Microcomputer Applications, Vol. 9, No. 3, 1990.

MA90

A TOOL FOR ESTIMATING SOFTWARE TESTING REQUIREMENTS

James M. Bieman(*)

Abstract

We describe a prototype software tool that estimates the number of test cases required to apply particular testing strategies to program subroutines. The tool was used to evaluate the practicability of data flow testing of a commercial text analysis system which runs on microcomputers. Our system was developed from formal specifications and is implemented in Prolog. It is easily adaptable and can be used to evaluate proposed testing techniques, estimate the resources required to test a software system, and identify hard-to-test subroutines. The testing effort estimating tool is one component of a software analysis research environment.

Key Words

Software tools, software testing, software measures, program analysis.

1. Introduction

The size of available microcomputer random access and disc memory and the speed of microcomputer processors have been increasing, while microcomputer prices have dropped. As capacity increases, the demands on microcomputer software in terms of functionality, size, and complexity also increase. Microcomputer software vendors are discovering that developing large software systems is much more difficult than developing systems for the limited microcomputers of the past. Development schedules are hard to keep, partially because testing takes an unexpected amount of time. Finally, systems are delivered with more bugs than vendors are willing to admit.

Such development problems were discovered years ago by the developers of software for main frame computers [1]. Today, microcomputers actually have the capacity, in both memory size and processing speed, of many of the main frame computers of the 1970's. Furthermore, the demand for easy to use software that can perform complex tasks is even higher for today's microcomputers. A microcomputer user has direct access to a machine and does not have the support of a data processing department to write specialized software. As a result, microcomputer software must have excellent user interfaces and must be usable by non-technical people. The added demands on microcomputer software means that the software must be very complex.

Software with increased complexity is much more difficult to test. Predicting when software will be ready for delivery is also a serious problem. The unpredictable duration of testing has become a major component of this uncertainty. Thus, we must develop techniques to assist software engineers in estimating the effort required to test a system and to identify hard-to-test software components. One common test strategy is to look at the structure of a program to determine test cases. A software system is not completely tested until the tests have touched every statement, branch, data interaction, or some other structural criterion. Our aim is to develop tools and techniques to support the analysis of program structure for use in software testing.

In this research, our goal is to develop prototype tools that estimate the required number of test cases necessary to apply particular structural testing strategies. In previous research, Tai, Weyuker, Ntafos, and Laski use a worst case analysis as a measure of the required number of test cases [2-5]. Our aim is to directly estimate the number of test cases required for a specific program to be tested to satisfy a particular criterion. Only through empirical studies can we determine whether the worst case analyses are overly pessimistic.

As data for our study, we use an actual software system currently being used commercially on microcomputers. The system that we study is a natural language text analysis system (NLTAS). The NLTAS is a product of Iris Systems Inc. and is used commercially for market research. The NLTAS is written in Pascal and runs on microcomputers (PC-compatibles).

In work related to ours, Frankl and Weyuker developed a system, ASSET, which determines whether a given set of test runs satisfies testing criteria and indicates those portions of the program that are still untested [6,7]. This information can then be used to strengthen the test data. ASSET is designed for use on a specific subset of Pascal.

Rather than monitor an ongoing testing process, our aim is to develop tools to estimate the number of tests required by testing criteria. We also seek to develop tools which operate on full Pascal and are easily adapted to other languages. The tool described in this paper can determine the approximate number of test cases necessary to apply data flow testing strategies. It was used in an empirical study of the NLTAS microcomputer system.

(*) Department of Computer Science, Colorado State University, Fort Collins, Colorado 80523 USA

(paper no. 307-479)

Testing strategies, testing criteria, and the data flow criteria which our tool currently supports are described in Section 2. Section 3 describes the software analysis research environment where our tool resides. The design of the testing tool is described in Section 4. Section 5 describes how our tools are used and summarizes the results from applying it to the NLTAS. Conclusions and plans for the future appear in Section 6.

2. Structural Testing Strategies

2.1 Common Testing Criteria

Generally, structural testing strategies use structural testing criteria to select program paths to be tested. Testing criteria aid in selecting the smallest set of test cases that will uncover as many errors as possible. Structural testing criteria use the flow of control and/or the flow of data to determine portions of a program that need to be tested.

Common criteria based on the flow of control include the *all statements, all branches,* and *all paths* criteria. The all statements criterion requires that all program statements be executed during testing, and the all branches criterion requires that all control flow branches be tested. Branch testing is considered a minimum control flow coverage requirement and is described in [8-10].

Other criteria are based on the flow of data through a program [11-14]. These data flow criteria suggest the testing of specific sets of paths that follow the flow of data from expressions through assignments to other expressions. In general, data flow criteria tend to be more effective in uncovering errors than criteria based solely on control flow [15].

The most discriminating path-based testing criterion is the all paths criterion, which requires testing all possible program paths. Unfortunately, the all paths criterion requires an infinite number of test cases in programs with loops. Path-based testing criteria are generally used for unit testing. The criteria help determine the level of coverage of individual procedures or functions.

2.2 Data Flow Testing Criteria

The data flow criteria of Rapps and Weyuker [14] focus on the program paths that connect the definitions and uses of variables (du-paths). A variable definition is a statement or expression that changes the value of the memory location referenced by a variable. A definition may occur via an assignment statement, a procedure invocation, or a function invocation which has a side effect. A variable use is a statement or expression that references or uses the value stored for a particular variable. Rapps and Weyuker distinguish between variable uses within computations (c-uses) and uses within predicates or decisions (p-uses). Thus a variable used in the right-hand-side of an assignment is considered a c-use, while a variable used in the boolean expression controlling a while loop is a p-use.

Often, errors in variable definitions are not uncovered until the variable is referenced at a distant point in the program. Data flow criteria allow the tester to determine the extent to which the paths from variable definitions to variable uses have been tested.

We use the example Pascal binary search procedure from Dromley [16] shown in Figure 1 to demonstrate the definition-use testing criteria. The binary search procedure determines whether or not a particular integer value (parameter x) is stored in an array (array a). We use an individual procedure to demonstrate definition-use criteria since they are unit test criteria — they are designed for testing individual procedures or functions.

```
    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.

A flowgraph is constructed from the source code of the procedure where the nodes represent straight line code (basic blocks) and the edges represent control choices. The binary search procedure has the flowgraph shown in Figure 2. The variables defined and referenced in each program basic block are represented by flowgraph nodes and are shown in Table 1. We distinguish between c-uses and p-uses because p-uses are associated with the out-edges from predicate nodes rather than the nodes themselves.

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, and uses.

The all-du-paths criterion requires testing all of the cycle-free paths between variable definitions and references. The all-uses criterion requires the testing of

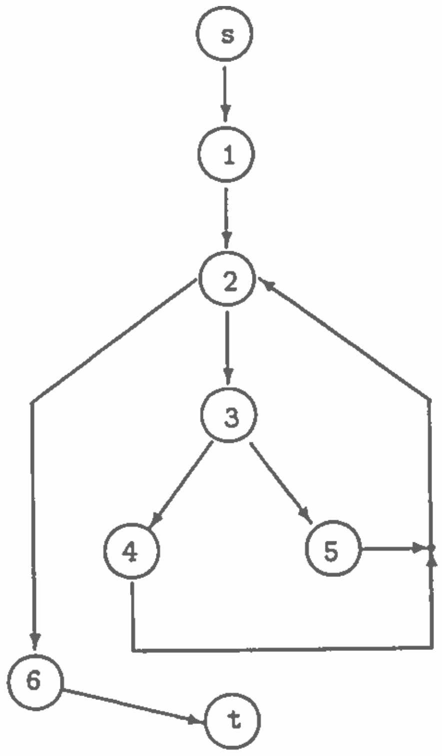

Figure 2. Binary search flowgraph.

at least one du-path (if one exists) between every node pair. All of the du-paths in the binary search procedure are shown in Table 2.

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.

Branch coverage is probably the most commonly used testing criterion in industry. The branch coverage criterion requires the testing of each edge at least once; it does not require the testing of particular paths containing more than one edge. Also, it does not require the testing of any edge more than once. Testing all of the du-paths requires that each branch within the while loop be executed as the last iteration (du-paths ⟨4,2,6⟩ and ⟨5,2,6⟩) and as a non-last iteration (du-paths ⟨4,2,3⟩ and ⟨5,2,3⟩). Du-path testing is more likely than branch testing to reveal errors that only occur when these particular paths are executed.

Our tool can identify the du-paths and estimate the number of test cases required to test these paths.

3. Software Analysis Research Environment

We are currently developing a software analysis research laboratory. The Software Analysis Laboratory is a center for the analysis of software documents. Requirements, specifications, designs, implementations, testing strategies, and the relationships between software documents are included in the research of the Software Analysis Laboratory. Research in the Software Analysis Laboratory has a firm basis in formal specifications: all tools and techniques are formally specified to make our results unambiguous and to allow others to repeat the research. The results reported in this paper are from the investigation of techniques for estimating the effort required to apply particular testing strategies.

To allow us to formally specify our measurement tools in a language independent fashion, we have developed a formal representation of imperative language programs, or the standard representation (Standard Rep) [17]. Imperative or procedural languages such as Fortran, Pascal, C, Cobol, or assembler can be easily modeled by the Standard Rep. The Standard Rep is not appropriate for modeling non-procedural languages such as Lisp or Prolog. The Standard Rep models a program as an annotated flow-graph (or flow chart) where basic blocks (straight-line code) are represented as flow-graph nodes and control branches are represented as flow-graph edges. Nodes are annotated with information concerning the variables whose values are changed or referenced within the basic block represented by a node. The Standard Rep is designed to hide proprietary information to allow us easier access to commercial software for our research. The Standard Rep and our test path measurement tools are specified in the SPECS specification language [18-20].

The Standard Rep is the basis for our software analysis research environment. We can convert Pascal programs into a Standard Rep using a translator implemented by Doh [21]. The information needed for our testing criteria research is extracted from the Standard Rep and stored in a Prolog data base (PDB). The test path analysis tools are implemented in Prolog and operate on the PDB. The overall design of our software analysis research environment is illustrated in Figure 3. This paper focuses on the Prolog Data Base and "Data Flow Testing Effort Estimators" component of Figure 3. The solid lines in Figure 3 represent software tools that are currently implemented and used in our research. The dashed lines represent planned software tools. A translator that converts C programs into the Standard Rep has been designed and is currently being implemented. The testing analysis tool described in this paper makes use of PDB's produced by the tools in our research environment. Since the testing tool uses programs in Standard Rep form, as soon as we have a translator that creates a Standard Rep from a particular language, our tools will be effective on programs in the new language. Thus, when the C to Standard Rep translator implementation is completed, we can immediately use our testing effort estimation tool on software implemented in C.

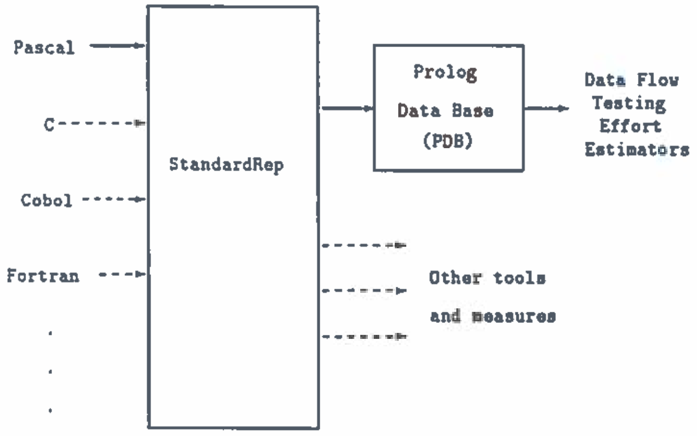

Figure 3. Software analysis research environment.

4. Tool Design

4.1 Implementation Language & Environment

The prototype software testing effort estimator tools were implemented in c-prolog. C-prolog was developed at the University of Edinburgh, and has a standard Prolog syntax without extensions. These programs were implemented without any machine dependent code, and should run on any machine with a Prolog interpreter or compiler. Our first implementation was on a DEC VAX 11/780 running Berkely UNIX. We ported the software to Sun 3/50 workstations and HP 9000/350 systems without any modifications. We anticipate no problems in running our software on any 386 based PC with at least 1 MByte of random access memory. Complete Prolog source listings are available in a technical report [22].

4.2 A Prolog Representation of Programs

The input to the software testing effort estimator is a Prolog representation of the control and data flow of Pascal procedures. This Prolog representation is extracted from the standard representation of Pascal programs (Standard Rep). The Standard Rep is generated by tools in the Software Analysis Research Environment described in Section 3.

The Prolog representation is a Prolog data base (PDB) which contains the control flow information and data flow information necessary to identify the du-paths. A PDB is essentially an annotated flow graph in a form easily digested by Prolog programs — Prolog rules. Figure 4 shows the PDB for the binary search example. The contents and structure of the PDB in Figure 4 match the information in Table 1. The PDB also includes the control flow information, which consists of Prolog rules representing control flow edges. Since the PDB is referenced by Prolog programs, Prolog syntax is used. Thus, the path ⟨2,3,4⟩ is represented by Prolog lists as [2,3,4]. In describing the estimation algorithm, Prolog syntax will be used for such lists or paths.

```
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 4. PDB for binary search procedure.

4.3 Estimation Algorithm

The testing effort estimation is performed in three steps:

1. The du-paths are identified.
2. Redundant du-paths are eliminated.
3. A set of complete paths (paths from the start to the end of a program which cover all of the du-paths) is identified.

The size of the set of complete paths produced in step 3 is an estimate of the fewest number of complete paths that include all of the du-paths.

Searching for Du-paths

Each ordered pair (x, y) of flow graph nodes is examined. If there are variables which are defined in node x and referenced in node y, then a depth first search is performed seeking all cycle-free paths from x to y. For a path to be included, at least one variable defined in x must not be redefined in any node on the path. Figure 5 outlines the algorithm used to find the du-paths. This figure is an attempt to describe a Prolog algorithm in a procedural fashion. Unfortunately, Prolog backtracking makes this presentation somewhat awkward. Prolog normally uses a depth first search approach, and Prolog backtracking allows us to try different sub-paths when the current one reaches a dead end. To find all du-paths the Prolog rule FAIL is used to force backtracking after finding one du-path for a particular node pair. This allows us to find more than one path connecting two nodes.

```
DUP := {};
For each pair (x,y) of flowgraph nodes:
 CVARS:=set of variables defined in x
         and used in a computation in y
 PVARS:=set of variables defined in x
         and used in a decision in y
 if CVARS & PVARS are empty then
     no du-paths between x and y
 else
 SP := [x]; {Search Path}
 Repeat until no more paths to find
  Repeat and Backtrack until CVARS={} and PVARS={}
              or tail(SP)=y
    Find an immediate successor node z
      to tail(SP)
     (z must not have been tried before
       or be on the current SP)
    VZ := {variables defined in z}
    CVARS := CVARS - VZ
    PVARS := PVARS - VZ
    if CVARS<>{} and PVARS<>{}
    then 1. SP := SP||z
         2. if z=y then DUP = DUP U {SP}
```

Figure 5. Algorithm for finding du-paths.

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],
[]]).
```

Eliminating Redundant Du-paths

When searching for du-paths, Prolog will often generate duplicate and redundant paths. Duplicate paths are clearly unnecessary. Du-paths that lie entirely on another du-path are also redundant. Consider the du-paths [1,2,3,4,5] and [2,3] and test cases which cause execution to traverse the first path. The second path has also been traversed and is redundant. The second path may be eliminated from our set of du-paths without consequence.

The du-paths are examined and the paths that lie on another du-path in the list are eliminated from the set of du-paths. After removing the redundant du-paths for the binary search routine, the following "condensed" (conlist) of dupaths is produced:

```
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]]).
```

Counting Complete Paths

The counting of complete paths uses the condensed list of du-paths. The program overlaps and merges as many du-paths as possible along one complete path. A counter is incremented and each selected du-path is deleted from the list of du-paths. The process is repeated until the list of du-paths is empty.

The process begins by a search for a du-path that begins with the start node s. Each time a du-path is selected, it is deleted from the list of du-paths and then a search is conducted for a du-path which begins with the tail of the selected path. If the path [s,1,2,3,4,5] is selected, the program then searches for a du-path with the initial sequence of [1,2,3,4,5,...]. If such a path is not found, the program searches for [2,3,4,5,...], then [3,4,5,...], etc. Should the search reach [5,...], the algorithm seeks du-paths that start with the closest successors to node 5. If no du-paths are found after trying all successor nodes, the count is incremented and the search for a du-path that begins with the start node s is repeated. The program terminates when the list of du-paths becomes empty. Figure 6 presents the algorithm in procedural form.

The final value of the counter may not be the minimum number of complete paths that cover all of the du-paths. When searching for a du-path, there may be several du-paths with the desired initial sequence. The program selects the first path it finds in the list, even though one of the other choices may allow more du-paths to be included along the complete path. Thus, the tool computes only an estimate of the required number of complete paths needed to meet the all-du-paths criterion. However, the algorithm does determine the size of a set of complete paths that covers all of the du-paths.

```
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 6. Algorithm for counting complete paths.

4.4 Ease of Modification

The research tools are very easy to modify when estimating the number of required tests for different testing strategies and when searching for paths with different properties.

For example, only one modification was required for the tool to support the all-uses criterion rather than the all-du-paths criterion. The set of Prolog clauses that searches for all du-paths uses the clause `fail` to force Prolog to search for all du-paths. The Prolog `search` clauses search for all du-paths between two nodes `N1` and `N2`. The search for paths (c_paths) terminating in computation uses (c-uses) and paths (p_paths) terminating in predicate uses (p-uses) are performed by separate clauses. The clauses used to search for all-du-paths include the following:

```
search(_, [], []).
search([N1,N2], C_VARS, P_VARS) :-
    c_path([N1,N2], C_VARS, [N1]),
    fail.
search([N1,N2], C_VARS, P_VARS) :-
    p_path([N1,N2], P_VARS, [N1]),
    fail.
search(_, _, |).
```

The all-uses criterion only requires that *one* du-path be tested between any two nodes rather than *all* du-paths. To modify the Prolog program to support the all-uses criterion, the `fail` clauses are removed:

```
search(_, [], []).
search([N1,N2], C_VARS, P_VARS) :-
    c_path([N1,N2], C_VARS, [N1]),
search([N1, N2], C_VARS, P_VARS) :-
    p_path([N1,N2], P_VARS, [N1]),
search(_, _, _).
```

In our current research, we have modified the Prolog clauses to perform additional analyses of our data. One of these studies involves taking complexity measures suggested by Tai which are based on the effect of control flow on the data flow of a program [23]. This measure requires the introduction of the definitions and uses of a hypothetical variable into a flowgraph representation of a program. This variable is introduced in a manner that maximizes the number of live definitions of the variable which reach successive use. This measure is the sum of the number of live definitions that reach each use of the hypothetical variable. We found it easy to adapt our tool to take this measurement and perform related analysis.

A production tool would use parameters to determine which analysis to perform. Since we are using our tool for research and want the maximum flexibility to change our analysis techniques, we prefer to make modifications to the code itself. Once we are sure about the appropriate analysis to use in a production tool, then such a tool with parameterized choices can be designed and implemented.

5. Analyzing a Microcomputer System

5.1 The NLTAS

Our tool for estimating software testing requirements was first applied to a microcomputer system currently used commercially. The system, a natural language text analysis system (NLTAS), is used to analyze verbatim responses to open ended surveys used in market 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 Standard Rep from the original source code and performed the analysis using the Standard Rep of the system.

5.2 Empirical Study Results

Using our estimating tool, we determined how many test cases are actually needed to test the NLTAS and satisfy the all-du-paths testing criterion [20,22]. The results indicate that the criterion is much more practical than previous analytical results have suggested.

The all-du-paths criterion is comparatively effective in revealing errors, but it may require 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 [3].

Although the all-du-paths criterion can potentially require an exponential number of test cases, our results indicate that the worst case scenario is rare. For most of the subroutines in the NLTAS, the all-du-paths criterion can be satisfied by testing fewer than ten complete paths. Only one subroutine requires the testing of an exponential number of paths. One other subroutine requires a comparatively large number of 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 NLTAS subroutines, the weaker all-uses criterion can be met by testing a reasonable number of complete paths. One of these subroutines requires testing on the order of 2^{32} complete paths to satisfy the all-du-paths criterion. The all-uses criterion requires testing a estimated 463 complete paths. The other subroutine requires testing an estimated 10,000 complete paths to satisfy the all-du-paths criterion. Satisfying the all-uses criterion for this subroutine requires the testing of an estimated 28 complete paths. The detailed results of this empirical study are reported in [20].

Our results confirmed similar results from an empirical study conducted by Weyuker [24, 25]. Weyuker determined how many test cases were required to apply data flow testing strategies using a suite of programs collected by Kernighan and Plauger [26]. The programs included in Weyuker's study were relatively small since they were designed for publication, and the study itself required considerable effort by human testers. The programs in our study are production software currently being used commercially.

5.3 Evaluation of the Tool

We experienced some difficulties when applying our tool to large NLTAS subroutines or NLTAS subroutines with large numbers of du-paths. Analyzing several of these subroutines required an hour or two of processing time when running on a Sun 3/50 workstation. Due to the nature of the c-prolog interpreter that we were using and the mutually recursive nature of our programs, we ran out of system stack space when analyzing subroutines with large flowgraphs. Flowgraphs with greater than approximately 30 edges caused the tool to exhaust the system stack space. We were able to handle these large subroutines after recoding portions of the Prolog code into iterative rather than recursive algorithms.

On the anomalous subroutines with exponential numbers of du-paths, the tool requires an exponential amount of time and space. Obviously, the tool will not be able to complete such computations. In testing the tool on the NLTAS software, we found only one subroutine exhibiting this anomalous behavior. This routine was identified because of the lack of progress and vast amount of disk space used by our tool. Currently, setting a limit on the amount of time that the tool is allowed to run is a pragmatic solution to the problem of identifying these anomalous subroutines.

We found that using Prolog as a language for developing this tool was a good idea. We were able to quickly implement the system and easily modify it to examine different testing criteria. Our only problems occurred when using the Prolog implementation to analyze very large subroutines. These large subroutines are the subroutines that will be the most difficult to test. We can easily adapt our tool to explicitly search for these difficult-to-test subroutines.

6. Conclusions

In this paper, we described a prototype software analysis tool. This tool analyzes the testability of program subroutines. The measure of testability is the number of test cases necessary to apply particular structural testing criteria. Given a software system under development, the tool can estimate the number of test cases required to apply specific structural testing strategies. The tool that we developed focuses on data flow criteria and can be easily adapted to other kinds of testing criteria. Given a Pascal program, our prototype tool will estimate the number of tests required to satisfy the all-du-paths or all-uses data flow testing criteria of Rapps and Weyuker [14]. The tool can be used to identify the subroutines most difficult to test.

The tool is one component of a flexible software analysis research environment. The input of our tool is a representation of programs. This standard representation of Pascal programs is produced by a translator in our environment. As we develop additional translators, we will be able to analyze programs in other languages without modifying the tool. The tool can be easily adapted to support different testing criteria or other methods of program analysis. A production quality testing tool with improved performance could be designed using our research tool as an executable specification.

We used this tool to analyze a commercial PC based natural language text analysis system. Our analysis suggests that data flow based testing strategies are much more practical than previously thought.

Our contribution includes the development of techniques for estimating the number of test cases required to apply particular testing strategies. The software analysis tool we designed applies these techniques. It can identify the most complex subroutines that are difficult to test using selected testing strategies. Our tool is implemented as an executable specification in Prolog and is designed from formal specifications. Thus, the experiments can easily be duplicated by other researchers, and our tool can be re-designed for production-level performance with minimal effort.

We are currently working to improve the performance of the tool. We want the tool to run faster and use less memory. We are also working on ways to identify, in polynomial time, the subroutines that require an exponential number of test cases. We are also adapting our tool to analyze alternative testing strategies. The results from these investigations should help turn our prototype tool into an industrial-strength tool.

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 acknowledge the work of Janet L. Schultz, who implemented initial versions of the testing effort estimation software. We also appreciate the effort of Kyung-Goo Doh, who implemented the Standard Rep generator. We thank both Albert Baker and Norman Fenton for their valuable comments and suggestions.

References

[1] F.P. Brooks, *The Mythical Man-Month: Essays in Software Engineering.* (Reading, MA: Addison-Wesley, 1975).

[2] K.C. Tai, "Program testing complexity and test criteria." *IEEE Trans. Software Engineering* (November 1980), SE-6(6), 531–538.

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

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

[5] J.W. Laski, "On the comparative analysis of some data flow testing strategies." *Technical Report 87-05,* School of Engineering & Computer Science, Oakland University, Rochester, MI, 1987.

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

[7] P.G. Frankl and E.J. Weyuker, "A data flow testing tool." In *Proc. Softfair II* (December 1985).

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

[9] W.E. Howden, "Methodology for the generation of program test data." *IEEE Trans. Computers* (May 1975), C-24(5), 554–559.

[10] W.E. Howden, "Functional program testing." *IEEE Trans. Software Engineering* (March 1980), SE-6(2), 162–169.

[11] P.M. Herman, "A data flow analysis approach to program testing." In *The Australian Computer Journal* (November 1976), 8(3), 92–96.

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

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

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

[15] S.J. Zeil, "Selectivity of data-flow and control-flow path criteria." In *Proc. Second Workshop on Software Testing, Verification, and Analysis* (July 1988), 216–222.

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

[17] 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* (January 1988), 8(1), 13–37.

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

[19] J.L. Schultz, "Measuring the cardinality of execution path subsets meeting the all-du-paths testing criterion." M.S. Thesis, Department of Computer Science, Iowa State University, Ames, IA, 1988.

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

[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, IO, 1986.

[22] J. Bieman and J. Schultz, "An empirical evaluation of the all-du-paths testing criterion." *Technical Report CS-89-118,* Computer Science Dept., Colorado State University, Fort Collins, CO, 1989.

[23] K.-C. Tai, "A program complexity metric based on data flow information in control graphs." in *Proc. 7th International Conference on Software Engineering* (1984), 239–245.

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

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

[26] B.W. Kernighan and P.J. Plauger, *Software Tools in Pascal.* (Reading, MA: Addison Wesley, 1981).