044-1058 (1986).
25. W. W. Agresti, V. E. Church, et al., Designing with Ada for satellite simulation: A case study, in *Proceedings of the First International Conference on ADA Applications for the NASA Space Station,* 1986, pp. F.1.3.1-14.
26. V. R. Basili and B. T. Perricone, Software Errors and Complexity: An Empirical Investigation," *Commun. ACM* 27, 42-52 (1984).
27. D. G. Kafura and G. R. Reddy, The Use of Software Complexity Metrics in Software Maintenance, *IEEE Trans. Software Engineering* 13, 335-343 (1987).
28. D. N. Card and W. W. Agresti, Resolving the Software Science Anomaly, *J. Systems Software* 7, 29-35 (1987).