Structurally Guided Black Box Testing

Harish V. Kantamneni **Sanjay R. Pillai**
Yashwant K. Malaiya
Department of Computer Science
Colorado State University
Ft. Collins, Colorado 80523
Tel. (970) 491 7031
Email : {kantamne, pillai, malaiya}@cs.colostate.edu

Abstract

Black-box testing [1, 13, 11] can be easily automated and involves less processing than white box testing because it does not use information about the program structure. However it is very hard to achieve high coverage with black-box testing. Some branches can be very hard to reach. These branches influence the testability of the code they encapsulate and the module in turn. A technique which can help black-box testing to cover these *hard to test* branches easily will significantly enhance test effectiveness. In this paper, we propose a simple guided approach which makes black-box testing more effective using some easily available structural information.

In this paper, we propose a new dynamic measure termed *potential* of a branch. During testing, we extract useful structural information and combine it with changing coverage information to evaluate the current potential of a branch. We use this measure to guide the black-box testing techniques so that the new tests generated are more likely to exercise branches which are so far not covered. We also present an instrumentation approach called *magnifying branches* which may enhance the effectiveness of the new approach. These magnifying branches help the guided approach to increase the focus on these harder branches. The coverage results of a black-box testing technique (random testing) with and without guiding are then compared. Further we compare these coverage results with those obtained after instrumenting the programs with the magnifying branches. The results show that this simple guided technique significantly improves coverage, especially for programs with complex structural properties.

1 Introduction

Software testing is an extensive and difficult process for any realistic software system. By automating the test generation process, the overall cost can be significantly reduced. There are several ways of classifying software testing techniques. One way is to classify them by the the amount of program information they use. *Black-box* or functional testing [1, 13, 11] strategy uses the specifications or the required behaviour of the software as a starting point to design test cases. *White-box* testing[1, 13], on the other hand, uses internal structure of the program to derive test cases. Black-box testing is conceptually simpler and can be easily automated. It is a commonly followed strategy for testing a wide variety of software systems. This provides the motivation for us to focus on improving the efficacy of black-box testing techniques.

Black-box testing has a limitation in that we do not know how much of the code has been covered. To overcome this limitation, test data adequacy criteria can be used to measure what has been tested. There are several adequacy criteria [26] that can be classified by the source of information used to specify testing requirements(specification or program-based) or by their underlying testing approach(structural, fault or error-based). We select a structural coverage criterion : *branch coverage*. The branch coverage criterion is formally described in [26] : " A set *P* of execution paths satisfies the branch coverage criterion if and only if for all edges *e* in the flow graph, there is at least one path *p* in *P* such that *P* contains the edge *e*".

This has been alternatively described as *Decision Coverage Criterion* [13] and can be stated as "each branch direction (true,false) must be traversed at least once". In actual practice, for large programs, it may not be feasible to completely satisfy the branch coverage criterion. If the branch coverage achieved is some minimum acceptable level (85% - 90%), then testing may be regarded as reasonably thorough. However increasing the coverage achieved helps to improve the reliability of the software.

Depending on the decision logic in software programs certain branches are harder to cover, or enter, than others. They are not infeasible but they need more testing effort to cover. We term these branches as *hard to test* branches. Simple black-box testing techniques are not effective for these branches. This is because they do not use any structural information. This combined black-box/white-box approach is found to increase coverage especially as we approach high coverage regions[5] i.e 90% to

100% coverage. We identify certain structural features that make branches hard to test. In this paper we propose a simple approach which makes use of this readily extracted structural information to improve the branch coverage capabilities of black- box testing techniques. We call this a guided approach since structural information is used to automatically guide black-box test generation. We have evaluated this approach by using 4 publicly available benchmark programs. The results show that a high coverage is obtained with considerably less effort i.e. number of test vectors required has been at least halved.

1.1 Testability and Hard to Detect Branches

In this section, we explain how covering the hard to test branches improves the testability of software programs.

Voas and Miller [20] defines software testability as “the probability that a piece of software will fail on it’s next execution during testing (with a particular assumed input distribution) if the software includes a fault”. While testability is a complex issue, it is usually considered as an attribute of a module or a software as a whole. Voas et al [18] give a method to calculate the testability of a module based on the PIE model. Further, Voas et al [19] define a metric, *fault revealing ability* of a test case as a measure of the likelihood that the test case will produce a failure if a fault were to exist anywhere in the program. This metric is also calculated based on the PIE model. This metric is similar in concept to the idea of *detectability* of a testing method as defined by Howden and Huang[7]. The PIE model is a technique that is based on the three part model of software failure. The three necessary and sufficient conditions for a fault to actually cause a failure which is detectable is

1. Execution : the location where the fault exists or has an impact on must be executed.

2. Infection : the program data state is changed by the erroneous computation.

3. Propagation : the erroneous data state is propagated to the program’s output causing an error in the output.

Probabilities for each stage have to be calculated for the PIE model. It uses all three probabilities to calculate either the testability of the code or used to distinguish test cases by calculating the fault revealing ability metric for each test case. However the PIE model restates this : If the code is never executed in the first place then faults can never be detected. This is then primarily a control flow problem, which means that we have to look at branches that influence which locations are executed, i.e. they affect the probability of execution of those locations. Thus testability of a software can be improved with respect to different test cases used.

If we can execute locations more often we can improve the testability. This is where branch coverage and particular test data generation schemes becomes important: to find out which branches are hard to cover and come up with ways to cover them. This would lead to execution of straight line code that lies within branches. To further support this notion we could introduce a modified definition of the branch coverage criterion as in '*a branch is said to be covered when it is executed or entered n times*'; where *n* is related to the testability of the straight line code enclosed by the branch under examination.

1.2 Related Work

There is a lot of literature on various testing techniques. We attempt here to briefly discuss some of the more relevant approaches. Since our technique aims to guide testing using structural information, structural testing is of interest to us. This can be subdivided into *program-based* and *specification-based*[26]. In program-based testing a classification can be based on which criteria are used. The criteria are explained below.

- **Control Flow Adequacy Criteria** Branch coverage is one criterion that falls within this category. Statement coverage is the most basic criterion that one can use. Path Coverage Criterion[26, 23] is another which is used as a basis for pathwise test data generation methods[17]. Two of these methods are symbolic execution[2, 6] and execution-oriented [8] test data generation.

  Of particular interest is Korel' dynamic test data generation method[9]. This differs from the execution oriented test data generation method in that there is no longer a focus on covering paths. The goal now is to execute a selected location. The program is executed with random input and based on the program execution flow an event sequence is generated. An event sequence is similar to a path but less formal in that it is focused towards achieving the goal. Based on this, the search procedure decides which branch should be taken to achieve the goal. If a certain branch in the event sequence is not taken, than a real valued function is associated and function minimization search algorithms are used to find input that will cover that branch. This basic idea has been extended to use data dependence analysis[4] and to handle procedures in programs. Our approach is much simpler. It focuses on covering branches, at least currently. Moreover it does not need to extract a function for each branch. So this makes our method computationally cheaper. This would make our approach more viable for any realistic software system.

- **Data Flow Adequacy Criteria** Data flow analysis of test adequacy criteria is based on covering the flow of data within the program. Based on a variety of data flow artifacts *c-use*, *p-use*, *du-paths* there are several criteria [21, 3, 10, 15]. These use far more information from the program and though complex can be used to detect errors much better than control flow based testing methods. Our objective being to extract as little information from the program, and to simplify the computational aspects, we do not consider methods based on these criteria to be comparable.

The above two categories fall into the white-box category because of their dependence on information extracted from the program and the amount and complexity of information needed. Another class of methods are *domain testing*[22, 16]. Domain analysis and domain testing try to partition the input-output behaviour space into sub-domains and to come up with test cases that can test these sub-domains. This is a black-box testing technique since it uses only the functional information about the program. There are also methods which generate sub-domains based on some information extracted from the program. The path condition for a particular path in the program is derived based on the predicates of the branches in that path. This serves to define a sub-domain. Then by testing the sub-domain, the branches that constitute the path will get covered. Our approach is conceptually similar in that we use some structural information to guide black-box testing. However, we do not derive sub-domain conditions and so do not focus on paths nor on deriving path conditions based on the branch predicates.

The organization of the paper is as follows. Section 2 presents our metric and the approach in detail. Section 3 contains the description of the experiment we ran and the results obtained. Section 4 contains our conclusions and lays out the future direction of our work.

2 The Guided Approach

Our proposed approach makes it possible to generate tests so as to maximize the branch coverage criterion as quickly as possible. The actual test generation mechanism is not fixed. What we have is a framework where we can guide a test generation mechanism to satisfy the branch coverage criterion. This is primarily a framework for black box testing techniques. However the guiding algorithm calculates metrics based on the structure of the branches within the code, thus this could be labeled a gray-box framework. Any test generation technique or even a suitable search procedure could be placed into the framework. The framework only guides the test generation process based on the metric it calculates.

2.1 Factors that make a branch hard to test

What makes a branch hard to test is in itself an interesting problem. Based on random testing several programs, we have come up with the follow structural criteria that make a branch hard to test.

1. Nesting factor : Branches that are deeply nested are more difficult to cover. In the potential measure that we have come up with, we quantify this difficulty associated with the nesting factor.

2. Branch Predicates : Branches with certain operators in their predicates are usually harder to cover. We have identified three such operators and they are : *equal to* (=), *not equal to* (!=) and *logical And (&&)*. We instrument the programs with magnifying branches around such branches in order to help the guiding algorithm cover these branches faster.

2.2 The Metric : Potential of a branch

We introduce a term **potential** of a branch. It is a dynamic metric that is calculated for each branch after the execution of each test case. Briefly it is the number of branches that are nested within a branch which are yet to be covered. It is a combination of both the nesting level and the number of branches which have not yet been covered. More significantly it indicates what the *potential* is, i.e.how many new branches is it possible to cover if we can somehow cover the current branch (for which we are calculating this metric). Formally we define it as follows.

Potential of branch i with respect to a test vector is 0 if it has not been covered. If branch i has been covered/entered with respect to the test vector then it's potential is the sum of the number of un-covered branches nested immediately inside this branch i and the potentials of the covered branches nested immediately inside branch i. If all branches within branch i have been covered the potential is 0.

The pseudo-code for calculating the potential of a branch is given below.

```
potential(branch i ) # with respect to a test vector
begin procedure
      if branch i is covered
         potential = 0
         for x in branches j to k  # where j to k are branches nested
                                   # immediately inside branch i
             if ( branch x is covered by this test vector )
                potential = potential + potential ( branch x )

             if ( branch x has never been covered )
                potential = potential + 1
         return potential
      else
         return 0
end procedure
```

An example for the calculation of the potential for a slice of code.

```
if (a <= b)  # branch i
       {
          if (a > c)   # branch a
             {
               :
             }
          if (a < d) # branch b
             {
                if (d > b) # branch x
                   {
                     :
                   }
                else # branch y
                   {
                     :
                   }
             }
          else # branch c
             {
                :
             }
         }
```

After a test case (or set of test cases),

- If branch i has not been covered then potential (i) = 0.
- If branch i has been covered and if only branch c was covered then potential(i) = (a,b) + potential(c) = 2 + 0 = 2.
- If only branch b was covered then potential(i) = (a,c) + potential(b) = 2 + 2 = 4.

2.3 Components of the Framework

The framework itself consists of the metric calculation after each test vector. Based on the changes in the potential for each branch, the branch on which the test generation is to be focus on is obtained. Based on the changes of the potential for that branch, range reduction or range expansion are carried out to provide constraints for the test generation mechanism. This is a simple search method to find the data point that would cover a certain branch or a set of uncovered branches. During this process, the branch being focused upon might change when for a test vector some other branch might seem more promising (this is decided based on the potential metric).

The components which are needed to use this framework are

1. Test Generation Mechanism

   A test generation scheme is needed to generate test vectors. These test generation schemes need to be able to adjust the test generation based on constraints which will be specified based on the changes in the potentials of the branches in the program under test. The test generation schemes can be Random, Anti-random or any other black-box test generation techniques. But we currently have used the random generation technique. In the future we plan to use the anti-random technique [11, 25] with this framework.

2. Range Adjustment Mechanisms

   This would involve specifying a constraint to the test generator to focus on a certain portion of the input space with a certain test vector as the center of the focus i.e. to localize the input space. The way this is done would be dependent on the test generation technique being used.

   - Reduction. The constraint would reduce the input space under consideration for the test generator. For the random test generation method, this would imply reducing the range for the random generator.
   - Expansion. The constraint would increase the input space under consideration for the test generator. For the random test generation method, this would imply increasing the range for the random generator.

     However the reduction and expansion would be different for the different methods. In the case of the anti-random test generation method, reduction could be taken as reducing the maximum hamming distance that is used to obtain the next test vector. In this way, the reduction and expansion operations would be very much dependent on the test generation method being used.

2.4 Description of the algorithm

Initially the test generation technique that has been selected is used without the guiding mechanism, to cover those branches which are easy to cover. During the next stage, the potential metric is used to guide further testing. After exercising the program with each test vector the potentials of all the branches in the program under test are calculated and the branch with the highest potential is selected. If the potential of this branch has increased, then the last test vector (after which the metric was calculated) is made the center of the input space and range reduction is carried out so as to localize the input space for further test generation. If there is no change in the potential after several test vectors (number of retries), then range expansion is carried out so that other promising regions could be identified in the input space. When another promising region is found, range reduction is done so as to focus the test generation in this newly found promising region. The number of retries is the consecutive number of test cases generated in a localized region for which there is no change in the potential of the branch with the maximum potential and the potential of all the other branches stays lower than this potential. If the number of retries is exceeded, range expansion takes place. A flowchart of the guiding algorithm is shown in Fig 1.

The flow chart includes these steps:

1. Generate a test vector with the selected test generation mechanism
2. If any new branches get covered by this input, then the number of retries is reset to zero. Otherwise the number of retries is incremented.
3. If the number of retries exceeds the maximum number of retries permitted, then range expansion is carried out.
4. The potential for each of the branches with respect to this test vector is calculated.
5. The branch i with the maximum potential is identified.
6. The current maximum potential is compared with the previous chosen potential. If the current potential is greater, then the number of retries is initialized to zero and range reduction is carried out with a view to focus on branch i.
7. Start from step (1) again till the specified coverage is achieved.

2.5 Magnifying Branches

In order to increase the effectiveness of the above framework, we can instrument the code. What we propose to instrument with are *magnifying branches*. These magnifying branches are used to surround branches that have predicates which may make the branch difficult to cover. These magnifying branches help the guiding mechanism cover this branch faster. The magnifying branches' predicates are derived from the predicate of the branch under scrutiny. They serve two purposes

1. to make more effective use of the the range reduction mechanism

2. to increase focus on that branch since the magnifying branch is easer to cover and has a higher potential.

The magnifying branch is meant to be easier to cover because it's predicate is derived from the original branch' predicate by relaxing the original predicate. So more test vectors satisfy this new predicate than that of the original branch. So whenever this magnifying branch is entered, its potential becomes higher because of the uncovered branch nested within. When this potential becomes higher, the guiding algorithm makes the test generation mechanism focus on the branch encapsulated by this magnifying branch. In this way, the magnifying branches enhance the effectiveness of the guiding mechanism.

A few examples to illustrate the instrumentation process are shown below.

example 1 `if ( a == b )`

```
after instrumentation becomes

if ( ( a >= ( b - LOCALITY_RANGE ) ) && ( a <= ( b + LOCALITY_RANGE) ) )
  if ( a== b )
```

`LOCALITY_RANGE` is a measure of how much a difficult predicate has been relaxed by. So the magnifying branch is easier to cover since the range of values that satisfy it has been increased.

example 2 `if ( a && b )`

```
after instrumentation becomes

if ( a )    magnifying branch
   if ( b )   magnifying branch
     if ( a && b )
```

In this way, other difficult predicates too can be magnified. This can be done recursively if each expression is a compound expression. This instrumentation is done at the beginning before the test process is started.

3 Experiments and Results

3.1 Programs used

An experiment was carried out to assess the effectiveness of these techniques. We selected 4 programs of varying sizes from literature or which have accepted standard implementations. The programs are listed in Table 1. They are a mix of program sizes, number of branches, predicate complexity and nesting level. Some of the programs have been slightly modified to allow it to interface properly with our tool.

Triangle: This program is a standard example for most testing related literature [24]. It accepts the lengths of three sides of a triangle and classifies it as scalene, isosceles, equilateral or not a triangle at all. This is a small example but has a complex decision logic.

Calendar: Calendar is the UNIX calendar program. This is the GNU implementation of cal. It is a relatively large program. It accepts th month, year and certain options. The output is the calendar for the specified range of dates.

Roots: This is an implementation of algorithm 326 from the Collected Algorithms of the ACM [14]. It accepts the coefficients of a bi-quadratic equation and calculates it's roots.

Max: This is a simple program used to test the tool we built initially. It can be found in any basic programming text. It accepts 3 numbers and outputs the greatest of the 3 numbers.

Program	Number of lines	Number of branches
triangle	90	20
calendar	419	42
roots	245	41
max	37	19

Table 1: Some details of the programs used in the experiment.

3.2 Description of the Experiment

We use a tool written by us which automates the technique that we have described in previous sections. Each program under test is instrumented by the tool to track which branches are covered. The tool also keeps track of which test covered which branches. It provides an interface to the test generator component to calculate the potential of any branch after each test vector is applied. Currently the magnifying branches are introduced manually but they could be automated easily. There are a lot of tools [12] that can calculate the coverages so those tools could be used to handle this feature of our tool. However to keep our tool integrated with advanced features of our technique it includes even these simple features too.

The test generator generates random test vectors. It interfaces with the tool in order to use the potential information. The test generator uses this information along with the algorithm outlined in the previous section to localize or expand the ranges for the random test generator. The test generator has two phases

1. Pure random test vectors are generated to cover all the easy to cover branches.

2. Random vectors are generated within ranges guided by our scheme to cover the harder branches.

For each of the programs, we test using three different methods.

Simple Random testing: We carry out pure random testing because the test generator component used with the guided scheme is a random test generator. So this serves as a benchmark to determine how the guided random test generator performs better.

Guided testing: The tool is used to help guide the random test generation mechanism.

Guided testing with Magnifying branches: The program is instrumented with magnifying branches. Then guided testing is performed on the instrumented program.

For each of the above methods, we stop the testing after reaching a certain percentage of coverage.

cover any new (i.e. previously uncovered) branch within a specific number of retries. If this is not successful the phase ends indicating failure to satisfy the required coverage criterion goals. For the current experiment testing was carried out till 100% branch coverage was achieved or till a plateau w

3.3 Results

The results of these tests are shown in graphs Fig (2) - Fig (5). Each graph plots the % of coverage achieved for the three methods for each of the programs under study. Table 2 summarizes the complete results, showing the maximum coverage achieved for each program and the number of tests required by each method. In the graphs, the scales of the x and y axes have been chosen so as to show the differences between the methods clearly.

Program	% Coverage Achieved	Number of Test Vectors		
		Random Testing	Guided Testing	Using Magnifying Branches
triangle	100	5527	166	316
calendar	97.62	408	127	243
roots	92.68	3426	1598	429
max	100	8	8	8

Table 2: Coverage achieved (%) Vs Number of tests needed by each Method

For all programs, the basic technique of potential guided random testing performs at least as well as random testing by itself. In case of the **triangle**, **calendar** and **roots** programs it performs much better, giving a high coverage with far fewer test cases. In case of the **max** program, it performs the same as random testing. This is to illustrate the case when the program is easily testable.

The magnifying branches method is still a preliminary idea. We are still studying the benefits of this method. Currently, it does not always perform better than the guided random testing method. The reason for this behaviour is that we magnify branches in the program under test irrespective of whether they are easy or hard to test. So even for the relatively easy branches, the magnifying branches cause quite a few unnecessary tests. The number of unnecessary tests is exacerbated by the retry factor. So the number of retries can be adjusted to improve the performance of the magnifying branches method. The number of retries also affects the simple guided testing method but to a much lesser (negligible) extent.

For the **triangle** program (figure 2), both the guided random testing methods perform much better than the pure random testing method. The graph shows that in order to obtain about 95% branch coverage, the pure random testing, guided testing and the magnifying branches methods require 442, 90 and 303 test vectors respectively. This in particular shows the effectiveness of the guided testing scheme for particularly complex decision logic.

For the **cal** program (figure 3) also, both the guided random testing methods perform better than the pure random testing method. The graph shows that in order to obtain about 90% branch coverage, the pure random testing, guided testing and the magnifying branches methods require about 400, 120 and 170 test vectors respectively. However the simple guided testing performs better than that with magnifying branches. From the graph, we can see the point at which these two diverge. From the study of the sequence of tests for this program it shows that from this point on, our previous reasoning for this anomaly holds. This could be rectified by combining the simple guided and the guided with magnifying branches methods into a more effective method. This can be done by having a method in which after the benefit of the simple guided method is realized, the magnifying branches can be made use of. For this program, the maximum coverage achieved is 97.62% i.e. one branch was not covered. This is because of the weakness of the random test generator.

The results for the **roots** program (figure 4) mirror that of the Calendar program. The pure random testing scheme takes as much as 10 times (as seen in the table) the number of test cases as does the best method, guided testing with magnifying branches. The graph shows that in order to obtain about 92% branch coverage, the pure random testing, guided testing and the magnifying branches methods require about 717, 246 and 400 test vectors respectively. Also for this program, the guided testing method with magnifying branches performs much better for the uncovered (the harder branches) branches after reaching about 90% coverage than the simple guided method. This is because it turns out that only the branch that was hardest to test qualified to have a magnifying branch. The maximum coverage achieved for this program is 92.68% i.e. 3 branches were not covered. In order to understand this, one must know the structure of this program. There are three functions in this program. They are quadratic, cubic and biquadratic root calculation functions. Our driver tests the biquadratic function only because this in turn makes use of the other two functions. There is some redundant code in these functions since each of them was coded to be used independently. Since only the biquadratic function is being used directly, the redundant code in the other functions is never exercised and thus the 3 uncovered branches.

The **max** program (figure 5) is used to point out the fact that programs with in which the branches are easy to test do not benefit from the guiding technique. All three methods achieve 100% branch coverage after 8 test vectors. This is because for any of the methods, initially the pure random method is run till saturation after which the guiding is used.

4 Conclusion and Future Work

The basic framework which guides testing using the potential metric works well in all the cases we have seen. It should be noted that the metric, and thus the framework, focuses on hard to test branches. If the program does not have such behavioural characteristics, i.e. all branches are easily tested, then this guided approach has the same effectiveness as that without the use of guiding for that program. In actual practice, all large programs have hard to test branches and would benefit from such an approach. Sometimes testing is not evaluated using a structural adequacy criterion like the branch coverage criterion. However it has been shown that higher coverage generally implies better defect finding capability. Branch coverage has been criticized as being a weak test adequacy criterion. This can be addressed by using more stringent criteria. We are looking into ways to devise similar metrics for the other coverage criteria also. If high test coverage is desired, then this approach is helpful and can guide testing. The results show remarkable reduction in the number of test vectors needed as we have already seen.

Our current implementation of magnifying branches in conjunction with the guided testing method does not always show improvement over the simple guided testing method. The reason for this behaviour is that we magnify branches in the program under test irrespective of whether they are easy or hard to test. So even for the relatively easy branches, the magnifying branches cause quite a few unnecessary tests. We are looking into ways to combine the simple guided and the guided with magnifying branches methods into a more effective method. This could be done by having a method in which after the benefit of the simple guided method is realized, the magnifying branches can be made use of.

A modification to the framework could be to use an Antirandom test generator in place of the random test generator current being used. Antirandom testing [25, 11] has an advantage in that it is able to uniformly test programs with different input domains. We also are in the process of formalizing our rules for hard to test branches and come up with a statically determinable quantitative measure to describe the detectability or testability of a branch.

References

[1] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 2nd edition, 1990.

[2] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, 2(3):215 – 222, Mar. 1976.

[3] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15(11):1318 – 1331, Nov. 1989.

[4] R. Ferguson and B. Korel. The chaining approach for software test data generation. Technical Report CSC-94-003, Computer Science Department, Wayne State University, 1994.

[5] G. Hohenstrater and B. Mundle. Elements of a reliable and productive test process. In *International Symposium on Software Reliability Engineering (ISSRE 98)*, volume 9, pages 241 – 250, Paderborn, Germany, Nov. 1998. IEEE Computer Society,.

[6] W. E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions in Software Engineering*, 4(4):266 – 278, Apr. 1977.

[7] W. E. Howden and Y. Huang. Software trustability analysis. *ACM Transactions on Software Engineering and Methodology*, 4(1):36 – 64, Jan. 1995.

[8] B. Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870 – 879, Aug. 1990.

[9] B. Korel. Automated test data generation for programs with procedures. In S. J. Zeil, editor, *International Symposium on Software Testing and Analysis*, pages 209 – 215, San Diego, CA, Jan. 1996. ACM, ACM Press.

[10] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions in Software Engineering*, 9(5):33 – 43, May 1983.

[11] Y. K. Malaiya. Antirandom testing: Getting the most out of black-box testing. Technical Report 96-129, Computer Science Department, Colorado State University, Fort Collins, CO, 1996.

[12] B. Marick. *General Coverage Tool (GCT)*, Jan. 1993. freeware.

[13] G. Myers. *The Art of Software Testing*. John Wiley and Sons, New York, NY, 1979.

[14] T. R. F. Nonweiler. Roots of low-order polynomial equations. *Communications of the ACM*, 11(4):269, Apr. 1968.

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

[16] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676 – 686, June 1988.

[17] C. Ramamoorthy, S.Ho, and W. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4):293 – 300, Apr. 1976.

[18] J. M. Voas. A dynamic testing complexity metric. *Software Quality Journal*, 1(2):101 – 114, June 1992. Chapman and Hall.

[19] J. M. Voas and K. W. Miller. The revealing power of a test case. *Journal of Software Testing, Verification and Reliability*, 2(1):25 – 42, May 1992. John Wiley and Sons.

[20] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Software*, May 1995.

[21] E. J. Weyukar. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912 – 919, Sept. 1993.

[22] E. J. Weyukar and T. J. Ostrand. Theories of program testing and the application of revealing sub-domains. *IEEE Transactions in Software Engineering*, 6(3):236 – 246, May 1980.

[23] L. J. White. Basic mathematical definitions and results. In B. Chandrasekaran and S. Radicchi, editors, *Computer Program Testing*, pages 367 – 375. North Holland, 1981.

[24] W. E. Wong. *On mutation and dataflow*. PhD thesis, Computer Science Department, Purdue University, 1993.

[25] H. Yin, Z. Lebne-Dengel, and Y. K. Malaiya. Automatic test data generation using checkpoint encoding and antirandom testing. Technical Report 97-116, Computer Science Department, Colorado State Univeristy, Fort Collins, CO, 1997.

[26] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):365 – 427, Dec. 1997.

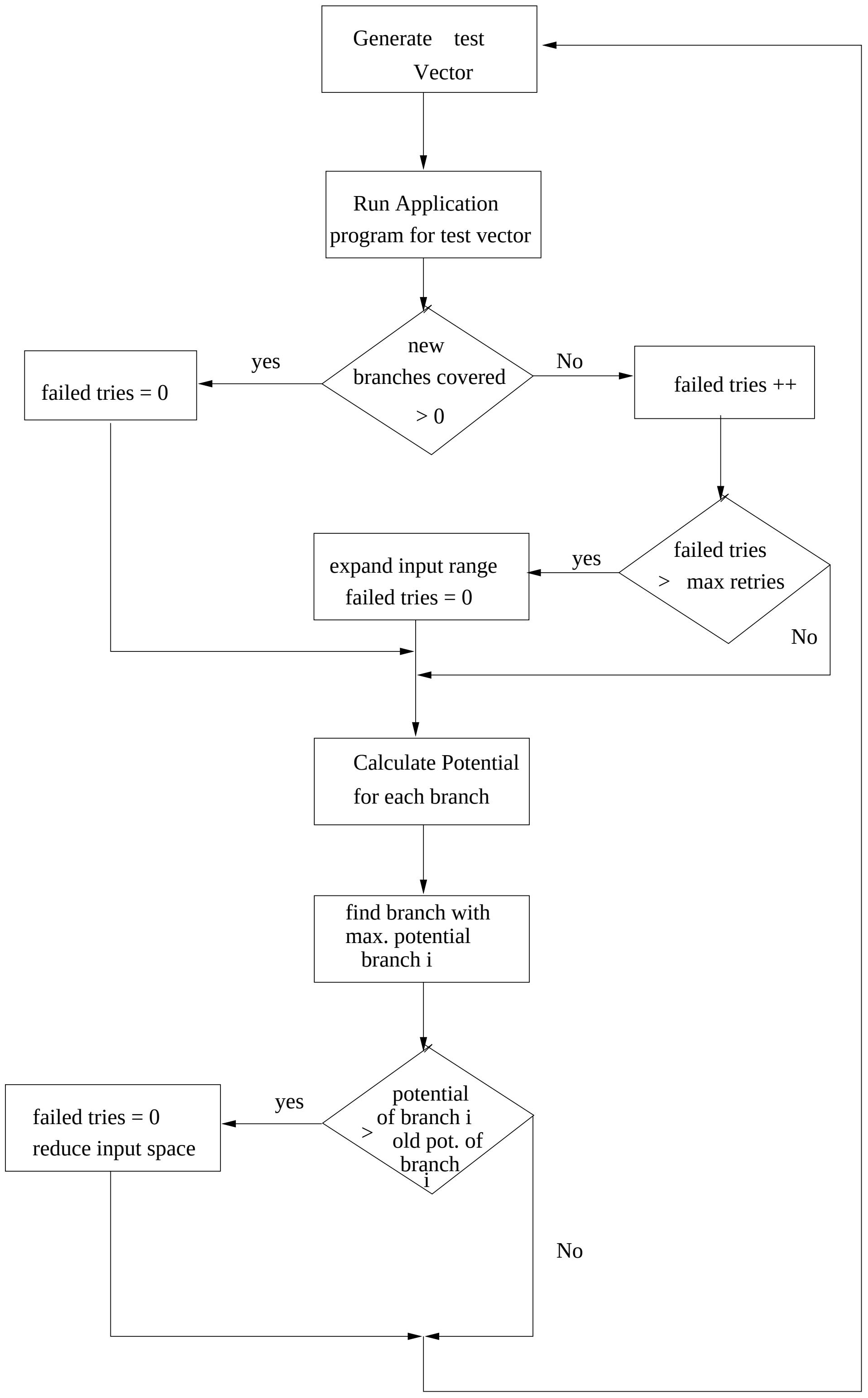
Generate test Vector
Run Application program for test vector
new branches covered > 0
yes
No
failed tries = 0
failed tries ++
failed tries > max retries
yes
No
expand input range failed tries = 0
Calculate Potential for each branch
find branch with max. potential branch i
potential of branch i > old pot. of branch i
yes
No
failed tries = 0 reduce input space

Figure 1: Flow Chart of the Guiding Algorithm

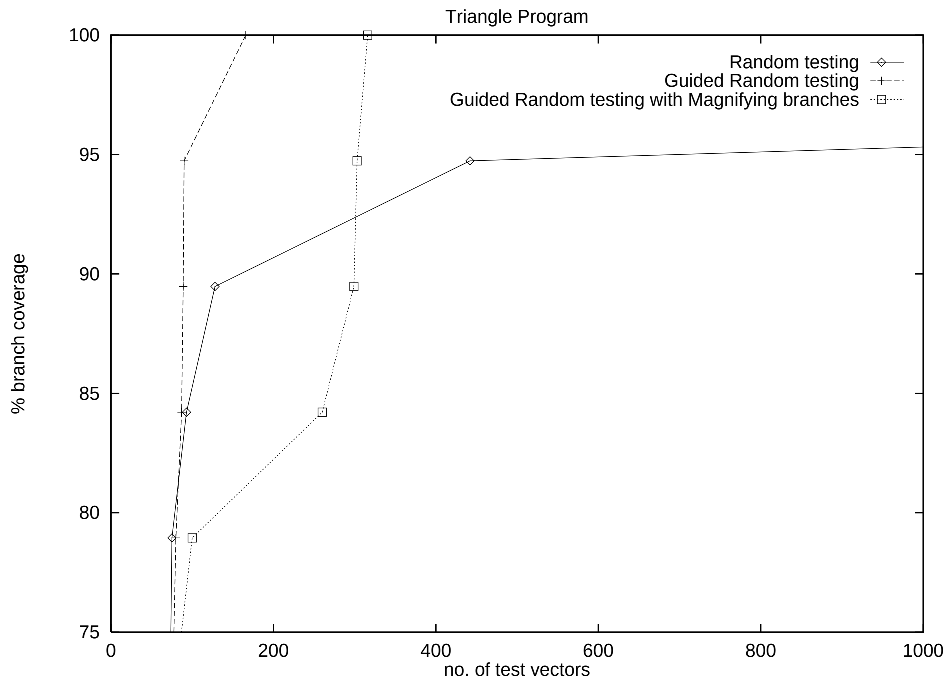

Figure 2: Results for the Triangle Program

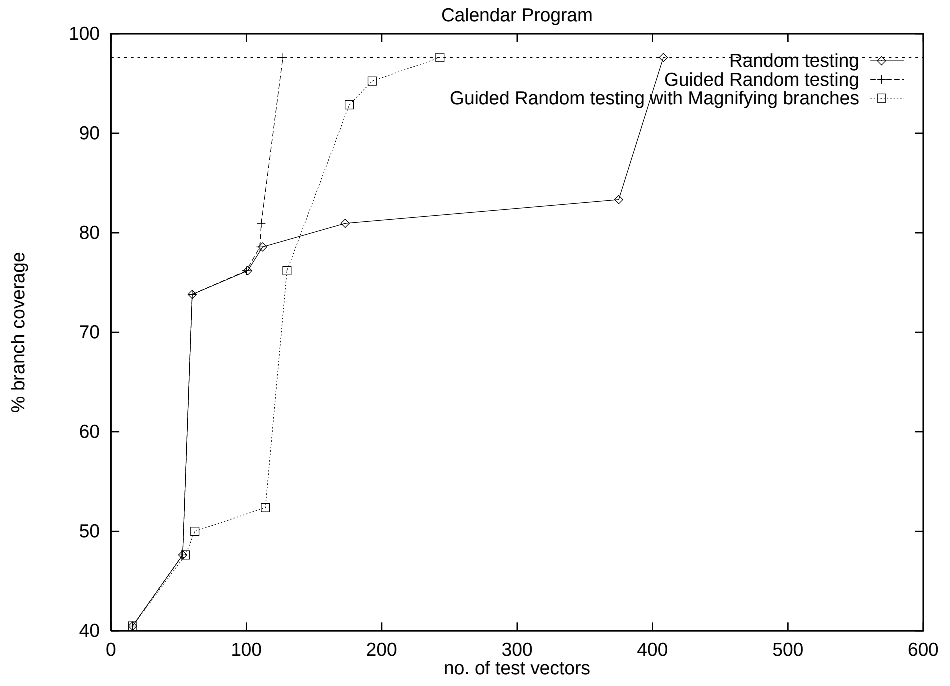

Figure 3: Results for the Calendar Program

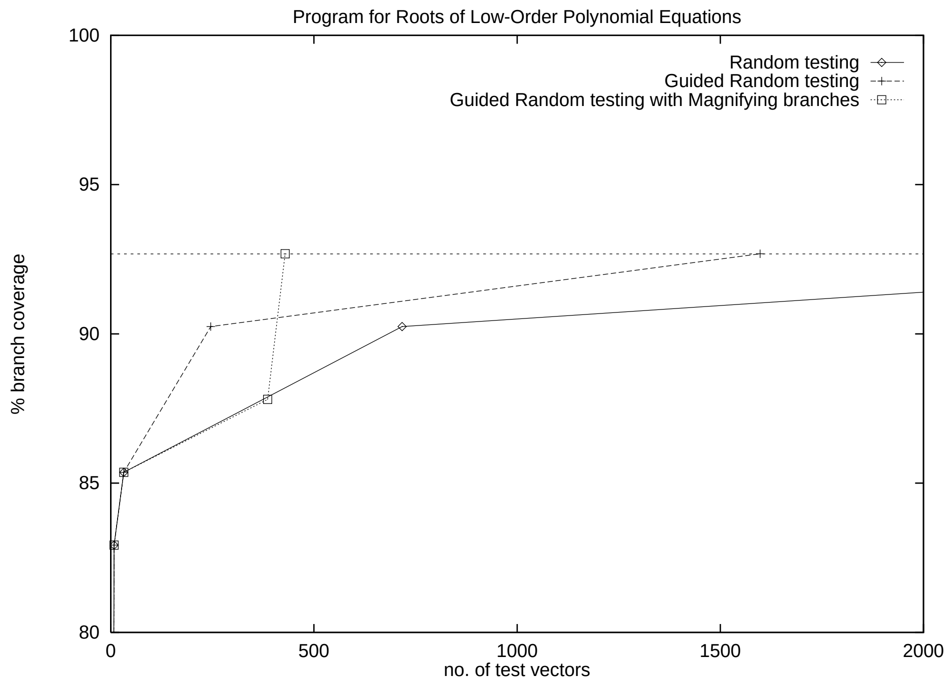

Figure 4: Results for the Roots of Low-Order Polynomials

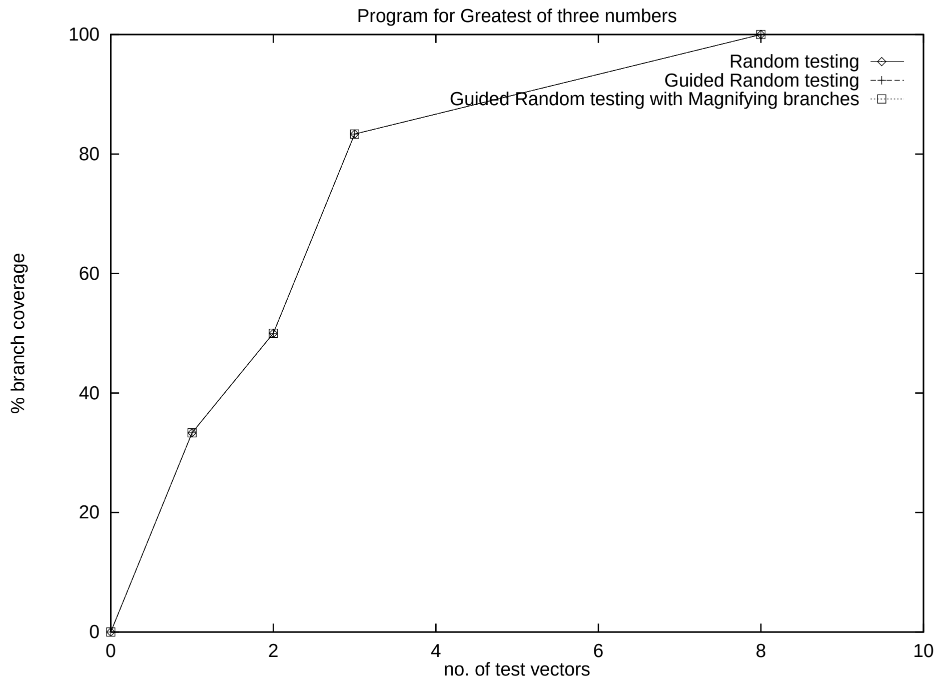

Figure 5: Results for the Greatest of three numbers Program