Relating computer program maintainability to software measures

by A. R. FEUER
Bell Telephone Laboratories
Piscataway, New Jersey

and

E. B. FOWLKES
Bell Telephone Laboratories
Murray Hill, New Jersey

MOTIVATION

It is no longer a surprise that program maintenance dominates the total cost of a large software system over its lifetime. In response to these costs, the emphasis in program design has largely shifted from the time and space issues of machine efficiency to issues of clear and flexible program structures that can be easily maintained.

The goal of this project is to identify measurable program properties that influence maintainability. More precisely, we examine the effect of various program characteristics on the subsequent frequency and magnitude of program errors.

PROGRAM MAINTAINABILITY

This paper presents a study of 123 PL/I program modules from a business data processing application. The modules fall roughly into three categories: high-level control, numerical and data base management. The maintenance record of each module was followed for approximately one year.

We characterize the maintenance performance of a program module by two values—the total maintenance time spent on the module and the number of maintenance changes made to the module. We considered a module in “maintenance” from the time it left the development programmer and entered system testing. The maintenance performance data came from two sources:

1. Informal (and incomplete) time records, recorded by hand, that include time spent on and cause of maintenance activities.
2. A formal (and complete) maintenance activity data base, recorded automatically when a program is changed, that does not include time spent per activity.

In this paper we present an analysis based upon the informal time records, chiefly:

- The number of errors per program module.
- The total time spent repairing those errors.

PROGRAM MEASURES

Our starting hypothesis was that maintenance performance for a program depends upon:

- The complexity of the algorithm coded.
- The clarity of the coding.

(It is beyond the current scope of this work to consider influences on performance beyond the program itself, such as frequency of execution or instability of user requirements.)

Based upon previous work and upon programming folklore we compiled a set of program properties for which the qualifiers complex and clear might have meaning. The list includes:

- Complexity of program control flow.
- Clarity of program control structure.
- Clarity of program data usage.

The next step was to develop measures to quantify these properties. Two primary criteria were adhered to in developing the measures:

1. Each measure should be largely language-independent.
2. Each measure should be noncoercible.

A measure is language-independent if:

- It can be meaningfully and consistently applied to programs written in several programming languages.
- The ordering the measure assigns to a set of algorithms...
remains roughly constant when the algorithms are coded across languages of similar power.

One language-dependent measure that has been used to characterize programs is a count of the GOTO statements appearing in a module.\textsuperscript{1,13} Counting GOTO statements is language-dependent because the practical meaning of GOTO changes as control structures change across languages. One language may require a GOTO to implement an innate control structure of another language.

A measure is noncoercible if it in fact measures the underlying program property of interest. Identifying program characteristics that may coincidently vary with a fundamental property tells us little about the property.

Coercible measures that have been used to characterize programs include calculating the mean length of variable names as a measure of name meaningfulness,\textsuperscript{13} and counting the number and length of comment blocks as an indicator of program documentation.\textsuperscript{13,15} We have called these measures coercible since they can (quite easily) be influenced without any corresponding effect on the property they seek to measure.

**Measuring the complexity of program control flow**

Program control flow was characterized by measurements taken from the control flow graph for each program. A control flow graph is a directed graph with nodes corresponding to the simple clauses of a program and arcs indicating the sequence of control. The graph is connected, unreachable clauses are ignored. And the graph is single-entrance-single-exit—an entry node points to all entry points and an exit node is pointed to by all exit points (Figure 1).

Two simple graph measures were included for study:

- A count of the nodes in the graph.
- The ratio of binary decisions embedded in the graph to total nodes. (The number of binary decisions at a node is one less than the number of outgoing arcs from the node.)

Two other graph measures, sensitive to the configuration of the graph, were also studied:

- A count of the possible paths through the graph.
- The mean number of decisions per path through the graph.

We define a path through a graph to be a sequence of nodes, from the entrance node to the exit, such that no cycle is repeated. Even with this restriction the number of paths through a typical graph can be very great. For the more complex programs the total path count was estimated by a lower bound, and the mean path length was estimated from a sampling of the paths.

**Measuring the clarity of program control structure**

Structured programming has evolved as a guide to writing more easily understood programs. It is generally agreed that well structured code is composed from blocks, each with a single entrance and a single exit. In the strictest sense, the lowest level block can be taken to be a simple clause in a program, a node in the control flow graph. A simple reduction rule exists to collapse graphs of strictly well structured programs to a single node:

1. Choose a node \( n \) that has at most one incoming arc and at most one outgoing arc.
2. Replace \( n \) by an arc, preserving direction, that connects \( n \)’s neighbors (\( m \) is a neighbor of \( n \) if there is an arc between \( m \) and \( n \)).
3. Remove any redundant arcs:
   - Two arcs in the same direction between the same nodes is a redundancy.
   - An arc from a node to itself is a redundancy.

The three steps are repeatedly applied until they are no longer applicable (Figure 2).

After applying the reduction rule to a graph, some subgraph will remain. The subgraph will be a single node for
**REDUCTION RULE**

**STEP A:** COLLAPSE ANY NODE THAT HAS AT MOST ONE INCOMING ARC AND AT MOST ONE OUTGOING ARC

**STEP B:** REMOVE ANY EXTRANEOUS ARCS

REPEATEDLY APPLY EACH STEP UNTIL NEITHER IS APPLICABLE

---

strictly well structured programs; the subgraph will be more complex for other programs. We characterize the degree to which a graph can be reduced by the ratio of binary decisions embedded in the subgraph to binary decisions embedded in the original graph.

**Measuring the clarity of program data usage**

One characteristic of data usage is the degree to which variable use has been localized. Measuring the span of a variable is one indicator of localization. A span of a variable is one plus the number of intervening statement clauses between two successive references to the variable. The average span of a variable is the sum of all spans for the variable divided by the number of spans. The variable span for a program is the sum of average variable spans for all the variables within the program (Figure 3).

**Software science measures**

The field of software science is attempting to establish meaningful relationships among the primitive components of algorithms—operators and operands (Figure 4). Work by others has indicted that some of the software science variables suggested by Halstead may reliably characterize the overall complexity of a program.

In our analysis we studied six software science variables.

All can be derived from counts of: Unique operators \(n_1\), unique operands \(n_2\), instances of operators \(N_1\), and instances of operands \(N_2\).

- **Length**—\(N_1 + N_2\). Length is a program size measure of finer granularity than a count of program statements or clauses. A virtue of length is that it is largely insensitive to horizontal (deeply-nested expressions) versus vertical (simple expressions) programming style.

- **Expected length**—\((n_1 \log_2 n_1)(n_2 \log_2 n_2)\). Empirical studies have shown expected length and length correlate highly for published, hence presumably well polished, programs. We used the calculation \((length - expected length)/length\) to indicate the agreement between length and expected length.

- **Volume**—\(\log(n_1 + n_2)\). Volume is an estimate of the minimum number of bits needed to represent the executable statements of a program.

- **Level**—\((2/n_1)(n_2/N_2)\). Level is a measure of the match between the operations performed by a program and the primitive operators and functions used by the program. In a program at the highest level the operations performed is implemented by a single operator.

- **Effort**—\(volume/level\). Intuitively, effort is a function of the quantity of information represented by a program and the power of the statements with which the information is encoded. Effort rises with increasing information content and with decreasing statement power.

---

**DATA ANALYSIS**

An analysis of 12 carriers, or independent variables, and two responses, or dependent variables, is presented here. Figure 5 summarizes the range of values for each of the carriers.

The carriers were first studied singly and in pairs. Normal quantile-quantile plots for each carrier revealed that most had non-normal, asymmetric distributions. Transformations (usually the natural logarithm) were carried out which rendered the distributions approximately normal and symmetric. We used the carrier node count as a simple measure of module size. Scatter plots of the eleven other carriers versus node count revealed that seven were highly correlated with module size and four were not (Figure 5).

In order to determine whether there was a significant stratification of the modules according to size, the hierarchical clustering algorithm of Johnson, using Manhattan distance and the complete linkage method, was used for the 123 modules according to the eight standardized size-related carriers. A dendogram of the result is shown in Figure 6. The figure shows strong clustering. If the tree is cut at a distance of 9.0, there appear to be six clusters.

An attempt was next made to relate the response time to repair errors to module size. Program maintenance data showed that there were a total of 124 errors on 45 modules. The 124 errors were classified into six groups according to the assignment of the modules with errors to the six size
MEAN SPAN FOR VARIABLE

\[
\frac{\text{LAST REFERENCE} - \text{FIRST REFERENCE}}{\text{NUMBER OF REFERENCES} - 1}
\]

SPAN FOR A PROGRAM = SUM OF MEAN SPANS FOR ALL VARIABLES

for this graph:

- yes: \[
\frac{4-2}{1} = 2
\]

- no: \[
\frac{6-2}{1} = 4
\]

- vote: \[
\frac{5-3}{1} = 2
\]

- abstain: \[
\frac{9-9}{1} = 0^*
\]

*By convention a variable referenced only once has a span of zero

Figure 3—Variable Span.
DO  WHILE  yes < no ;

IF  vote = 'y'  THEN  yes = yes + 1 ;
ELSE  IF  vote = 'n'  THEN  no = no + 1 ;
ELSE  abstain = abstain + 1 ;

END ;

<table>
<thead>
<tr>
<th>OPERATORS</th>
<th>INSTANCES</th>
<th>OPERANDS</th>
<th>INSTANCES</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO-END</td>
<td>1</td>
<td>yes</td>
<td>3</td>
</tr>
<tr>
<td>WHILE</td>
<td>1</td>
<td>no</td>
<td>3</td>
</tr>
<tr>
<td>&lt;</td>
<td>1</td>
<td>vote</td>
<td>2</td>
</tr>
<tr>
<td>;</td>
<td>5</td>
<td>'y'</td>
<td>1</td>
</tr>
<tr>
<td>IF-THEN</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>= (equality)</td>
<td>2</td>
<td>'n'</td>
<td>1</td>
</tr>
<tr>
<td>= (assignment)</td>
<td>3</td>
<td>abstain</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>3</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>ELSE</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

9  20

Figure 4—Software science primitives: operators and operands.

clusters. Average node count was calculated for each group, and the times to repair (in quarter-hour units) were plotted versus average node count (Figure 7). (A random normal deviate from \( N(0, e) \) was added to each of the \( x \) and \( y \) coordinates of every point so that the density of points in each group could be seen.) A striking dependency can be seen; the spread of time to repair increases with node count.

Actually, since at node counts of 50, 100 and 220 there appear to be one or two points quite separated from the rest (the observation of 15 hours at node count 220 is not a
mistake), it is not unreasonable to hypothesize two mechanisms in operation here. Figure 7 suggests a model for such a phenomenon superimposed on the data. The model says that there are two types of errors that occur, easy-to-repair and difficult-to-repair. The easy-to-repair errors occur with large probability and the difficult-to-repair with small probability. Points that might have come from the difficult-to-repair distribution have been circled in Figure 7. An exponential fit was estimated for these points, but this estimate may be very unreliable because of the small number of points. Nevertheless, the model does provide an interesting framework to study the dependency of time to repair on module size.

An important question addressed next in the analysis was whether there might be some residual or second-order effect of the seven size-related carriers other than node count on the responses. Size-adjusted variables were created for the seven carriers by performing simple linear regressions of the log carrier on log node count. Simple linear regression appeared to be adequate for each case. The seven sets of residuals about regression formed new adjusted carriers that were relatively free of node count.

A hierarchical clustering of the 123 modules according to the standardized residual carriers was conducted using Manhattan distance and the complete linkage method. A dendogram is shown in Figure 8. The dendogram appears to show reasonably strong clustering. If the tree is cut at a distance equal to 8.5, 13 clusters result. As a check, clusterings using different methods and different metrics were tried. It was seen that modules combined in the tree at different heights and in slightly different orders, but the basic grouping remained stable. In the analysis that follows, only the response error density will be considered. Error
density is defined as the total number of errors for a group of modules divided by the number of modules in the group which contain errors.

The seven residual carriers were next investigated for multi-collinearity. A principal components analysis of the correlation matrix of these carriers revealed that only three of the seven carriers were not collinear since the first three principal axes accounted for 99% of the total variability. Before proceeding further, the multi-collinearity of the carriers was removed. Weighted (by reciprocal variance) averages of residual length and volume and residual path count and average path length were taken, thereby reducing the number of carriers to five. Next, the first and fourth carriers of the five-carrier set were eliminated. These carriers had the lowest simple correlation with the response error density. This leaves as carriers residual level, residual expected length, and a weighted average of residual path count and residual average path length. A principal components analysis for these three variables showed no singularity.

One approach to relating the error density to the carriers is to group the modules with errors according to the 13 clusters found previously, calculate average error density for each group, and regress these averages on representative values of the adjusted carriers for each group. The representative value for each group was taken to be the centroid of the final three adjusted carriers. Since each group has a different number of modules with errors, the average responses are based on differing numbers of items and weighted regression is appropriate. In carrying out this grouping-averaging process, regression analysis was used in its classical sense of estimating the mean of the conditional distribution of the response, conditional on values of the carriers. There was a large amount of inherent variability in the original data, in part because the data was not collected as part of a designed experiment. The averaging carried out above reduced this variability and allowed the marginal effects of an adjusted carrier to be seen more clearly.

Next we formed a regression model of error density and our three-carrier set. However, it might be possible for the carriers to affect the response interactively. To account for this, we included cross-product terms in the regression model. When this is done, six carriers result (Figure 9).

In order to select usable models containing combinations of the six carriers, the $C_p$ statistic was calculated for each possible regression containing the carriers. These statistics were plotted against $p$, the number of terms in each regres-
sion, and are shown in Figure 10. Here the maximum \( p \)
equals seven because a constant term was added in each
regression. The line \( C_p = p \) was added in order to indicate
good results. Any combination of variables that lie on this
line are good models for the regression. A dashed line was
drawn to show reasonable models for the data. We consid­
ered a model reasonable if it had a mean square error at
least as small as the model with all the terms in it. All the
models include variable 1, residual level, which by itself
almost provides a reasonable model for the data. Notice that
some of the candidate models include interaction terms with­
out containing the corresponding main effects; such models
were not considered acceptable. One of the best and most
plausible models comprises the terms 1, 2, 3, and 4. These
are residual level, residual expected length, their interaction
and the residual path carrier. The result of this analysis
shows that there is an apparent residual effect of three of
the seven original carriers.

The regressions for models that were considered reason­
able were all significant at the \( \alpha = .01 \) level but are not strong
enough for accurate prediction (the multiple \( R^2 \) is less than
.40 for regressions performed on the raw, unaveraged data).
However, they do provide ideas for future experiments and
food for thought.

SUMMARY OF FINDINGS

Both the data collection and analysis are at an early stage;
yet a few results are evident.

1. Most of the variables we studied have a large size
   component.
2. Module size, by itself, appears to be a good indicator
   of maintenance performance for the module, though
   we have not studied the tradeoff of many small modules
   versus fewer large modules.
3. When adjusted for size, level appears to be a fair in­
   dicator as to how a group of modules will perform.

While we do not expect to predict the maintenance per­
formance of a single module based upon a static analysis as
presented here, we do expect to find program properties
characteristic of poor performance. Such properties would

---

1. Residual level
2. Residual expected length
3. Weighted average of residual path count and residual mean path length
4. Cross product of carriers 1 and 2
5. Cross product of carriers 1 and 3
6. Cross product of carriers 2 and 3

---

Figure 8—Clustering of standardized residuals from regression on node count using manhattan distance.

Figure 9—Reduced Carrier Net
lead us to firmer ground for establishing guidelines of good programming practice.

ACKNOWLEDGMENTS

We are indebted to J. C. Gear and D. E. Pinkston for their assistance with this work.

REFERENCES