Automatic Test Generation using Checkpoint Encoding and Antirandom Testing *

Huifang Yin, Zemen Lebne-Dengel† and Yashwant K. Malaiya
Computer Science Dept.
Colorado State University
Fort Collins, CO 80523
malaiya@cs.colostate.edu

† Graphics Technology Lab
Hewlett-Packard Co.
Fort Collins, CO 80525

ABSTRACT

The implementation of an efficient automatic test generation scheme for black-box testing is discussed. It uses checkpoint encoding and antirandom testing schemes. Checkpoint encoding converts test generation to a binary problem. The checkpoints are selected as the boundary and illegal cases in addition to valid cases to probe the input space. Antirandom testing selects each test case such that it is as different as possible from all the previous tests. The implementation is illustrated using benchmark examples that have been used in the literature. Use of random testing both with checkpoint encoding and without is also reported. Comparison and evaluation of the effectiveness of these methods is also presented. Implications of the observations for larger software systems are noted. Overall, antirandom testing gives higher code coverage than encoding random testing, which gives higher code coverage than pure random testing.

Keywords: antirandom testing, random testing, checkpoint encoding, test coverage, software testing.

*This research was supported by a BMDO funded project monitored by ONR

1 Introduction

Testing of software requires a significant commitment of resources [12, 17]. It is of considerable practical and theoretical importance to explore ways to reduce the testing effort while maximizing test effectiveness [3, 6, 13, 15].

There are many testing techniques discussed in the literature that can be termed *black-box testing* [6, 8, 9, 13, 20]. *Random testing* [8] chooses tests randomly based on some input distribution, without attempting to exploit information gained by tests applied earlier. It considers the program's input domain as a single whole and randomly selects test inputs from this domain. There are different opinions regarding the effectiveness of random testing. Meyers [17] claims that random testing is an ineffective strategy to uncover errors in the software. Other studies such as that of Duran and Natfos have shown that random testing under certain situations can be effective and is worth considering [8], specially considering the relative ease of test generation and potential for automation.

A number of test data selection strategies [6, 11, 26] have been discussed in the literature. In *partition testing* approach, the program's input domain is divided into subsets, and one or more tests from each of these subsets are selected to exercise the program. In dividing the input space into subdomains, it is expected

that software responds to all the points within the same subdomain in a similar way by validating program correctness or by uncovering a fault or exhibiting illegal behavior [1, 20, 23]. While this expectation is an idealization, it allows us to feel reasonably assured that only one or few test cases within a subdomain are enough to cover the expected behavior for the whole subdomain. Economizing on test data selection this way can make the cumulative number of input test cases manageable and can make testing much more cost effective.

In real programs, however, this idealized scenario of clean and nonoverlapping partitioning into subdomains happens rarely [9, 10, 19, 23, 25]. Useful heuristics [6] of selecting test data are designed to exercise boundary values [2, 25] as well as uncover simple errors that tend to have a coupling effect [8] to larger errors (errors that violate the software specifications). This has the advantage of reducing testing effort while preserving testing effectiveness. Another approach to increasing test effectiveness and efficiency is to reduce the number of tests required by limiting the number of combinations of tests to be considered. Orthogonal latin squares [15] and combinatorial design [3] are among the approaches that have been discussed in the literature.

Recently, Malaiya [13] introduced the concept of antirandom testing for black-box testing. It is based on the view that testing is efficient if the next test in the sequence is chosen to have maximum *distance* from all previous tests that have been applied. Antirandom test sequences are constructed using this approach. Hamming and Cartesian maximum distance measures are defined to help generate these antirandom test sequences. Unlike random testing, antirandom testing generates input test sequences designed from the outset to exploit information about the tests that were applied in the past.

In this paper, techniques for automatic test generation using checkpoint encoding are investigated using both antirandom testing and random testing. These two testing approaches are compared with random testing based on some input distribution. For this study we take three common benchmark programs which have been used in the literature. We present possible approaches for checkpoint encoding to ensure that we are probing the input space in an efficient way.

Various approaches can be taken to gauge the effectiveness of testing [21, 22]. Here effectiveness of the various testing approaches was evaluated by measuring code coverage. This is an acceptable approach, since higher test coverage generally implies better defect detection capability [14, 27]. Test generation in this case is based on the external specification of the problem (black-box). Test coverage, unlike testing effort, is a direct measure of how well the software under test has been exercised [14].

The purpose of this paper is to demonstrate the feasibility of automatic test generation using approaches that are not random, and to show the promise of such approaches. In future work we will apply the methods developed here to larger programs to do a systematic comparison of the proposed approaches to random testing.

In the next section, we introduce the approach using antirandom testing and checkpoint encoding. In the third section the three benchmark examples are described, and for each example, the encoding scheme and the code coverage results are shown. Finally concluding remarks are presented and future areas of investigation are briefly discussed.

2 The CEAR test generation scheme

The Checkpoint Encoded Antirandom testing (CEAR) scheme used here was proposed by Malaiya [13]. This scheme integrates antirandom testing with checkpoint encoding as explained below, and is designed to process input test vectors on the fly automatically and to exercise the software under test, thus making the scheme cost effective. The CEAR scheme has three major components:

- The MHDATS(MCDATS) binary sequence generator.
- The random value generator.
- The binary-to-actual input translator.

As shown in Figure 1, the CEAR is a collection of software tools that produce actual input vector for the software under test. The MHDATS(or MCDATS) binary sequence generator calculates the next binary vector in the antirandom sequence. Its bit values are examined for a match to the bit values assigned to the fields in the checkpoint encoding definition. The appropriate actual input test vector is generated and fed to the software under test.

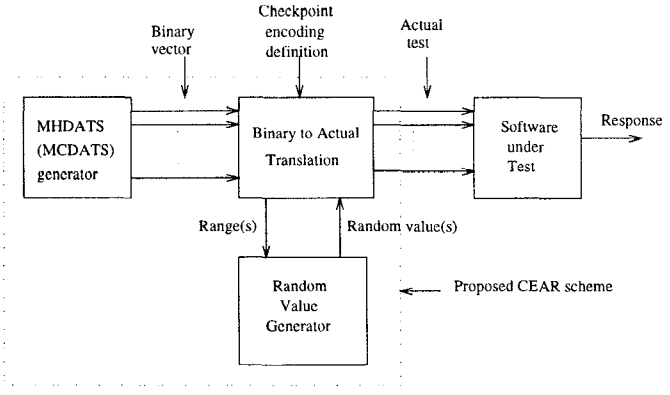

Figure 1: The CEAR scheme with encoded antirandom vectors

2.1 Antirandom testing

In antirandom testing, each test in a sequence is defined to be maximally distant from all of the previous tests. Test vectors which are closer together are likely to exercise the software in a similar way, and no new information is likely to be gained. However, in the antirandom testing paradigm, each test in the sequence attempts to exercise different areas of the software and thus has the potential of getting higher code coverages.

In antirandom test vector generation, distance is defined using either the Hamming or the Cartesian distance measure. If we assume binary vector encoding is used to represent the input variables, one first chooses the initial binary test vector t_0 to be all 0's, without lose of generality. The next binary test vector in the sequence, t_1 is then obtained by calculating the maximum hamming or maximum Cartesian distance away from t_0. Construction of maximal hamming distance antirandom test sequence (MHDATS) and maximal Cartesian distance antirandom test sequence (MCDATS) is discussed in detail in [13]. Each subsequent test vector t_i is then chosen such that the total distance between t_i and all the previous tests $t_{i-1}, t_{i-2}, ..., t_0$ is a maximum. The procedures presented by Malaiya have been implemented in the Antirandom Testing Generation ARTG program [30].

An integral part of antirandom testing is the checkpoint encoding scheme that enables the efficient capture of proper combinations of typical, boundary and illegal tests cases so that the test coverage is as high as possible.

2.2 Checkpoint encoding

In general, the desire is to exercise not only expected or usual program behavior but also corner or boundary cases. Moreover, regions of expected illegal behavior need to also be tested to ensure the software under test is responding appropriately. The objective of checkpoint encoding is to make the testing effort as effective as possible by converting the problem to that of constructing binary antirandom sequences. Sample points representing the range of input characteristics (e.g. typical, boundary and illegal) are encoded into binary. These sample points (or checkpoints) are then obtained by automatic translation.

Typically, a boundary value in the input space maps to a specific field encoding which then results in the generation of a test tailored to exercise that boundary condition. For homogeneous values, corresponding to a subdomain partition in the input space, the checkpoint field definition and encoding consists of multiple values which are then randomly selected using the random value generator in accordance with some input distribution assumption. Uniform distribution has been used for this study.

In checkpoint encoding the design of the encoding scheme needs to be carefully considered. We need to decide how many bits to use and need to allocate the combinations to the typical, illegal or boundary case situations. Careful attention needs to be given to how much of the black-box information needs to be cap-

tured by checkpoint encoding and to what level of resolution. Also one has to balance the number of bit encodings that are assigned to legal range versus illegal or boundary cases in the input space. For instance, if the bit assignment in the encoding scheme is weighed heavily toward the illegal input combinations, the software under test will have initially low coverage as the common cases (or legal operations) in the input space are not being exercised as often in comparison to the illegal situations. Detailed observations are presented in the next section.

In devising an encoding scheme, one starts from the problem specification and tries to conceptually partition the problem space into subdomains, where each of the subdomain has a common or homogeneous characteristic. Another way to look at the subdomain classification is to see the problem as made of possibly one or more dimensions, each dimension occupying a hyperplane in the hamming space.

3 Experimental methods, results and analysis

Three typical benchmark programs frequently mentioned in the literature are used in our investigation. They are a string matching program STRMAT [29], a triangle classification program TRIANGLE [6, 8, 12, 17] and FIND which can be part of a sorting program [6, 13, 29]. Coverage measures are used to quantitatively compare the testing approaches. The GCT coverage tool [16] is used to instrument the programs to get quantitative code coverage measures using branches, loops, multiple conditions and relational operations covered. Test coverage measures have been demonstrated to have a relationship to defect coverage [5, 14].

3.1 Testing data generation procedure

Automatic test generation [4, 7, 24] is designed to ease the test effort. Here we have used three different approaches for automatic test generation.

1. Antirandom with checkpoint encoding
2. Random with checkpoint encoding.
3. Random without checkpoint encoding.

In the plots and the tables, these are respectively indicated by *AE*, *RE* and *RW1* or *RW2*.

We first need to analyze the program specifications and the natures of the problem. Then according to some general encoding rules, we decide the specific encoding scheme for each program. After determining the number of binary bits, we use the ARTG program to generate the necessary antirandom test sequences. Each antirandom binary vector is then decoded to actual input value for each variable. In decoding the binary vector, we use randomly generated value within the range specified in the encoding scheme. In checkpoint encoding, we use the random function to generate the binary test vectors, then decode them to actual input values just like in antirandom testing. In purerandom testing, for each program, we choose two different seeds to generate the actual input values randomly according to the range specified to illustrate the possible variation of the results.

3.2 Testing code coverage evaluation

Once a test suite is prepared based on the testing approaches discussed earlier, the GCT tool [16] is used to instrument the program.

The coverage measures used are:

- Branch coverage: Complete branch coverage requires every branch be exercised at least once in both the true and false directions.
- Loop coverage: Complete loop coverage requires that a loop condition be executed once, several times and also should be skipped (without ever entering the loop) in some test condition.
- Multi-condition coverage: This has a stronger requirement than branch coverage. It checks for all parts of a logical expression being used. That is, each of the logical expression components must evaluate to TRUE in some test, and to FALSE in some other test. Multi-condition coverage is

stronger requirement compare to branch coverage.

- Relational coverage: this checks for tests that probe common mistakes regarding relational operators. A likely mistake could be using "<" when "<=" is intended.

3.3 Experimental results

3.3.1 The STRMAT program

This example has also been used by Wong et al. [27, 28] to investigate test coverage issues. The program is given as input a string of zero to 80 characters, and a pattern at most 3 characters long. The objective is to see if the pattern is matched in the string. If so, the pattern position in the string is returned.

In choosing the checkpoint encoding scheme one is interested in dividing the problem space into subdomains that conceptually can be seen as consisting of orthogonal dimensions. The text length can be seen as a variable in one dimension, the pattern length can be seen as a variable in a second dimension orthogonal to the first. Finally, the pattern position can be seen as a third dimension orthogonal to the first two. As will be seen later, it turns out that checkpoint encoding with antirandom testing is particularly effective and is superior to the other testing schemes when many dimensions are characteristic of the problem specification.

The encoding scheme for STRMAT string matching program chosen is shown in Table 1. The following subdomains can be identified from the problem specification in considering black-box testing.

Table 1: Encoding scheme for STRMAT

Field	Bits	Value	Significance
text length	b2,b1,b0	110	0
		010	80(tmax)
		011	80 < tlen < 100 (illegal)
		rest	1–79
pattern positions	b5,b4,b3	110	no pattern
		010	beginning
		011	end
		rest	middle
pattern length	b8,b7,b6	110	0
		010	3 (pmax)
		011	3 < plen < 10 (illegal)
		rest	1–2

Ranges:
text length: $0 \leq texlen \leq 80$
pattern positions: 1,textlen, middle, outside
pattern length: $0 \leq patlen \leq 3$

Using this encoding scheme for the STRMAT program, coverage measures (branch, loop, multicondition, relational and total coverages) for antirandom testing(AE), random testing with checkpoint encoding (RE), and pure random testing with two different seeds (RW1 and RW2) are shown in Tables 2–5 and Figures 2–5.

From the results we can make three observations:

1. Antirandom testing generally gives better coverage values.
2. Testing using the checkpoint encoding is generally better than purerandom testing.
3. As expected, coverage may vary when applying random tests generated using different seeds as shown in Figures 2–5. In some situations, it can be better than random testing with checkpoint encoding.

In exercising the STRMAT program as instrumented by GCT, we find that there are 18 binary conditions, 9 loop conditions, 4 multiple conditions, 15 relational operator conditions. So the total conditions for STRMAT is 46, and the total coverage is given by the sum of coverage percentage for each condition.

In general, there is no significant difference among the testing approaches in the first few test vector applications. Sometimes the coverage achieved by random testing appears to rise quickly. This is because random testing has a better chance of probing homogeneous

cases more effectively at the beginning. Antirandom testing, however, tries to maintain a balance between the boundary cases and the homogeneous cases, with the objective of achieving overall better test coverage after a reasonable number of test vectors have been applied.

Table 2: Branch coverage(%) for STRMAT

Test No.	AE	RE	RW1	RW2
1	72.22	72.22	77.78	72.22
2	72.22	72.22	77.78	77.78
6	83.33	72.22	94.44	77.78
10	100	83.33	94.44	83.33
14	100	94.44	94.44	83.33
16	100	100	100	83.33
20	100	100	100	83.33

Table 3: Loop coverage(%) for STRMAT

Test No.	AE	RE	RW1	RW2
1	33.33	33.33	33.33	33.33
2	44.44	44.44	33.33	33.33
6	77.78	44.44	55.56	44.44
10	88.89	44.44	55.56	44.44
14	88.89	66.67	55.56	44.44
16	88.89	66.67	66.67	44.44
20	88.89	66.67	66.67	44.44

Table 4: Relational Operator coverage(%) for STRMAT

Test No.	AE	RE	RW1	RW2
1	53.33	53.33	60	60
2	53.33	53.33	60	60
6	66.67	66.67	93.33	73.33
10	93.33	80	93.33	86.67
14	93.33	80	93.33	86.67
16	93.33	93.33	93.33	86.67
20	100	100	93.33	86.67

Table 5: Total code coverage(%) for STRMAT

Test No.	AE	RE	RW1	RW2
1	58.70	58.70	63.04	60.87
2	60.87	60.87	63.04	63.04
6	76.09	65.22	86.96	69.57
10	95.65	73.91	86.96	76.09
14	95.65	82.61	86.96	76.09
16	95.65	91.30	91.30	76.09
20	97.83	93.48	91.30	76.09

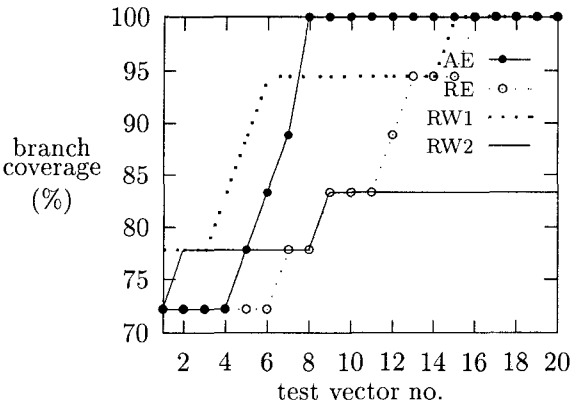

Figure 2: Branch coverage for STRMAT

3.3.2 The TRIANGLE program

This triangle example is used by Jorgenson [12]. Demillo [6] has also discussed test data selection for this program. Given three integers as input values for the three sides, TRIANGLE classifies whether we have a legal triangle or not. If the triangle is legal , there is a further classification whether it is isosceles, equilateral or scalene triangle. Any combination of input sides where the sum of the inputs of any given two sides is less than or equal to the third side is classified as "Not a Triangle".

Table 6 shows the checkpoint encoding used.

Table 6: Encoding scheme for TRIANGLE

Field	Bits	Value	Significance
Not a Triangle	b4,b3,b2,b1,b0	x1111	a+b < c, a!=b or a=b
		x1001	b+c < a, b!=c or b=c
		x0011	a+c < b, a!=c or a=c
		x0100	a+b=c, a!=b or a=b
		x0101	b+c=a, b!=c or b=c
		x1100	a+c=b, a!=c a=c
Legal Triangle	b4,b3,b2,b1,b0	01010	a=b
		11010	a=c
		00110	b=c
		10110	a=b=c
		rest	scalene

Side a,b,c are integer values in [1..200]
Here x indicates both 0 and 1

We used the triangle example to examine how dif-

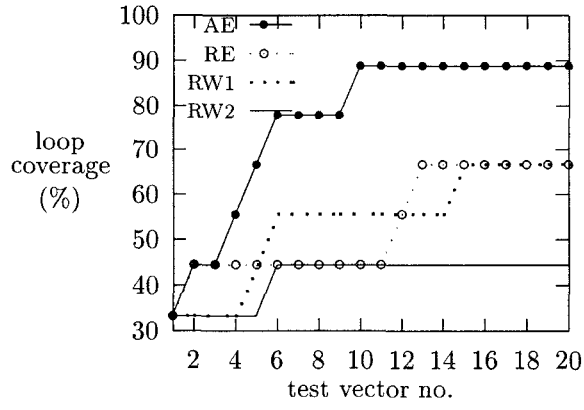

Figure 3: Loop coverage for STRMAT

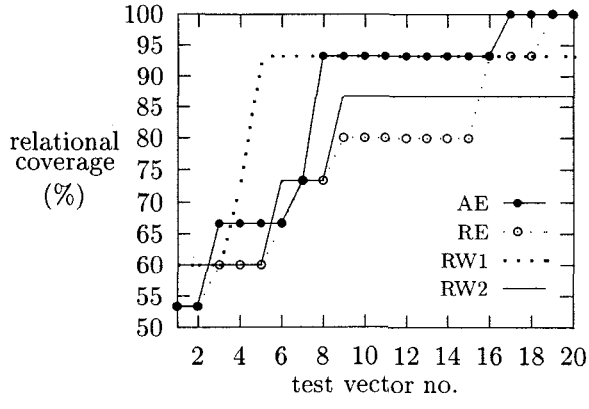

Figure 4: Relational coverage for STRMAT

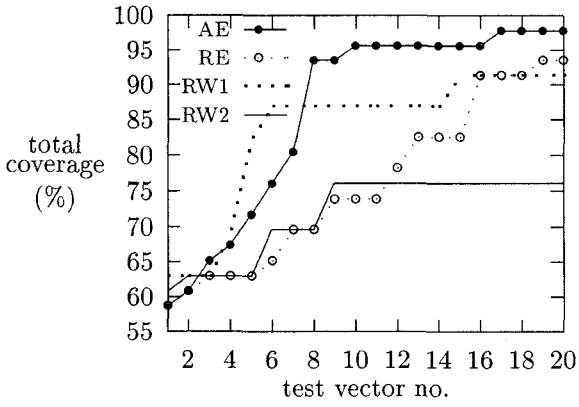

Figure 5: Total code coverage for STRMAT

ferent encoding schemes affect test coverage. The motivation for this is to see if we can come up with efficient encoding scheme and to understand the underlying reasons why some encoding scheme give better coverage than others.

The triangle example demonstrated that the checkpoint encoding exercise can actually force the tester to look more closely at the specification. Studies [12, 17] have shown that one of the causes for software bugs that are not being identified early enough is that testers were not exercising the specification fully. At first, our encoding did not take into account the "equal" part in the requirement that says if the sum of any two sides is less than or equal to the third does not constitute a triangle. The poor coverage results for the initial encoding scheme alerted us to this missing specification. This indicated that we were not capturing the specification fully.

Another lesson learned in the checkpoint encoding exercise for the triangle is that it is important the bit values assigned in the encoding scheme should map to the combinations occurring in antirandom test vector sequence if the number of tests applied is small. Otherwise, some of the conditions identified for checkpoint encoding can have bit values assigned that may not be triggered by the antirandom test vectors. This concern does not apply to random testing with checkpoint encoding because the input vector sequence is obtained randomly. However, for antirandom test sequence each test vector is chosen to be as far away from all previous test vectors as possible. When dealing with less than exhaustive testing, we can choose encoding such that each antirandom test vector in the sequence exercises a different aspect of the software

under test as compared with earlier tests. This is specially important in getting higher coverage quickly when dealing with much less than exhaustive testing.

After experimenting with several encoding schemes, we obtained the encoding scheme which is shown in Table 6. Using this encoding scheme, we can map the various boundaries to the early part of the antirandom test sequences, thus resulting in the highest coverage. For the special "Not a Triangle" case, we assign each vector for each situation. For the special case of an equilateral triangle, one input vector was assigned. Similarly, one input vector each was assigned to the three situations that make a triangle isosceles (there are three situations where any two sides are equal).

In exercising the TRIANGLE program as instrumented by GCT, we find that there are 22 binary conditions, 8 multiple conditions, 18 relational operator conditions. So the total conditions for TRIANGLE is 48, and the total coverage is given by the sum of coverage percentage for each condition.

Note that in this case, an encoding scheme that uses a minimum number of bits results in a single dimension as given in Table 6. This makes the effectiveness of antirandom testing very dependent on the code assignment used. Real problems would often be complex and would involve multiple dimensions. There the results would have a smaller dependence on code assignment.

For this encoding scheme, the total code coverage obtained for TRIANGLE is shown in Table 7 and Figure 6.

Table 7: Total code coverage(%) for TRIANGLE

Test No.	AE	RE	RW1	RW2
1	25	25	25	25
4	54.17	56.25	25	33.33
7	66.67	66.67	33.33	33.33
10	77.08	72.92	35.42	33.33
14	83.33	77.08	35.42	33.33
18	85.42	79.17	35.42	47.92
22	87.50	79.17	35.42	47.92
26	89.58	79.17	35.42	47.92
28	89.58	79.17	35.42	47.92
30	91.67	79.17	35.42	47.92

3.3.3 The FIND program

This program takes an integer array B of size $S \geq 1$ and index F. The program sorts the array elements

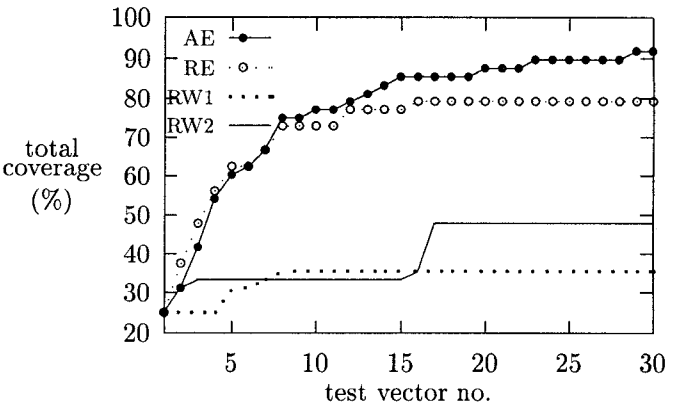

Figure 6: Total code coverage for TRIANGLE

such that all elements to the left of B(F) are no larger than B(F), and all elements to the right of B(F) are no smaller than B(F). The legal range for F is $1 \leq F \leq S$. In [13], Malaiya examined this program to illustrate how checkpoint encoding can be used.

The encoding scheme for this program shown in Table 8 is similar to what was described in [13]. The following subdomains can be identified from the problem specification in considering black-box testing.

Table 8: Encoding scheme for FIND

Field	Bits	Value	Significance
Array Size	b1,b0	01	1,2
		rest	> 2
Array status	b4,b3,b2	110	already ordered
		100	reverse ordered
		011	all equal
		rest	randomly ordered
Element Values	b7,b6,b5	010	all positive
		101	all negative
		rest	mixed
F points to	b9,b8	10	first element
		01	last element
		rest	a middle element

Ranges:

Array size: $1 \leq n \leq 10$

Index: $1 \leq F' \leq 10$

Element values: $0 \leq V' \leq 511$, $-256 \leq V \leq 255$

In assigning bit values to various subdomains and the categories within the subdomains, the first consideration is how many bits to assign to each subdomain. The second issue is to distribute the values among the categories within the subdomain suitably. A subdo-

main usually would incorporate a homogeneous or a normal range that largely spans the subdomain space. In addition, within the subdomain are the categories or boundaries and special situation that are special and require a category assignment. The homogeneous case normally would have more bits combinations so that a larger fraction of the assigned values would fall in this range. The boundary or the corner cases are assigned fewer specific bit combinations.

In exercising the FIND program as instrumented by GCT, we find that there are 20 binary conditions, 15 loop conditions, 4 multiple conditions, 27 relational operator conditions. So the total conditions for FIND is 66, and the total coverage is given by the sum of coverage percentage for each condition.

As shown in Figure 7, test coverage quickly reaches high coverage for all three test approaches by the application of only a few test vectors and no significant distinction among the three is observed. There are two reasons for this. First, the implementation of FIND is quite simple and separate handling of the special cases is not implemented. Thus the identification of the categories for the subdomains does not assist in achieving higher coverage. Secondly we note that good coverage for all the testing approaches is obtained with only 3 or 4 inputs. That implies that this is a highly testable program. It is known that random testing is quite effective in exercising attributes that have high testability. For the real problems, such a situation is unlikely to occur. For a large and complex program there will be many potential defect sites that will be hard to reach and exercise, these are the kind of defects that constitute the real challenge to the testers. Purely random testing will be not efficient in testing for such defects which may be triggered only under special input conditions. The checkpoint encoding and antirandom testing are formulated to generate such test cases.

In black-box testing one can not make any assumptions about how the problem is implemented as would be done in case of white-box testing. However, the encoding scheme should be general enough while identifying reasonable categories that are likely to be implemented in a special way and thus end up in a different section of code in a practical implementation. In fact, in an industry setting where performance is an important criteria, a common performance optimization technique is to handle special cases separately. For instance, there may be an upfront test to check for the equality of all array element values. If they are all equal, then you bypass the sorting part and the array index, which presumably is initialized to the first element of the array is left unmodified. The program then quickly exits rather than blindly trying to sort the elements, even though it is not needed in this case. In this situation, the encoding scheme we outlined having a subdomain category for all equal values will quickly generate a test to exercise this situation. Random testing would be very unlikely to exercise this scenario and thus under certain conditions the checkpoint encoding scheme would give us better coverage quickly. Even checking for equal values may not be necessary if say the program has a third argument that passes a hint such as all array element values are equal or are already sorted. We can see that the encoding scheme we have proposed maintains a balance between the general and the particular or boundary conditions based on the problem specification. It is much easy to identify possible special cases and generate test cases for them.

Table 9: Total code coverage(%) for FIND

Test No.	AE	RE	RW1	RW2
1	77.27	25.76	78.79	25.76
2	77.27	75.76	84.85	71.21
3	80.30	89.39	84.85	81.82
4	83.33	92.42	84.85	90.91
5	84.85	92.42	84.85	90.91
6	86.36	92.42	84.85	90.91
7	90.91	92.42	84.85	90.91
8	90.91	92.42	84.85	90.91
9	90.91	92.42	86.36	90.91
10	90.91	92.42	86.36	90.91

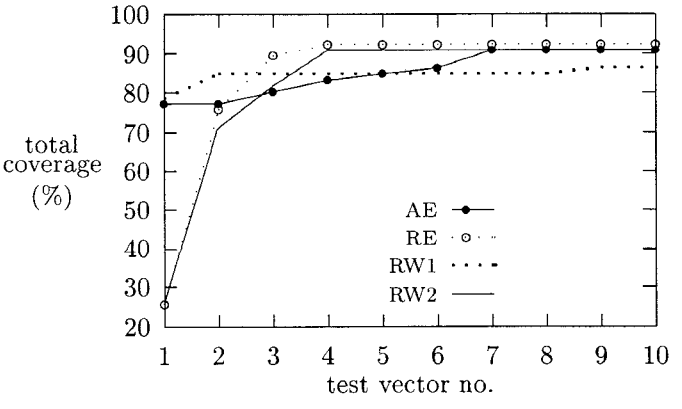

Figure 7: Total code coverage for FIND

3.4 Infeasible path condition

The code coverage obtained above (e.g., figure 7 - Total code coverage for FIND) would actually reach higher coverage percentage values quickly if steps were taken to remove infeasible path conditions. Infeasible path conditions are ones that the coverage instrumentation tool, GCT, flags as not traversed in its report but close examination of the code reveals that test path can never happen based on the specification of the problem. For instance, a report from GCT for the FIND program after a few test vectors are applied shows the following:

```
"find.c", line 17:  loop zero times:  0, one
time:  1, many times:  15.
"find.c", line 26:  loop zero times:  0, one
time:  1, many times:  15.
```

Examination of lines 17 and 26 as shown in the program listing (see Appendix [29]) shows that it is impossible to traverse the for loop 0 times, since the problem specification say that the array size, n, is greater than 1:

```
17 for(i=1;i<=n;i++)
18 {
19 scanf("%d", &a[i]);
20 gets(mystr);
21 }


26 for(i=1;i<=n;i++)
27 printf("%5d\n", a[i]);
```

GCT [16] does have a mechanism to edit the report and remove infeasible path conditions, once the tester determines which ones they are. This was not done in the above coverage plot data, as it was not germane to the issues that are the subject of the paper.

4 Concluding Remarks

In this work we have demonstrated that it is possible to have automatic test generation that can be more efficient than random testing. The benchmark examples considered here give us insight into some important considerations:

1. Even with random testing there is an aspect of systematic testing in the sense of deducing from the problem specification, the range of values for the specific dimension from which random values are selected. Understanding of the subdomain dimension help narrow the space from which random tests are chosen.

2. Using a coverage measure as an indicator of effectiveness is limited, because code coverage does not ensure uncovering errors.

3. In antirandom testing with checkpoint encoding, we are much more likely to probe boundaries than random testing, and thus the antirandom technique may detect faults that may not be directly associated with some of the coverage measures.

This investigation suggests certain basic rules for a checkpoint encoding scheme to be efficient:

a) It is important to look at the problem specification and the subdomains that have been identified and recognize what are the legal, illegal and boundary conditions in the input space that need to be exercised. Here the emphasis is to cover as much as possible using a suitable of homogeneous, boundary and illegal cases with the test vectors being applied.

b) To decide how many bits are appropriate for the encoding scheme, we need to identify different ranges of identifiable similar characteristics. One needs to analyze each subdomain and assign binary combinations to each range within the subdomain. For instance, the problem statement may involve illegal conditions, but the things that make it illegal could be due to many ways that results in illegal behavior.

c) In assigning bits for the encoding scheme within each subdomain we need to give more weight to the most common cases, which almost always map to the legal input range.

d) The total bit length of the encoding scheme is the concatenation of the bits in each subdomain.

e) In devising the checkpoint encoding scheme, it is important that the subdomains in the input space that we identify really exercise different aspects of the problem. If a subdomain is not orthogonal to the rest, then we can encounter a situation where some combinations for the input test vector may not be generated resulting in low test coverage. Also, one needs to ensure that conflicting assignments for different fields do not occur.

For this study, the examples used are small and are not representative of real problems. Future research would use larger programs to apply the concepts developed here. For larger programs the coverage measurements would be more significant. Antirandom testing is specially formulated to be effective when there are multiple dimensions. Larger programs will be able to demonstrate the capabilities of this scheme better.

References

[1] V. Basili and D. Weiss, "A methodology for collecting valid software engineering data," *IEEE Trans. Software Engineering*, Vol. SE-10, pp. 728-738, Nov. 1984.

[2] T. Y. Chen and Y. T. Yu, "On the expected number of failures detected by subdomain testing and random testing," *IEEE Trans. Software Engineering*, Feb. 1996, pp. 109-119.

[3] D. M. Cohen, S. R. Dalal, J. Parelius and G. C. Patton, A., "The combinatorial design approach to automatic test generation," *IEEE Software*, Sept. 1996, pp. 83-88.

[4] D. M. Cohen, S. R. Dalal, A. Kajla and G. C. Patton, A., "The automatic efficient test generator (AETG) system," *Proc. ISSRE*, Nov. 1994, pp. 303-309.

[5] S. R. Dalal, J. R. Horgan and J. R. Kettenring, "Reliable Software and Communication: Software Quality, Reliability and Safety," *Proc. International Conference on Software Engineering*, 1993, pp. 425-435.

[6] R. A. Demillo, R. J. Lipton and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, Apr. 1978, pp. 34-41.

[7] R. A. Demillo and A. J. Offutt, "Constrained based automatic test data generation," *IEEE Trans. Software Engineering*, Vol. SE-17, No. 9, pp. 900-910, Sept. 1991.

[8] J. W. Duran and S. C. Natfos, "An evaluation of random testing," *IEEE Trans. Software Engineering*, July 1984, pp. 438-444.

[9] R. G. Hamlet and R. Taylor, "Partition testing does not inspire confidence," *IEEE Trans. Software Engineering*, Vol. SE-16, No. 12, pp. 1402-1411, Dec. 1990.

[10] D. Hamlet, "Are we testing for true reliability," *IEEE Software*, July 1992, pp. 21-27.

[11] W. E. Howden, "The theory and practice of functional testing," *IEEE Software*, Sept. 1995, pp. 6-17.

[12] Paul C. Jorgensen, *Software Testing: A Craftsman's Approach*, CRC Press, New York 1995.

[13] Y. K. Malaiya, "Antirandom Testing: Getting the most out of black-box testing," *Proc. International Symposium On Software Reliability Engineering*, Oct. 1995, pp. 86-95.

[14] Y. K. Malaiya, N. Li, R. Karcich and B. Skbbe, "The relationship between test coverage and reliability," *Proc. International Symposium On Software Reliability Engineering*, Nov. 1994, pp. 186-195.

[15] R. Mandl, "Orthogonal Latin Squares: An application of experiment design to compiler testing," *Comm. ACM*, Oct. 1985, pp. 1054-1058.

[16] Brian Marick, *The Generic Coverage Test (GCT) User's Manual*, 1981.

[17] G. Meyers, *The Art of Software Testing*, John Wiley & Sons Inc., New York, 1979.

[18] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *Journal of Systems*, Vol. 14, 1984, pp. 289-300.

[19] K. C. Tai, "Condition based software testing strategies," *Proc. COMPSAC* 1990, Oct. 1990, pp. 564-569.

[20] Markos Z. Tsoukalas and Joe W. Duran, "On some reliability estimation problems in Random and Partition testing," *IEEE Transactions on Software Engineering*, Vol 19, No 7, pp. 687-697, Jul. 1993.

[21] M. D. Weiser, J. D. Gannon and P. R. McMullin, "A comparison of structural test coverage metrics," *IEEE Software*, Vol. 19, no.6, 1989, pp. 80-85.

[22] N. Weiss, "Comparing test data adequacy criteria," *Software Engineering Notes* vol. 14, no. 6, pp. 42-49.

[23] E. J. Weyuker, S. N. Weiss and R. G. Hamlet, "A comparison of program testing strategies," *Proceedings of the fourth Symposium on Software testing, Analysis and Verification*, Victoria, Canada, Oct. 1991, pp. 154-164.

[24] E. J. Weyuker, T. Goradia and A. Singh, "Automatically generating test data from a boolean specification," *IEEE Trans. Software Engineering*, May 1994, pp. 353-363.

[25] E. J. Weyuker and B. Jeng, "Analyzing partition testing strategies," *IEEE Transactions on Software Engineering*, Vol. SE-17, No 7, pp. 703-711, July 1991.

[26] L. White and E. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Engineering*, May 1980, pp. 247-257.

[27] W. E. Wong, J. R. Horgan, S. London and A. P. Mathur, "Effect of test set minimization on fault detection effectiveness," *IEEE International Conference on Software Engineering*, 1995, pp. 41-50.

[28] W. E. Wong, J. R. Horgan, S. London and A. P. Mathur, "Effect of test size and block coverage on fault detection effectiveness," *Fifth International Symposium on Software Reliability Engineering*, 1994, pp. 230-238.

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

[30] Huifang Yin, "Antirandom test patterns generation tool," *Project Report*, Colorado State University, Computer Science Dept., Fall 1996.

5 Appendix - FIND Program listing [29]

```
 1  #include <stdio.h>
 2  #define max    256
 3  typedef enum boolean {false, true} BOOLEAN;
 4  int  a[max+1];
 5  int  n, f;
 6
 7  main()
 8  {
 9    char *mystr;
10    int  i;
11    char *gets();
12    mystr = (char *)malloc (80);
13    scanf("%d", &n);
14    gets(mystr);
15    scanf("%d", &f);
16    gets(mystr);
17    for(i=1;i<=n;i++)
18    {
19      scanf("%d", &a[i]);
20      gets(mystr);
21    }
22    find(n, f);
23    printf("%5d\n", n);
24    printf("%5d\n", f);
25
26    for(i=1;i<=n;i++)
27     printf("%5d\n", a[i]);
28  }
29
30  find(n, f)
31  int n;
32  int f;
33  {
34    int m, ns, i, j, w;
35    BOOLEAN b;
36    b = false;
37    m = 1;
38    ns = n;
39    while ((m < ns) || b)
40    {
41       if (!b)
42       {
43         i = m;
44         j = ns;
45       }
46       else
47         b = false;
48       if (i>j)
49       {
50          if (f>j)
51          {
52             if (i>f)
53               m = ns;
54             else
55               m = i;
56          }
57         else
58           ns = j;
59       }
60       else
61       {
62         while (a[i] < a[f])
63              i = i+1;
64         while (a[f] < a[j])
65              j = j-1;
66         if (i <= j)
67         {
68           w = a[i];
69           a[i] = a[j];
70           a[j] = w;
71           i = i+1;
72           j = j-1;
73         };
74         b = true;
75       }
76     }
77  }
78
```