An Empirical Evaluation (and Specification) of the All-du-paths Testing Criterion (Extended Version)

James M. Bieman
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523 USA
(303) 491-7096
bieman@cs.colostate.edu

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

Abstract

The all-du-paths structural testing criterion is one of the most discriminating of the data flow testing criteria. Unfortunately, in the worst case, the criterion requires an intractable number of test cases. In a case study of an industrial software system, we find that the worst case scenario is rare. Eighty percent of the subroutines require ten or fewer test cases. Only one subroutine out of 143 requires an intractable number of tests. However, the number of required test cases becomes tractable when using the all-uses criterion. This paper includes a formal specification of both the all-du-paths criterion and the software tools used to estimate a minimal number of test cases necessary to meet the criterion.

Keywords: Software testing, software measures formal specifications, data flow analysis.

Department of Computer Science Technical Report #CS–91–105
Colorado State University (Revision of Technical Report #CS–89–118)

A version of this paper was published in ***IEE/BCS Software Engineering Journal*****, 7(1):43-51, Jan. 1992.**

This research was partially supported by the NATO Collaborative Research Grants Program under RG. 0343/88.

1 Introduction

A major focus of software testing research is the derivation of 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 which are more capable of uncovering errors tend to require a greater number of test cases. Consider criteria which require testing specific sets of program control flow paths. Paths with particular properties are selected for testing. The "all branches" criterion requires the testing of all branches in a program, while the "all statements" criterion requires that the test data cause all statements to be executed at least once. Myers describes statement coverage as a "weak criterion" and demonstrates the advantages of branch coverage over statement coverage [1]. The "all branches" criterion can uncover more errors than the "all statements" criterion and usually requires more tests. The most discriminating path-based criterion is the "all paths" criterion which requires the testing of all possible program paths and requires an infinite number of test cases in programs with loops. Since we want testing to be completed eventually, any practical and useful criterion must be met with a finite and acceptably small number of test cases.

A number of published analytical studies compare various structural testing criteria [2, 3, 4, 5, 6]. These studies examine the inclusion orderings and worst case complexity of the criteria. Criterion A *includes* criterion B if and only if any test data and program that satisfies criterion A also satisfies criterion B. For example, 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 meet 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, where n is some measure of program size [6]. Such strategies are most likely to require a reasonable and practical number of test cases. 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.

Rapps and Weyuker define a family of path selection criteria based on data flow relationships [5]. These criteria focus on the program paths that connect the definitions and uses of variables (du-paths). Consider the Pascal binary search procedure from Dromley [7] shown in Figure 1. The binary search procedure has the flowgraph shown in Figure 2. For each basic block represented by flowgraph nodes, the variables defined and referenced and the program code are shown in Table 1. A distinction is made between variable uses within computations (c-uses) and uses within predicates or decisions (p-uses). This distinction is made because p-uses are associated with the out-edges from predicate nodes rather than the nodes themselves. All of the du-paths in the binary search procedure are shown in Table 2. Testing all of the du-paths requires that each branch within the while loop be executed as the last iteration (du-paths $\langle 4,2,6\rangle$ and $\langle 5,2,6\rangle$) and as a non-last iteration (du-paths $\langle 4,2,3\rangle$ and $\langle 5,2,3\rangle$). The branch coverage criterion requires testing each edge at least once and does not require testing all of these du-paths.

Of the criteria defined by Rapps and Weyuker, the all definition/use criterion (all-du-paths) is the "strongest". The all-du-paths criterion requires that the test data exercise all du-paths in a program. This criterion includes all of the other data flow criteria of Rapps and Weyuker and requires the greatest number of paths in a program to be tested. Thus, the all-du-paths criterion should theoretically be the most effective of these data flow criteria in discovering errors.

The all-du-paths criterion may require a large number of test cases. In the worst case, it can take an enormous number of test cases. 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 [2]. Note that data flow

```
procedure binarysearch (a: nelements;
                        n,x: integer;
                        var found: boolean);
  var lower, upper, middle: integer;
  begin
   lower := 1;
   upper := n;
   while lower < upper do
     begin
       middle := (lower + upper) div 2;
       if x > a[middle]
       then lower := middle + 1
       else upper := middle
     end;
   found := (a[lower]=x)
  end;
```

Figure 1: Pascal Binary Search Procedure

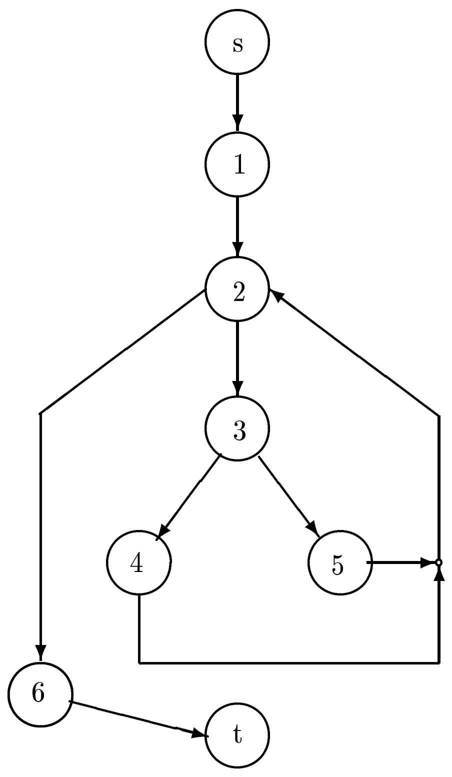

Figure 2: Binary Search Flowgraph

Block	Code	c-uses	p-uses	definitions
s	input parameters a, n, x;			a,n,x,

1	lower := 1;
	upper := n;
		n		lower,
				upper

2	while lower < upper do		lower, upper	

3	middle := (lower + upper) div 2;
	if x > a[middle]
		lower, upper	x, a, middle	middle

4	then lower := middle + 1	middle		lower
5	else upper := middle	middle		upper
6	found := (a[lower]=x)	a, lower, x		found
t				

Table 1: Binary Search Basic Blocks, Definitions, & Uses

DU Paths
⟨s,1⟩
⟨s,1,2,3,4⟩
⟨s,1,2,3,5⟩
⟨s,1,2,6⟩
⟨1,2,3⟩
⟨1,2,6⟩
⟨3,4⟩
⟨3,5⟩
⟨4,2,3⟩
⟨4,2,6⟩
⟨5,2,3⟩
⟨5,2,6⟩

Table 2: Binary Search Definition/Use Paths

testing criteria of Ntafos [8], and the criteria of Laski and Korel [9] strictly subsume the all-du-paths criterion with the modifications suggested by Clarke *et al.* [10]. Thus, these "stronger" criteria may require even more test cases than the all-du-paths criterion.

A testing strategy that requires an intractable number of test cases is not realistic. However, the actual number of test cases can be much less than the worst case. For example the all-du-paths criterion can be satisfied for the binary search procedure with eight or fewer tests since several du-paths can be covered on one test. For example, the du-paths in Table 2 can be covered by three test cases which cover the complete paths ⟨s,1,2,3,4,2,3,4,2,6,t⟩, ⟨s,1,2,3,5,2,3,5,2,6,t⟩, and ⟨s,1,2,6,t⟩. Only empirical studies can determine whether the worst case scenario is common.

One empirical study of the du family of criteria was conducted by Weyuker [11, 12]. This study determined how many test cases were required to meet criteria using a suite of programs collected by Kernighan and Plauger [13]. The study was performed using the ASSET system, described later in this section, and required considerable effort by human testers. Four human testers developed and ran sets of test cases on the suite of programs. Out of the suite of programs, only those with 5 or more decision statements were included in the study. A total of 29 programs were included. Weyuker found that the all-du-paths criterion required approximately the same number of test cases as the all-uses criterion. The all-uses criterion required far fewer than the worst case analysis. No cases requiring an intractable number of tests were reported. The programs in our study are production software in use commercially. Our study includes 143 subroutines, of which 34 subroutines have 5 or more decisions, 7 suboutines have more than 10 decisions, and three subroutines have 35 or more decisions. One program contains 65 decisions. Weyuker does not report the sizes of individual procedures.

We examined an industrial software system to determine how many test cases are actually required to satisfy the all-du-paths criterion. In this case study, we first determine the du-paths in the software system. 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 to terminal node of a program flowgraph) that covers all of the du-paths [14]. 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 intractable number of complete paths. One other subroutine requires on the order of 10,000 complete paths. Thus, the all-du-paths criterion can be used to test most of the subroutines in the software system under study. In the two anomalous subroutines, the weaker all-uses criterion can be met with a reasonable number of complete paths.

Our contributions include the specification of a software measurement tool that estimates a minimal number of test cases required to meet the all-du-paths criterion. We use this tool to determine the estimated number of tests required to meet the all-du-paths criterion. The basis for our measurement tools is a formal specification of the all-du-paths criterion. The specification is written in a language-independent fashion so that our tools can be easily adapted to apply to programs in many standard imperative languages.

The number of complete paths needed to meet the criterion is an estimate of the number of test cases required. Some complete paths of a program may be infeasible — no input data exists that can cause such a path to be executed. In addition, some du-paths may be infeasible. Thus, it might be impossible to select test data that satisfies the particular criterion. As are many problems in testing, determining whether a particular path is feasible is undecidable [15, 16, 17]. Frankl and Weyuker suggest the use of heuristics to identify infeasible paths [18]. 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.

In developing our research tools, we rigorously define the all-du-paths criterion using the SPECS specification language [19]. The criterion is specified in terms of a language-independent representation *(StandardRep)* of imperative programs [20]. Our formal definition of the all-du-paths criterion is applicable to *any* imperative language that can be mapped to the representation. Because our empirical study is based on a formal specification, the research can be repeated by others. Note that the *StandardRep* can be used to specify a number of useful software structural measures.

We utilize a tool [21] that generates a *StandardRep* from an ISO Standard Pascal [22] 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 [23, 24]. 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 a 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. Section 2 presents the formal definition of the all-du-paths criterion. This formal definition is the basis of our analysis tools. We formally specify 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. Appendix A contains tables of the case study results and Appendix B includes the Prolog programs used.

2 Specification of the All-du-paths Criterion

The software tools used in this study are specified in the SPECS specification language [19]. SPECS is an abstract model specification language and is similar to VDM's meta-iv [25] and Z [26]. SPECS specifications are defined as abstract data types in terms of mathematical sets, sequences, labelled tuples, integers, booleans, and operations on these primitives. A SPECS specification consists of an abstract domain specification and the specification of operations on objects in the domain. SPECS type declarations use a syntax similar to Pascal. The operations are specified using pre and post conditions in first order predicate calculus (FOPC). Our tools use an abstract representation of imperative programs as the abstract domain. The all-du-paths criterion and other software tools are specified as operations on the domain.

2.1 A Representation of Programs

The abstract domain used in specifying the all-du-paths criterion and other software tools incorporates concepts that are common to imperative language programs — control flow, data dependency, and procedure interfaces. This program representation is the *StandardRep* of Bieman et al [20]. Because we use the *StandardRep* as the basis for our specification rather than the source language, our specification applies to programs in any imperative language which can be mapped to the *StandardRep*.

In a *StandardRep*, a program is represented by a set of program units, where each program unit represents a procedure or function.

$$\text{StandardRep} \quad = \quad \text{set of UnitRep}$$

Each program unit has its own internal structure and a specific interface with the rest of the program.

$$\text{UnitRep} \quad = \quad \text{tuple(Interface: Header, UFS: UnitFlowStructure)}$$

The *Interface* contains the unique name of the program unit and the information necessary to determine the data interface with the rest of the program — the procedure or function name, the list of formal parameters, and the set of global variables which are used or defined by the program unit.

```
Header  =  tuple(
             UnitName: UnitID,
             FormalParams: sequence of VarID,
             Globals: set of VarID)
```

Program unit control flow is modeled by a flowgraph in which nodes represent basic blocks and edges represent transfers of control between blocks.

```
UnitFlowStructure  =  tuple(
                        Nodes: set of NodeType,
                        Edges: set of EdgeType,
                        Start: NodeID,
                        Terminal: NodeID)
```

```
EdgeType  =  tuple(FromNode: NodeID, ToNode: NodeID)
```

A basic block consists of code that is always executed in order starting with the first token and ending with the last token. Program unit data flow is represented as the sequence of definitions and uses in the nodes that represent basic blocks. Following Hecht [27], a variable *definition* is source program code that, when executed, can (potentially) modify the value stored by a program variable; a variable *use* or *reference* is code that, when executed, references the value stored by a program variable. Information concerning variable definitions and uses are embedded within the nodes. Nodes also contain information about invoked procedures and functions. A distinction is made between uses in definitions and uses in predicates. The characterization of a node consists of four parts: a node identifier, a list of variable definitions, and a list of predicate uses.

```
NodeType  =  tuple(
               NID: NodeID,
               LocalDefinitions: sequence of DefinitionType,
               Predicate: ExpType)
```

Nodes in the *StandardRep* as defined in [20] have an additional component to record the occurrences of operators and operands. This component is omited here since the information is not needed to define the all-du-paths criterion.

A definition of a variable occurs when either an assignment is made to that variable, or the variable is defined by a procedure call.

```
DefinitionType  =  SimpleDefinition | ProceduralDefinition
```

A *SimpleDefinition* has two components: the name of the variable being defined, and the list of items used in the definition. A procedure call is represented by the procedure name and the sequence of actual parameters. The representation of a procedure call, combined with the control flow and data dependency information in the *UnitFlowStructure* of a called procedure, allows us to deduce potential data dependencies resulting from the call.

SimpleDefinition = tuple(DefinedVariable: VarID, Expr: ExpType)

ProceduralDefinition = tuple(
ProcName: UnitID,
ActualParams: sequence of ExpType)

For our purposes, an expression results in a particular sequence of uses. The order is determined by the parsing.

ExpType = sequence of ExprComponent

Each item used in a *SimpleDefinition* may have any of three forms: the item may be a variable, a constant, or a function call.

ExprComponent = VarID | ConstID | FunctionUse

FunctionUse = tuple(
FunctionName: UnitID,
ActualParams: sequence of ExpType)

Fields of *StandardRep* objects are referenced in a specification in the following manner:

- Given an object *UFS* of type *UnitFlowStructure*, the nodes (*set of NodeType*) may be referenced with *Nodes(UFS)*. Similarly, the edges (*set of EdgeType*) may be referenced with *Edges(UFS)*.
- Given an object *N* of type *NodeType*, the node ID may be referenced with *NID(N)*, the local definitions with *LocalDefinitions(N)*, and the predicate with *Predicate(N)*.
- Given an object *U* of type *UnitRep*, the name of the unit may be referenced with *UnitName(Interface(U))*, and the formal parameters with *FormalParams(Interface(U))*.
- Given a sequence S, the first item may be referenced with S_1, the i^{th} item with S_i, and the last item with $S_{length(S)}$. The expression *header(S)* refers to the sequence without the last item and *trailer(S)* refers to the sequence without the first item.

We can generate a *StandardRep* from ISO Standard Pascal programs using the generator implemented by Doh [21]. The mapping from Pascal to the representation is formally specified by Bieman et al [20]. The output from the generator is used as input to the analysis tools used in this research. We specify the all-du-paths criterion as an operation on *StandardRep* objects.

2.2 All-du-paths Criterion

The all-du-paths criterion is based on data flow relationships. Variables are tracked from their points of definition to their points of use. All paths between every definition-use pair are selected.

Rapps and Weyuker define their family of criteria on a simple, formal programming language [5]. We rigorously define the all-du-paths criterion on the *StandardRep* in a manner consistent with the original definitions. In doing so, we have extended the definitions to apply to any imperative programming languages that can be mapped to the *StandardRep*.

To incorporate procedures and functions in our definition, we need to 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 [21]. 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" [28]. 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*. In the remainder of this section we specify *AllDUPaths* in SPECS as an abstract function in terms of the *StandardRep*.

The SPECS specification of *AllDUPaths* has a Pascal-like function header, and pre and post conditions expressed in first order predicate calculus (FOPC). The post-condition utilizes named expressions, which allow us to modularize complex post-condition expressions and improve readability. *AllDUPaths* is defined in a "top-down" manner—the post-condition is defined using named expressions, each named expression is defined in an expression definition, and expression definitions may also have named expressions.

The abstract function *AllDUPaths* and some of its associated expression definitions utilize the following additional type definition:

$$type\ PathType = sequence\ of\ NodeType$$

Definition 2.1 The function *AllDUPaths* has objects of type *UnitRep* and *StandardRep* as formal parameters. The *StandardRep* parameter is needed to determine definitions and uses resulting from procedure calls. This function generates the set of all du-paths in *UR*.

```
function AllDUPaths(UR : UnitRep,
                    SR : StandardRep) : set of PathType
   pre: UR ∈ SR
   post: AllDUPaths(UR,SR) = CDUPaths(UFS(UR),SR) ∪ PDUPaths(UFS(UR),PR)
```

Given the Pascal binary search procedure in Figure 1, *AllDUPaths* specifies the set of du-paths in Table 2.

The named expressions *CDUPaths* and *PDUPaths* are also defined formally. *CDUPaths* specifies the set of all *c-paths* — du-paths with a computation use or *c-use* of a variable in the last node of each path. A c-use is a variable reference which appears on the right-hand side of an assignment statement or in a procedure or function call. *PDUPaths* specifies the set of all *p-paths* — du-paths with a predicate use or *p-use* of a variable in the second-to-last node of each path. A predicate use appears in a conditional statement, such as an "if" or "while" statement. We may think of a predicate use as attached to the edges following the predicate in a flowgraph.

Definition 2.2 *CDUPaths* specifies the du-paths where for any variable v that is globally defined in the first node, v is not defined between the first and last nodes in the path, and the last node contains a global c-use of v. Such a path must also be a simple path — a path with no embedded cycles.

```
CDUPaths (UFS : UnitFlowStructure,
          SR : StandardRep        ) as set of PathType
   such that
      CDUPaths(UFS,SR) = $\{P : PathType \mid SimplePath(P, UFS) \wedge$
                                          $\exists v[GlobalDefs(P_1, v, SR) \wedge$
                                             $DefClearPath(P, v, SR) \wedge$
                                             $GlobalCUses(P_{length(P)}, v, SR)]\}$
```

The set of CDUPaths in the binary search example is { ⟨s,1⟩, ⟨s,1,2,6⟩, ⟨1,2,3⟩, ⟨1,2,6⟩, ⟨3,4⟩, ⟨3,5⟩, ⟨4,2,3⟩, ⟨4,2,6⟩, ⟨5,2,3⟩ } .

Definition 2.3 *PDUPaths* specifies the du-paths where for any variable v that is globally defined in the first node, v is not defined between the first and the last nodes in the path, and the second-to-last node contains a p-use of v. The last node can be any successor of the second-to-last node. All nodes except the last must comprise a loop free path.

```
PDUPaths (UFS : UnitFlowStructure,
          SR : StandardRep        ) as set of PathType
   such that
      PDUPaths(UFS,SR) = $\{P : PathType \mid LoopFreePath(header(P), UFS) \wedge$
                              $\exists v[GlobalDefs(P_1, v, SR) \wedge$
                                 $DefClearPath(P, v, SR) \wedge$
                                 $PUses(P_{length(P)-1}, v) \wedge$
                                 $P_{length(P)} \in Nodes(UFS) \wedge$
                                 $(NID(P_{length(P)-1}), NID(P_{length(P)})) \in Edges(UFS)]\}$
```

The set of PDUPaths in the binary search example is {⟨s,1,2,3,4⟩, ⟨s,1,2,3,5⟩, ⟨1,2,3⟩, ⟨1,2,6⟩, ⟨3,4⟩, ⟨3,5⟩, ⟨4,2,3⟩, ⟨4,2,6⟩, ⟨5,2,3⟩, ⟨5,2,6⟩}.

SimplePath, GlobalDefs, DefClearPath, etc., are also named expressions in the definitions for *CDUPaths* and *PDUPaths*. These named expressions are also specified using expression definitions.

Definition 2.4 A simple path in a *UnitFlowStructure* is a path in which no two nodes are the same, except possibly the first and last.

```
SimplePath (P : PathType,
            UFS : UnitFlowStructure) as boolean
```

such that

SimplePath(P,UFS) $\equiv (length(P) \geq 2 \wedge$
$\forall i[1 \leq i \leq length(P) \Rightarrow P_i \in Nodes(UFS)] \wedge$
$\forall i[1 \leq i < length(P) \Rightarrow (NID(P_i), NID(P_{i+1})) \in Edges(UFS)] \wedge$
$\forall m \forall n[1 \leq m < length(P) \wedge 1 \leq n < length(P)$
$\wedge m \neq n \Rightarrow P_m \neq P_n] \wedge$
$\forall m \forall n[2 \leq m \leq length(P) \wedge 2 \leq n \leq length(P)$
$\wedge m \neq n \Rightarrow P_m \neq P_n])$

Definition 2.5 A loop free path in a *UnitFlowStructure* is a path in which all nodes are distinct.

LoopFreePath (P : PathType,
UFS : UnitFlowStructure) as boolean

such that

LoopFreePath(P,UFS) $\equiv (\forall i[1 \leq i \leq length(P) \Rightarrow P_i \in Nodes(UFS)] \wedge$
$\forall i[1 \leq i < length(P) \Rightarrow (NID(P_i), NID(P_{i+1})) \in Edges(UFS)] \wedge$
$\forall m \forall n[1 \leq m \leq length(P) \wedge 1 \leq n \leq length(P)$
$\wedge m \neq n \Rightarrow P_m \neq P_n])$

Definition 2.6 *GlobalDefs* determines whether or not a variable, v, is defined in a given node. Such a new value of v may be used in other nodes.

GlobalDefs (N : NodeType,
v : VarID,
SR : StandardRep) as boolean

such that

$GlobalDefs(N, v, SR) \equiv v \in Defs(LocalDefinitions(N), SR)$

Definition 2.7 *GlobalCUses* determines whether there exists a definition in the sequence of definitions in a node such that v is used, and v is not defined in any previous definition of the node.

GlobalCUses (N : NodeType,
v : VarID,
SR : StandardRep) as boolean

such that

$GlobalCUses(N, v, SR) \equiv \exists i[1 \leq i \leq length(LocalDefinitions(N)) \wedge$
$v \in CUses(LocalDefinitions(N)_i, SR) \wedge$
$v \notin Defs(\langle LocalDefinitions(N)_1, ..., LocalDefinitions(N)_{i-1}\rangle, SR)]$

Definition 2.8 *PUses* uses *ProcessExpType* to find the set of p-uses in a node, then determines whether or not v is in this set.

PUses (N : NodeType,
v : VarID) as boolean

such that

$PUses(N, v) \equiv v \in ProcessExpType(Predicate(N))$

Definition 2.9 *Defs* determines the set of all variables which are defined in a given sequence of definitions. If the first definition is a simple definition, the left-hand side of the assignment statement is included in the returned set. If it is a procedure call, *ProcDefs* finds the defined variables in the call. This expression definition is defined recursively to include the rest of the definitions in the sequence.

```
Defs(Dseq : sequence of DefinitionType,
    SR : StandardRep                    ) as set of VarID
  such that
      $Dseq = \langle\rangle \Rightarrow$
          $Defs(Dseq, SR) = \{\}$
  $\wedge$  $Dseq \neq \langle\rangle \Rightarrow$
          $Dseq_1 \overset{type}{\in} SimpleDefinition \Rightarrow$
              $Defs(Dseq, SR) = DefinedVariable(Dseq_1) \cup Defs(trailer(Dseq), SR)$
      $\wedge$  $Dseq_1 \overset{type}{\in} ProceduralDefinition \Rightarrow$
              $Defs(Dseq, SR) = ProcDefs(Dseq_1, SR) \cup Defs(trailer(Dseq), SR)$
```

Thus, the *Defs* for the sequence of definitions *lower := 1; upper := n;* in Block 1 of the binary search procedure is {*lower,upper*}.

Definition 2.10 *CUses* specifies the set of variables which are used in either the right side of an assignment statement or in a procedure call.

```
CUses(D : DefinitionType,
      SR : StandardRep  ) as set of VarID
  such that
      D $\overset{type}{\in} SimpleDefinition \Rightarrow$
          $CUses(D, SR) = ProcessExpType(Expr(D))$
  $\wedge$  D $\overset{type}{\in} ProceduralDefinition \Rightarrow$
          $CUses(D, SR) = ProcUses(ActualParams(D), SR)$
```

The CUses set for the definition *middle := (lower + upper) div 2* in the binary search procedure is {*lower,upper*}.

Definition 2.11 A definition clear path with respect to a variable v is a path in which all nodes other than the first and last do not have a definition of v.

```
DefClearPath (P : PathType,
              v : VarID,
              SR : StandardRep) as boolean
  such that
      DefClearPath(P,v,SR) $\equiv$
          $\forall i[1 < i < P_{length(P)} \Rightarrow v \notin Defs(LocalDefinitions(P_i), SR)]$
```

Definition 2.12 *ProcessExpTypeSeq* specifies the set of all variables occurring in a sequence of expressions, or *ExpTypes*. Each *ExpType* in the sequence is analyzed by *ProcessExpType*. *ProcessExpTypeSeq* is defined recursively to include the rest of the expressions in the sequence.

```
ProcessExpTypeSeq(ETseq : sequence of ExpType) as set of VarID
  such that
      ETseq = $\langle\rangle \Rightarrow$
          $ProcessExpTypeSeq(ETseq) = \{\}$
  $\wedge$  ETseq $\neq \langle\rangle \Rightarrow$
          $ProcessExpTypeSeq(ETseq) = ProcessExpType(ETseq_1) \cup$
                                      $ProcessExpTypeSeq(trailer(ETseq))$
```

Definition 2.13 *ProcessExpType* specifies the set of all variables occurring in an expression, which can be viewed as either an *ExpType* or a sequence of *ExprComponent*. The type of the first item in the sequence is determined and the corresponding result is obtained. This expression definition is defined recursively to include the rest of the sequence.

```
ProcessExpType(ECseq : sequence of ExprComponent) as set of VarID
    such that
        ECseq = $\langle\rangle \Rightarrow$
            ProcessExpType(ECseq) = $\{\}$
    $\wedge$  ECseq $\neq \langle\rangle \Rightarrow$
            $(ECseq_1 \overset{type}{\in} VarID \Rightarrow$
                ProcessExpType(ECseq) = $ECseq_1 \cup ProcessExpType(trailer(ECseq)))$
        $\wedge$  $(ECseq_1 \overset{type}{\in} ConstID \Rightarrow$
                ProcessExpType(ECseq) = $ProcessExpType(trailer(ECseq)))$
        $\wedge$  $(ECseq_1 \overset{type}{\in} FunctionUse \Rightarrow$
                ProcessExpType(ECseq) = $ProcessExpTypeSeq(ActualParams(ECseq_1)) \cup$
                                        $ProcessExpType(trailer(ECseq)))$
```

Definition 2.14 *ProcDefs* specifies the set of all variables defined in a procedure call. The actual parameters, formal parameters and the sequence of definitions in the start node of the called procedure are evaluated by the named expression *ProcessProcDefs*.

```
ProcDefs(PD : ProceduralDefinition,
            SR : StandardRep            ) as set of VarID
   such that
   ProcDefs(PD,SR) =
      $\{v : VarID \mid \exists u \exists n [u \in SR \wedge$
                          $UnitName(Interface(u)) = ProcName(PD) \wedge$
                          $n \in Nodes(UFS(u)) \wedge$
                          $NID(n) = s \wedge$
                          v $\in ProcessProcDefs(ActualParams(PD),$
                                              $FormalParams(Interface(u)),$
                                              $LocalDefinitions(n))]\}$
```

Definition 2.15 *ProcessProcDefs* examines each actual parameter-formal parameter matching of a procedure call to determine which variable(s) in the actual parameter are defined. This expression is defined recursively to include all of the parameters. If the formal parameter of the match in question is not used in the start node of the called procedure, the corresponding actual parameter is a call-by-reference and is added to the result. Otherwise, the result is not affected by this particular parameter matching.

```
ProcessProcDefs (AP : sequence of Exptype,
                 FP : sequence of VarID,
                 LD : sequence of DefinitionType) as set of VarID
 such that
 $AP = \langle\rangle \Rightarrow$
   $ProcessProcDefs(AP, FP, LD) = \{\}$
$\wedge AP \neq \langle\rangle \Rightarrow$
```

$(FP_1 \notin CUses(LD) \Rightarrow$
$ProcessProcDefs(AP, FP, LD) = ProcessExpType(AP_1) \cup$
$ProcessProcDefs(trailer(AP), trailer(FP), LD))$
$\wedge (FP_1 \in CUses(LD) \Rightarrow$
$ProcessProcDefs(AP, FP, LD) =$
$ProcessProcDefs(trailer(AP), trailer(FP), LD))$

Definition 2.16 *ProcUses* is similar to *ProcDefs*, but specifies the set of all variables used rather than defined in a procedure call. The actual parameters, formal parameters and the sequence of definitions in the start node of the called procedure are passed to the expression definition *ProcessProcUses*, which does the actual analysis. Included in the result of *ProcUses* are all global variables accessed in the called procedure.

```
ProcUses(PD : ProceduralDefinition,
         SR : StandardRep           ) as set of VarID
 such that
 $ProcUses(PD, SR) = \{v : VarID \mid \exists u \exists n[u \in SR \wedge$
                    $UnitName(Interface(u)) = ProcName(PD) \wedge$
                    $n \in Nodes(UFS(u)) \wedge$
                    $NID(n) = s \wedge$
                    $v \in (ProcessProcUses(ActualParams(PD),$
                                           $FormalParams(Interface(u)),$
                                           $LocalDefinitions(n)) \cup$
                        $Globals(Interface(u)))]\}$
```

Definition 2.17 *ProcessProcUses* examines each actual parameter-formal parameter matching of a procedure call to determine which variable(s) in the actual parameter are used. This expression definition is defined recursively to include all of the parameters. If the formal parameter of the match in question is used in the start node of the called procedure, the corresponding actual parameter expression is a call-by-value and all of its variables are added to the result. Otherwise, the result is not affected by this particular parameter matching.

```
ProcessProcUses (AP : sequence of Exptype,
                 FP : sequence of VarID,
                 LD : sequence of DefinitionType) as set of VarID
 such that
  $AP = \langle\rangle \Rightarrow$
   $ProcessProcUses(AP, FP, LD) = \{\}$
 $\wedge AP \neq \langle\rangle \Rightarrow$
   $(FP_1 \in CUses(LD) \Rightarrow$
     $ProcessProcUses(AP, FP, LD) = ProcessExpType(AP_1) \cup$
                                   $ProcessProcUses(trailer(AP), trailer(FP), LD))$
   $\wedge (FP_1 \notin CUses(LD) \Rightarrow$
     $ProcessProcUses(AP, FP, LD) =$
                                   $ProcessProcUses(trailer(AP), trailer(FP), LD))$
```

The foregoing formal specification of the all-du-paths testing criterion is used to specify and design the research tools used in the case study.

3 Research Tools

This section describes the tools used to identify du-paths and count the minimum or near-minimum number of complete paths required to satisfy the all-du-paths path selection criterion for a program unit. 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* [20]. The original *StandardRep* generator takes a Pascal program as input and produces the corresponding *StandardRep* as output [21]. 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 [29] and the original *StandardRep* generator is in [21].

A *PDB* consists of an annotated flowgraph for one program unit, or *UnitRepType.* A *PDB* contains the information needed for our analysis. The structure of a *PDB* may be represented abstractly 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 specified as follows:

```
function ProducePDB (UR : UnitRepType,
                     SR : StandardRep) : PDB
    pre: UR ∈ SR
    post:
        $Nodes(ProducePDB(UR,SR)) =$
        $\{x : NodeID \mid \exists n[n \in Nodes(UFS(UR)) \wedge NID(n) = x]\}$
    $\wedge$  $DCP(ProducePDB(UR,SR)) = \{x : DCPType \mid \exists n[n \in Nodes(UFS(UR)) \wedge$
                                      $NID(n) = NID(x) \wedge$
                                      $D(x) = \{v \mid GlobalDefs(n,v,SR)\} \wedge$
```

$$C(x) = \{v \mid GlobalCUses(n, v, SR)\} \wedge$$
$$P(x) = \{v \mid PUses(n, v, SR)\}]\}$$
$$\wedge \quad Edges(ProducePDB(UR, SR)) = Edges(UFS(UR))$$

GlobalDefs, GlobalCUses and *PUses* are given in Definitions 2.6, 2.7 and 2.8, respectively. A *PDB* is represented by Prolog "facts" using Prolog lists. Figure 3 illustrates a *PDB* as represented with Prolog data objects for the binary search procedure.

The terminal node t is not included in the list of nodes in the Prolog *PDB*, since there are no global definitions or uses in t.

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 is 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. The Prolog code for Count is listed in Appendix B.

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 as defined in Section 2 are found. The following three expression definitions, 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
$$puses(P, NID) = \{v \mid \exists x[x \in DCP(P) \wedge NID(x) = NID \wedge v \in P(x)]\}$$

```
nodes([s,1,2,3,4,5,6]).

global_defs(s,[a,n,x]).
global_c_uses(s,[]).
p_uses(s,[]).

global_defs(1,[lower,upper]).
global_c_uses(1,[n]).
p_uses(1,[]).

global_defs(2,[]).
global_c_uses(2,[]).
p_uses(2,[lower,upper]).

global_defs(3,[middle]).
global_c_uses(3,[lower,upper]).
p_uses(3,[x,a,middle]).

global_defs(4,[lower]).
global_c_uses(4,[middle]).
p_uses(4,[]).

global_defs(5,[upper]).
global_c_uses(5,[middle]).
p_uses(5,[]).

global_defs(6,[found]).
global_c_uses(6,[a,lower,x]).
p_uses(6,[]).

edge(s,1).
edge(1,2).
edge(2,3).
edge(3,4).
edge(3,5).
edge(4,2).
edge(5,2).
edge(2,6).
```

Figure 3: PDB for Binary Search Procedure

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. Recall that 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, the search is repeated until 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. The following is the du-paths in the binary search procedure example in Prolog form:

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

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 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 expression definition:

Included (*P1 : sequence of NodeID,*
P2 : sequence of NodeID) *as boolean*
such that

$$\mathit{Included(P1,P2)} \equiv \exists i[1 \leq i \leq (\mathit{length}(P2) - \mathit{length}(P1) + 1) \wedge \forall j[1 \leq j \leq \mathit{length}(P1) \Rightarrow P1_j = P2_{i+j-1}]]$$

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. When the du-paths for the binary search routine are condensed we get:

```
conlist([[s,1,2,3,4],
[s,1,2,3,5],
[s,1,2,6],
[4,2,3],
[4,2,6],
[5,2,6],
[5,2,3]]).
```

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. The intuitive idea behind this step of the algorithm is 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. Figure 4 presents the algorithm for computing a number of complete paths that include all of the du-paths in procedural form.

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

```
Initialize:
 DUP := set of condensed du-paths;
 SP := [s]; {Search Path}
 Count := 1;
While DUP is non-empty do
 if SP = [N] {contains a single node}
 then if there is a du-path starting
          with a successor to N
      then 1. choose a path P which
              starts with the closest
              successor;
           2. DUP := DUP - {P};
           3. SP :=  tail(P);
      else 1. increment Count;
           2. SP := [s];
 else if there is a du-path P
           such that SP is a prefix
      then 1. D := D - {P};
           2. SP := tail(P);
      else SP := tail(SP);
end While.
```

Figure 4: Algorithm for Counting Complete Paths.

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 and can only be an estimate of the required number of complete paths needed to meet the all-du-paths criterion.

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.

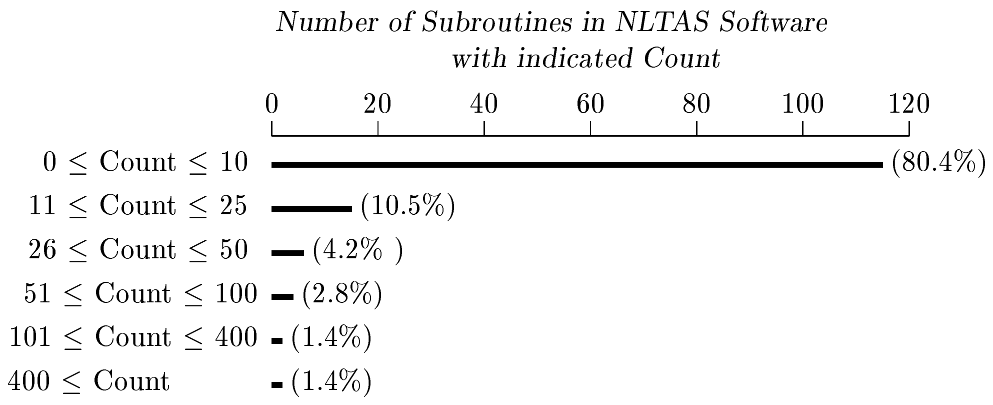

Figure 5: Required Number of Complete Paths

5 Case Study Results

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

1. Number of lines of code not including comments (Lines),
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 Appendix A. The number of decisions can be calculated from the flowgraph dimensions using the following formula: $d = e - n$ where d is the number of decisions, e is the number of edges, and n is the number of nodes in the flowgraph [30].

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 criterion can be met with 25 or fewer complete paths. Figure 5 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) requires testing a clearly intractable number of complete paths. A18 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.

The results in Appendix A 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 intractable number of complete paths necessary in this case.

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

Define X: $X := Y;$
if P_1 *then* $S_1;$

Program Unit	Du-Paths	Condensed	Count
A59	346	139	28
A18	672	568	463

Table 3: All-uses Criterion Applied to A18 and A59

```
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 foregoing is rare; only subroutine A18 requires an intractable 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.

Subroutines A18 and A59 would require the testing of an impractically large number of complete paths to meet the all-du-paths criterion. To estimate the number of tests required using a weaker criterion, we rewrote the Count program to measure the number of complete paths necessary to meet the all-uses criterion. The *all-uses* criterion requires that at least one du path (rather than all du paths) for each du pair be included in testing. Using the all-uses criterion on subroutine A18 and A59, the number of required complete paths drops dramatically, see Table 3. These results should be expected since the worst case complexity for the all-uses criteria is bounded by a polynomial [2]. Using the all-uses criterion, the preceding code structure for A18 could be covered with only one path rather than the 2^{32} paths required for the all-du-paths criterion. Additional paths required to satisfy the all-uses criterion result from definitions and uses other than the pair that causes the anomolous situation for the all-du-paths criterion.

In most cases, the Prolog programs used in this study executed fast enough and made efficient enough use of memory for our purposes. However, due to performance problems in computing the measures, an iterative version of Count was used for five of the subroutines. The performance problems on these routines result from inefficient stack use by C-Prolog. We suggest that the Prolog programs in Appendix B be treated as executable specifications. The programs will run correctly, but may exhaust available memory when processing large PDB's. Recoding into iterative routines can resolve the problem.

Unfortunately, were not able to determine the percentage of the du-paths or complete paths that are actually feasible. We were not able to actually run the programs or do an analysis of the feasibility of particular paths, because we had access to only the StandardRep of the programs. The actual source code is proprietary. We were able to gain access to the code only in the StandardRep form, which hides too much of the program semantics for a path feasibility analysis.

6 Conclusions

In this paper, we formally specify both the all-du-paths criterion and a software 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 intractable 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. All of the subroutines (including the one "worst case" subroutine) can meet the all-uses criterion with a tractable number of 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 intractable number of test cases. Of course, the criterion is not practical for testing these subroutines and alternative testing or verification strategies must be used. The other data flow criteria of Rapps and Weyuker require fewer test cases than the all-du-paths criterion; the all-uses criterion can be met with significantly fewer test cases.

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 [23, 24]. And Count can be used to predict how many test cases may be required. Supplying the data and drivers for the required tests is still a difficult problem.

Our research tools have been formally specified to allow other similar studies to be conducted. The *StandardRep* serves as an excellent basis for specifying data flow and other structural software measures. The c-prolog code for the Count program (Appendix B) may be viewed as an executable specification. We would not expect the program to have satisfactory performance when used to analyze subroutines that are significantly larger than those in this study. The performance of the Count program can be improved by using either a more efficient Prolog implementation or transforming Count into an iterative program.

In order to generalize from these results we are expanding our study to include additional commercial software from various application domains. We are also developing similar estimation tools for additional testing criteria.

Acknowledgment

This research was partially supported by the NATO Collaborative Research Program under RG. 0343/88.

We are grateful to Iris Systems, Inc. for allowing us to use their natural language text analysis system as data for this case study. We also appreciate the effort of Kyung-Goo Doh who implemented the *StandardRep* generator. We thank Albert Baker for his suggestions, and Norman Fenton for his comments on an earlier draft of this paper.

References

[1] G. J. Myers. *The Art of Software Testing.* John Wiley & Sons, New York, 1979.

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

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

[4] 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*, pages 244–251,

1985.

[5] 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.

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

[7] R. G. Dromey. *How to Solve it by Computer.* Prentice-Hall International, London, 1982.

[8] S. C. Ntafos. On required element testing. *IEEE Trans. Software Engingeering*, SE-10(6):795–803, November 1984.

[9] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. Software Engineering*, SE-9(3):347–354, May 1983.

[10] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Trans. Software Engineering*, 15(11):1318–1332, November 1989.

[11] E. J. Weyuker. An empirical study of the complexity of data flow testing. *Proc. Second Workshop on Software Testing, Verification, and Analysis*, pages 188–195, 1988.

[12] E. J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Trans. Software Engineering*, 16(2):121–128, February 1990.

[13] B. W. Kernighan and P. J. Plauger. *Software Tools in Pascal.* Addison Wesley, Reading, Massachusetts, 1981.

[14] J. Bieman and J. Schultz. Estimating the number of test cases required to satisfy the all-du-paths testing criterion. *Proc. Software Testing, Analysis and Verification Symposium (TAV3–SIGSOFT89)*, pages 179–186, December 1989.

[15] L. J. White. Basic mathematical definitions and results in testing. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 13–24. North-Holland, 1981.

[16] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Engineering*, SE-1:156–173, June 1975.

[17] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. Software Engineering*, SE-2(3):208–215, 1976.

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

[19] A. Baker, J. Bieman, and P. Clites. Implications for formal specifications – results of specifying a software engineering tool. *Proc. COMPSAC87*, pages 131–140, October 1987. Tokyo, Japan.

[20] 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.

[21] K. Doh, J. Bieman, and A. Baker. Generating a standard representation from pascal programs. Technical Report 86-15, Dept. of Computer Science, Iowa State University, Ames, Iowa, 1986.

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

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

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

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

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

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

[28] 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.

[29] Janet L. Schultz. Measuring the cardinality of execution path subsets meeting the all-du-paths testing criterion. Master's thesis, Department of Computer Science, Iowa State University, Ames, IA, 1988.

[30] T. J. McCabe. A complexity measure. *IEEE Trans. Software Engineering*, SE-2(4):308–320, 1976.