Measuring Data Dependency Complexity

by

James M. Bieman
William R. Edwards

TECHNICAL REPORT 83-5-3

July, 1983

Copyright (c) 1983
Computer Science Department
The University of Southwestern Louisiana
P.O. Box 44330, Lafayette, Louisiana 70504
(318) 231-6284
All Rights Reserved

MEASURING DATA DEPENDENCY COMPLEXITY

1 Introduction.

Although much research has been directed towards measuring the complexity of the programmer/program interface, most of this work has centered on the measurement of flow of control complexity. McCabe's [76] presentation of cyclomatic complexity is an example of the modeling and measurement of flow of control complexity.

Weiser [82] found that programmers seem to work backwards when debugging, examining only instructions that affect the variables in error. 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.

Before measures of data dependency complexity can be derived, we must have a model of the data dependencies within a program. The model we have developed is the data dependency graph (DDG).

The DDG is a directed graph, with each node representing a variable definition and each edge representing a data dependency. To construct a DDG it is necessary to collect live definitions from alternate pathways to determine

dependencies of a given variable definition.

We present an algorithm to generate DDG's from arbitrary programs written in procedural languages and we use the DDG to develop measures of software complexity. The algorithm is driven by a scanner that can recognize variable definitions and flow of control constructs, and uses knowledge of the live definitions and the variables that determine control flow.

We use the DDG and the condensed DDG to derive measures of definition and variable activity, coverage, parallelism, adjacency, and self dependency.

2 Data Dependencies.

A data dependency exists when the value of a variable may depend on the value of another variable or an earlier value of the same variable. Data dependency complexity is the complexity of the programmer/program interface resulting from the combination of all of the data dependencies in a program.

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. As

Weiser [81] found, experienced programmers seem to decompose programs into *slices* when debugging. A *slice* is a program after the deletion of all statements that do not affect the value of a specified variable at a given statement. 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 particular execution. The more complex the data dependencies in a program, the larger the search space that the programmer must contend with when searching for the source of a bug, or, in Weiser's terms, the larger the slice.

3 *The Data Dependency Graph.*

The DDG is a directed graph representation of the possible data dependencies in a program. We say that a data dependency exists in a program when the value of one variable may affect the value of another or possibly the same variable during execution of the program. The variable that is modified is called the dependent variable and the variable that may affect it is called the independent variable. The DDG is a representation of all of the individual data dependencies within a program.

The DDG consists of nodes and edges, the nodes representing data objects and the edges representing the

dependencies.

In constructing a DDG representation of a program, a node is used to represent each variable definition, rather than each variable. A variable *definition* is a statement that may modify the value of a variable, such as assignment statements, procedure calls, and input statements [Hecht77]. We will assume that initialization or the initial state of a variable is a definition. Each node is labeled with a name identifying the variable that the node represents and a subscript. The subscript distinguishes between nodes representing different definitions to the same variable and and are sequentially numbered based on the relative position of the definition in the source code.

Edges represent all possible data dependencies resulting from all variable definitions. Consider definition d of variable v. Definition d is dependent on a set of independent variables I. An edge is constructed for each definition of variables in I that can reach d. 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. Oveido [80] used similar notions to develop a measure of data flow complexity; however he did not represent the complexity of the entire program with a graph.

3.1 Dependency Sources.

A variable definition may depend on the value of several independent variables for a variety of reasons. A dependency can be (1) a *direct dependency* that can be determined by evaluating an individual statement or (2) a *control dependency* that is determined from the mechanism that determines 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 (isolated from the rest of the program) is represented by an edge. Statements that cause a change in the value of variables include assignment statements, procedure calls, and iterative control structures.

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. There are 3 dependencies evident from the statement since the assignment to Y depends on A, B, and C. A DDG of the above assignment statement is shown in figure 1.

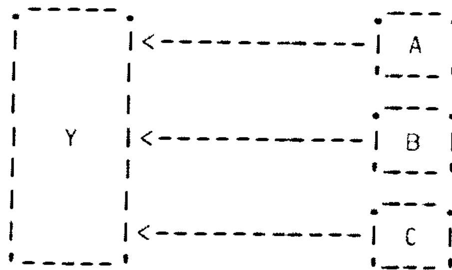

Figure 1. Simple Assignment DDG

Because we cannot determine the actual data dependencies that may result from an external procedure call, we must include all possible data dependencies 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 all of the parameters to a procedure are called by reference and cannot be protected from modification by the called procedure, then each argument may be affected by the value of all of the arguments. Consider the procedure call statement of the form

```
CALL P (X, Y)
```

Assuming that the parameters are passed using the call by reference convention and each argument may be modified by procedure P, figure 2 illustrates the DDG for the external procedure call statement isolated from the remainder of the program.

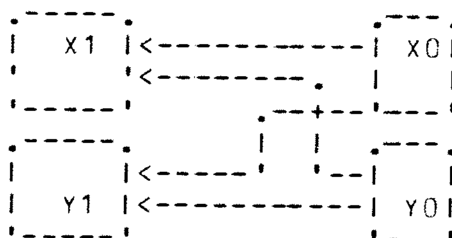

Figure 2. External Procedure Call DDG

Note that in figure 2 the new definitions of X and Y are represented by duplicate nodes with incremented subscripts.

Global variables that may be referenced or modified by a called procedure results in additional data dependencies. To include the influence of global variables in the model, we treat global variables referenced or modified by the procedure as if they are arguments of the procedure call.

It is clear that iterative control structures are also a source of a direct dependency. In a statement of the form

```
FOR I = A TO B BY J,
```

the value of I depends on the values of A, B, and J, and the DDG of the above statement is illustrated in figure 3

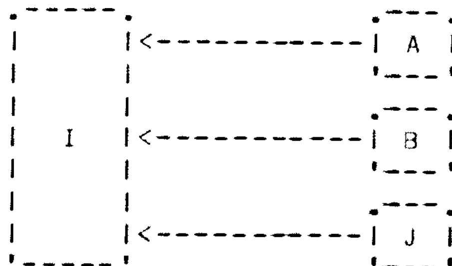

Figure 3. Iterative Control Structure Graph

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 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

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

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 DDG representation of the IF-THEN-ELSE statement is shown in figure 4

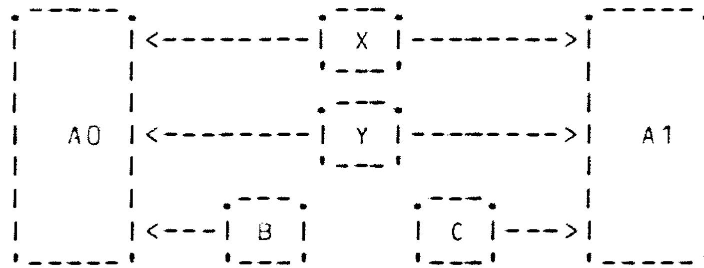

Figure 4. IF-THEN-ELSE DDG

The assignment of value to D in the code segment above is not directly dependent on the variables in the boolean expression following the IF because it is out of range of the IF-THEN-ELSE construct. However, because it is possible that A received its value in either the IF assignment or the ELSE assignment, D is dependent on both assignments to A. To incorporate the assignment to D in the DDG graph illustrated in figure 4, edges must be drawn from the nodes labeled A0 and A1 as shown in figure 5.

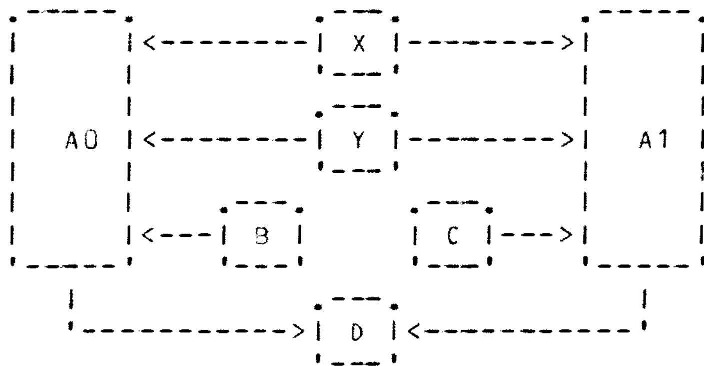

Figure 5. IF-THEN-ELSE DDG with assignment

Loop control structures require special examination.

Consider the following code segment

```
A := B

WHILE X > Y DO
   C := A
   A := D
END WHILE
```

All variable assignments in the body of the loop are dependent on the variables X and Y within the boolean expression, since these variables determine how many times the body of the loop is executed. The assignment to C is 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 6.

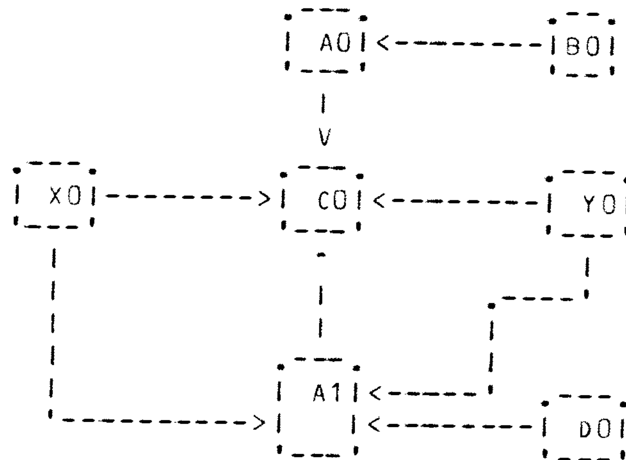

Figure 6. WHILE loop DDG

We assume that loops terminate; thus the range of the effect of the variables in the loop exit controlling boolean

expression is limited to the body of the loop.

3.2 *Live Definitions*.

In order to use the correct independent definitions of variable y to use as the source node at variable definition d dependent on y, we must know which definitions of y can reach d. A definition is considered *live* at statement s if the value assigned may still be present on execution of s. Determining the live definitions is a prerequisite for constructing the DDG.

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

A variable definition for x *kills* all definitions that were live for the straight line code just before the new definition, unless the new definition is a *probable definition* such as a procedure call parameter. Since in that case the 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. 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 many of the array elements remain the same as before the assignment, the new definition is dependent on the old definition.

3.3 DDG Algorithm

The DDG algorithm is designed to be included in a scanner that can recognize variable definitions, loop constructs, branch statements, and labels for an ALGOL-like language. The input to the scanner is the program and the output is a file containing ordered pairs of labeled nodes or an incidence matrix, where, again, the nodes represent variable definitions and the pairs represent the individual data dependencies. Note that the algorithm generates no multiple edges. The algorithm and the main data structures that store critical information during a parse are described below.

Data Structures

- A set DEF which, for each variable v defined in statement n, contains an element vn representing the variable

definition.

- For each variable v, a tree LIVE(v) where each node contains a set of variable definitions which is a subset of DEF. The LIVE trees are used to save and restore the names of live definitions as necessary during flow of control branches and joins. It is necessary to use a tree because the live definitions along alternative pathways must be saved on tree branches so that at a join point the alternative sets of live definitions can be merged.

- A stack CONTROL whose members are sets of variable definitions, subsets of DEF. The CONTROL stack is used to save and restore the names of definitions used in flow of control boolean expressions.

- A set EDGES which is a subset of DEF X DEF, may be represented as a list of pairs, an incidence matrix, or a table. EDGES contains the running list of DDG edges as the algorithm recognizes them. It becomes the final output of the program.

- A set PATCHES which is a set of ordered pairs (a, b), where a is a string used for identification and b is a member of DEF. The PATCH set is used to store a list of modifications to EDGES to be done after the completion of the parse.

- For each internal procedure p, a set GLOBALS_SET(p) and a set GLOBALS_REF(p) where the members of both sets are variable names. The GLOBAL sets are used to determine which global variables to treat as input and/or output arguments at procedure call statements.

Algorithm

A. During an initial parse, the GLOBAL_SET and GLOBAL_REF sets are filled in with the names of global variables set and/or referenced by internal procedures.

B. Sequentially processing statements in a syntactically legal program, For each statement n of a program execute one of the cases depending on the statement type.

(1) Scalar assignment, y := f(x1, x2,..., xi) :

- Using the LIVE trees, determine the live definitions, L, for x1 to xi.
- Using the CONTROL stack, determine the set of definitions, C, that determine current flow of control.
- For all elements v of L U C, add (v, yn) to EDGES.
- Replace the current set of live definitions on the LIVE(y) tree with {yn}.

(2) Array assignment, y(x1, ..., xj) := f(xj+1,..., xi) :

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

y := f(y, x1, ..., xi)

(3) Procedure call statement,

CALL P(x1,..., xk : y1,...,ym) :

(Note: x1,...,xk are input arguments and y1,...,ym are output arguments)

- If P is an internal procedure use the GLOBAL(P) sets to determine the global variables, R, that are referenced and the global variables, S, that are set by procedure P.
- For each element u of y1 U ... U ym U S, use the following procedure:
  - Using the LIVE trees, determine the set of live definitions, L, for x1 to xi and R.
  - Using the CONTROL stack, determine the set of definitions, C, that determine current flow of control.
  - For all elements t of L U C, add (t, un) to EDGES.
  - Add un to the current set of live definitions on the LIVE(u) tree.

(4) Decision statement,

IF B(x1,..., xi) THEN S1 ELSE S2 :

(where B is a boolean expression)

- Push onto the CONTROL stack the union of the current top of the stack and the current live

definitions of x1...xi.

- Create two child nodes of the current node in each LIVE tree with each child containing a copy of the contents of the current node.
- Push onto CONTROL a copy of the top of CONTROL.
- Using the first child node in each LIVE tree as the current node, process S1.
- Pop the top off of CONTROL.
- Push onto CONTROL a copy of the top of CONTROL.
- Using the second child node in each LIVE tree as the current node, process S2.
- Pop the top off of CONTROL.
- Replace the contents of the parent LIVE nodes with the union of the contents of the child nodes.
- Free the child nodes making the updated parent node the current LIVE node.
- Pop the top off of CONTROL.

(5) Loop statement, WHILE B(x1,..., xi) do S:

- Push onto CONTROL the union of the top of CONTROL and the live definitions for x1,...,xi.
- Add a loop_dummy(v,n) definition to each LIVE(v) tree.
- Process S.
- Replace each occurrence in EDGES of the loop_dummy with the current definition for the respective

variable.

(6) Labeled statement, L: S :

- Add to the top of CONTROL a control_label_dummy(L).
- Add a live_label_dummy(L,v) to each LIVE(v) tree. (The dummy definitions are added so that they can be replaced by the correct sets of control and live definitions after all of the go to statements that use L are parsed.)

(7) Go to statement, GO TO L :

- For each element u in the top of CONTROL add to the PATCH set (CONTROL_LABEL_DUMMY(L),u).
- Pop the top off of CONTROL.
- Push the empty set onto CONTROL since the current control definitions cannot affect code following a go to statement.
- For each variable v in the program, add to the PATCH set (live_label_dummy(L,v), CURRENT(v)) which will be used to replace instances of live_label_dummy(L,v) in EDGES at the conclusion of the parse.
- Replace the contents of the current node in each LIVE tree with the empty set since the current live definitions are not necessarily live in

statements following a go to.

(8) End of program statement:

- For each dummy element u in the PATCH list, where u = (dummy_id, z), for each element w of EDGES where dummy_id is contained in w, add w' to EDGES, where w' is w but with each instance of dummy_id replaced with z.
- delete all edges containing dummy elements from EDGES.

4 Data Dependency Complexity Metrics.

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 [McCabe76, Chen78, Harrison82], the DDG can be used to derive data dependency complexity measures. All possible data dependencies in a program are modelled by its DDG and, since the DDG is a graph, measurable graph features of the DDG are candidates for complexity measures.

4.1 Features of the DDG.

Some of the features of the DDG that are useful in defining metrics include:

- Order(g) - the number of nodes in graph g.
- Connectivity Graph (x,g) - maximal connected subgraph of graph g that includes node x.
- Condensed DDG(p) - A graph constructed from DDG(p) such that there is a node for every variable in p rather than for each definition. Edges correspond to the edges in DDG(p). The DDG(p) nodes for one variable are combined into one node in the condensed graph. For each edge that is incident to node vn (representing a definition) in the DDG, there a node incident to node v (representing a variable) in the condensed DDG.
- Definition Slice Graph(x,p) - Subgraph of DDG(p) that includes node x and all nodes in DDG(p) that are predecessors of node x. Edges include all paths from predecessors to x.
- Definition Influence Graph(x,p) - Subgraph of DDG(p) that includes node x and all nodes in DDG(p) that are successors of x. Edges include all paths from x to predecessors.
- Indegree(x) - Number of edges entering node x.

- Outdegree(x) - Number of edges leaving node x.
- Degree(x) - indegree(x) + outdegree(x)
- Bridge - A bridge is an edge contained in a connected graph whose removal from the graph results in a disconnected graph.
- Cycle - A cycle is a sequence of edges with the same initial and terminal node.

4.2 Proposed Metrics.

The DDG, the condensed DDG, and subgraphs of either graph can be used as a basis for deriving complexity measures. The DDG can be mapped to actual statements and can be used to measure the complexity of individual definitions. The condensed DDG is useful for determining activity around specific variables and memory locations and can be used to show variable self dependencies. Candidate metrics are described in the remainder of this section.

Definition Activity - Definition Activity can be measured as the indegree and outdegree of DDG nodes. Important program activity complexity measures include mean, median, and standard deviation of the indegree and outdegree of the DDG nodes in a program. High activity nodes based on high indegree and/or outdegree indicate highly active definitions. Note that the sum of the indegree for all nodes in a program is similar to Oviedo's [80] data flow complexity measure except that Oviedo used sequential code blocks rather

than definitions as nodes and the indegree of a DDG node includes the effect of control dependencies. The activity measures could also be applied to the condensed DDG and would indicate _variable activity_ rather than definition activity.

Coverage Metrics - Based on Weiser's [81] candidate slicing metrics, coverage metrics measure how much of the DDG is contained in each definition slice or definition influence. We propose the following coverage metrics:

- Definition Slice Coverage(x,p) =

$$\frac{\text{Order(Definition Slice(x))}}{\text{Order(DDG(p))}}$$

where x is a definition and p is a program.

- Program Slice Coverage(p) is the mean definition slice coverage for all definitions in program p.

Similarly we can determine _Definition Influence Coverage_

- Definition Influence Coverage(x,p) =

$$\frac{\text{Order(Definition Influence(x,p))}}{\text{Order(DDG(p))}}$$

- Program Influence Coverage(p) is the mean definition influence coverage for the entire program.

We can also derive connectivity measures such as

- Definition Connectivity Coverage(x,p) =

$$\frac{\text{Order(Connectivity Graph(x,p))}}{\text{Order(p)}}$$

- Program Connectivity Coverage(x,p) is the mean of the Definition Connectivity Coverage for all definitions in a program.

Parallelism. Another of Wiser's [81] proposed slicing metrics is parallelism which can be applied to a DDG. We will say that *DDG Parallelism* is the number of different connectivity graphs in a DDG. Another possible parallelism measure is the number of bridges in a DDG.

Adjacency. The adjacency measure is the average of the difference in statement numbers between adjacent nodes. The adjacency measure will show how spread out the interdependent definitions are in the source code. We assume that programmer effort is related to the distance between dependent statements. The adjacency measure parallels Elshoff's [76] span metric.

Self Dependency Measurement. Self dependency is the dependence of a definition node or variable upon itself. These measures could be applied to the condensed DDG as well as the DDG and include

- self dependency ratio =

$$\frac{\text{Number of Nodes in DDG(p) that are successors to themselves}}{\text{Order(DDG(p))}}$$

- number of cycles in a DDG

5 Summary and Research Directions.

In this report, the data dependency graph was described, aspects of DDG construction were discussed, an algorithm for building the DDG was outlined, and a set of candidate data dependency metrics were introduced.

We plan to continue to refine the models of data dependency complexity and the associated metrics. Empirical work will be conducted in order to evaluate the proposed metrics and compare these new metrics with existing metrics.

References

[Chen78] Chen, Edward T. "Program Complexity and Programmer Productivity," *IEEE Transactions on Software Engineering*, SE-4, 3 (1978), 187-194.

[Elshoff76] Elshoff, J.L., "An Analysis of Some Commercial PL/1 Programs," *IEEE Transactions on Software Engineering*, SE-2, 2 (1976), 113-120. Also in Victor R. Basili, editor, *Tutorial on Models and Metrics for Software Management and Engineering*, IEEE Catalog No. EHO-167-7, (1980), 266-273.

[Harrison82] Harrison, Warren, K. Magel, R. Kluczny, and A. DeKock, "Applying Software Complexity Metrics to Program Maintenance," *Computer*, 15, 9, September (1982), 65-79.

[Hecht77] Hecht, Matthew S., *Flow Analysis of Computer Programs*, Elsevier North-Holland, 1977.

[McCabe76] McCabe, Thomas, "A Complexity Measure," *IEEE Transactions on Software Engineering*, SE-2, 4 (1976), 308-320.

[Oviedo80] Oviedo, E., "Control Flow, Data Flow and Program Complexity," _Proc. COMPSAC80_, 146-152.

[Weiser81] Weiser, Mark, "Program Slicing," _Proceedings of the 5th International Conference on Software Engineering_, (1981), 439-449.

[Weiser82] Weiser, Mark, "Programmers Use Slices When Debugging", _Communications of the ACM_, 25, 7, July 1982, 446-452.