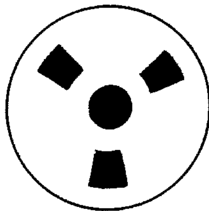
MODELING AND MEASURING SOFTWARE DATA DEPENDENCY COMPLEXITY

by
James Michael Bieman
and
William R. Edwards

June 1985

TR #85-18

DEPARTMENT OF COMPUTER SCIENCE
IOWA STATE UNIVERSITY/AMES, IOWA 50011



MODELING AND MEASURING SOFTWARE DATA DEPENDENCY COMPLEXITY

by
James Michael Bieman
and
William R. Edwards

June 1985

TR #85-18

Modeling and Measuring Software Data Dependency Complexity

James Michael Bieman
and
William R. Edwards

Abstract

We have developed a model of the data dependencies within a program - the data dependency graph (DDG). The DDG is a directed graph: nodes represent variable definitions and edges represent possible data dependencies. A data dependency exists when the value stored in a variable may be directly affected by the value of another variable or an earlier value of the same variable. For example, after an assignment Y := f(X) is executed, the new value of Y depends on the current live definitions of X and the control variables that determine whether the assignment is executed. We present algorithms to generate DDG's from arbitrary programs written in procedural languages, use the DDG to develop measures of software complexity, and report the results of an initial validation effort. The complexity measures proposed are the cyclomatic complexity and the rooted spanning tree complexity of a DDG.

Index Terms

data dependency, software structure, software complexity measurement, programmer/program interface, software reliability, software engineering tools

I. Introduction

The objective of this research effort is to construct a data dependency model of program complexity that is viewed from the programmers perspective and is automatable. Most of the research directed towards measuring the complexity of the programmer/program interface has centered on the measurement of flow of control complexity. McCabe's presentation of cyclomatic complexity is an example of the modeling and measurement of flow of control complexity [13].

Weiser found that programmers seem to work backwards when debugging, examining only instructions that affect the variables in error [19]. Weiser's results demonstrate the importance of data dependencies and indicate that a measure of data dependency complexity would be a useful tool for the development of reliable and maintainable software.

A model of the data dependencies within a program is required before measures of data dependency complexity can be derived. The model developed in this research effort is the data dependency graph (DDG) [2].

The DDG is a directed graph, with each node representing a variable definition and each edge representing a possible data dependency. A variable definition is considered live at a specified statement if it is possible for the value placed in the variable at the definition to be referenced at the statement. When constructing a DDG, live definitions are collected from alternate pathways in the program to determine possible dependencies of a given variable definition. The algorithms for generating a DDG recognize variable definitions and flow of control constructs, and use knowledge of the live definitions and the variable definitions that determine control flow.

The algorithms can generate DDG's from programs with unstructured constructs such as GOTO statements, multiple entry/exit points, and global variables. Good programming practices should preclude the use of unstructured code. However, addressing the problems involved in constructing a DDG from unstructured code is justified. The model can be most generally applied if there are minimal restrictions on code structure. Furthermore, the metrics based on the DDG may help quantify and support subjective judgments concerning the merits of structured programming. Also, there are pragmatic reasons for including unstructured programs in the model. A large amount of software currently in use is not structured or is written in languages such as FORTRAN 66 where structured constructs must be simulated. The model can be applied to existing unstructured code since the issues associated with unstructured programs are addressed.

Certain general measures of DDG complexity are proposed as software complexity measures. These measures are the cyclomatic complexity and rooted spanning tree complexity of the DDG.

The DDG models program complexity from the programmers point of view, and is an abstraction of the complexity that the programmer must deal with. The programmer views a program as a static representation of a dynamic entity, a program in execution. The program specifies more than what will occur during an individual execution, rather the program specifies the possible actions during all executions. Weiser defines a <u>slice</u> to be a program after the deletion of all statements that do not affect the value of a specified variable at a given statement [18]. A slice is one representation of the search space that the programmer must examine in detecting the source of an error and is created by using a subset of the data dependencies in a program segment. The DDG is a representation of all of the possible data dependencies

that may exist during any execution. The more complex the data dependencies in a program, the larger the search space that the programmer must contend with when trying to comprehend the program for the purpose of enhancement or when searching for the source of a bug.

II. The Data Dependency Graph

The DDG <N,E> is a directed graph representation of the possible data dependencies in a program. N is a set of nodes where each node denotes a variable definition and E is a set of edges in NxN. If (a,b) is a member of E then the definition represented by b is possibly dependent on the definition represented by a.

A variable definition is a statement that may modify the value of a variable, such as assignment statements, procedure calls, and input statements. Initialization, or the initial state of a variable even when undefined, is assumed to be a definition. For convenience, we label each node with a name identifying the variable that the node represents and a subscript. The subscripts distinguish between nodes representing different definitions of the same variable and are sequentially numbered based on the relative position of the definition in the source code.

DDG edges represent possible dependencies between definitions. Consider a statement S_n in program P of the form Y := f(A). S_n is a definition of variable Y and would be represented by a node Y_n in the DDG. The definition represented by Y_n is directly dependent on each definition of variable A that can reach S_n, and each of the dependencies is represented in the DDG by an edge of the form (A_x,Y_n). A definition can reach a statement if there is a path clear of redefinition in the control flow graph from the definition to the statement. Oviedo used similar notions to develop a measure of data flow

complexity [14]; however, he did not represent the complexity of the entire program with a graph.

A. Dependency Sources.

A variable definition may depend on the value of several variable definitions for a variety of reasons. A dependency can be (1) a direct dependency that can be determined by evaluating an individual statement in context or (2) a control dependency that results from the definitions that determine the flow of control.

Direct dependencies can be determined by examining an individual statement. Every possible data dependency resulting from the execution of the statement is represented by an edge. Statements that cause a change in the value of variables include assignment statements, procedure calls, and iterative control structures.

For the most obvious case, consider an assignment statement of the form

Y := f(A,B,C)

where Y, A, B, and C are variables and f represents some combination of operations on A, B, and C. Assuming only one definition of A, B, and C can reach the statement, three dependencies are evident in the statement - the assignment to Y depends on the values of A, B, and C.

The actual data dependencies that may result from an external procedure call cannot be determined. Therefore, all possible data dependencies resulting from an external procedure call are included in the DDG. The direct data dependencies that may result from an external procedure call statement depend on the modifiability of the procedure arguments. For example, if the parameters are called by reference and not protected from modification by the called procedure, then each argument may be affected by the value of the

arguments. Consider the procedure call statement of the form

CALL P (X, Y)

Assume that each argument may be modified by procedure P and only one definition of X and Y can reach the statement. Figure 1 illustrates the DDG for the external procedure call statement, isolated from the remainder of the program.

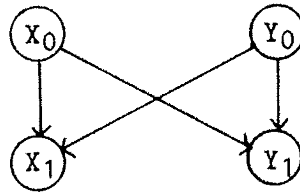


FIGURE 1. External Procedure Call DDG

Note that in Figure 1 the new definitions of X and Y are represented by duplicate nodes with incremented subscripts. Also, note that, because definitions X_1 and Y_1 are possible definitions and not absolute, X_0 and Y_0 remain live after the procedure call.

Global variables that may be referenced or modified by a called procedure are implicit arguments of a procedure call and therefore result in additional data dependencies. To include the influence of global variables in the model, global variables referenced or modified by the procedure are treated as if they are explicit arguments of the procedure call.

All of the dependencies resulting from one specific statement cannot be determined by examining the statement in isolation. An assignment to a variable in statement S is dependent on the variables used in the control constructs that determine whether or not S will be executed. The effect of

control constructs on the DDG, in particular the effect of IF-THEN-ELSE and loops, can be illustrated with an example.

Consider the if-then-else construct

```
1:  IF X < Y
2:  THEN A := B
3:  ELSE A := C

4:  D := A
```

Because the assignments to A are within the range of the effect of the boolean expression in the IF statement, the assignments to A are dependent on the variables in the boolean expression in addition to the variables in the expression-side of the assignments. The assignment of a value to D is not directly dependent on the variables in the boolean expression following the IF because the assignment is out of range of the IF-THEN-ELSE construct. However, because of the possibility that A received its value in either the IF-THEN branch or the IF-ELSE branch, D is dependent on both assignments to A. The DDG representation of the above code segment is shown in Figure 2. In Figure 2, the subscripts represent the source code line number of the definition. X_0, Y_0, B_0 & C_0 represent the initial definition of X, Y, B, & C. (We assume only one definition of the variable can reach the if-then-else statement.)

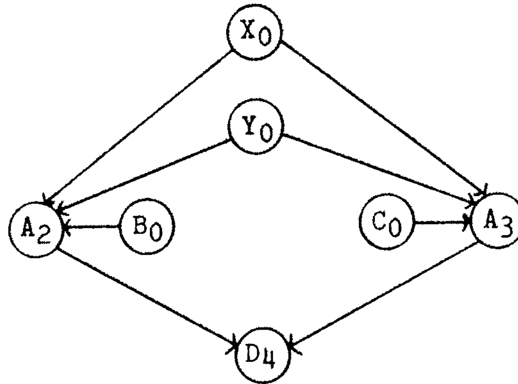


FIGURE 2. IF-THEN-ELSE DDG

Loop control structures require special examination. Consider the following code segment

```
1:  A := B
2:  WHILE X > A DO
3:      A := F(A,B)
4:  END WHILE
```

All variable assignments in the body of the loop are dependent on the variables X and A within the boolean expression, in that these variables determine how many times the body of the loop is executed. The assignment to A is considered to be dependent on both assignments to A, since it is possible, at that point, that the value of A could have been set either before or within the body of the loop. The DDG of the code segment above is illustrated in Figure 3. Again, we assume only one definition of each variable can reach the loop; these definitions have a 0 subscript in Figure 3.

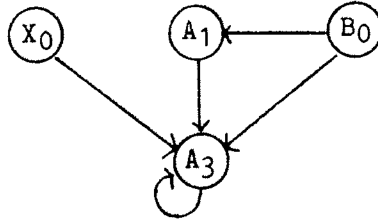


FIGURE 3. WHILE loop DDG

The statement executed immediately after a loop terminates is a common successor to all paths from the decision that controls the loop exit. Thus, the range of the effect of the variables in the loop exit controlling boolean expression is limited to the body of the loop. A detailed discussion of the influence of a decision will follow in a later section of this paper.

B. Live Definitions

One must know which definitions of Y can reach D in order to use the correct independent definitions of variable Y as the source node at variable definition D if D is dependent on Y. A definition is considered live at statement S if the value assigned may still be present on execution of S.

There are three control constructs that must be considered in determining which variable definitions are live at any particular statement - branches, joins, and sequential code.

Usually a variable definition for x kills all definitions that were live for the straight line code just before the new definition. A previous definition for x may remain live when the new definition is a probable definition such as a procedure call parameter. Since a procedure parameter might not be modified, earlier definitions for the parameter remain live. At a branch point all live definitions remain live on the branches until

redefined. The live definitions after a join point consist of the union of all of the definitions that were live on each predecessor of the join.

Loops are special cases of branches and joins. The loop exit is a branch, with one successor being the loop entrance and one successor being the code following the loop. The loop entrance is a join, where one predecessor is the code for the first entrance and the other predecessor is the loop exit.

An assignment to an array element must be considered. Assume that the array, rather than its elements, is the data object of interest. Since an assignment to an array element modifies the contents of the array, the assignment creates a new definition and kills the old definition. However, since the values of many of the array elements remain unchanged after the assignment, the new definition is dependent on the old definition. Complex data objects other than an array would be treated similarly.

C. Determining Live Definitions

A program can be viewed as a sequence of statements S_1, S_2, ..., S_n or as a control flow graph which can be formally defined as a flowgraph. A *flowgraph* is a 4-tuple F = (N, E, s, f). N is the set of nodes where each node n is an element of N and represents a basic block of the program. A *basic block* B consists of an ordered set of statements, s_1, s_2, ..., s_n such that s_1 is the only statement in B to which control can be transferred. Once s_1 is executed all of the statements in B will be executed sequentially. The final statement in a basic block is a decision statement which can transfer control to other basic blocks. E is a set of ordered pairs (i, j) where $i, j \in N$ and E represents the possible transfers of control from one basic block to another. There is an initial node s and a final node f where s and f are elements of N such that there is a path from s to every node in {N - s}

and there is a path from every node in {N - f} to f. If a program has more than one entry point, a single start node can be added to the graph with edges leading to nodes representing each entry point. Should a program have more than one terminal node (return statement), a final node can be added with edges from all terminal nodes. Any node that cannot be reached from the start node is dead code and can never be executed and therefore can be removed from the graph.

Associated with each statement S_x of program P is a pair $L_x = (l_{x,i}, l_{x,f})$ where $l_{x,i}$ is the set of all variable definitions in P that are live just prior to the execution of S_x and $l_{x,f}$ is the set of all variable definitions that are live just after the execution of S_x. The $l_{x,i}$ of each statement S_x of a program must be determined before a data dependency graph can be constructed.

Hecht presents algorithms for determining the definitions that reach each node in a flowgraph [10]. Hecht describes the algorithms as solutions to the "reaching definition" problem. The solution for the reaching definition problem can be transformed into a solution for the live definition problem by treating each statement as a basic block. Consider successive statements S_n and S_{n+1} within the same block: $l_{n,f}$ is the same as $l_{n+1,i}$. Using one of the reaching definition algorithms, one can determine the $l_{x,i}$ and $l_{x,f}$ for each statement x in a program. Hecht's "interval analysis" algorithm has a worst case complexity for "anomalous flowgraphs" of O(r**2), where r is the number of edges, and an average complexity of O(r).

D. The Influence of a Decision

Consider decision node D in a flowgraph, where the next node to be executed after D depends upon the evaluation of a boolean expression B in D.

The resulting path taken is determined by the value of the variables in B. The definitions of variables in B that are live when D is evaluated influence all definitions on the alternative paths from D, at least until the node where all paths from D merge.

A node S is defined by Carre to be an (a,b)-<u>separating node</u> when every path from node a to node b passes through S [5]. Consider a decision node D in flowgraph F. Node T is a D-<u>decision terminator</u> if T is a (D,f)-separating node where f is the final node in F and T is a predecessor of every other (D,f)-separating node in F.

The D-decision terminator is the node where all paths from D merge and is the inverse dominator of D. The D-decision terminator can be simply determined for structured control constructs. Structured control constructs always have one exit node which is also the decision terminator. The exit from a structured control construct can be easily determined from one pass through a program. Unstructured control constructs resulting from the use of GOTO statements complicate matters. However, Targin has published an algorithm for finding dominators in an arbitrary directed graph with a time bound of $O(n \log n + r)$, where n is the number of nodes and r is the number of edges in the graph [15].

In this model, a decision D influences all successor nodes of D in the flowgraph that are predecessors of the D-decision terminator. The algorithm for determining control influence uses an annotated flowgraph as input. The flowgraph is annotated with live definition information associated with each statement. The algorithm will associate with each statement n a set of control influence definitions C_n. The algorithm follows.

Find Control Influence (F)

A. For each decision node D, identify and mark the D-decision terminator. Tarjan's algorithm for finding immediate dominators (in this case inverse dominators) can be used [15].

B. For each decision node D of statement d with boolean expression B containing variables V,

   1. let S consist of the live definitions of variables in V from $l_{d,i}$
   2. follow all paths from D until the D-decision terminator is reached and for each statement n in nodes passed, let $C_n := C_n \cup S$.

Theorem 1 The worst case time complexity of the Find Control Influence algorithm is 0 (r**2) where r is the number of edges in the input flowgraph.

Proof:

Tarjan shows that Step A can be completed with a time bound of O(V logV + r) where V is the number of nodes [15].

Step B can be completed via a maximum of m passes through the flowgraph where m is the number of decision nodes. During each pass every edge is visited at most once. So Step B can be completed in a maximum of r * m steps. The limiting complexity of Step B is of O(r**2) should m approach r.

Combining the complexity of Step A and B, the worst case complexity is O(r**2).

III. DDG Algorithm

The input to the Construct DDG algorithm is the flowgraph F of a program and the output of the algorithm is a set Edges of ordered pairs of definitions. For simplicity, assume all procedures are external - the actual data dependencies of called procedures is unknown.

Construct DDG (F)

A. For each statement x in the program annotate flowgraph F with $(l_{x,i}, l_{x,f})$ where $l_{x,i}$ is the set of definitions that are live before execution of x and $l_{x,f}$ is the set of definitions that are live after execution of x. One of the algorithms of Hecht can be used to determine l_x for each statement x [10].

B. Use the Find Control Influence (F) algorithm to annotate each statement x in F with C_x.

C. For each definition statement n in F use the appropriate routine below depending on the type of definition statement:

(1) Scalar assignment, $Y := f(x_1, x_2, \ldots, x_i)$:

- Using the $l_{n,i}$ determine the live definitions L for x_1 to x_i.
- For all elements v of $L \cup C_n$ add (v, Y_n) to Edges.

(2) Array assignment, $Y(x_1, \ldots, x_j) := f(x_{j+1}, \ldots, x_i)$:

- Use case (1) above to process as if the assignment were:

  $Y := f(Y, x_1, \ldots, x_i)$

(3) Procedure call statement,

CALL $P(x_1, \ldots, x_k : Y_1, \ldots, Y_m)$:

(Note: $x_1, \ldots, x_k$ are input arguments and $Y_1, \ldots, Y_m$ are output arguments)

- Using $l_{n,i}$ determine the live definitions L for $x_1, \ldots, x_k$.
- For each element u of $Y_1 \cup \ldots \cup Y_m$,

  For all elements t of $L \cup C_n$ add (t, u) to Edges.

Theorem 2. The worst case complexity of the Construct DDG algorithm is O(r**2) where r is the number of edges in the input flowgraph.

Proof:

The worst case complexity of Step A is O(r**2) [10]. Theorem 1 shows that the worst case complexity of Step B is O(r**2). Since only one pass through the input flowgraph is required to complete Step C, each edge is traversed once. Therefore the worst case complexity of Step C is O(r). Combining the complexity of Step A, B, and C the complexity of the entire algorithm is O(r**2).

IV. DDG Complexity Measurement

The DDG is an abstract model of the data dependency complexity of a program. Using modified versions of the algorithm presented, a DDG can be constructed for arbitrary programs written in most procedural languages.

As the control flow graph has been used as a basis for deriving control complexity measures [13, 6, 8], the DDG can be used to derive data dependency complexity measures. All possible data dependencies in a program are modeled by its DDG and, since the DDG is a graph, measurable graph features of the DDG are candidates for complexity measures.

We propose the use of rooted spanning tree complexity (RSTC) along with the cyclomatic number to determine the complexity of the DDG.

The cyclomatic number can be simply calculated from the number of nodes(N), arcs(E), and connected components(N) by the formula:

$$\text{Cyclomatic Number} = E - V + N$$

It is well known, for example Berge's text, that the cyclomatic number is the maximum number of linearly independent circuits in a strongly connected graph [1]. The tree complexity measure is based on techniques for determining the number of spanning trees contained in a graph.

We define a *tree* to be a connected graph with no cycles and the *root* node of a tree is a node that is a predecessor of all other nodes in the tree. Consider a directed graph G = (V, E), where V is the set of nodes and E is the edge set of G. A spanning tree of G is a graph ST(G) = (V, E'), where E' is a subset of E and ST(G) is a tree (that is, a tree that includes every node in G). We define the *rooted spanning tree complexity* with root node x (RSTC(x)) of graph G as the number of distinct spanning trees with root x that can be constructed from the graph consisting of the nodes and arcs of G that are successors of x. The notion of using the number of spanning trees in a graph as a measure of complexity has been described in graph theory literature [4,16].

The following theorem shows how to calculate the number of distinct rooted spanning trees with a specified root contained in a directed graph [17, as cited in 1].

Theorem. Let G = (X,U) be a directed graph, and let x_1 be a member of X. The number of trees with root x_1 contained in G equals the determinant of a matrix A where

$$\det(A) = \begin{vmatrix} \sum_{i \neq 2} a(i,2) & -a(2,3) & \cdots & -a(2,n) \\ -a(3,2) & \sum_{i \neq 3} a(i,3) & \cdots & -a(3,n) \\ \vdots & \vdots & \cdots & \vdots \\ -a(n,2) & -a(n,3) & & \sum_{i \neq n} a(i,n) \end{vmatrix}$$

where a(i,j) is the number of edges directed from node x_i to node x_j.

By including the number of rooted spanning trees in a graph along with cyclomatic complexity we incorporate a measure of both the noncyclical and cyclical complexity of a graph.

We define the definition slice graph of a DDG of program P at node X to be a subgraph of the DDG of P that includes node X and all nodes in the DDG of P that are predecessors of node X. Edges of the definition slice graph include all paths from predecessors to X. Because programmers appear to decompose programs into slices when debugging, we hypothesize that the RSTC of the definition slice graph of a specified node in a DDG is a significant measure. We take the definition slice graph of the specified node and reverse the directions of the edges and then determine the RSTC of the definition slice. To calculate the RSTC of an entire program we use the definition that represents the output of the program as a root. Should there be more than one output node, a node R, with edges directed from each output node to R, is added to the DDG. R is then used as the root for calculating the RSTC.

V. Example Complexity Calculation

To illustrate the procedure for calculating the RSTC and the cyclomatic number of the DDG of a program, we present the process as performed on a segment of a FORTRAN program designed to calculate the sine function:

```
1:        SIN = X
2:        TERM = X
3:        DO 20 I = 3, 100, 2
4:        IF (DABS(TERM) .LT. E) GO TO 30
5:           TERM = -TERM * X**2 / FLOAT(I * (I - 1))
6:           SIN = SIN + TERM
7:    20  CONTINUE
8:    30  RETURN
```

Using the DDG algorithm, we construct the DDG for the program. The DDG cyclomatic number is calculated as 10 by counting the number of nodes (N) and

edges (E) in the DDG. To calculate the RSTC of the example, we examine the code to determine which of the definitions represent possible outputs of the program segment. The variable SIN is the only output variable and at the end of the code segment there are 2 live definitions of SIN. We add an output node to the DDG and construct edges for the nodes representing the 2 live definitions of SIN to the output node. The modified DDG is illustrated in Figure 4 below.

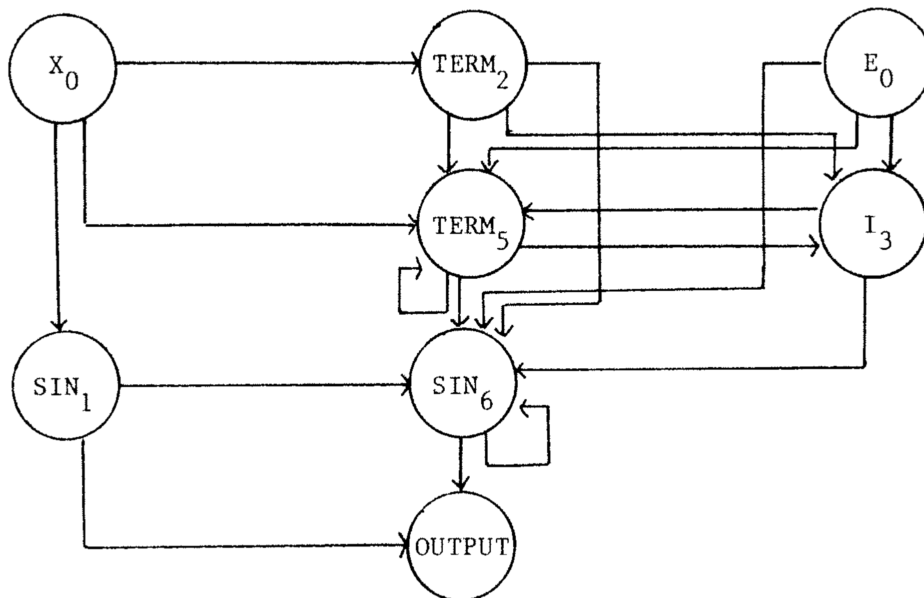


FIGURE 4. DDG of Example Program Segment With Output Node

To calculate the RSTC of the DDG the direction of the edges is reversed and the matrix described in Section IV is constructed using the output node as the root. The determinant of the matrix is the RSTC of the graph. The matrix and determinant for the example program is

$$
\begin{array}{l} SIN_6 \\ SIN_1 \\ TERM_5 \\ TERM_2 \\ X_0 \\ I_3 \\ E_0 \end{array}
\left|\begin{array}{ccccccc}
1 & -1 & -1 & -1 & 0 & -1 & -1 \\
0 & 2 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 2 & -1 & -1 & -1 & -1 \\
0 & 0 & 0 & 3 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & -1 & -1 & 0 & 2 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 3
\end{array}\right| = 162
$$

VI. Initial Validation

A series of tests were conducted to provide initial experimental support of the DDG model and the associated measures [3]. Each test compared a set of measures applied to two or more program segments that represent solutions to the same problem. The measures applied were the RSTC and cyclomatic complexity of the DDG representation of the program, the software science effort measures [9], the flow of control cyclomatic complexity [13], and a subjective ranking of program clarity. The program segments included in the tests were selected from the set of programs that Gordon used in his effort to validate software science measures [7]. Gordon reported software science measures and the clarity rankings of two or more implementations of 46 problems from the literature. In the effort to validate the DDG model and measures, nineteen program segments that are solutions to nine problems were selected from Gordon's testbed.

Table 1 summarizes the results of the validation. In each test, version A is described as less clear than the version B. For Test 1, which has three program versions, version B is described as less clear than version C.

Relative clarity is the reported subjective judgment of Kernigham, Wirth, or Knuth [11,20,12].

Program Segment	Halstead Effort Number	Control Cyclo. Comp.	DDG Cyclo. Comp.	RSTC
1a	2008	4	6	64
1b	300	3	1	6
1c	54	1	0	1
2a	1869	5	26	77,760
2b	1020	3	9	384
3a	6263	4	25	2100
3b	4833	4	14	240
4a	4507	3	10	96
4b	3485	3	10	162
5a	4445	4	17	300
5b	2931	4	16	480
6a	4275	5	26	1260
6b	1974	4	20	320
7a	3447	3	9	32
7b	2821	3	12	80
8a	952	3	7	9
8b	620	3	6	4
9a	2605	5	9	360
9b	2160	4	5	40

Table 1. Metrics From Testbed Program Segments

In each case the Halstead E measure distinguished between clear and less clear implementations. However, one must keep in mind that the purpose of Gordon's study was to support the software science metrics.

The RSTC was unable to match the reported subjective clarity ranking in Test 4, Test 5, and Test 7. The DDG cyclomatic complexity measure was unable to match the clarity ranking in Test 4 and Test 7. A close look at the reasons given for the clarity judgments in the experiment reveals the subjective nature of "clarity" and provides explanations for the inability of the DDG based metrics to match the human ratings.

In the three cases of disagreement the reported rankings are a result of semantic differences rather than clarity or readability differences. Program 4a, which is designed to calculate the sine function, fails to initialize three of the variables. Program Segments 5a and 5b are designed to input a sequence of numbers, select the first n distinct numbers, and assign the numbers to an array with n elements. Program Segment 5b is described as "more economical" than Program Segment 5a because, in Program Segment 5a, a loop exit test is not always necessary [20]. Program Segments 7a and 7b are designed to multiply two nonnegative integers using only addition, doubling, and halving. Program Segment 7a is criticized because the program segment may cause a semantic error on some machines in the form of numerical overflow [20].

The flow of control cyclomatic complexity fails to distinguish relative complexity in half of the cases, probably because of its small range of values. Recall that the value depends only on decision count. Out of all nineteen testbed programs and program segments the flow of control cyclomatic complexity only had four different values.

Note that the value of the RSTC for Program Segment 2a is very high while the E measure is relatively low. An investigation of Program Segment 2a may explain the discrepancy:

```
      DO 10 I=1,M
      IF(BP(I)+1.0)19,11,10
11    IBN1(I)=BLNK
      IBN2(I)=BLNK
      GO TO 10
19    BP(I)=-1.0
      IBN1(I)=BLNK
      IBN2(I)=BLNK
10    CONTINUE
```

Because of the computed go to statement in the program segment, the data dependency complexity is high. Each variable definition is dependent on many other definitions because of the complex flow of control. Since the E measure is based only on the counts of operators and operands it does not account for the added complexity resulting from the complex interactions among variables.

The subjectivity of human judgment of program clarity was a serious problem in the validation effort. However, the results show that DDG based measures are sensitive to some aspect(s) of complexity that E is not. A short program with few operators and operands can be complex. Program Segment 2a is one case where E is relatively low and RSTC and DDG cyclomatic complexity are extremely high for a short nine line program. The flow of control cyclomatic complexity measure appears to be relatively insensitive to clarity.

VII. Summary and Conclusion.

In this paper, the data dependency graph was described, aspects of DDG construction were discussed, algorithms for building the DDG were outlined, measures of DDG complexity were introduced, and the results of an initial validation effort were presented.

The DDG represents a model of software complexity that is necessary in order to measure the complexity of the data dependencies of software. By transforming a program into a graph theory based abstraction of the program's data dependencies, we are able to examine data relationships without unnecessary details. We can then apply the ideas, theorems, and algorithms from graph theory. Since a graph can be converted into a matrix representation of a graph, we can apply the tools of linear algebra to a matrix that is an abstraction of a program.

The DDG forms the basis for a set of complexity measures that should prove valuable in learning how to algorithmically identify troublesome programs.

The algorithm for constructing a DDG from a program demonstrates that a DDG can be produced automatically. Thus, data dependency measures may be taken from large numbers of programs and, if future empirical studies verify the validity of the metrics, aid in the development of more reliable software.

The initial validation demonstrates that data dependency graphs from actual programs can be built and the DDG based measures can be calculated. The results show that the interdependency of variable definitions is a component of program complexity. However, the initial validation was limited and the subjectivity of clarity judgments was a problem.

This research effort suggests future research in empirical, theoretical, and applied areas.

Additional empirical work is necessary to verify the model and associated metrics and to learn more about the complexity of existing software. Empirical work may be performed by studying large samples of industry software and through controlled experiments.

The model and metrics developed in this research could be refined further. A theoretical study of data dependency graphs will help us learn more about the programs that the graphs represent. For example, if we can see how to modify a DDG to reduce complexity, we may learn how to map the change in the DDG to a change in the program. We also feel that additional studies of graph complexity measures will lead to a greater knowledge of how to derive more meaningful program complexity measurement tools.

Finally, we see numerous application areas to be explored. The DDG could be used as a basis for providing a rich set of programming tools. Cross reference lists, produced from a DDG at compile time, could show a programmer exactly which statements could have affected the value of a variable at a particular statement. The DDG could be produced at run time and include only actual dependencies as edges. Such a run time DDG could allow for much more powerful debugging tools.

AFFILIATION OF AUTHORS

James M. Bieman is Assistant Professor, Department of Computer Science, Iowa State University, Ames, Iowa 50011. William R. Edwards is Associate Professor, Computer Science Department, University of Southwestern Louisiana, Lafayette, Louisiana 70504.

REFERENCES

[1] C. Berge, Graphs and Hypergraphs, North-Holland, Amsterdam, The Netherlands, 1973.

[2] J.M. Bieman, Measuring Software Data Dependency Complexity, Ph.D. Dissertation, Computer Science Department, University of Louisiana, Lafayette, Louisiana, 1984.

[3] J.M. Bieman and W.R. Edwards, "Experimental Evaluation of the Data Dependency Graph For Use in Measuring Software Clarity", Proc. of 18th Hawaii Conf. on Systems Science, Vol. 2, pp. 271-276, 1985.

[4] N. Biggs, Algebraic Graph Theory, Cambridge University Press, 1974.

[5] B. Carre, Graphs and Networks, Clarendon Press, Oxford, 1979.

[6] E.T. Chen, "Program Complexity and Programmer Productivity", IEEE Trans. on Software Eng., Vol. SE-4, No. 3, pp. 187-194, May 1978.

[7] R.D. Gordon, "Measuring Improvements in Program Clarity", IEEE Trans. on Software Eng., Vol. SE-5, No. 2, pp. 74-90, March 1979.

[8] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "Applying Software Complexity Metrics to Program Maintenance", Computer, Vol. 15, No. 9, pp. 65-79, Sept. 1982.

[9] M.H. Halstead, Elements of Software Science, Elsevier, New York, 1977.

[10] M.S. Hecht, Flow Analysis of Computer Programs, Elsevier North-Holland, 1977.

[11] B.W. Kernighan and P.J. Plauger, "Programming Style: Examples and Counterexamples", ACM Computing Surveys, Vol. 6, No. 4, pp. 303-319, Dec. 1974.

[12] D.E. Knuth, "Structured Programming With Go-To Statements", ACM Computing Surveys, Vol. 6, No. 4, pp. 261-301, Dec. 1974.

[13] T. McCabe, "A Complexity Measure", IEEE Trans. on Software Eng., Vol. SE-2, No. 4, pp. 308-320, July 1976.

[14] E. Oviedo, "Control Flow, Data Flow and Program Complexity", Proc. of the IEEE Computer Society's 4'th Int. Computer Software and Applications Conf. (COMPSAC80), pp. 146-152, Nov. 1980.

[15] R. Tarjan, "Finding Dominators in Directed Graphs", SIAM J. Computing, Vol. 3, No. 1, pp. 62-89, March 1974.

[16] H.N.V. Temperly, Graph Theory and Applications, Ellis Horwood Limited, 1981.

[17] W.T. Tutte, "The Dissection of Equilateral Triangles into Equilateral Triangles", Proc. of the Cambridge Philosophical Society, Vol. 44, pp. 203-217, 1948.

[18] M. Weiser, "Program Slicing", Proc. of the 5th Int. Conf. on Software Eng., 1981, pp. 439-449.

[19] M. Weiser, "Programmers Use Slices When Debugging", _Communications of the ACM_, Vol. 25, No. 7, July 1982, pp. 446-452.

[20] N. Wirth, "On the Composition of Well-Structured Programs", _ACM Computing Surveys_, Vol. 6, No. 4, pp. 247-260, Dec. 1974.