Estimating the Number of Test Cases Required to Satisfy the All-du-paths Testing Criterion

James M. Bieman
Department of Computer Science
Colorado State University
Fort Collins, CO 80523 USA

Janet L. Schultz
Department of Statistics
Iowa State University
Ames, IA 50011 USA

Abstract

The all-du-paths software testing criterion is the most discriminating of the data flow testing criteria of Rapps and Weyuker. Unfortunately, in the worst case, the criterion requires an exponential number of test cases. To investigate the practicality of the criterion, we develop tools to count the number of complete program paths necessary to satisfy the criterion. This count is an estimate of the number of test cases required. In a case study of an industrial software system, we find that in eighty percent of the subroutines the all-du-paths criterion is satisfied by testing ten or fewer complete paths. Only one subroutine out of 143 requires an exponential number of test cases.

Keywords: Data flow analysis, software testing, software measures.

1 Introduction

A major activity in software testing research is deriving criteria that aid in selecting the smallest set of test cases that will uncover as many errors as possible. Structural testing uses the control and data flow of a program to select test cases.

Testing criteria that are more discriminating in uncovering errors tend to require a greater number of test cases. For example, the "all branches" criterion is more discriminating than the "all statements" criterion and usually requires more tests. The most discriminating criterion is the "all paths" criterion which requires an infinite number of test cases in programs with loops. Since we want testing to be completed eventually, a useful criterion must be met with a finite and acceptably small number of test cases.

Several analytical studies compare structural testing criteria with respect to inclusion and worst case complexity of the criteria [Wey84, WGM85, CPRZ85, RW85, Nta88]. The "all branches" criterion *includes* the "all statements" criterion because when the "all branches" criterion is met the "all statements" criterion is also satisfied. The complexity of a criterion refers to the worst case growth in the number of test cases required to satisfy the criterion as a size attribute of a program increases.

Ntafos suggests the use of strategies that require at most $O(n^2)$ test paths [Nta88]. Ntafos also notes that these worst case bounds may not reflect the actual number of required test cases. Our research is directed towards determining whether testing strategies that have an exponential worst case bounds may actually be feasible on real programs.

Some of the more discriminating structural testing criteria are based on data flow analysis. Rapps and Weyuker define a family of path selection criteria based on data flow relationships [RW85]. These criteria focus on the program paths that connect the definitions and uses of variables (du-paths). Of these criteria, the all definition/use criterion (all-du-paths) is the "strongest". The all-du-paths criterion requires that the test data cover all du-paths in a program. This criterion includes all of the other data flow criteria and requires the greatest number of paths in a program to be tested. Thus, the all-du-paths criterion should be the most effective of the data flow criteria in discovering errors.

Unfortunately, it may take an exponential number of test cases to meet the all-du-paths criterion. Weyuker shows that the all-du-paths criterion requires 2^t test cases in the worst case, where t is the number of conditional transfers [Wey84].

A testing strategy that requires an exponential number of test cases is not realistic. However, the actual number of test cases can be much less than the worst case. Only an empirical study can determine whether the worst case scenario is common.

In this study, we determine the du-paths in a software system in use commercially. The software is a natural language text analyzer used for marketing research. After identifying the du-paths, we estimate the minimum number of test cases necessary to satisfy the all-du-paths criterion. Our estimate is based on finding a minimal sized set of complete paths (paths from the start node to the terminal node of a program flowgraph) that covers all of the du-paths. Each complete path can be "exercised" by one test case if appropriate input data can be found. We find that for most of the subroutines, the all-du-paths criterion can be satisfied with fewer than ten complete paths. Only one subroutine requires an exponential number of complete paths. One other subroutine requires a comparatively large number of paths. Thus, our results indicate that the all-du-paths criterion can be used to test most of the subroutines in the software system under study.

The number of complete paths needed to meet the criterion is an estimate of the number of test cases required to meet the criterion. Some complete paths of a program may be infeasible — no input data exists that can cause such a path to be executed. Similarly, some du-paths may be infeasible. Thus, it might be impossible to select test data that satisfies the particular criterion. Unfortunately, determining whether a particular path is feasible is undecidable. Frankl and Weyuker suggest the use of heuristics to identify infeasible paths [FW88]. Since some complete paths and du-paths may be infeasible, our measure of the necessary number of complete paths will tend to be higher than the number of feasible paths.

Our research tools are based on a formal specification of the all-du-paths criterion [Sch88]. This specification is written in the SPECS specification language [BBC87] in terms of a language-independent representation *(StandardRep)* of imperative programs [BBC*88]. This specification of the all-du-paths criterion is applicable to *any* imperative language that can be mapped to the *StandardRep*.

We utilize a tool [DBB86] that generates a *StandardRep* from an ISO Standard Pascal [JW85] program. The output routines of the generator are modified to produce a Prolog data base for each program unit. Each data base represents an augmented flowgraph in which the nodes and edges are included, along with the variables which are defined and used in each node. A Prolog program estimates the minimum number of complete paths that satisfy the all-du-paths criterion for the corresponding program unit. All tools used in this research are rigorously specified using SPECS.

A related system developed by Frankl and Weyuker, ASSET, determines whether a given test set is adequate with respect to the criterion, and produces a list of any node pairs required by the criterion but not exercised by the test data [FWW85, FW85]. This list can then be used to strengthen the test data set. In contrast, our research tools are designed to estimate the number test cases required by the all-du-paths criterion. ASSET is designed for use on a specific subset of Pascal, while our tools operate on the *StandardRep*.

This paper has the following organization. The next section describes the all-du-paths criterion. We describe the tools used in the study in Section 3. These software tools include the programs that (1) generate a Prolog data base from a *StandardRep*, (2) compute all du-paths in a program, (3) remove redundant du-paths, and (4) estimate the number of test cases needed to meet the all-du-paths criterion. Section 4 describes the natural language text analysis system that is the object of this study. The case study results are described in Section 5 and the conclusions follow in Section 6. The Appendix contains tables of the case study results.

2 All-du-paths Criterion

The all-du-paths path criterion is based on data flow relationships. Variables are tracked from their points of definition to their points of use. A du-path is a program path that connects the definition of a variable to its use. By *definition* we mean the modification of the value associated with a variable via an assignment, input statement, or procedure invocation. A *use* is a reference of the value of a variable usually within an expression. In addition, cycles are restricted within a du-path. For testing to satisfy the all-du-paths criterion, the test data must cover all du-paths for all variables in a procedure. Thus, all acyclic paths between every definition-use pair must be tested.

Rapps and Weyuker define their family of criteria on a simple, formal programming language [RW85]. We use a rigorous specification of the all-du-paths criterion in terms of a language independent representation of imperative programs, the *StandardRep* [Sch88, BBC*88]. Our specification is consistent with the original definitions, and applies to any imperative programming languages that can be mapped to the *StandardRep*.

To incorporate procedures and functions in our definition, we determine which of the actual parameters are defined and which are used at the point of a call. In a procedure call, we assume that a variable which represents a call-by-reference parameter is defined. We also assume that the variables in an expression which represents a call-by-value parameter are used. All call-by-value formal parameters are assigned to local variables in the start node of the called procedure by the *StandardRep* generator [DBB86]. We also assume that all global variables accessed by the called procedure are used at the point of a procedure call, and all variables in the actual parameters of a function call are used.

Our specification makes use of the notion of "path subset criterion" [BHB86]. Consider a flowgraph $G = (N, E, s, t)$, where $E \subset N \times N, s \in N, t \in N$, and all nodes $x \in N$ lie on a path from s to t. Any path P from s to t is a *complete path*. A *path subset criterion* is a boolean function that, given a set of complete paths in a flowgraph, outputs *true* if and only if the set of paths *satisfies the criterion*. When using a path subset criterion as a testing criterion the set of paths is finite.

Let *AllDUPaths(G)* denote the set of all du-paths in flowgraph G, and let *all-du-paths(FS,G)* be a path subset criterion that determines if a particular set of complete paths, *FS*, includes all du-paths in G. Then *all-du-paths(FS,G)* is true if and only if every member of *AllDUPaths(G)* is included along some path in *FS*. The complete formal specification of the all-du-paths criterion is in [Sch88]. In this specification, *AllDUPaths* is defined in SPECS as an abstract function in terms of the *StandardRep*. Since we can generate a *StandardRep* from ISO Pascal [DBB86], our specification can be used to implement tools that operate on Pascal programs. As soon as *StandardRep* generators for other languages are implemented, our tools can be applied to programs in these languages.

3 Research Tools

Our research tools identify du-paths and count the minimum or near-minimum number of complete paths required to satisfy the all-du-paths criterion for a program unit (procedure or function). The analysis is performed in two distinct phases. In the first phase, a Prolog data base *(PDB)* is produced for each unit of a Pascal program. In the second phase, a Prolog program Count takes a *PDB* as input, finds all of the du-paths, and outputs a count of the number of du-paths, the number of non-redundant du-paths, and an estimate of the number of complete paths, or test cases, necessary to satisfy the criterion.

3.1 Producing the *PDB*'s for a Program

The *PDB*'s for a Pascal program are generated from a *StandardRep* [BBC*88]. The original *StandardRep* generator takes a Pascal program as input and produces the corresponding *StandardRep* as output [DBB86]. The output routines of the original generator were modified to produce a *PDB* for each program unit. A listing of the modified C output routines appears in [Sch88] and the original *StandardRep* generator is in [DBB86].

A *PDB* consists of an annotated flowgraph for one procedure or function. A procedure or function is represented as a *UnitRepType* in the *StandardRep*. A *PDB* contains just the information in a *UnitRepType* object that is needed for our analysis. The structure of a *PDB* may be specified in SPECS by the following type definitions. (In the following definitions, *DCP* stands for definitions, c-uses and p-uses.)

```
PDB = 3-tuple(
          Nodes : set of NodeID,
          DCP : set of DCPType,
          Edges : set of EdgeType)

DCPType = 4-tuple(
              NID : NodeID,
              D : set of VarID,
              C : set of VarID,
              P : set of VarID)
```

For an object X of type *PDB* for a program unit U, *Nodes(X)* and *Edges(X)* represent the nodes and edges in the corresponding flowgraph of U. Each element E in *DCP(X)* contains data flow information for node *NID(E)*, including the global definitions, $D(E)$, global c-uses, $C(E)$, and p-uses, $P(E)$.

The abstract operation that produces a *PDB* from a particular *UnitRepType* is formally specified in [Sch88]. A *PDB* is represented by Prolog lists. Figure 1 illustrates a *PDB* as represented with Prolog data objects for a program unit named "test".

Note that the terminal node t is not included in the list of nodes in the Prolog *PDB*, nor is any information for node t given. (t still appears in the edges, though). Since there are no global definitions or uses in t, t cannot contribute to any du-paths.

Also note that the global c-uses and p-uses in a Prolog *PDB* include not only variables, but constants as well. Constants are included because the implementation of the original *StandardRep* generator does not distinguish between a variable ID and a constant ID. Since a constant can never occur as a global definition, the constants that appear in the other lists do not contribute to any du-paths. In effect, these constants may be ignored.

A Prolog *PDB* is the input to the Prolog program Count which identifies the du-paths and computes the various path counting measures.

3.2 The Prolog Program Count

Count is implemented in Prolog. The built-in backtracking features of Prolog are well suited for graph searches, and allow our algorithms to be specified at a higher level than possible using a conventional programming language such as Pascal or C. However, by using Prolog, we sacrifice execution speed. This sacrifice is not significant except when examining large *PDB*'s.

```
nodes([s,1,2]).

global_defs(s,[input]).
global_c_uses(s,[]).
p_uses(s,[]).

global_defs(1,[x,y]).
global_c_uses(1,[y]).
p_uses(1,[x,y,2,3]).

global_defs(2,[stop,10,x,z]).
global_c_uses(2,[]).
p_uses(2,[stop,100]).

edge(s,1).
edge(s,2).
edge(1,2).
edge(2,t).
```

Figure 1: An example *PDB*.

We describe the algorithm in terms of the abstract representation of a *PDB*. There are four main steps in the algorithm:

1. Find all of the du-paths using the *PDB*. The number of du-paths is output.
2. Find the successor nodes for each node in the *PDB*.
3. Remove redundant du-paths found in Step 1. The number of remaining du-paths is output.
4. Determine the cardinality of a "small" set of complete paths that include all of the du-paths from Step 3. These complete paths correspond to (potential) test cases and the number of such paths is output.

3.2.1 Finding the du-paths

In this step, all du-paths of a program unit are identified. The following three specification expressions, which are defined in terms of the abstract *PDB* type, will be utilized in the discussion. They specify the set of global definitions, global c-uses and p-uses for a given node ID.

gdefs (P : PDB, NID : NodeID) as set of VarID
such that gdefs(P, NID) =
$$\{v \mid \exists x[x \in DCP(P) \wedge NID(x) = NID \wedge v \in D(x)]\}$$

cuses (P : PDB, NID : NodeID) as set of VarID
such that cuses(P, NID) =
$$\{v \mid \exists x[x \in DCP(P) \wedge NID(x) = NID \wedge v \in C(x)]\}$$

puses (P : PDB, NID : NodeID) as set of VarID
such that puses(P, NID) =
$$\{v \mid \exists x[x \in DCP(P) \wedge NID(x) = NID \wedge v \in P(x)]\}$$

To calculate the du-paths, all ordered node pairs i and j, where $i, j \in Nodes(PDB)$, are examined in turn for the du-paths between them. For a graph with n nodes, there are $n(n-1)$ distinct node pairs. We do not include the terminal node t in any node pairs.

The following sets are calculated for each ordered node pair:

$$c\_vars(i,j) = gdefs(PDB,i) \cap cuses(PDB,j)$$
$$p\_vars(i,j) = gdefs(PDB,i) \cap puses(PDB,j)$$

When both *c_vars* and *p_vars* are empty there are no du-paths between i and j. Such a pair is discarded and the next node pair is examined.

If either *c_vars* or *p_vars* is non-empty, Prolog conducts a depth-first search for a du-path between nodes i and j. As each new node k is examined along the path, *c_vars* and *p_vars* are recalculated as follows:

$$c\_vars(i,j) := c\_vars(i,j) - gdefs(PDB,k)$$
$$p\_vars(i,j) := p\_vars(i,j) - gdefs(PDB,k)$$

The recalculation is necessary because a definition of a variable v on a path from node i to node j makes the definition of v in i unusable. The redefinition of v "kills" the earlier definition of v. So if both sets become empty, the current node is discarded and backtracking occurs to search for an alternate path. If and when node j is reached, a du-path has been found and it is written to an output file.

After each du-path is found and written to the output file, FAIL is used to force Prolog to backtrack and find another du-path from the point at which it left off. Thus, *all* du-paths for a particular node pair are found. Backtracking is employed at two points: after each du-path is found and when a new node is encountered along a potential du-path that causes both *c_vars* and *p_vars* to become empty.

The du-paths appear in the output file as a Prolog list of lists:

```
du_paths([[1,2,3],
[1,2,4],
[1,3,4],
[1,2,4,5],
[1,3,4,5],
[6,5],
[6,5,6],
[6,5,7]]).
```

3.2.2 Node Successors

In this step we determine the successor nodes for each node in the node list of the PDB. A sequence of successor nodes for a particular node n_1 consists of all n_2 in the node list such that there exists a path from n_1 to n_2. Prolog searches the edges in such a way that the sequence of successor nodes is in nondecreasing order according to the lengths of the paths from n_1 to n_2. All of these sequences of successor nodes are written to the output file and accessed in the Step 4.

3.2.3 Condensing the du-paths

A number of the du-paths found in Step 1 must be eliminated to prevent the inclusion of one or more redundant complete paths. For example, if we have the du-paths [1,2,3,4,5] and [2,3] and test cases cause execution to traverse the first path, we have also traversed the second one. As a result, the second path may be eliminated without consequence.

To condense the list of du-paths, each du-path is examined and if it is "included" on another du-path in the list it is eliminated. The concept of one path (P1) being included on another path (P2) can be expressed by the following specification expression:

Included (*P1 : sequence of NodeID,*
P2 : sequence of NodeID) *as boolean*
such that Included $(P1, P2) \equiv$
$\exists i[1 \leq i \leq (length(P2) - length(P1) + 1) \wedge$
$\forall j[1 \leq j \leq length(P1) \Rightarrow P1_j = P2_{i+j-1}]]$

For example, [2,4,5], [4,5,8] and [5,8,9,10] are all included on [1,2,4,5,8,9,10], but [8,9,10,11,12] is not.

3.2.4 Counting complete paths

The final step of the Prolog algorithm estimates the fewest number of complete paths that include all of the du-paths. This part of the algorithm is described as follows. We start with a count of 1. We then "overlap" and "piece together" as many du-paths as possible along one complete path. The count is then incremented and each selected du-path is deleted from the list of du-paths. This process is repeated until the list of du-paths is empty. The final output of Count is the final value of the count. During this step the output file is accessed for the list of condensed du-paths found in Step 3 and the successor nodes found in Step 2.

For example, we begin by looking for a du-path that starts with the start node s. If [s,1,2,3,4,5] is initially selected, the next du-path we look for in the list should begin with the initial sequence of [1,2,3,4,5,...]. (There cannot be another du-path that begins with [s,1,2,3,4,5,...] due to the condense step.) If such a path does not exist, we try [2,3,4,5,...], then [3,4,5,...], etc. If we come to [5,...], we then start looking at the successors of node 5, trying to find a du-path that starts with the closest successor node. If, after trying all successor nodes, we have no luck, we increment the count and look again for a du-path that starts with the start node s. When we do select a du-path, the next du-path we look for should start with the tail of the selected path. The selected du-path is deleted from the list. The program terminates when the list of du-paths becomes empty.

The algorithm does not guarantee that the final value for Count is the minimum number of complete paths that include the du-paths. Suppose we are looking for a du-path that starts with a particular sequence of nodes, and that more than one such du-path exists. Count picks the first path it finds in the list. But perhaps one of the other choices would allow more du-paths to be included along the complete path. A lower value for Count could result.

Thus, the value of Count is dependent upon the order of the du-paths in the list. However, since Count tries to make the best choice for every selection, the algorithm is a reasonable one for finding a value close to the minimum number of complete paths required to include the du-paths.

4 Case Study Data: the NLTAS

A natural language text analysis system (NLTAS) is the software that is the data for our study. The NLTAS is used to analyze verbatim responses to open ended surveys used in marketing research. An expert analyst uses the system to identify, within natural language text, the words and phrases that correspond to a specified set of "meaning units." The NLTAS is a product of Iris Systems, Inc. and has been in commercial use since 1985. The system consists of five Pascal programs with a total of 143 subroutines (procedures and functions). The system has a total of 7,413 lines of code (including comments). Thus the average length of a subroutine is 52 lines of code. The longest subroutine is 367 lines of code. All but ten of the subroutines are shorter than 100 lines of code.

We generated a *StandardRep* from the original source code, and performed the analysis using the *StandardRep* of the system. The *StandardRep* is an abstraction of the code that contains the information necessary for our analysis, but hides proprietary details.

5 Results

For each procedure or function in the NLTAS we record the following data:

1. Number of lines of code (Lines) not including comments,
2. Number of nodes in the flowgraph representation of the program unit (Nodes),
3. Number of edges in the flowgraph (Edges),
4. Number of du-paths (Du-Paths),
5. Number of non-redundant or condensed du-paths (Condensed), and
6. Estimated minimal number of complete paths required to meet the all-du-paths criterion (Count).

The above measures for each of the program units are in the Appendix.

The most striking finding is that in 115 of the 143 subroutines (80%) the all-du-paths criterion can be met with ten or fewer complete paths. And, in 91% of the subroutines, the all-du-paths crition can be met with 25 or fewer complete paths. Figure 2 illustrates these results.

Only four subroutines or 2.8% require more than 100 complete paths. Two of these subroutines require the testing of more than a practical number of tests. One subroutine (A18) exhibits exponential behavior and requires the testing of at least 2^{32} complete paths. Another subroutine (A59) requires the testing of on the order of 10,000 complete paths. Due to machine and time limitations, exact counts of the required number of complete paths for subroutines A18 and A59 could not be computed.

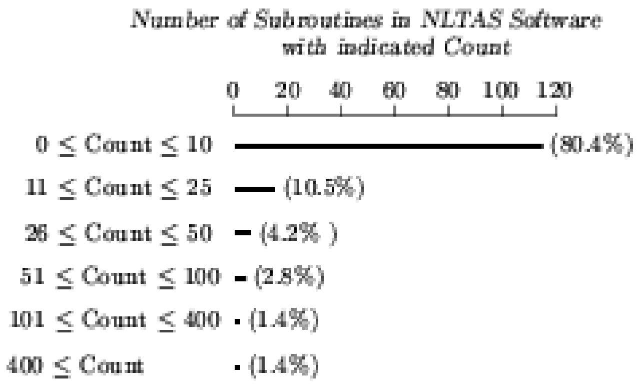

Figure 2: Required Number of Complete Paths

The results in the Appendix appear to indicate that the Count is dependent on subroutine length (Lines). However, the longest subroutine in the system (A68) with 367 lines of code has a Count of 76. An examination of A18 reveals the cause of the required exponential number of complete paths necessary in this case.

The code in A18 that causes the exponential result has the following structure:

```
Define X:  X := Y;
if $P_1$ then $S_1$;
if $P_2$ then $S_2$;
   ⋮
if $P_{32}$ then $S_{32}$;
Use X:  Y := F(X);
```

where X is not modified in statements S_1 through S_{32}. There are 2^{32} paths between the definition of X and the use of X and each path is a distinct du-path.

In the NLTAS, code with a structure similar to the forgoing is rare; only subroutine A18 requires an exponential number of complete paths. In almost all of the subroutines, the all-du-paths criterion is satisfied by testing a reasonable number of complete paths. These results indicate that the all-du-paths testing criterion can be used on most of the subroutines in the system.

6 Conclusions

In this paper, we describe a tool that estimates the number of test cases required to meet the all-du-paths testing criterion. We use this tool to empirically evaluate the practicality of the criterion. In the worst case, the all-du-paths criterion requires an exponential number of test cases. However, in this case study, the worst case scenario only occurs in one subroutine out of 143. Eighty percent of the subroutines would require ten or fewer test cases.

These results demonstrate that the all-du-paths criterion may be a more realistic criterion than the theoretical results indicate. The all-du-paths criterion should not be completely avoided because of the few subroutines that require an exponential number of test cases. Of course, the criterion is not practical for testing these subroutines and alternative testing strategies must be used. Note that the other data flow criteria of Rapps and Weyuker require fewer test cases than the all-du-paths criterion.

A tool similar to the Count program can be used to identify these anomalous subroutines and either recode them or use an alternative testing strategy. One could use a tool such as ASSET to assist in finding input data to meet the criterion. And Count can be used to predict how many test cases may be required.

Acknowledgment

We thank Iris Systems, Inc. for allowing us to use the natural language text analysis system as data for this case study. We appreciate the effort of Kyung-Goo Doh who implemented the *StandardRep* generator. Finally, we thank Albert Baker for his comments and suggestions.

References

[BBC87] A. Baker, J. Bieman, and P. Clites. Implications for formal specifications – results of specifying a software engineering tool. *Proc. IEEE Computer Society's Eleventh Annual International Computer Software & Applications Conference (COMPSAC87)*, 131–140, October 1987. Tokyo, Japan.

[BBC*88] J. Bieman, A. Baker, P. Clites, D. Gustafson, and A. Melton. A standard representation of imperative language programs for data collection and software measures specification. *The Journal of Systems and Software*, 8(1):13–37, January 1988.

[BHB86] Albert L. Baker, James W. Howatt, and James M. Bieman. Criteria for finite sets of paths that characterize control flow. *Proc. 19th Hawaii International Conference on System Sciences (HICSS-19)*, IIA:158–163, January 1986.

[CPRZ85] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A comparison of data flow path selection criteria. *Proc. 8th International Conference on Software Engineering*, 244–251, 1985.

[DBB86] K. Doh, J. Bieman, and A. Baker. *Generating a Standard Representation from Pascal Programs.* Technical Report TR 86-15, Dept. of Computer Science, Iowa State University, Ames, Iowa, 1986.

[FW85] P. G. Frankl and E. J. Weyuker. A data flow testing tool. *Proc. Softfair II*, December 1985.

[FW88] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Trans. Software Engineering*, 14(10):1483–1498, 1988.

[FWW85] P. G. Frankl, S. N. Weiss, and E. J. Weyuker. Asset: a system to select and evaluate tests. *Proc. IEEE Conference on Software Tools*, 72–79, April 1985.

[Hay87] I. Hayes (editor). *Specification Case Studies.* Prentice-Hall International, London, 1987.

[Hec77] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier, New York, 1977.

[Jon86] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, London, 1986.

[JW85] K. Jensen and N. Wirth. *Pascal User Manual and Report, Third Edition.* Springer-Verlag, New York, 3rd edition, 1985.

[KP78] B. W. Kernighan and P. J. Plauger. *The Elements of Programming Style, 2nd Ed.* McGraw-Hill, New York, 1978.

[Nta88] S. C. Ntafos. A comparison of some structural testing strategies. *IEEE Trans. Software Engineering*, 14(6):868–874, June 1988.

[RW85] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. Software Engineering*, SE-11(4):367–375, April 1985.

[Sch88] J. L. Schultz. *Measuring the Cardinality of Execution Path Subsets Meeting the All-DU-Paths Testing Criterion.* Master's project, Department of Computer Science, Iowa State University, Ames, IA, 1988.

[Wey84] E. J. Weyuker. The complexity of data flow criteria for test data selection. *Information Processing Letters*, 19:103–109, August 1984.

[WGM85] M. D. Weiser, J. D. Gannon, and P. R. McMullin. Comparison of structured test coverage metrics. *IEEE Software*, 2(2):80–85, March 1985.

Appendix: Table of Case Study Results

The following tables contain the results of the case study. The number of lines (Lines) for each program unit was obtained from the actual program code. The number of nodes (Nodes) and edges (Edges) for each unit was obtained from the corresponding PDB. (The node list in a PDB does not contain the terminal node, *t*, so the values for nodes in this report have been increased by one.) The value for the number of du-paths (DUP), condensed du-paths (Cond), and the number of complete paths required to satisfy the all-du-paths criterion (Count) were generated as described in Section 3.

Program A

Unit	Lines	Nodes	Edges	DUP	Cond	Count
A1	40	14	18	57	24	7
A2	41	18	23	47	20	7
A3	24	8	9	14	5	3
A4	43	15	20	35	17	10
A5	59	11	14	14	6	5
A6	50	21	28	83	32	12
A7	18	9	11	23	10	5
A8	10	8	9	24	12	3
A9	21	8	10	22	6	4
A10	53	16	22	67	24	17
A11	21	5	5	4	2	2
A12	24	14	19	18	7	7
A13	32	12	15	33	14	8
A14	32	16	20	74	25	15
A15	13	7	8	14	8	4
A16	47	12	15	28	11	5
A17	16	8	9	19	9	4
A18	252	118	183	[a]	[a]	[a]
A19	25	8	9	21	8	3
A20	10	3	2	1	1	1
A21	13	7	8	14	7	3
A22	47	12	17	28	11	6
A23	36	15	18	55	18	5
A24	57	15	18	50	17	6
A25	62	27	35	110	42	8
A26	24	8	9	19	9	3
A27	40	14	17	40	15	5
A28	24	6	6	11	4	2
A29	35	12	14	31	13	6
A30	53	23	31	96	39	16
A31	13	7	8	14	7	3
A32	19	10	12	25	11	5
A33	50	14	20	46	15	8
A34	40	20	26	72	26	10
A35	84	27	35	114	45	12
A36	25	6	6	10	4	2
A37	44	24	31	87	39	13
A38	35	14	17	54	18	5
A39	37	10	12	20	4	4
A40	19	6	6	9	4	2
A41	31	8	9	18	6	2
A42	20	8	9	27	9	3
A43	32	6	7	14	3	3
A44	68	32	48	172	59	44
A45	66	17	23	115	50	9
A46	100	40	53	267	93	33
A47	41	13	17	39	16	4
A48	96	26	36	117	40	24
A49	48	20	25	68	28	4
A50	76	22	29	94	34	7
A51	101	25	32	76	27	9
A52	19	7	8	7	3	3
A53	49	13	16	45	17	4
A54	18	5	5	7	5	3
A55	66	12	15	38	16	7
A56	31	8	9	21	10	3
A57	91	29	40	1071	576	336
A58	59	19	26	91	31	24
A59	309	92	128	10822	[b]	[b]
A60	38	12	15	55	25	6
A61	15	4	4	3	2	2
A62	23	9	11	10	4	4
A63	21	9	11	22	10	5
A64	11	5	5	7	5	3
A65	29	9	11	32	16	5
A66	33	11	14	39	14	7
A67	32	11	14	39	14	7
A68	367	71	110	554	376	76

[a]The number of du-paths, condensed du-paths, and Count is on the order of 2^{32} and is not practical to compute.

[b]The Prolog rule containing the du-paths is too large (2.07 Mbytes) for c-prolog to read. Our c-prolog implementation can read rules of a maximum size of 256 Kbytes.

Program B

Unit	Lines	Nodes	Edges	DUP	Cond	Count
B1	14	7	8	11	7	3
B2	97	22	30	306	166	36
B3	47	13	16	25	10	6
B4	47	12	15	39	14	3
B5	85	17	15	150	52	9
B6	55	26	34	91	38	18
B7	51	19	24	57	18	7
B8	29	10	12	24	10	4
B9	27	7	8	16	6	3
B10	22	6	7	12	5	4

Program C

Unit	Lines	Nodes	Edges	DUP	Cond	Count
C1	46	14	18	47	18	18
C2	138	31	44	286	152	60
C3	13	5	5	7	4	3
C4	73	22	31	118	47	33
C5	33	10	12	24	10	4
C6	9	3	2	0	0	1
C7	25	9	11	10	4	4
C8	13	3	2	1	1	1
C9	28	11	14	38	17	6
C10	20	10	12	34	11	4
C11	17	4	4	5	2	2
C12	21	8	10	29	8	8
C13	29	11	14	38	17	6
C14	21	10	12	34	11	4
C15	17	4	4	5	2	2
C16	28	7	8	11	3	3
C17	40	6	7	14	4	4
C18	26	10	12	26	10	5
C19	43	10	13	45	16	4
C20	31	9	11	33	10	6
C21	53	18	25	199	66	60
C22	53	18	25	199	66	60
C23	134	15	20	48	15	9

Program D

Unit	Lines	Nodes	Edges	DUP	Cond	Count
D1	33	11	14	39	14	7
D2	32	11	14	39	14	7
D3	37	23	29	76	34	7
D4	256	47	64	1338	406	295

Program E

Unit	Lines	Nodes	Edges	DUP	Cond	Count
E1	19	11	14	28	10	8
E2	12	3	2	0	0	1
E3	28	11	13	32	14	4
E4	43	15	20	35	17	10
E5	68	13	17	22	9	7
E6	62	16	22	63	22	14
E7	11	5	5	4	2	2
E8	24	13	19	18	7	7
E9	25	8	9	21	8	3
E10	10	3	2	1	1	1
E11	50	21	22	83	32	12
E12	13	7	8	14	7	3
E13	47	12	17	28	11	6
E14	34	12	14	41	16	4
E15	57	15	18	50	17	6
E16	34	17	21	58	21	8
E17	53	23	31	96	39	16
E18	13	7	8	14	7	3
E19	19	10	12	25	11	5
E20	50	14	20	46	15	8
E21	40	20	26	72	26	10
E22	84	27	35	108	43	12
E23	17	6	6	9	4	2
E24	26	5	6	12	3	3
E25	67	17	23	111	48	9
E26	100	40	53	267	93	33
E27	41	13	17	39	16	4
E28	68	19	26	54	21	12
E29	48	20	25	68	28	4
E30	78	22	29	94	34	7
E31	17	9	10	20	7	3
E32	14	8	9	19	7	3
E33	16	7	8	5	4	3
E34	39	12	15	44	21	6
E35	13	4	4	3	2	2
E36	14	8	9	24	9	3
E37	16	7	8	15	5	3
E38	215	18	23	156	62	10