NORTH-HOLLAND

Quantitative Models of Cohesion and Coupling in Software

Harpal Dhama

Development and Methodology / Tools, The MITRE Corporation, Bedford, Massachusetts

Our project goal is to specify, implement, and verify quantitative models for measuring cohesion and coupling (C & C) in software modules. This article is our project interim report on the specification of the C & C quantitative models and preliminary verification effort. To quantify cohesion, we subdivided it into four categories and then quantified each category. Coupling is subdivided into four categories, followed by the quantification of each category. Although the C & C concepts are applicable to any procedural language such as FORTRAN, PASCAL, or Ada, we chose to apply the C & C formulas to Ada programs. We have hand-calculated C & C values for a number of programs, but here we report and discuss in detail only a typical result of our calculations obtained by applying the C & C formulas to two different implementations of an algorithm. We have found that the formulas are sensitive enough to distinguish between the two implementations, and the obtained quantitative values agree with the qualitative assessment of the implementations.

1. INTRODUCTION

Software life cycle costs depend on software quality factors such as complexity, maintainability, reusability, reliability, and portability. The two properties of software that have a great impact on software quality are cohesion and coupling (C & C). Figure 1 shows that 8 of the 13 software quality factors identified by the Rome Laboratory (Bowen et al., 1983) are dependent on C & C. Thus, the identification, measurement, and management of C & C in software can have a major influence on reducing software costs.

Cohesion in a module refers to that software property that binds together the various statements and other smaller modules comprising the module. We define a module to be a compilation unit of code. A module can contain other smaller modules. Therefore, a function, a procedure, or any combination of these is referred to as a module. Cohesion is an intramodule property that reflects the design considerations for integrating the various components of the module into one unit. It is the glue that holds a module together, and it is a measure of the logical strength of a software module. The strength and consequently the "quality" of the module increase with increasing cohesion.

Coupling is a measure of the interdependence between two software modules. It is an intermodule property. Because it is desirable that the changes made in a module affect another module as little as possible, the "quality" of a module increases as module coupling decreases.

2. OBJECTIVES

Qualitative evaluation of C & C has been used to measure the "good" qualities of software. However, the subjective judgment and consequent inconsistency inherent in qualitative assessments have raised questions about the consistency and credibility of such evaluations. Our goal is to build quantitative models of C & C and then use the models to evaluate C & C for existing software modules. To verify our models, we will run a controlled experiment in which we will take ~20 algorithms implemented in more than one way. These implementations will be

Address correspondence to Harpal Dhama, Development Methodology / Tools, The MITRE Corporation, 202 Burlington Road, Bedford, MA 01730. e-mail: dhama@mitre.org

J. SYSTEMS SOFTWARE 1995; 29:65–74

655 Avenue of the Americas, New York, NY 10010

0164-1212/95/$9.50
SSDI 0164-1212(94)00128-A

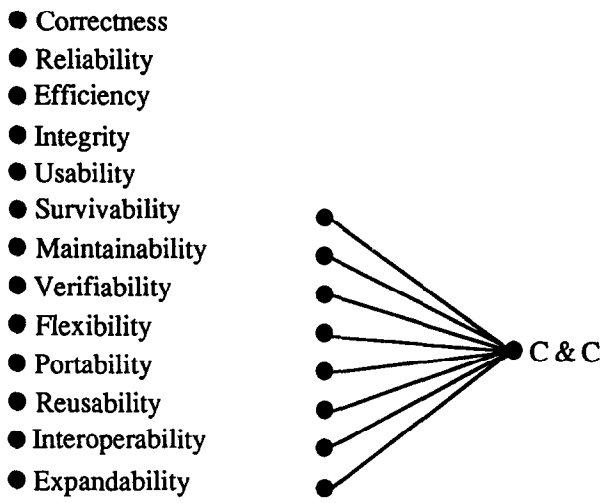

Figure 1. Hierarchy of software qualities as proposed by Rome Laboratory (Bowen et al., 1983).

evaluated by use of our C & C formulas and will also be evaluated qualitatively by experts for C & C. The two evaluations will then be compared to demonstrate that the models are a valid characterization of C & C. We have chosen Ada software for our experiment. The results of this initial pilot study will be used as feedback to tune our model as needed.

Because hand calculation of C & C values is tedious and error prone, we are in the process of building a tool to apply our formulas. After running our controlled experiment, the tool can be used to calculate C & C values for a large number of software modules from existing Ada software. This will allow us to establish an approximate average and range of C & C values. These values can then be used to construct a scale of "goodness" to be used for software evaluation purposes.

In this article, we discuss and describe only the first part of our project: the development of the quantitative models of C & C and the results of our hand application of the C & C formulas to a few sample programs. This represents our initial effort at building these models.

2.1 Motivation for this Research

The significance of C & C was established by Yourdon, Constantine, and Myers (1976) in the period 1973–1979, and their analysis forms a part of Yourdon and Constantine's (1979) book, *Structured Design.* A large portion of the subsequent work on software metrics has been done by Rome Laboratory–funded research, and in the early 1980s, there were a number of Rome Laboratory publications (Bowen et al., 1983) promoting and outlining the use of software metrics.

Although the importance of C & C in software metrics has been well established, there have been limited attempts to quantify C & C. In 1989, Ott and Thuss at Michigan Technological University developed a quantitative model of cohesion using a slice-based[1] methodology. In June 1992, Zage et al. built a model of module coupling and showed that the number of module errors increases as coupling increases. In an empirical study done by Card et al. in 1986 at Computer Sciences Corporation, high cohesion values have been related to fewer software errors. Because a correlation between C & C and number of errors has been established, the C & C metric can be used for

- forecasting testing costs and reliability
- allocating the testing effort according to the error proneness of the module
- software quality assurance
- assessing the quality of reusable software

3. MODEL DEVELOPMENT

There are only a few quantitative models in the software metrics area, and of these, only a handful have been verified empirically. In the development of our prototype model of C & C, we make some assumptions and then carry out experiments to verify the assumptions. The model that we propose has a number of constants built into it. As a first estimate, these constants have been assumed to be 1 and 2. These values may change, depending on our experimentation results.

To build quantitative models for C & C, we have divided cohesion into functional, data flow, action-bundling, and logical bundling cohesion; coupling has been divided into data and control flow, global, and environmental coupling. We have then quantified the various categories of C & C by analyzing the source lines of Ada code and gathering statistics that characterize the code. To gain confidence in our theoretical work, we have hand calculated C & C values for ~15 Ada modules, and in each case, the C & C values agree with the qualitative evaluation of the modules. As a sample of our calculations, Ap-

[1]A slice-based methodology is the reduction of a program to a minimal form that reflects a chosen program behvior called the slicing criteria. For example, a subset of a program consisting of only the statements that affect the value of a particular variable X would be a program slice with respect to X.

pendix A shows the results of the hand application of the C & C formulas to two implementations of a "selection sort" algorithm, which sorts integers in ascending or descending order. The first implementation, titled MODULE1, consists of a single module, whereas the second implementation, titled MODULE2, consists of four submodules but uses the same algorithm to do the sorting. Although we have hand calculated values of C & C for other programs, the chosen example offers some interesting insights into the C & C calculation process; the example is by no means a verification of our theoretical work. In this application, as in others, we have found that our C & C formulas are sensitive enough to distinguish between the two implementations, and the obtained quantitative values agree with the qualitative assessment of the implementations.

When a module contains a number of other submodules, we calculate the C & C values of each of the submodules, and the average of these values is used in determining the C & C value of the containing module. Because C & C is an interval scale, we feel that this is a legitimate approach and has been used by other researchers (Bowen et al., 1983; Henry and Kafura, 1981; Zage et al., 1992).

3.1 Development of a Theoretical Model of Cohesion

We categorize cohesion into the following four categories:

- Functional cohesion
- Data flow cohesion
- Action-bundling cohesion
- Logical bundling cohesion

The definition and development of a metric for each of these different types of cohesion follows. In the development of our C & C formulas, we have a series of constants, $q_1, q_2, q_3, \ldots$ that are used to weigh the effect of the various factors that influence C & C. These constants have been assumed to be 2 and may be revised at a later date.

3.1.1 Functional cohesion. This type of cohesion results from the single-purpose functional design of the module. The more focused the module goal, the greater its functional cohesiveness. The functional strength is inversely proportional to the generality of the functional purpose of the module, the meaning of generality being the ability to perform multiple functions within a given capability. If we can now find a measure of generality, then we have indirectly evaluated functionality.

One of the measures of module generality is the *action content* of the module. The action content in turn is dependent on the number of parameters in the interface of the module and on the number of global and local variables. The intuitive notion used here is that the module capability depends on the number of different items available for manipulation within the module, whether the items come into the module from the outside or are locally defined. McCabe Associate's (1993) object-oriented software evaluation tool uses this aspect of cohesion to measure cohesion in a "method" for manipulating objects. We also postulate that variables used as control variables, for example, those used in IF-THEN or WHILE statements, have the potential of increasing functionality by twice that of data variables. The action content of a module also depends on the number of other modules called because this represents a group of related actions to be carried out by a subordinate. Lacking any other guidelines, we make the simple assumption that such calls have twice as much effect on functionality as simple data variables.

To quantify generality, let us consider a module. Let

F = functional cohesion of a module
$i1$ = in data parameters
$i2$ = in control parameters
$u1$ = out data parameters
$u2$ = out control parameters
$l1$ = number of local variables used as data
$l2$ = number of local variables used as control
$g1$ = number of global variables used as data
$g2$ = number of global variables used as control
w = number of modules called

Because we have assumed a singular functionality to be inversely proportional to generality, there is an inverse relationship between functional cohesion and the variables enumerated above.

Now let $p = i1 + q_1 i2 + u1 + q_2 u2 + l1 + q_3 l2 + g1 + q_4 g2 + q_5 w$, where q_1, q_2, q_3, q_4, and q_5 are constants, and, as a first heuristic estimate, are assumed to be 2 in our calculations.

Then,

$$F \alpha \frac{1}{p}$$

When a module is called, there is an implicit transfer of control to the module, that is, an implied system control parameter is being passed to the module. Therefore, when all the constituents of p

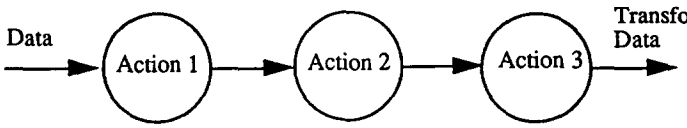

Figure 2. Actions of a module linked together by data flow.

are 0, then the minimum value of p is assumed to be 1. With this assumption, functional cohesion is an interval scale with values > 0 but < 1. Therefore,

$$F = K1/\text{maximum}(p, 1)$$

where $K1$ is the proportionality constant. Assuming $K1 = 1$,

$$F = 1/\text{maximum}(p, 1) \quad (1)$$

From the calculations of Appendix A, the functional cohesion of the two modules is as given below:

F(MODULE1) = 0.08

F(MODULE2) = 0.12

Not only is the functional cohesion of MODULE2 higher than that of MODULE1, but the functional cohesion of each of the submodules of MODULE2 is higher than or equal to the functional cohesion of MODULE1. This is to be expected, because each individual submodule of MODULE2 does a smaller amount of work than MODULE1 and is consequently more functionally cohesive. The results of this calculation are in keeping with the defined notion of functional cohesion. We also note that the calculations can distinguish MODULE1 from MODULE2 on the basis of functional cohesion.

3.1.2 Data flow cohesion. Data flow cohesion describes the interdependencies among the different statements of the module depending on the processing of data. Data flow exists when a piece of data, after undergoing some transformation in a statement, must undergo another transformation in a following statement. This is shown in Figure 2, where the actions (statements) of the module are linked together like a chain by the data that flows from one action to the next. As a simple example, if a module was constructed to read, calculate, and write the sum of two values, then the sequence of statements to accomplish that would have data flow cohesion.

In considering data flow cohesion between statements, the type of the statement and the position of the variable in the statement are taken into account. The general criterion for two statements to be linked by data flow cohesion is that a piece of data, after undergoing a transformation, be used in another transformation or action. For example, let us consider the pairs of adjacent statements as shown in Figure 3. Data flows from the lefthand side of the first statement to the righthand side of the second statement. In Figure 3, we have considered an assignment statement followed by three different types of statements; similarly, we must consider each of the n different types of statements of the language followed by any one of the n different types of statements. Each pair of statements will then be evaluated for data flow cohesion depending on the position of the identifiers used in the pair of statements. Therefore, we have an $(n \times n)$ data flow connectivity matrix in which the rows and columns are the same and show the functional position of a variable in a statement. The intersection of a row and column indicates that the row statement with a variable in a given functional position is followed by the column statement with the same variable. We mark the intersection of those statements that have data flow cohesion.

In Figure 3, we have considered adjacent pairs of statements, but a case for data flow cohesion can be made when the pairs of statements linked by data flow are separated by two or more intervening statements. In general, let the pair of statements be separated by j number of statements; then j takes on values from 1 to $(s - 1)$. The data flow bond weakens as it is stretched to cover a bigger span. Therefore, we assume that the value of data flow cohesion decreases as the value of j increases. As a result of our many hand calculations, we have found that the increase in data flow cohesion due to the contribution of those data-cohesive statements that are separated by three or more statements is marginal. Therefore, we have limited our calculation of data flow cohesion to $j \leq 3$ or $j \leq (s - 1)$, whichever is smaller.

Data flow cohesion in a module increases as the

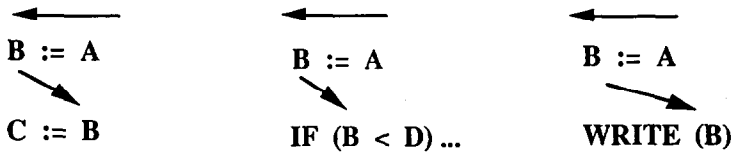

Figure 3. Example of statements linked by data flow cohesion. In each pair of statements, the variable B, after being on the lefthand side in the first statement, is used again on the righthand side. The flow of data is marked by arrows.

number of statements connected by data flow increases. The ratio of such statements to the total number of statements is used as a measure of the data flow cohesion of a module. Based on the rationale given above, we have the following model of data flow cohesion. Let

D = data flow cohesion in a module
s = total number of statements in the module
j = number of statements separating pairs of statements being evaluated for data flow cohesion. When the pair of statements are adjacent, then $j = 1$. The value of j goes from 1 to $(s - 1)$.
d_j = number of pairs of statements that have data flow cohesion when $j = 1, 2, \ldots (s - 1)$.

Now

$$D \alpha d_j$$

$$D \alpha \frac{1}{j}$$

Therefore, for a single variable, data flow cohesion normalized over the total lines of code s is given by

$$D = K2/s \sum_{j=1}^{j=(s-1)} d_j/j$$

where $K2$ is a constant. Assuming $K2$ to be 1, data flow cohesion

$$D = 1/s \sum_{j=1}^{j=(s-1)} d_j/j$$

The same data flow considerations have to be given for each variable in the module, that is, the calculations shown above have to be repeated for each variable in the module. We now make the simple assumption that the contribution of each variable toward the module data flow cohesion is additive. Therefore, summing up the data flow cohesion contribution of each variable, if v = total number of variables in the module, then

$$D = 1/s \sum_{i=1}^{i=v} \sum_{j=1}^{j=(s-1)} d_{ij}/j \quad (2)$$

In the example worked out in Appendix A, we have limited ourselves to $j = 1, 2, 3$, because the value of data flow cohesion drops off sharply as j increases. After more empirical work, we may be in a position to recommend a value of $j < (s - 1)$. For the modules in the example, we have

D(MODULE1) = 0.64

D(MODULE2) = 0.71

The results are in keeping with the expectation that data flow cohesion would be higher in MODULE2 than in MODULE1. In MODULE1, all the data and the actions to be performed on this data are in one unit, so there is greater likelihood of interleaved transformations as opposed to sequential transformations on the same data. The statements in the submodules of MODULE2 are partitioned into smaller units of code according to the data manipulated; after some statement has manipulated the data, it "flows" on to the next statement in the same module. This establishes strong data flow ties. For example, the statements that manipulate data to find the minimum value are concentrated in the subprogram FIND_MIN, and after finding the minimum value, the data are passed on to a separate subprogram. Subprogram EXCHANGE is an exception to this because it does not perform sequential transformations on one particular data item.

3.1.3 Action-bundling cohesion. Action-bundling cohesion takes place as a result of the collection of several actions to be performed on a single piece of data (Figure 4). Suppose the task was to read, up-

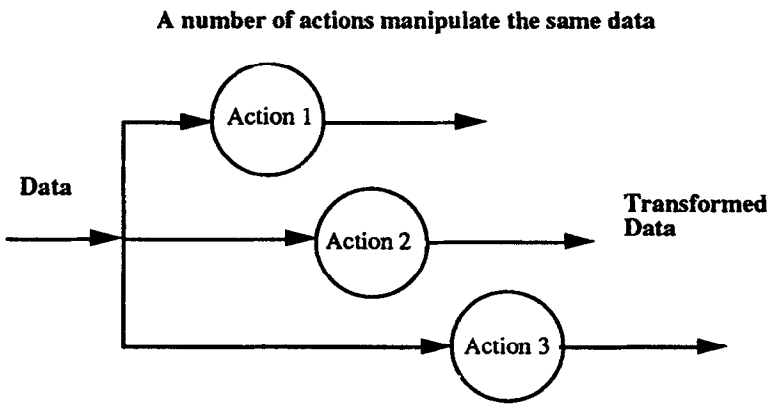

Figure 4. Action-bundling cohesion.

date, and write out an array or a file; if all these actions were gathered together in one unit of code, then this unit of code would exhibit action-bundling cohesion.

The general rule for determining action bundling between two statements is to examine if the same data item is being manipulated in the two statements. For example, Figure 5 illustrates three pairs of adjacent statements that exhibit action-bundling cohesion. The first statement of each pair is an assignment statement, followed by three of the many different types of statements of the language. Action-bundling cohesion between the statements of each pair exists because in each pair the variable C is used without being transformed, that is, the actions of the two statements are "bundled" because of the variable C.

The algorithm used for determining action-bundling cohesion is the same as that used in determining data flow cohesion except that the rules for determining cohesion between statements are different. This merely means a change in the connectivity matrix used for determining the relevant cohesive pairs of statements. We have the following model of action-bundling cohesion. Let

A = action-bundling cohesion in a module
s = total number of statements in the module
j = number of statements separating pairs of statements being evaluated for action-bundling cohesion. When the pair of statements are adjacent, then $j = 1$. The value of j goes from 1 to $(s - 1)$.
t_j = number of pairs of statements that have action-bundling cohesion when $j = 1, 2, \ldots (s - 1)$.

Now

$$A \alpha t_j$$

$$A \alpha \frac{1}{j}$$

Normalizing over s, the total number of lines of code, we have, for a single variable, the action-bundling cohesion

$$A = K3/s \sum_{j=1}^{j=(s-1)} t_j/j$$

where $K3$ is a constant. Assuming $K3$ to be 1, action-bundling cohesion

$$A = 1/s \sum_{j=1}^{j=(s-1)} t_j/j$$

The same action-bundling considerations must be given for each variable in the module. Using the same argument as for summing up the data flow contribution of each variable, we sum the action-bundling cohesion contribution of each of the variables in the module. Therefore, if v = total number of variables in the module, then

$$A = 1/s \sum_{i=1}^{i=v} \sum_{j=1}^{j=(s-1)} t_{ij}/j \qquad (3)$$

With reference to the example worked out in Appendix A,

A(MODULE1) = 1.01

A(MODULE2) = 1.43

Although two of the subprograms of MODULE2 have lower action-bundling cohesion than MODULE1, the total action-bundling cohesion of MODULE2 is higher than that of MODULE1. This is primarily due to the main program of MODULE2 (procedure A in Table A1), which has the highest value of action-bundling cohesion. This is an expected result because this procedure calls the other subprograms and coordinates their activity; it parcels out all the data and receives all the results, so that its actions are closely related. Procedure D of MODULE2 also exhibits high action-bundling cohesion because all of the procedure's actions are performed on two data items, causing the actions to be closely related.

Once again, on the basis of action-bundling cohesion, MODULE1 can be distinguished from MODULE2, signifying that the developed formula is sensitive to varying implementations.

3.1.4 Logical bundling cohesion. It is common to have a "logical block" of program statements executed based on a Boolean condition. Such a condition can be part of an IF-THEN statement or the terminating condition of a WHILE, REPEAT, or FOR loop. The block structure of this type is held together by the common Boolean condition that forms the basis for the execution of the block. The

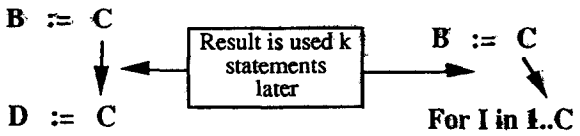

Figure 5. Example of statements exhibiting action-bundling cohesion. In each pair of statements, the variable C is used in both the statements on the righthand side.

cohesion exhibited by the statements of the block is categorized as logical bundling cohesion because the statements "bundle" together following a "logical" decision.

Logical decisions are often nested, that is, logical blocks of statements are contained within logical blocks. In such cases, the statements in the core block are logically more cohesive because they share a larger number of logical decisions than those statements that are in the outer blocks. As the number of statements increases, the logical binding weakens. An analogy is that of a bundle of sticks tied together by a string. The bundle becomes more secure (more cohesive) as the number of loops around the bundle increases and less secure as the number of sticks in the bundle increases.

Let

L = logical bundling cohesion
w = nesting depth of block in which a block of statements is located
t = number of statements in the block under consideration
k = total number of blocks in the module

Then for a block i

$L_i \alpha w_i$

$L_i \alpha 1/t_i$

Therefore, $L_i = (K4 * w_i)/t_i$ where $K4$ is a constant. Assuming $K4 = 1$, we have for the ith block $L_i = w_i/t_i$. To find the logical bundling cohesion of the module, we follow the same reasoning except that we take the average depth of all the statement blocks in the module and divide by the average number of statements in a block.

$$\text{Average nesting depth} = 1/k \sum_{i=1}^{i=k} w_i$$

$$\text{Average number of statements in a block} = 1/k \sum_{i=1}^{i=k} t_i$$

Therefore

$$L = \sum_{i=1}^{i=k} w_i \Big/ \sum_{i=1}^{i=k} t_i \qquad (4)$$

With reference to the example worked out in Appendix A,

L(MODULE1) = 0.54

L(MODULE2) = 0.40

It is not surprising that the logical bundling cohesion of MODULE1 is higher than that of MODULE2; the smaller units of code of MODULE2 are likely to have fewer loops than MODULE1, in which the entire logic is represented in one unit.

3.1.5 Cohesion summary and discussion. Our investigation shows that the four types of cohesion are independent of each other. In adding up these four types of cohesion to obtain the total cohesion, we have chosen to weigh them equally because there is no evidence for any other weighting scheme. However, in the literature, various authors have rank ordered the categories of cohesion in the order that they have been presented here. If in our follow-up empirical work we find that an unequal weighting scheme is warranted, we will consider it at that time.

The total cohesion of a module = $F + D + A + L$ (the sum of the four types of cohesion). From the example in Appendix A,

Cohesion (MODULE1) = 2.27

Cohesion (MODULE2) = 2.65

We make the following observations regarding these final results:

- The calculated value of the total cohesion supports the expected qualitative notion of cohesion. MODULE2, with its small components, is qualitatively judged to be more maintainable, easier to reuse, and less error prone than MODULE1. On the basis of this metric, it is also possible to distinguish MODULE1 from MODULE2. Therefore, it appears that the metrics developed are sufficiently sensitive to implementation.
- More empirical work needs to be done to be able to draw some general conclusions regarding software using cohesion as a metric.
- The cohesion metrics can be used for comparative purposes to pick between two modules that implement the same function. The module with the higher cohesion index will indicate a better design from the cohesion point of view.

3.2 Development of a Theoretical Model of Coupling

When modules are linked together, the four categories of coupling exhibited and their causes are as shown below:

- Data flow coupling caused by the parameters at the interface
- Control flow coupling also caused by the parameters at the interface
- Global coupling caused by global variables
- Environmental coupling caused by calling and being called by other modules

Any parameter or shared variable can be used as data or as a "control variable"; the latter is defined as a variable that determines operational execution, such as when used in a LOOP or IF-THEN type of statement. A variable used as a diagnostic or error recovery variable is also classified as a control variable. As a start, our premise is that control variables result in twice as tight a coupling as an equal number of data variables. At present, we also assume that the extent of coupling exercised by parameters and global variables is the same. However, we agree with the qualitative notion that global variables cause more insidious coupling than parameters. We will be in a better position to make comments on this issue after some empirical work to evaluate a large number of modules for coupling.

We would like the coupling between modules to be as low as possible, because this lowers the probability that a change in a module will cascade to the module coupled to it. In the development of our formula for coupling, low coupling will be indicated by high numbers because increasing numbers are generally associated with increasing "goodness." This association is also in keeping with our calculations of module cohesion, in which high numbers indicate high cohesion, which is "good." We have the following quantitative model for the four types of coupling. Let

C = total module coupling

For data and control flow coupling:

$i1$ = in data parameters
$i2$ = in control parameters
$u1$ = out data parameters
$u2$ = out control parameters

For global coupling:

$g1$ = number of global variables used as data
$g2$ = number of global variables used as control

For environmental coupling:

w = number of modules called
r = number of modules calling the module under consideration

Now let $m = i1 + q_6 i2 + u1 + q_7 u2 + g1 + q_8 g2 + w + r$, where q_6, q_7, and q_8 are constants, and, as a first heuristic estimate, are assumed to be 2 in our calculations. The minimum value of r is 1, because every module executed, including a main program, must be called by some other program.

$$C \alpha \frac{1}{m}$$

Therefore, $C = K5/m$ where $K5$ is a constant. Assuming $K5 = 1$, we have

$$C = 1/m \tag{5}$$

From the example worked out in Appendix A, we have the following results:

C(MODULE1) = 0.10

C(MODULE2) = 0.15

We note that, although MODULE2 has four interfaces, the coupling of MODULE2 is better than the coupling of MODULE1. It may have been expected that because of the increased number of interfaces, the average coupling would be worse than the coupling in the single interface of MODULE1. The coupling for each individual module of MODULE2 is also better than that of MODULE1. This may be related to the fact that each of the small modules of MODULE2 has a higher functional cohesion and requires less interaction with the outside world. This result can be an input to answer the often-asked question, "How many modules are optimal to implement a given functionality?" The answer is, "The coupling of each of the modules should be better than or equal to that of the single module."

The coupling values obtained for MODULE1 and MODULE2 are also in agreement with the qualitative notion of good coupling, in which four smaller modules are expected to be easier to understand and modify than a single monolithic module. The coupling formula is also sensitive enough to distinguish between the two implementations.

We are by no means suggesting that C & C be the only metric considered in evaluating a software module; software has many facets and C & C is but one of them. However, we feel that C & C is an important facet because so many of the higher level software quality factors depend on it.

4. FUTURE WORK

In this article, we have quantified C & C, and the preliminary calculations appear to support our analysis. On the basis of the hand-calculated samples, no conclusions can be reached. The remainder of our project will address this problem. We are nearing completion of a tool that will do the C & C calculations for Ada programs. We will then verify the formulas and build a C & C scale to be used for evaluating programs. After the completion of our project, there are several avenues open for research to answer the following types of questions:

- What is the correlation between the number of errors found in a module and its C & C values?

There is also a need to establish such correlations between C & C and a number of management and other software quality metrics, such as productivity and complexity.

- Are there optimal values for C & C, and are there some general guidelines for achieving them?
- Can the C & C concept be used to evaluate software architecture?
- How sensitive are the formulas to the various factors that enter into the C & C calculations?
- How applicable are the C & C metrics to "methods" of object-oriented programs?

REFERENCES

Bowen, T. P., Post, J. V., Tsai, J., Presson, P. E., Schmidt, R. L., Software Quality Measurement for Distributed Systems, Guidebook for Software Quality Measurement, RADC-TR-83-175, Vol. II, Final Technical Report, Rome Air Development Center, Air Force Systems Command, Griffis Air Force Base, NY, 1983.

Card, D. N., Church, V. E., and Agresti, W. W., An Empirical Study of Software Design Practices, *IEEE Trans. Software Eng.* SE-12, 264–271 (1986).

Henry, S., and Kafura, D., Software Structure Metrics Based on Information Flow, *IEEE Trans. Software Eng.* SE-7 (1981).

McCabe Associates, OO Tool Features New Metrics, *The Outlook* (Fall 1993).

Myers, G. J., *Software Reliability Principles and Practices,* John Wiley and Sons, New York, 1976.

Ott, L., and Thuss, J., The relationship between slices and module cohesion, in *IEEE 11th International Conference on Software Engineering,* 1989, pp. 198–204.

Yourdon, E., and Constantine, L. L., *Structured Design: Fundamentals of a Discipline of Computer Program and System Design,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1979.

Zage, W. M., Zage, D. M., Bhargava, M., and Gaumer, D. J., Design and code metrics through a DIANA-based tool, in *11th Ada-Europe International Conference,* 1992, pp. 60–71.

APPENDIX A: SAMPLE CALCULATION OF C & C VALUES

This Appendix contains two implementations of a "selection sort" algorithm, which sorts integers in ascending or descending order. The first implementation, titled MODULE1, consists of a single monolithic module, whereas the second implementation, titled MODULE2, consists of four subprograms but uses the same algorithm to do the sorting. The quantitative metrics for C & C that have been developed are applied to the two implementations to show that (1) the metrics are in keeping with the qualitative evaluation of C & C in the two modules, and (2) on the basis of the calculated C & C metrics, the two modules can be distinguished from each other.

MODULE1—Implementation #1–a single procedure implements a "selection sort."

MODULE1

```
package sort1 is
type array_type is array (1..1000) of integer;
```

Specifications of the procedure

```
procedure sort1 (n : in integer;
                 to_be_sorted: in out array_type;
                 a_or_d: in character);
end sort1;
```

The code in the body is evaluated

```
package body sort1 is

procedure sort1 (n : in integer;
                 to_be_sorted: in out array_type;
                 a_or_d: in character) is
location, temp : integer;
begin
  for start in 1..n loop
  location := start;

  —loop to get min or max each time

  for i in (start + 1)..n loop
    if a_or_d = 'd' then
      if to_be_sorted(i) > to_be_sorted(location) then
        location := i;
      end if;
    elsif to_be_sorted(i) < to_be_sorted(location) then
      location := i;
    end if;
  end loop;
```

The exchange

```
    temp := to_be_sorted(start);
    to_be_sorted(start) := to_be_sorted(location);
    to_be_sorted(location) := temp;
  end loop;
end sort1;
end sort1;
```

MODULE2—Implementation #2—four procedures implement a "selection sort."

MODULE2

```
package sort2 is
type array_type is array (1..1000) of integer;
```

Specifications of the procedures

```
procedure sort2 (n : in integer;
                 to_be_sorted: in out array_type;
                 a_or_d: in character);
procedure find_max (n : in integer;
                    to_be_sorted: in out array_type;
                    location : in out integer);
procedure find_min (n, start : in integer;
```

Table A1. Calculation of the Different Types of Cohesion for the Two Implementations

Type of Cohesion	Single Procedure in Implementation 1	Four Procedures in Implementation 2				Average Cohesion Values in Implementation 2
		A	B	C	D	
Functional	0.08	0.08	0.11	0.11	0.17	0.12
Data flow	0.64	1.17	0.75	0.75	0.17	0.71
Action bundling	1.01	2.87	0.50	0.50	1.85	1.43
Logical bundling	0.54	0.38	0.60	0.60	0.0	0.40
Total cohesion*	2.27	4.50	1.96	1.96	2.19	2.66

*The different types of cohesion are equally weighted.

Table A2. Calculation of Coupling for the Two Implementations

Coupling	Single Procedure in Implementation 1	Four Procedures in Implementation 2				Average Coupling Values in Implementation 2
		A	B	C	D	
C	0.10	0.13	0.14	0.14	0.20	0.15

```
                    to_be_sorted: in out array_type;
                    location : in out integer);
procedure exchange (start : in integer;
                    to_be_sorted: in out array_type;
                    location :in out integer);
end sort2;
package body sort2 is
```

Only body code evaluated

Procedure find_max body ('Proc. B' in Tables A1 and A2)

```
procedure find_max (n, start : in integer;
                    to_be_sorted: in array_type;
                    location : in out integer); is
begin
  location := start;
  for i in start + 1..n loop
    if to_be_sorted(i) > to_be_sorted (location) then
      location := i;
    end if;
  end loop;
end find_max;
```

Procedure find_min body ('Proc. C' in Tables A1 and A2)

```
procedure find_min (n, start : in integer;
                    to_be_sorted: in array_type;
                    location : in out integer) is
begin
  location := start;
  for i in start + 1..n loop
    if to_be_sorted(i) < to_be_sorted (location) then
      location := i;
    end if;
  end loop;
end find_min;
```

Procedure exchange body ('Proc. D' in Tables A1 and A2)

```
procedure exchange (start : in integer;
                    to_be_sorted: in out array_type;
                    location : in integer) is
temp : integer;
begin
  temp := to_be_sorted(start);
  to_be_sorted(start) := to_be_sorted(location);
  to_be_sorted(location) := temp;
end exchange;
```

Procedure sort2 body ('Proc. A' in Tables A1 and A2)

```
procedure sort2 (n : in integer;
                 to_be_sorted: in out array_type;
                 a_or_d: in character) is
location : integer;
begin
  for start in 1..n loop
    if a_or_d ='d' then
      find_max(n, start, to_be_sorted, location);
    else
      find_min(n, start, to_be_sorted, location);
    end if;
    exchange (start, to_be_sorted, location);
  end loop;
end sort2;
end sort2;
```