COLORADO STATE UNIVERSITY

November 30, 1999

WE HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER OUR SUPERVISION BY SANJAY R. PILLAI ENTITLED ISSUES IN AUTOMATION OF CHECKPOINT ENCODING FOR ANTI-RANDOM TESTING BE ACCEPTED AS FULFILLING IN PART REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE.

Committee on Graduate Work

Adviser

Department Head

THESIS

ISSUES IN AUTOMATION OF CHECKPOINT ENCODING FOR ANTI-RANDOM TESTING

Submitted by

Sanjay R. Pillai

Department of Computer Science

In partial fulfillment of the requirements

for the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Spring 2000

ABSTRACT OF THESIS

ISSUES IN AUTOMATION OF CHECKPOINT ENCODING FOR ANTI-RANDOM TESTING

Checkpoint Encoding is the process of representing any input domain of a software system into a binary valued domain. Any input data value (i.e.an element of any application domain) can now be translated with minimal loss of information into a binary valued string. This abstraction of any application domain into an uniform format allows a variety of testing techniques to be applied consistently and universally. Anti-random testing is one such scheme that takes advantage of binary representation that checkpoint encoding provides and has shown good results. The checkpoint encoding process is usually applied manually and in an arbitrary fashion. The result varies depending on the choices made by individual test engineer carrying out the process. This thesis attempts to improve checkpoint encoding by automating checkpoint encoding and, by understanding what factors control the effectiveness of checkpoint encoding.

The first part of this thesis deals with the automation of checkpoint encoding. The need for automation is self evident as it is a prerequisite for any testing method to be used by the industry. If checkpoint encoding cannot be automated any testing method that uses checkpoint encoding consequently suffers. Automation is carried out using Z specifications. A method to automate checkpoint encoding based on extracting partitions from Z specifications is proposed. Verification of this method is done by comparing its results with that from previous studies. By extending this method, using partitions extracted from Z specifications, the finite state machine of a software system can be extracted. Using the finite state machine specification of a system to test the system is an important strategy especially for concurrent systems. This thesis extends checkpoint encoding to use the finite state machine to improve its applicability to increasingly complex systems.

In the second part of this thesis, different checkpoint encoding schemes are carried out to understand what makes a good checkpoint encoding scheme. This understanding is quantified using a graphical measure to visually describe how a checkpoint encoding scheme can detect hard to cover branches. Results prove that allocating more bits towards incorrect or illegal input data values

improves the ability of a testing scheme using such a checkpoint encoding scheme, to test hard to cover branches.

Lastly the thesis introduces a new method of testing that targets these same hard to cover branches. Preliminary results indicate that this method is effective in this regard. A proposed combination of this method and anti-random testing using checkpoint encoding would combine their individual advantages. By looking into several different sub-areas this thesis has increased the understanding of, and consequently, the effectiveness of checkpoint encoding. Till such time as another method is attempted, checkpoint encoding is best used with the anti-random test data generation method.

Sanjay R. Pillai
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Spring 2000

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Issues in testing

The need to test

Testing is both a risk reduction exercise and a process of assuring the quality of delivered software products. Effective testing significantly enhances the likelihood of delivering real business benefits while reducing the likelihood of serious problems after delivery. Even when software is developed using a rigorous discipline, it will contain a significant number of bugs.

Testing software is often viewed as either an exercise to show that a program is correct or that a program is incorrect. Both these views are misconceived. In the case of the latter, a single test can show a program isn't correct, but an infinite number of tests are required to show a program is correct. In the case of the latter, every error discovered requires a test and to discover all possible errors would require all possible tests. Since both views require an infinite number of tests to be performed, the expectation that a test set will uncover every error in a program is unrealistic.

For every test we perform, whether successful (the test finds an error) or not (the test finds no error), we increase our confidence, in a probabilistic sense, that the program developed works correctly. Therefore, the view that testing is waste of time and resources if it reveals no errors is a misnomer. The idea of testing giving confidence is closely linked to the idea of test metrics. Much of the early work in formalizing testing was concerned with the question of what a test is. In a seminal paper, Goodenough and Gerhart (Goodenough and Gerhart 1975) introduced the validity and reliability metrics. A test criterion is valid if and only if at least one of the tests it selects causes an incorrect program to fail. A reliable criterion is one where for all test sets which can be selected using it, the program either succeeds on all the tests in the set or fails on them all. If a test is reliable then we can draw conclusions about the correctness of the program.

Generation of a test set which is valid and reliable is an unsolvable problem since we would be

able to distinguish between those programs that are correct and those that aren't. In (Weyukar and Ostrand 1980), the authors introduce the concept of test adequacy criterion which allow the tester to determine when testing should stop, i.e. the point where we can have reasonable confidence that the program behaves correctly. This problem is also unsolvable. However, in (Weyukar 1988) the author defines certain properties which an adequate test set should have.

The argument of an infinite number of tests being required to show a program is correct is frequently used by those who decry the need for testing. Instead of testing, they state that program proving (i.e. a mathematical proof) is better at showing program correctness than testing. But this is not necessarily valid since proofs do not take into consideration the environment of the system. Testing provides quantitative evidence that a program meets the requirements of the customer as given in a requirements specification. Proofs only carry weight if people are willing to accept them and for many customers, testing provides a more tangible form of evidence that a program is required. By testing, we can see that the program and the platform work together correctly. A test checks a real product in a real environment, a proof checks a real product in an abstract environment. We are also implicitly testing that the tools used in the development have produced an acceptable program.

The arguments for and against proof and testing usually give too much of a polarization. For some developments a full proof may be required, for others a suite of test cases may be sufficient to demonstrate that the program performs as desired. Program proving consumes large quantities of time and money, a suite of test cases is likely to be more cost effective. For most programs, a mix of the two techniques should be used as appropriate. The choice of proof or testing should be based on the risk resulting from the program still containing errors after the proof or test is performed. In high risk applications, a high level of confidence that the number of remaining errors is very small should be achieved.

Automation in testing

The need to automate the testing process is self evident when we consider it. Software development organizations face three major challenges in validation and verification process : maintain high product quality, deliver fastest to market, minimize testing costs. All the while the software complexity grows. Testing consumes up to 60% of the time of a project (Beizer 1990). What also needs to be taken into consideration is that much of the time spent in activities classified as development or maintenance is actually involved in testing of new functions or upgrades to existing systems. While the time needed for most phases of the software life-cycle i.e. requirements analysis, design, coding etc. goes down as the system ages, the testing levels remain constant : which represents a major

obstacle to reducing cycle times.

Traditional manual approaches are no longer effective. Most of testing is repetitive and involves a huge number of tests. Testing tools can do most of the tedious work : they are robust, can work continuously, and are better suited for regression testing. It can test more number of cases and do it faster. Most of testing is guided intuitively. Even when done manually but systematically there are chances for error to creep in. Software testing lags behind hardware testing where use of tools is regarded mandatory. The main deterrents are the learning curve involved in using tools and the resistance to adopt new tools or techniques, especially during product development.

In the creation of test cases automation lags behind the most. When people think of automation, the "capture/playback" paradigm is commonly perceived. This paradigm involves recording the input actions taken to execute a test case by a human(capture) and repeating them automatically under direction of a script(playback). While automation of test execution is important, and there are many tools for this purpose, this does solve all our problems vis-a-vis test automation. The test case or more realistically the test data generation process is far more difficult. Thus any testing method that can provide an automated process would have a clear advantage over other methods. This is the rationale for automation of checkpoint encoding in this thesis.

The role of specification in testing

The achievement of the full potential of software testing requires an integrated approach which involves every phase of the life cycle. Not only must testing be carefully planned, but completion of planned testing must also be measurable to provide managers with a clear indication of the level of risk remaining at every stage. Testing is perhaps the least understood stage of the software life-cycle, and since it is a "tail end" activity (as distinct from a "front end" activity such as requirements analysis), testing is often hastily carried out so that a token effort is seen to have been made.

The waterfall model is perhaps the most commonly used model of the software life-cycle. Each stage depends only on the stage directly before it. Consider module testing which depends on the coding stage for the white-box testing of the module. Testing solely against the code developed can show that the code works correctly (verification), but only with respect to the code itself. It cannot show that the code behaves as required by the specification (validation) without performing tests derived from the specification. This indicates that the specification plays an important part all stages of testing, and this forms the basis of automation in this thesis.

The notion of using specifications to generate tests comes from an observation of current software engineering practices. The information used to develop tests in the testing stages usually comes

directly from either the coding stage or from earlier testing stages, with little regard to the specification. Any program that claims to be an implementation of the specification should pass tests generated from it, and any deviations from the specification that have occurred during coding will be caught. Additionally, the time between the specification and the testing phases of the software may be lengthy. By using the specification as the basis of the tests, the problem to be solved is re-considered at numerous points in the later stages of the cycle. Many authors (e.g. Beizer (Beizer 1990)) state that designing tests before a program is coded is a good way to avoid introducing errors. Another advantage is that the specifications can provide test oracles. By utilizing the specification we can also reduce the number of tests by removing any superfluous tests or combinations of tests. However the dependence on specifications means we do not test the non-functional aspects of the program.

When generating tests from specifications formal specifications are chosen over informal ones so that the tests can be founded on an unambiguous statement of what the program should do. Bouge (Bouge, Choquet, Fribourg, and Gaudel 1986) state that the introduction of formal methods has provided a sound basis for black box testing, since prior to this the only formal object in the software life-cycle was the program itself. It is a result of this that current testing methods tend to be structurally based.

Test Effectiveness

Test effectiveness is a measure of how well the testing process has achieved it's goals. This is very difficult to measure. The goal of testing is ideally to remove all errors. This is not practically feasible for any software system of reasonable size or complexity. The goal of testing is defined in far more detail for each system under test. Depending on various factors including type of software, usage, time, cost etc. we come up with different goals for test effectiveness. Depending on the goals different strategies are formulated to achieve them.

A measure of test effectiveness could be the degree to which the tested software can be considered *dependable*. One approach to dependability estimation is reliability analysis; in which random testing is used to determine the probability of failure under various operational environments. Another approach is based on the concept of trustability, defined as the confidence in the absence of faults. Test effectiveness in turn depends on a couple of other measure : *detectability* and *testability*. Detectability is a property of a testing method. It is the probability that a method will detect faults when there are faults present. Testability on the other hand is more a property of the program under test. Testability can be defined as the probability that a piece of software will fail on its next execu-

tion during testing *if* the program contains a fault (Voas and Miller 1995). However other definitions (Committee 1990) stress the fact that testability can be defined for different testing methods. Test effectiveness as a measure can be defined in various ways. It could be defined more generally as the ease with which some test criteria can be satisfied during testing. Thus improving test effectiveness of any testing scheme is important.

1.2 Anti-Random Testing.

In Anti-random(Malaiya 1996) 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 anti-random testing paradigm, each test in the sequence attempts to exercise different areas of the software and thus has the potential of being more effective.

In anti-random 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 loss 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 anti-random test sequence and maximal Cartesian distance anti-random test sequence is discussed in detail in (Malaiya 1996). Each subsequent test vector t_i is then chosen such that the total distance between t_i and all the previous tests $t_{i-1}, t_{1-2}, ..., t_0$ is a maximum. The procedures presented by Malaiya have been implemented in the Anti-random Testing Generation (ARTG) program (Yin, Lebne-Dengel, and Malaiya 1997).

An integral part of anti-random 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 effectiveness is as optimal as possible.

1.3 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(Malaiya 1996) is to make the testing effort as effective as possible by converting the problem to that of constructing binary anti-random 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 in the thesis.

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 captured 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.

In devising an encoding scheme, one starts from the problem specification and tries to conceptually partition the problem space into subdomains, where each 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.

1.4 Problem Definition

The kind of testing scenario this thesis looks at is the unit testing of software. Module testing or unit testing is the verification of a single program module, usually in an isolated environment. This is the most basic kind of testing. There are two classes of unit testing : black-box and white-box testing. Black box or functional testing strategy uses specifications or the required behaviour of the software to design test cases. Anti-random testing (Malaiya 1996), used as the testing method in this thesis, falls into this category. Effectiveness of this testing method is based on branch or decision coverage test criteria (Clarke, Podgurski, Richardson, and Zeil 1989; Myers 1979).

This thesis focuses on the checkpoint encoding process in anti-random testing. The goal is to make anti-random testing a practical testing method. Checkpoint encoding is crucial since it provides the encoding which the anti-random test data generation scheme is dependent on. The first issue

dealt with is automation of checkpoint encoding. To be able to automate anti-random testing we must first automate the checkpoint encoding process. A practical first step is to attempt to reduce the need for manual intervention as much as possible. This is achieved by using formal specifications to extract the needed encoding. The second issue is the application of checkpoint encoding (and thus anti-random testing) to complex software systems i.e. those with a non-trivial state machine controlling it. The third issue is to better understand the effectiveness of the checkpoint encoding scheme. The measure for effectiveness that is considered in this thesis is it's ability to cover "hard to test" branches. This thesis looks at how various checkpoint encoding schemes cover hard to test branches.

1.5 Organization of the thesis

A review of the field is carried out as part of the background needed for this thesis. Since several issues are covered in this thesis, the background is particularly extensive. A simple introduction to the testing field is followed by a discussion into partition testing theory and terminology. Also described are the various methods of specification based testing. This has been a well researched field with several approaches proposed over the last decade. A brief explanation on each method is made. The third chapter proposes a method of generating a checkpoint encoding scheme from a Z specification. This is explained using an example. The performance of this method is verified by comparing the checkpoint encoding schemes that it generates with that generated manually for some standard testing problems. It's advantages and shortcomings are discussed.

The fourth chapter examines the issue of state in software system and how checkpoint encoding relates to it. The need to use the finite state machine, representing an understanding of the working of the system, to create test cases is well understood. A method to extract the state machine of the software based on the Z specification is proposed. It is based largely on the method proposed in the previous chapter for checkpoint encoding. A method to use checkpoint encoding when programs have retained state is proposed. Results using this method are shown to be quite effective to increase the structural coverage vis-a-vis traditional checkpoint encoding schemes.

The fifth chapter deals with our understanding of checkpoint encoding. To be able to measure this, the ability to cover or detect branches not normally triggered is used as a measure of effectiveness. This is in addition to the standard goal of higher structural coverage. To understand how different checkpoint encoding schemes are effective a distribution of coverage of the branches is used. Checkpoint encoding schemes based on different rules are experimented with. The basic idea is to prove which test allocation scheme works the best depending on the allocation of more test cases

(or bits in a checkpoint encoding scheme) for illegal partitions or input values rather than "usual" or legal input values. Based on the experimental results and the later analysis some guidelines are laid out for creating an effective checkpoint encoding scheme. Lastly, the thesis concludes with an overviews of what the most significant results are and what should be done to extend the work carried out in this thesis.

Chapter 2

Background

This thesis draws on principal ideas from software testing. This is discussed first before concentration shifts to reviewing various theories in partition testing and specification based testing techniques.

The testing process consists of test case generation, test data generation and test sequencing. The meanings assigned to these terms are not uniform in the literature which makes it difficult to compare the techniques. In particular, the distinction between test data and test cases is often blurred. What is referred to throughout this thesis is usually test data generation. In this review, the terms will have the following definitions:

A test case is a statement about what the test covers. It will consist of input information (the test specification) and some indication of the expected outcome of the test (the acceptance criterion). Therefore a test case can be seen as a tuple - (input criteria, acceptance criteria).

Test data are values submitted to a program in order to test it. There is a major link between test cases and test data, the test data instantiates the input and acceptance criteria of the test case.For example, consider a test case with input criteria given by conjunction of two constrained integers

$$(0 \leq A \leq 4 \wedge B * 8)$$

. There are a range of values which A and B can take which define a number of instantiated test cases. For example (0,8), or perhaps (3,152). If we consider boundary value analysis (see later), then we might end up with test cases (0,8), (4,8) which define cases just inside the boundary and (-1,7), (5,7) which are just outside. (In practice we would also consider combinations of A and B lying inside or outside the boundaries). Thus a test case is a criterion or rule while test data are any input data that satisfy that criterion. It is easy to see why both these terms have been used interchangeably.

Placing a system into some state in order to perform a test may require a series of other tests,

which may have already been performed independently. To remove the repetition, sequencing can be employed which arranges the tests in such a way that all test cases are used, but no test is repeated any more than is necessary.

2.1 Testing techniques : discussion

Testing techniques or testing strategies can be classified in many ways.

- *Black box* vs *White box* testing (or functional vs structural). This is classified by the information used to identify test cases. Black box testing is concerned with testing the functions that a program should perform. It considers the program to be a black box (i.e. it knows nothing about the internal structure/logic). The functionality (the relationship between the input and output) is given in a specification (which is at a higher level of abstraction than the program) which may be formally or informally specified. White-box testing on the other hand the tester knows the internal structure of the program and can use this information to devise tests.

- *Static* vs *Dynamic* testing methods. Dynamic testing involves executing a program with some form of test data. Static testing techniques do not involve execution of the program, and are most often employed in tests that also fall under the white box banner. Black box testing always involve program execution, therefore, black box techniques are always dynamic. White box techniques may fall into either the dynamic class (e.g. domain testing, path based testing, mutation analysis) or static class (e.g. symbolic execution, program proving).

- Another classification is by underlying approach.

  - *Structural testing* : specifies testing requirements in terms of coverage of a particular set of elements in the structure of the program or it's specification. This could be dataflow or code coverage based testing.

  - *Fault-based testing* : focuses on detecting faults (defects) in the software. A fault, in standard IEEE terminology, is an incorrect program component, such as a wrong relational operator or a wrong constant in an expression. Fault based testing depends on the characterization of large classes of faults which a much smaller test set can reveal the presence of any fault in a class. Mutation analysis and perturbation testing are fault based methods.

  - *Error-based testing* : requires test cases to check the program on certain error prone points according to our knowledge about how programs typically depart from their specifications.

An error is defined as the incorrect behaviour resulting from a fault. Error detection methods depend on the characterization of a small number of outputs which can determine if all or almost all of the program outputs are correct. Domain testing falls into this category.

2.1.1 White Box Techniques

In white-box testing, tests are devised using the program code. Traditionally testing has been along these lines, and consequently there are numerous techniques.

Symbolic execution involves substituting symbols for program variables and the results of execution are expressions containing the substituted symbols. Mutation analysis (DeMillo 1989) takes the program under test and produces many variant or "mutant" programs which differ slightly from the original (e.g. $a <$ replaced by $a \leq$). The theoretical underpinning behind this relies on the "competent programmer hypothesis", which states that a competent programmer will produce code that is very close to being correct, and that the errors introduced by the mutations represent possible programmer errors. Each mutant is tested with a set of test data, and mutants which yield incorrect results are said to be "dead". If, following execution, a number of the mutants are still alive, then the test data has failed to detect the error introduced. The test data is then refined until an acceptable number of mutants have been killed. The usefulness of mutation testing arises due to the "coupling effect", which states that if the test data set can detect the small errors which have been introduced by mutation, then it is likely to detect larger and more important errors. Empirical results have shown that mutation analysis is a very effective technique.

Path based testing aims to show that every execution path through the code is executed at least once (i.e. no island or inaccessible code exists). For large programs this will involve a very high number of paths due to the loops and branches within the program. Loops are a particular problem since the number of paths through the program is multiplied by the number of iterations of each loop, as a result, it is unreasonable to try to test every path. To shorten the time taken for the testing, a limit can be given for the maximum number of iterations of a loop, even so, path based testing can by a lengthy process.

Code coverage criteria are often used in path based testing. They measure the percentage of statements that need to be executed in order for confidence in the program to be achieved. A fully tested program would have 100 % path coverage, often infeasible. To reduce this problem, the notion of "branch coverage" is introduced. For branch coverage, each branch has to be executed at least once. For loops this implies that there are two (rather than an infinite) number of conditions to

handle, one when the loop guard is satisfied, and one where it is not.

Domain based testing can be both a white-box and a black-box technique. In the context of white-box testing, paths through the code identify test domains. A test is run for each of the identified domains. If the test succeeds, then it can be assumed that for any value selected in the domain, the program will compute the correct result. The technique usually incorporates boundary value analysis, originally proposed in (Myers 1979) where values close to domain boundaries are selected since they are the most common source of errors. For example, if a branch is of the form *if* $(x \neq 20)$ *then* ..., boundary values of 19 and 20 would be selected, being the values on which the branch is executed or not executed respectively.

Domain analysis in either the black-box or white-box case makes the assumption that the program processes its input correctly, but that the domains it operates on have been incorrectly defined. Therefore, domain testing is not sufficient for producing well tested software, and should be augmented with other techniques.

2.1.2 Black Box Techniques

Black-box techniques assume no knowledge of the program under test which means that all tests must be either guessed or, more scientifically, devised from a program specification.

Random testing falls into the first category. The technique uses a random number generator to produce test inputs for the program. Any input to a program can be regarded as numeric because all inputs are binary at the lowest level. It is intuitive to think that the effectiveness of this technique would be poor, however, (Duran and Ntafos 1984) states that randomly generated tests are free from any assumptions that we as testers might make about which tests might be useful. They illustrate the use of random testing on a series of (mainly mathematical) programs which have some known error and present some promising results. For example, in a program to calculate the "sine" function, 50 tests were generated and 22 of them found the error. This does not sound impressive until the speed at which tests can be generated is considered. The errors found per unit time compares favourably with other techniques.

Cause-effect graphing (Myers 1979) takes a specification (or more commonly a sub-section of a specification) and identifies causes (e.g. inputs to the system) and effects (e.g. outputs from the system). A graph is constructed which links causes with their effect, and the graph is annotated with impossible causes and effects (i.e. those outside the operating domain). A trace of the state conditions represented by the graph is made, and this information is used to construct in a limited entry decision table where each column in the table represents a test case. This method was originally

used as a means of extracting the important information from informal specifications. To a certain extent, the use of formal methods (particularly the model based techniques) has removed the need for the earlier stages of this method.

Domain based testing is considered in (Beizer 1990), where the author states that, while the theory behind domain based techniques is primarily structural, the best results for domain based testing come from using specifications. Domains are generated by identifying partitions on the input space made in the specification. Test data values from these domains are then used to test the implementation. Additionally, Beizer notes that a specification may contain incomplete domains (those inputs for which no specifi cation exists) and inconsistent domains (two or more contradictory specifications for any given input). Neither of these are possible in an implementation. Incomplete domains don't exist because the program will do something for values it does not recognize, even if the effect is a crash. Inconsistent domains can't exist as the program will execute deterministically. This is described in more detail in the next section.

The variety of techniques available is quite broad. Each method has its own advantages and disadvantages, but judicious selection of a number of contrasting techniques from each strategy would enable a thorough set of test cases to be constructed.

2.2 Partition Testing

The basic idea of domain analysis and domain testing is to partition the input output behaviour space into subdomains (or subsets) so that the behaviour of the system on each subdomain is equivalent. That is if the software behaves correctly for one test case within a subdomain then we can assume that the behaviour of the software will be correct for all points within that subdomain (Zhu, Hall, and May 1997). The term *partition testing* is used in a very broad sense, to refer the family of domain testing strategies. However a "partition" is in a formal sense, a *disjoint* division of the domain into subdomains which together span the domain. Thus subdomain testing is the correct term when referring in general to these testing strategies. Partition testing is actually a subset of subdomain testing. Here we use it in the broader sense unless otherwise specified.

When a subdomain exhibits the ideal behaviour described above then it termed as a *homogeneous* subdomain. Such a subdomain in which every element gives an incorrect answer for the software is called a *revealing* subdomain (Weyukar and Jeng 1991). Each testing strategy attempts to first partition the input (and/or output space) subdomains. Then the question is how many test cases should be used in each subdomain and from where in the subdomain should these test cases be chosen.

2.2.1 Specification-based Input Space Partitioning

The software input space is partitioned according to the specification of the program. This is the basis for automation of checkpoint encoding. This is described in more detail in the next section on specification-based testing.

2.2.2 Program-based Input Space Partitioning

This is another approach to partition testing. The input space is partitioned according to the program logic. A subdomain is defined by an execution path in the software (Zhu, Hall, and May 1997). Thus two points in input space are considered to be in the same subdomain if they cause the same computation sequence of the program. This is also known as path analysis testing (Clarke, Hassell, and Richardson 1982) since we construct subdomains for (selected) paths in a program. Associated with a path is a "path domain" or subdomain of input space and "path computation" or the function computed by the statements in that path. The path computation is most often constructed by symbolic execution techniques.

Thus there are two types of errors that we check for. Domain errors are those due to incorrect subdomains i.e. incorrectness in the selection of boundaries for a subdomain. These can again be subdivided into path selection error and missing path error. Computation errors are those due to incorrectness of the implementation of the computation on the subdomain. These two errors give rise to two types of testing. Computation errors are detected by both program and specification input space partition testing. Domain errors are detected by domain testing which is an offshoot of program based input space partition testing.

2.2.3 Domain Testing

White and Cohen proposed domain testing which attempts to uncover errors in a path domain by selecting test data on and near the boundaries of the path domain. The goal is to detect domain errors. There are several variations involving the number of off (outside) and on the border test data points. $Nx1$ domain testing strategy involves N on and 1 off point, this can detect parallel shift in a linear border. NxN strategy involves N on and N off points and can detect parallel shift and rotation of linear borders. Clarke *et al* (Clarke, Hassell, and Richardson 1982) suggested the use of vertices instead of borders to improve efficiency.

There are also approaches involving combining specification and program based input partitioning. The revealing subdomains approach (Weyukar and Ostrand 1980) uses the intersection of problem partitions, created manually from the specification, and path partitions, derived from

program paths. This intersection creates a set of equivalence classes which are then used for testing. The equivalence partitioning approach (Richardson and Clarke 1985) is also similar involving the use of procedure partitions which are the intersection of program based path domain and specification based specification domain.

2.3 Specification Techniques

Specification techniques divide into three classes, formal, structured and informal. Formal methods are those which are based in some branch of mathematics. Structured methods are not mathematically based and often employ some sort diagrammatic notation (e.g. JSD - Jackson System Development Method (Jackson 1975)). Informal techniques those which are completely ad hoc and such techniques are still used in many organizations. There are a number of benefits associated with the use of formal methods over both structured and informal ones, the principal one being unambiguity. A discussion of the benefits of formal methods over informal ones can be found in (Meyer 1985). Formal specification techniques divide into two classes, model-based and algebraic. Model based techniques such as VDM (Jones 1986) and Z (Spivey 1992) are based around the notion of a system state and a set of operations on that state. Many model based techniques involve the use of special mathematical notation. The existence of tools for these techniques is present and continues to increase, e.g. VDM is supported by the IFAD VDM-SL Toolbox (Elmstrom, Larsen, and Lassen 1994) and Z is supported by CADiZ (Ltd 1994), Formalizer (Logica). For more details refer the WWW Virtual Library section on Formal methods (Bowen)for more details on tool support for VDM and Z.

Algebraic specifications (e.g. Larch (Guttag, Horning, and Wing 1985)) are based around sets of operations and the relationships between operations. Their presentation is similar to module interfaces in programming languages. Generally, an algebraic specification will consist of a set of sorts (which are similar to types in programming languages), a set of operations and a set of equations which are constraints on the behaviour of the operations. Tool support for algebraic techniques is relatively good in comparison to model-based techniques. Many of the techniques have their own development environments as well as prototyping and test generation facilities.

The use of formal methods is not a common as it might be. This is largely due to the stigma attached to them by people who write programs (as distinct from software engineers) working in the software industry, particularly those who came into the industry before the advent of formal approaches. As programming habits are hard to change, software is written in much the same way as it always has been. The common arguments presented against the use of formal methods are

dismissed in (Hall 1990) and (Bowen and Hinchey 1994). What is perhaps more worrying is that few programming courses show the need for formalism. Kung *et al* (Kung-Kiu, Bush, and Jinks 1994) note that students learning to program are often taught what amounts to "the hackers way". It is hoped that a shift in education to incorporate formal methods will lead (eventually) to a wider uptake in industry.

2.4 Review of specification based testing

All authors writing on the subject agree that there is a need for use of specifications in testing. This is reinforced by Hall in his paper citing the relationship between specifications and testing (Hall and Hierons 1991). However, the techniques proposed vary widely.

For the purposes of generating tests from specifications, algebraic techniques predominate. This is because the information required to be able to produce a test case is already present in the equation. In addition to this, algebraic techniques result in specifications which often closely model the interfaces to program modules. A test method assuming this correlation simply has to check that the conditions given by the equations hold for the functions provided in the implementation. Indeed, this method predominates. (Ostrand 1986; Arkko, Hirvisalo, Kuusela, and Nuutila 1990; Bouge, Choquet, Fribourg, and Gaudel 1986; Bernot, Gaudel, and Marre 1991) all introduce a notion of a mapping between the specification and implementation.

Methods which partition the input space are quite common, it forms a part of several methods (Hierons 1997a; Ostrand and Balcer 1988a; Ammann and Offutt 1994; Buttle 1995; Dick and Faivre 1993; Stocks and Carrington 1996). Once domains are identified, the test procedure is relatively well documented in contrast to other black box testing techniques. In his 1990 paper, (North 1990) considers three methods of specification (VDM, Miranda and Prolog) and their suitability for automatically generating tests. The choice of Miranda and Prolog is interesting, since a specification in these languages corresponds to the program itself, even though the code gives the "what" but not the "how". It is difficult to judge what constitutes a specification language, since any program can be seen as a specification. However, it is not usual to consider a programming language as a specification language.

The results of North's investigation indicate that both VDM and Miranda form a good basis for test generation. Prolog is seen as poor choice since a type system has to be added, which is a major task in itself. The trade off between the abstractness of VDM and the executability of Miranda will ultimately determine the method chosen. This question is a fundamental one for formal specifications. The line taken in this report is that specifications are primarily aids to assist

the understanding and elicitation of a problem, therefore, abstractness is more important than executability.

2.4.1 Ostrand 85

Ostrand (Ostrand 1986) briefly considers two methods of test case generation from algebraic specifications. The first finds suitable instantiations of the variables in the specification, calculates the result, then tests that the implementation produces the same result for the same set of inputs. This type of method has been adopted by a number of authors. His testing method also considers the possibility of boundary value analysis and a form of "error guessing" (originally proposed by (Myers 1979)) where "different" values (e.g. zero and maxint) are used as instantiations.

The second method provides an extension to the first by concentrating on the operations of the specification as well as the equations. Ostrand states that since all objects must be created/observed/destroyed by an operation, a test technique could generate many possible combinations of the operations, with the results being determined by appealing to the equations in the specification.

2.4.2 Bouge *et al* 86

In their 1986 paper, Bouge *et al* use a technique similar to Ostrand's first method for their test case generation scheme. Their work starts with the ideas of the regularity and the uniformity hypotheses.

The former considers the complexity of testing, and hypotheses that if a program works correctly for a given complexity, C, then it will work for any complexity greater than C. An example of complexity would be the number of iterations around a loop e.g. if a test for 3 iterations of a loop is successful, then 4..N iterations will also be successful. This hypothesis would appear to require some modification to be used in practice, since a complexity of zero constitutes a different case - where the loop isn't actually executed since the loop guard isn't satisfied. If a loop executes correctly for zero iterations, then it is not correct to infer from this that it will work for 1..N iterations. Altering the hypothesis to include this case would make it acceptable. Bouge notes that as C tends to infinity, testing tends to proof.

The uniformity hypothesis is based around similar arguments. The authors state that if a program behaves correctly for a randomly chosen value in a domain (the domain defined using some method of domain analysis) then it will work for every value in that domain, i.e.:

$$\exists\, x \bullet spec(x) = prog(x) \Rightarrow \forall\, x \bullet spec(x) = prog(x)$$

For test generation, the authors assume that there is a correlation between the operations defined in

the specification and the operations provided by the implementation. They state that this assumption is valid due to the use of specifications for software design, and hypothesize that as such usage is increased this correlation will be more and more apparent. Many examples of development from algebraic specifications closely obey this correlation principle.

For equations in the specification which do not contain conditional operators, testing proceeds by instantiating a specification equation with some value (for example the status of a stack) and calculating the result (either manually or by executing the specification). The uniformity hypothesis is applied to lower sorts (data types) for the specification and the regularity hypothesis is applied to the current sorts in the specification under test. For example, if tests were to be performed on a stack of integers, then the test generation assumes the uniformity hypothesis for the integer sort (i.e. that any integer from a given domain will result in the same behaviour for the object under test). The regularity hypothesis would then be applied to the sort stack (i.e. the current sort of interest).

The result obtained gives the acceptance criteria for the test case. The implementation receives the value(s) in the input criteria and the computed value is compared with the acceptance criteria. If both specification and implementation yield the same value then the implementation passes the test.

The situation changes for conditional equations (though the notion of appealing to the hypotheses remains the same), since values which satisfy the left hand side of the expression need to be searched for. The authors translate the conditional equations into positive Horn-clauses and then into Prolog rules. This enables an automated search of the possible instantiation space to be made.

The authors note that the use of Prolog is problematic due to its resolution algorithm, since if the first branch attempted in the search is infinite, the algorithm will never backtrack to find alternatives. This problem is solved by changing the resolution algorithm so that the search tree is pruned at a specified level.

For this type of specification, the solution proposed contains a fair level of automation. There is no consideration made for test sequencing, test data generation or how the tests developed can be performed on the implementation.

2.4.3 Bernout *et al* 91

The system proposed by Bernout *et al* takes the formal notions given in Bouge *et al* and expands them to provide a full description of the testing process.

The Bernout technique starts with the notion of the exhaustive test set, i.e. one which will test

every possibility in the implementation. Obviously, this will be very large for all but the most trivial programs. They make a series of refinements to the test set in order to bring the size down to a more manageable level without compromising the effectiveness of the tests.

Their notion of a test case differs from the one defined above. A triple (H,T,O) is used where H is the set of hypotheses assumed for the test, T is the test set and O is an oracle which judges on the outcome of the tests.

2.4.4 Arkko *et al* 90

Arkko *et al* propose a test system which tests both specification and implementation. For specification testing, the data types used in the specification are themselves specified, and these specifications are used to generate inputs to the specification under test.

The authors use an attribute grammar to represent sets of initial states and values. The grammar rules specify the type of the operation, the level of the test (the number of operations to be performed, for example, the maximum number of iterations of loops), the operations to use and the argument(s) to the operation. These values can be supplied by the user, or left to be randomly instantiated by a test data generator. This method is very similar to that proposed by Bouge *et al*, and differs only in the representation. Indeed, each axiom in the specification is instantiated such that the value that satisfies it is found by searching a possible instantiation space using Prolog.

Expected outcomes for test cases are generated by executing the specification equations with the inputs generated from the data types in the specification resulting in a test case containing an instantiated test equation and the expected result. The authors state that an error generated at this stage corresponds to an error or an omission in the specification.

The authors assume that there is a known mapping between the specification and the implementation and, given this, a transformation can be applied to the test cases expressed in the specification language to generate the required implementation test cases. This method is slightly more adaptable than that proposed by (Bouge, Choquet, Fribourg, and Gaudel 1986), since the transformation stage sets up a mapping from the functions provided in the specification to the ones in the implementation. As noted when discussing the Bouge technique, this type of assumption is valid for algebraically specified programs. A scheme could be imagined where a single function in the specification is mapped onto a number of functions in the implementation that will result in the same functionality being provided.

2.4.5 Richardson *et al* 89

Richardson *et al* considers two types of testing, *error-based* and *fault-based*. The authors define an error-based test as that which will reveal an incorrect value produced in execution and a fault-based test is defined as that which will detect a bug in the source code. The authors further define two categories in which a specification can be used, one where the technique is applied to the specification itself, and one where the specification is used as an oracle to determine the correctness of results produced by the implementation.

From these categories, four methods of testing are defined:

1. Specification / Error-based
2. Specification / Fault-based
3. Oracle / Error-based
4. Oracle / Fault-based

The instantiations that the test case input criteria can take range from distinct values to conditions which describe a range of values. By defining a test case in this way the authors allow for some refinement of test cases from expressions to values.

- Specification / Error-based testing involves symbolically evaluating the specification to generate test domains. Given these domains, boundary values are calculated and these are used to test the pre- and post-conditions for validity. In addition to this, the authors state that "special" values may also be chosen, though the exact nature of such values is omitted. It is assumed that these values would be similar to the ones (Myers 1979) heuristically generates through "error guessing".

- Specification / Fault-based testing is based on the hypothesis that faults exist. Test cases are selected to determine potential faults in the specification. The authors adopt the RELAY model (Richardson and Thompson 1986), and from it use the notion of the revealing test case which sets up an incorrect internal state and propagates the effect, via execution of the specification under test, until an error manifests itself. The test case should be sufficient to distinguish between variations (mutants) of the specification. Both these specification based methods require that the specification is executable, which lends them to testing based on algebraic specifications.

Testing the specification while using the specification as the source of test cases contradicts the rationale given in section one for using the specification to test the implementation. Applying the reasoning for the use of specifications in implementation testing would infer that requirements should be used for specification testing.

- Oracle / Error-based testing uses path analysis of the implementation to select test domains. Special values of paths are also selected by what is assumed to be application of Myers' heuristics. The test cases developed provide a domain test, based on the boundary conditions, and a "computation" test, based on the special values. In both cases, the input criteria is the conjunction of the boundary/special value (as appropriate), the pre-condition and the negation of the post-condition. The acceptance criteria is given as false, meaning that if the input criteria exist, then an error has occurred since correct operation would result in the post-condition being validated.

  Incompleteness testing is also used as an oracle/error-based test. The implementation is tested for parts of the specification which have been omitted. The test case input criterion in this case is the negation of a pre-condition. The acceptance criterion are decided at the discretion of the tester as there is no specification to base the oracle on. Such testing is useful since it can be based on the specification, but test parts of the implementation that are not specified, resulting in more extensive testing. Additionally, this type of testing could be used as an aid to identifying the parts of the specification which have been omitted, (i.e. as a tool to test the completeness of specifications).

- Oracle / Fault-based testing uses the revealing condition technique covered above. A condition is selected and an attempt is made to force either the original code or a mutation of it to violate the post-condition. If this occurs a fault has been detected. This method is mutation analysis using a single mutation with test conditions drawn from the specification. The difficulty here, as with specification/fault-based testing, is the selection of the revealing condition. Given structural knowledge of the mutant would simplify this problem. The method would probably be more effective if it adopted a more traditional form of mutation analysis by using many mutants, although this would be at the cost of greatly increasing the number of test cases.

2.4.6 Hayes 86

In a seminal paper, Hayes (Hayes 1986) uses Z specifications as a source for generating tests. Instead of generating test cases from the specification, he generates code to include in the implementation

to test that properties of the specification hold. The additional code can be disabled for delivery but, if subsequent tests are to be carried out, it can be re-inserted.

Since we could generate these tests while we are refining the specification to code, the method neatly ties the act of specification with the act of testing. Designing tests hand in hand can often result in better tests being developed as the time between specification and testing is cut to a minimum. A variation on this theme has been proposed elsewhere, where, equations from algebraic specifications are used to devise further specifications that define test operations. Due to this, specifications exist for the testing phase as well as for the code.

For any operation, three tests are performed. Initially, a test to check if an invariant holds is made. The invariant is drawn from the specification, and the test is devised using both this information and information from the implementation. The invariant test is performed before the first operation (implementing the specification) then after every subsequent operation. The second test checks that a derived pre-condition holds on entry to the operation.

Hayes' final test checks what he refers to as an "input-output" relation. This final test is needed since an implementation passing the first two tests may still compute an incorrect result. Hayes states that this may be tested by comparing two different implementations of the same specification. The validity of such a test is questionable since it relies as much on the notions of redundancy as anything else. He assumes that by performing a check on two implementations we will be able to identify errors. If we can be sure that the implementation being tested against is correct, then there are practicalities that need to be addressed. For any implementation, some additional code would have to be provided that implements the same specification (and thus does the same as the implementation) to perform the check. The goal is then to show that two implementations are equivalent for a given set of test inputs. Performing such a test, therefore, is analogous to trying to solve the equivalence problem.

2.4.7 Hall 88

Hall also proposes a method for test generation from Z specifications. The method identifies test domains which, once defined, can be used with domain/boundary value analysis methods. The variable names given in the declarative part of the Z schema are used to identify the test domains. The domains are developed by selecting sets of values, given by the schema predicates, which each variable can have in the context of all the other variables. Therefore, if there are two declarations and each can exist in exactly two states, then there will be 4 test cases in all.

Data values which satisfy the predicates defining the domains are derived manually to produce

a set of test data to instantiate the test cases. The domains identified by Hall comprise of inputs, outputs and state changes, which does not make the nature of the test case very clear, nor does it fit in well with the notion of the test case described earlier.

Hall states that some method of executing the specification is required in order to automate this process which is not necessarily correct as a Z parser would be sufficient. He discusses the fact that it is not desirable to regenerate the state of the system for every test, and thus hints that the test cases should be sequenced.

2.4.8 Dick *et al* 93

The subject of test sequencing is not often considered but one notable exception of this is given in the paper by (Dick and Faivre 1993). The authors describe a testing method based around specifications written in VDM-SL which generates test cases from a specification, removes those which are unnecessary and finally sequences them. This is explained in detail in the next chapter.

2.4.9 Carrington and Stocks 93

The The Test Template Framework (TTF) (Stocks and Carrington 1993; Stocks and Carrington 1996) is a framework for defining and structuring specification-based testing. The framework consists of a formal, abstract model of tests and test suites and a method for using the model in testing. The framework involves test derivation, oracles and reification. It can incorporate guidance in choice of test strategies.The framework uses the Z notation as a test description language and to define the framework.

The TTF creates a hierarchy of tests with the valid input space (VIS) of the specification as it's root. The VIS is the precondition of the operations schema and is computed by hiding the final state and any outputs. The valid output space of an operation is used to define oracles but is not really prominent in the framework. The central unit for defining test data is a *test template* (TT). A TT is a constrained subset of the VIS and is described formally by a Z schema. They define a set of bindings from input variables to acceptable values. A TT does not define test data, only a set of data. The connection between a TT and a test case is made when the hierarchy is defined.

A structured approach is used to build a hierarchy of test templates. The hierarchy is created by applying test strategies to existing TTs (starting from the VIS) to derive additional TTs. This is just subdividing the input space into partitions. The TTF does define the choice of subdomains, it allows the use of various strategies within a common framework to determine this. This differs from previous approaches since they define a strategy rather than a framework. However the end

result does not differ since the result is essentially the same : a set of input and output partitions (or predicates which specify partitions). The output partitions are called oracles in their approach. The hierarchy includes test inputs and expected outputs (oracles) but is not used directly for test execution.

The main contribution of this approach aside from the idea of choosing from a set of strategies within a framework is the formal specification of a test. The TT is as abstract as the specification. This simplifies test derivation in two ways. First, it is usually easier to derive tests at higher levels of abstraction. Secondly, more information about the final implementation are be incorporated in small amounts, in stages. Most of the work with TTF has involved the Z specification notation but has been extended to include Object-Z (Murray, Carrington, MacColl, and Strooper 1997). For the purpose of this thesis this work is not very relevant since it does not define the strategies needed in detail, leaving that up to individual systems.

2.4.10 Weyukar *et al* 94

Weyukar *et al* considers an approach to testing based on boolean specifications. Test case inputs are based on the idea of meaningful impact. A literal has meaningful impact on a boolean formula if a change in the truth value of that term affects the truth value of the formula as a whole.

Testing proceeds by selecting a set of test points from sets of unique true points (those which cause exactly one term in the formula to evaluate to true) to test the true cases, and selecting points from the near false points (those that cause a formula to evaluate to false when exactly one of the literals in a term has been complemented) to test the false points. The selected near false points are disjoint from the set of false points (those causing the formula to evaluate to false). The test set generated from this may be reduced if a particular near false point will test two conditions.

The method relies on being able to express the specification using boolean algebra. This will be possible in some cases (but there will be problems for formulae quantified over infinite sets, for example natural numbers), although the resulting formulae many be quite large. However, the nature of the method lends itself well to automation, so in many cases the size of the formulae would not be problematic.

2.4.11 Tsai *et al* 90

Tsai *et al* use specifications written in relational algebra queries (RAQ) and produce a set of test cases (each being a test path and an acceptance criterion). The test cases generated are used for testing the specification and the implementation. As mentioned earlier, using a specification to test

a specification is questionable.

The authors create test cases by transforming the RAQ into a set of predicates. These predicates are then transformed into a set of systems of linear inequalities (SSLI) which can be displayed geometrically, resulting in a shape that encloses the test domain. The vertices of this shape are used to find values near (just inside or just outside) the boundary of the test domain.

The method presented is a novel, but rather complicated, way of creating domains for domain/boundary value analysis. The use of relational algebra is not very common for program specifications which obviously restricts the genericity of this method. However, the scheme that the authors propose would seem to be well suited to testing databases.

2.4.12 Summary

Specification based testing is still in its infancy. Consultation of any textbook on testing will reveal little on this method. It seems that each author who approaches the problem creates new ideas for its solution. The exception to this lies with the algebraic based techniques which are heavily based on the original ideas of (Ostrand 1986). For the model based approach, the wider choice of methods has not resulted in a wider choice of tools. The tools existing are limited in application and availability, with many being prototypes. However, it would seem that, given the wider use of model based methods for specification, methods making use of model based techniques may have a more practical future than those based on algebraic ones.

Generally applicable specification based testing tools seem to be a fair way off. The very generality which they would hope to support is the main factor in the difficulty of their development. A thought might be spared for the telecommunications industry, whose use of formal methods tends to be limited to the formal description techniques proposed by ISO (LOTOS (Interconnection 1989a), Estelle (Interconnection 1989b)) and SDL (Telegraphy and Telephony 1989)). In this field, the methods are used for the description of switching protocols, and this limitation (and the relative ease of the problem) has resulted in a wide variety of tools being available which produce tests from specifications.

Chapter 3

Specification based Checkpoint Encoding

This chapter describes a solution to the problem of automating checkpoint encoding. The eventual goal is the complete automation of anti-random testing. The phase of anti-random testing that is the most manual intensive is checkpoint encoding. It is therefore the focus of automation. The solution is adopted from the area of specification based testing. What all the methods considered have in common is the approach : partitioning the input space from the specification. Since checkpoint encoding is also fundamentally the same, this suggests the obvious approach to automating checkpoint encoding. Checkpoint encoding however does not end with extracting partitions, a encoding scheme still remains to be formulated. The encoding scheme actually distributes tests between usual, boundary and illegal equivalence classes. However that can be easily automated once the partitions are extracted.

This chapter discusses some methods of extracting partitions from formal specifications in an automated manner. The choice of specification technique affects the choice of testing, therefore, the technique to be used will be considered first. This is followed by an appraisal of several possible techniques which could be adopted. Then the proposed method is explained in detail. Finally the proposed method of automating checkpoint encoding is used with certain examples. The resultant checkpoint encoding schemes from both the new automated method and the old manual approach are compared. Based on the comparison the thesis attempts to state what can be achieved with the proposed method.

3.1 Z Specification Language

Generating tests from algebraic specifications appears to be more straight forward than from model based specifications. The underlying algebra of these techniques is suited to formal arguments of

validity and reliability of the generated test set. However, algebraic specifications are only popular in a small number of specialized fields. The use of model based approaches is more diverse. As a testing system should be as general as possible, the use of model based techniques is the most appropriate. In addition to this, tool support for model based techniques is limited. Further tools would be beneficial as they would encourage developers to use formal methods.

The most common model based techniques are VDM and Z. The Z notation is chosen on the grounds that Z tools have been developed before. The availability of full technical support for a major toolset is the principal reason for the choice of Z.

A secondary reason for the choice of Z relates to proof obligation. Standard Z does not include a proof system, but there have been a number of proposals for one. The most common method, introduced by Woodcock in (Woodcock and Brien 1992) is W which is based on sequent systems. CADiZ uses a modified version of W described in (Toyn and Hall 1994). The actual mechanism of proof is not of major concern here, rather the ability to include proof obligations within a specification. Proof obligations provide additional information which can be used for testing. This idea fits in well with the notion of tests proving properties about programs (given a set of assumptions such as those presented by (Bouge, Choquet, Fribourg, and Gaudel 1986)).

3.1.1 An Overview of Z

The Z language (Spivey 1992) is one of the many formal specification languages. There two components that make up the Z langauage. The first is the use of mathematical data types to model the data in a system. These data types are not oriented towards computer representation, but they obey a rich collection of mathematical laws which make it possible to reason effectively about the way a specified system will behave. The notation of predicate logic and set theory is used to describe abstractly the effect of each operation of a system, again in a way that enables us to reason about its behaviour.

The other main ingredient in Z is a way of decomposing a specification into small pieces called schemas. By splitting the specification into schemas, we can present it piece by piece. Each piece can be linked with a commentary which explains informally the significance of the formal mathematics. In Z, schemas are used to describe both static and dynamic aspects of a system. The static aspects include:

- the states it can occupy;
- the invariant relationships that are maintained as the system moves from state to state.

The dynamic aspects include:

- the operations that are possible;
- the relationship between their inputs and outputs;
- the changes of state that happen.

The schema language allows different facets of a system to be described separately, then related and combined. For example, the operation of a system when it receives valid input may be described first, then the description may be extended to show how errors in the input are handled. Or the evolution of a single process in a complete system may be described in isolation, then related to the evolution of the system as a whole. The schemas can be used to describe a transformation from one view of a system to another, and so explain why an abstract specification is correctly implemented by another containing more details of a concrete design. By constructing a sequence of specifications, each containing more details than the last, it is possible to eventually arrive at a program with confidence that it satisfies the specification.

The Z syntax and semantics cannot be completely explained within this document. Several references exist (Spivey 1992; Wordsworth 1992) that explain the Z language. This thesis assumes a working knowledge of the Z language.

3.1.2 Example Z specification

Consider a Z specification for a stack of integers. The specification begins with the definition of the stack. This is referred to as defining the state of the system. Base types, global constants and variables are first defined. Then one or more schemas are used to define the stack. This is the state schema. The operational schemas are defined next. In the current specification there are only two operations : Push and Pop. This specification is kept simple purposely, leaving out error handling schemas etc.

$$RESPONSE ::= OK \mid EMPTY \mid FULL$$

$$Max : \mathbb{N}_1$$

Stack

$$data : \operatorname{seq} \mathbb{N}$$
$$size : \mathbb{N}$$

$$size = \# data$$
$$size \leq Max$$

$StackOp \mathrel{\widehat=} Push \lor Pop$

Push

$\Delta Stack$
$item? : \mathbb{N}$
$r! : RESPONSE$

$((size < Max) \land (data' = item? \frown data) \land (size' = size + 1) \land (r! = OK)) \lor$
$((size = Max) \land (r! = FULL) \land (data' = data) \land (size' = size))$

Pop

$\Delta Stack$
$item! : \mathbb{N}$
$r! : RESPONSE$

$((size > 0) \land (item! = head\ data) \land (data' = tail\ data) \land (size' = size - 1) \land (r! = OK))$
$\bigvee((size = 0) \land (r! = EMPTY) \land (data' = data) \land (size' = size))$

The predicate part of the state schema Stack specifies an invariant on the state of the stack which is that the number of items in the stack (represented by the variable size) is equal to the length of the sequence representing the stack. The operational schemas Push and Pop define the two common stack operations of the same name. For the sake of simplicity, schemas defining the action if the pre-conditions of Push or Pop are not satisfied are not shown.

The following proof obligations may be inserted by the specifier:

$Theorem1\ [size : \mathbb{N} \mid size < Max \land Push] \vdash \#data' = \#data + 1$
$Theorem2\ [size : \mathbb{N} \mid size > 0 \land Pop] \vdash \#data' = \#data - 1$

These proof obligations are derived from the relationships between *size* and *size'* in Push and Pop and *size* and $\#data$ in the included schema *Stack*. They can be used in a test by evaluating $\#data$ before the relevant operation and $\#data$ after (i.e. $\#data'$) then checking that the consequent of the proof obligation holds. For example, consider the following test sequence:

$BEFORE := EVALUATE(\#data)$
$PUSH(item?)$
$AFTER := EVALUATE(\#data) -- i.e. \#data'$
if $(BEFORE < Max)$ *and* $(AFTER = BEFORE + 1)$ *then*
SUCCESS – the consequent of the proof obligation holds
else if $(BEFORE = Max)$ *and* $(AFTER <> BEFORE + 1)$ *then*
SUCCESS
else FAIL
endif

The proof obligation for Push states that the consequent of Push, given that the pre-condition holds, is that $\#data' = \#data + 1$, and this is checked by the code implementing the test. If the pre-condition does not hold, the consequent should not hold true. The evaluation can be made symbolically or by instantiation with specific values. The latter option was shown in the example above.

3.2 Test case generation from Z

3.2.1 Category-Partition Method

The paper by Ostrand et al (Ostrand and Balcer 1988b) appeared at a time when there were no firm guidelines on how to actually generate partitions. What it did was to go some way to filling that vacuum. It uses the functional specification of the software under test as the test information given. It converts the functional specification (which in this paper refers to a natural language description of the software) into some intermediate representation from which test cases and test scripts can be generated.

The steps of the method are

1. Analyze and decompose the functional specification into functional units that can be tested separately/independently.

2. Identify *parameters* and *environment conditions* that affect the function's execution behaviour. Parameters are the explicit inputs to a functional unit, supplied by either the user or another program. Environment conditions are characteristics of the system's state at the time of executing a functional unit.

3. Find *categories* of information for each parameter and environment condition.This is done by analyzing how the functional unit behaves with respect to the characteristic of the parameter or environment condition. A category is a major property or characteristic of a parameter or environment condition.

4. Partition each category into distinct *choices* that include all possible values for a category. The choices are the equivalence classes for the input domain.

5. Write categories and choices in a formal *test specification* for each functional unit. The test specification is written using **TSL (Test Specification Language)** that the authors created for this purpose.

This is then used to construct *test frames* . A test frame consists of a combination of choices from the specification with each category contributing 0 or 1 choice. This is instantiated into an actual test case by specifying a single element from each choice in the frame.

This paper focuses more on the latter part of actual test case generation and test scripts to run the test cases. The authors develop TSL and in later papers (Balcer, Hasling, and Ostrand 1989) describe further work with creating the generation tool. However for this thesis the first 3 steps are important. They are the first guidelines for generating partitions from specifications even if the process described involves manual intervention and is carried out using a natural language specifications.

3.2.2 Using Z specifications for Category Partition Testing

The work done by Amla & Ammann (Amla and Ammann 1992) and Ammann & Offutt (Ammann and Offutt 1994) build on the Category-Partition method. They attempt to mechanize as much of the process as possible. To address this they use formal specifications, which allows the automation of the first three steps. The formal specification language used is Z. They also address some open issues in this method, namely a minimal coverage criterion and a procedure to derive test cases from test scripts.

1. Identify functional units.

   This is simple due to the use of schemas. In Z, a schema that either changes or observes the state defines an independently testable unit. Also any schema that produces an output defines a functional unit.

2. Identifying the Parameters and Environmental Variables

   Parameters correspond to input variables of a functional unit/schema. Environment variables correspond to Z state variables. Usually only those used in preconditions of a schema are selected. Both are easily identifiable.

3. Identifying the Categories.

   In Z specifications there are two distinct sources of categories

   - Characteristics enumerated in the preconditions. This based on the predicates described using those variables under examination in the schema.
   - Characteristics of a parameter or environment variable by itself. This is based on the type of the variables under examination.

4. Partitioning the Categories.

   A category can always be broken into two partitions : **valid** and **invalid**. Finer partitioning is also possible. Here some heuristics can be used like selecting boundary and/or extreme values. This is difficult to implement differently for every software. We can use heuristics based on the type of variable or predicates involved.

5. Specify Possible Results

   In this step we have to extract what the expected output of the operation or test case should be. This has to entered into the TSL specification of the tests.This last step to completing the TSL specification is largely done manually; some assistance is obtained from the Z specification. This is an independent problem and can be considered a separate task by itself. One way of describing it is as a **test oracle** which is used to determine if the test has discovered any errors. This is not relevant to this thesis.

In (Ammann and Offutt 1994) a method to create test frames is proposed. A combination matrix is created with choices for each category. A *base* choice is defined for each category, this represents the default choice or normal choice based on an expected user profile. The test frames are created by selecting choice combinations based on a coverage criterion.

3.2.3 Generating Tests from a Z specification : Dick and Faivre

Dick and Faivre (Dick and Faivre 1993) describe several aspects of automation of the testing process from specifications. The four main aspects are

- *The partition analysis of individual operations.* This involves reducing the expressions into a Disjunctive Normal Form (DNF), which gives disjoint partitions representing domains of the operation.

- *The partition analysis of the system state.* The expressions of the system state are reduced into DNF which yields partitions of state values which can be used to construct a Finite State Automaton (FSA) model.

- *The scheduling of tests of different operations* The aim is to avoid redundancy in the testing process. Paths through the FSA covering all the required tests are found. Constraints for all the subpaths are composed to detect inconsistencies.

- *The generation of test values.* This involves selection of concrete values for each test case. These are substituted into the composed constraints.

- *The verification of the results* Verify the results using the reduced constraints against that of the specification.

In the specification an operation describes a partial relation between systems states as a logical expression involving the pre and post condition of the operation and the state invariant. This is at the core of the test generation process. The pre-conditions of the operation effectively provide a partition of the input space, insofar as it applies to the operation. The state invariant also provides information crucial to deriving the partitions. The entire relation is then split into a set of smaller relations by transforming it into a DNF. This allows each sub-relation (which may be a specific partition) to be treated independently. This is similar to path-coverage analysis of a program.

The steps of the process are, stated simply :

1. Extract the definition of the operation. In Z this is not necessary since the schema more or less provides this. Essentially all relevant pre/post-conditions, invariants etc. are collected.

2. Unfold all definitions and introduce basic types. Unfolding of recursive function and type definitions have to be limited. This is a limitation akin to loop unfolding. Usually it is unfolded a fixed number of times.

3. Transform relation into DNF to obtain disjoint sub-relations (domains). There are standard rules for doing this with propositional logic.

4. Simplify each relation by using inference rules based on first order predicate calculus.

The approach is based on using model based specifications and the language chosen is VDM-SL. This could be applied to Z specifications with slight differences.

3.2.4 Generating Tests from a Z specification : Hierons

Robert Hierons (Hierons 1997b) examines the automation of the testing process from Z specifications. The paper gives an algorithm to rewrite the specification to a form from which both a partition of the input domain as well as the states of a FSA model can be derived. The FSA is used to control the testing of the program. This idea is similar to that proposed by Dick and Faivre and is explained in Chapter 4.

This paper does advance any radically new concepts in the extraction of partitions from a Z specifications. It builds on work by Amla and Ammann (Amla and Ammann 1992), Stocks and Carrington (Stocks and Carrington 1993) and particularly on Dick and Faivre (Dick and Faivre 1993). What is different from Dick and Faivre, is it proposes a step by step method that rewrites

a Z specification into a form that has input and output subdomains. This then becomes one of the starting points for this thesis because of the formal approach taken as compared to largely informal ones prior to it.

In order to produce a partition, the specification is rewritten to the form

$$\vee_i \; P_i \wedge Q_i$$

, where each P_i represents a precondition and each Q_i represents a postcondition. Effectively, the specification is expressed as a set of behaviours with preconditions and the equivalent of the category-partition's test frames will be generated from the preconditions. The process of generating a partition can be broken down into the following steps.

1. Classify the variables.

   Each variable in the Z schema is classified as input, output or state variable. This step is important to try to identify input and output subdomains. Significantly output variables are treated equally with input variables.

2. Rewrite the specification to predicate logic.

   Using standard schema expansion, the specification is flattened and expressed as statements in first order predicate logic. This step includes breaking up of set expressions. If the specification is defined recursively, the process of flattening may not terminate. As suggested by Dick and Faivre a limit should be placed on the number of unfoldings allowed.

3. Categorize the predicates.

   The individual predicates are now categorized into either input or output predicates. This is based on the type of variables in the predicate. If there are more input variables than output variables then the predivate is an input predicate. It is not so easy to make this distinction, sometimes heuristics have to be used. The paper does not explain it in great detail. Effectively input predicates give the preconditions and output predicates give the behaviour of the software.

4. Rewrite to the required form.

   The basic rewrite system used is the one described for a specification with two classes of predicate c1 and c2, with the final form desired

   $$\vee_{1 \leq i \leq n} \; (X_i \wedge Y_i)$$

   where X_i and Y_i are predicates of class c1 and c2 respectively.

For a predicate p, class(p) is c1 if all atomic predicates are input predicates, c2 if all atomic predicates are output predicates, c3 if they are mixed. The goal is to separate input and output predicates. Certain forms are defined using Backus-Naur form.

$$X' ::= X \mid X' \vee X' \mid X' \wedge X'$$

$$Y' ::= Y \mid Y' \wedge Y' \mid Y' \vee Y'$$

$$S ::= X \mid Y \mid S \wedge S \mid S \vee S$$

$$N ::= X' \wedge Y'$$

$$N' ::= N \mid N' \vee N'$$

X' and Y' give the predicates of classes c1 and c2, respectively; S gives the set of predicates; N gives the conjunctions that do not need rewriting; and N' gives the set of structures that do not need rewriting (normal forms).

The rewrite rules are given in the form; preconditions above the line and rewrite rule below the line. J represents an index set $\{1, \ldots, m\}$ for some m; $J_k = \{j \in J \mid class(N_j) = ck\}$ where $k = \{1, 2, 3\}$. The 3 rewrite rules are

$$\frac{x : X', y : Y'}{y \wedge x \rightarrow x \wedge y} \tag{3.1}$$

$$\frac{\begin{array}{c}\forall j \in J \bullet N_j : N \\ class(\vee_{j \in J} N_j) = c3\end{array}}{S \wedge (\vee_{j \in J} N_j) \rightarrow (S \wedge (\vee_{j \in J_1} N_j)) \vee (S \wedge (\vee_{j \in J_2} N_j)) \vee (\vee_{j \in J_3} (S \wedge N_j))} \tag{3.2}$$

$$\frac{\begin{array}{c}\forall j \in J \bullet N_j : N \\ class(\vee_{j \in J} N_j) = c3\end{array}}{(\vee_{j \in J} N_j) \wedge S \rightarrow ((\vee_{j \in J_1} N_j) \wedge S) \vee ((\vee_{j \in J_2} N_j) \wedge S) \vee (\vee_{j \in J_3} (N_j \wedge S))} \tag{3.3}$$

The first rule 3.1 reorders within a conjunction while the second, 3.2 and third, 3.3 rules split up disjunctions into disjuncts with predicates of the same class. The second and third rules have the precondition that the N_j must be in normal form. It has been proved by Hierons (Hierons 1997b) that these rules produce the correct form, and are confluent and terminating.

5. Generate disjoint subdomains.

   A disjoint subdomain is a partition of the input domain. It is a set of conditions on the input variables. Given $\{P_i \mid 1 \leq i \leq n\}$, a partition $\{P_X \mid X \subseteq \{1, \ldots, n\} \wedge X \neq \{\}\}$ from which test cases is generated is given by

   $$P_X = (\wedge_{i \in X}\ P_i[Y_i / In_i]) \wedge (\wedge_{i \notin X}\ \neg P_i[Y_i / In_i])$$

. Here P_i represents a pre-condition and In_i denotes intermediate variables that are renamed to Y_i. Thus P_X is a combination of pre-conditions with intermediate variables renamed, leaving just input variables. It is forced to be disjoint by and-ing certain predicates with the negation of all the other predicates.

Hierons tries to formalize the method of extracting partitions. The method of creating partitions is not explained in detail or with examples. His method depends on ideal specification. If the specification is incomplete then his method fails at providing the best set of disjoint subdomains. Considering the relevance from the viewpoint of this thesis, his main contribution is formalizing the rules to rewrite the specification into an appropriate form for further analysis.

3.2.5 Summary

The idea of extracting partitions from Z or any model based specification is clearly advantageous. The pre/post-conditions are clearly specified or can be derived easily. This is why most of the work vis-a-vis automating partition testing has been done with respect to model based specifications rather than any other type of specification technique. Consider automation of the Category-Partition method.

- Identification of parameters and environment variables can be done directly from the Z specification.
- Z specifications are suited to the crucial phase of identification of categories. They can be derived from the type declarations themselves but primarily from the preconditions of the schemas.
- Further the preconditions of the schema can be computed rather than simply observed.
- Ultimately the Z specification can provide the test engineer with a reliable and precise reference point from which choices can be generated, and if necessary derive test specifications.

The work done by Ammann et al (Amla and Ammann 1992; Ammann and Offutt 1994) is not complete. They have taken a subset of Z and used very simple examples to illustrate their technique. Dick and Faivre (Dick and Faivre 1993) suggest that the specifications be flattened into the DNF form. They then apply precondition analysis to the disjuncts. Unfortunately this forces regions that have uniform behaviour to be split if they contain one or more disjunctions. Using DNF can thus lead to more subdomains. This is one of the main disadvantages of their approach i.e. the size can get unwieldy. Horcher and Peleska(Horcher and Peleska 1995) also carry out much the same

technique. Theirs is not as mathematically rigorous as Dick and Faivre's. Stocks and Carrington (Stocks and Carrington 1993)(Stocks and Carrington 1996) introduce the concept of a test template framework, a hierarchical tree of test templates. The test template therefore gives a structure within which test case generation algorithms such as those mentioned above can be applied. It does not provide the strategy itself. However the idea of a test being specified formally is in line with other approaches. Hierons does not strictly follow the category-partition method (neither does Dick and Faivre) but proposes a more formal algorithm. This algorithm holds the most promise in terms of being automated. In the next section, which explains the proposed technique, how it overcomes some shortcomings of Hierons' approach is also pointed out.

3.3 Proposed Approach

A stated goal of the thesis is to automate checkpoint encoding. Checkpoint encoding has two phases : defining the subdomains (or ideally partitions) of the input space, and encoding them to form a binary vector. The first phase, from the automation viewpoint, is the difficult one. The second phase is not difficult per se. There are few rules, formal or informal, for encoding which makes this more difficult than it is.

The proposed method consists largely of the first phase. The second phase is dealt with using simple heuristics. The first phase, of extracting partitions, is based on methods discussed earlier. It builds on the approach of extracting partitions based on information in the Z schemas. This approach is quite solid and serves the required purpose. Where the proposed technique differs is in the latter stages : that of generating subdomains based on information extracted using the standard approach. This thesis proposes an enhancement of that step to ensure completeness and correctness. The subdomains extracted must be complete i.e. they should sub-divide the entire domain and they should be correct i.e. they should have the correct boundaries. Implicit in the correctness objective is the fact that granularity of the partitions must also be right. They should be as large as possible while satisfying the uniformity hypothesis of partition testing (Beizer 1990). These objectives are limited by the specifications itself. If the specifications are not complete or correct i.e. they do not map the entire domain or are written incorrectly then no technique will guarantee that the subdomains generated are complete or correct.

3.3.1 Extracting Partitions from a Z Specification

The first phase involves refinement of the specification to a form in which the pre and post-conditions are clearly visible. This is of the form $\vee_i P_i \wedge Q_i$, where each P_i represents a precondition and each

Q_i represents a postcondition. The process used is based on Hierons explained above. The Stack specification is used throughout to illustrate the process.

1. Classify the variables.

   To be able to classify the predicates we have to classify the variables in them. This is an indirect mechanism but is the only one possible for Z. The syntax does not clearly delineate the pre and post conditions. Fortunately the syntax of Z does help. There are five kinds of variables in Z : global, input, output, state and intermediate variables. Input variables are suffixed with a ? and output variables with a !. Global variables are effectively parameters for the system as a whole. Operations on a system cannot change the value of a global variable. The distinction between intermediate and state variables is semantic. This is not a widely followed convention and is defined clearly for the purpose of this thesis. State variables are those that are defined in state schemas. Intermediate variables are defined mostly in operation schemas and are temporary in nature. They do not have any scope or significance outside the schema in which they are defined. All quantified variables also fall into this category. State variables can be further classified as an input state variable (unprimed) or an output state variable (primed).

   The conventions followed in Z allows to identify these automatically. In the case of the Stack specification we have
   Global variables : Max
   Input variables : $item?$
   Ouput variables : $item!$, $r!$
   Input State variables : $data$, $size$
   Output State variables : $data'$, $size'$
   The above results are slightly confusing. This is because we should actually define input and output variables separately for each schema. Thus $item?$ is an input for $Push$ and $item!$ an output variable for Pop. In the case of state variables the primed state indicates an after state while the unprimed state indicates a before state. Thus the variable fulfills two roles depending on it's state, and is classified as such..

2. Rewrite the specification to predicate logic.

   The specification is now rewritten and simplified. The target to rewrite the schema using only first order logical connectives $\vee, \wedge, \neg$ and atomic predicates. First we break up set expressions that contain several predicates i.e. the predicate of the form $x \in \{y : Y \mid p(y) \bullet f(y)\}$ can be

rewritten as $\exists\, y \bullet p(y) \land (x = f(y))$. Standard operators when being replaced can add added structure to the specification. Each such operator is replaced with a disjunction of behaviours with preconditions. One suggestion is to replace individual operators with standard partitions i.e the absolute value operator is replaced with $x < 0, x \geq 0$. Finally the quantifiers are moved to the beginning of the expression (prenex normal form). This may involve some renaming of quantified variables.

Thus we have

$$((size = \#data) \land (size \leq Max))$$

$$((size < Max) \land (data' = item? \frown data) \land (size' = size + 1) \land (r! = OK))$$
$$\lor ((size = Max) \land (r! = FULL) \land (data' = data) \land (size' = size))$$

$$((size > 0) \land (item! = head\ data) \land (data' = tail\ data) \land (size' = size - 1) \land (r! = OK))$$
$$\lor ((size = 0) \land (r! = EMPTY) \land (data' = data) \land (size' = size))$$

In the Stack specification not much had to be done to rewrite it. No functions or types were defined recursively so the problem of limiting the flattening process does not arise. However the state specification schema *Stack* has to be included in the operation schemas. The $\Delta Stack$ implies this but this must be done explicitly since we are flattening the specification. The Δ operator specifies that both pre and post-condition must satisfy the state schema. Also the operation of the specification is the disjunction of both *Push* and *Pop* schemas. Thus we have

$$((size = \#data) \land (size \leq Max)) \land ((size < Max) \land (data' = item? \frown data) \land (size' = size + 1)$$
$$\land (r! = OK)) \lor ((size = Max) \land (r! = FULL)) \land ((size' = \#data') \land (size' \leq Max))$$
$$\bigvee((size = \#data) \land (size \leq Max)) \land ((size > 0) \land (item! = head\ data) \land (data' = tail\ data)$$
$$\land (size' = size - 1) \land (r! = OK)) \lor ((size = 0) \land (r! = EMPTY)) \land ((size' = \#data')$$
$$\land (size' \leq Max))$$

3. Categorize the predicates.

   The individual predicates are now categorized into either input or output predicates. Effectively input predicates give the preconditions and output predicates give the behaviour. Any predicates that refer to input state variables, input variables and global variables are input

predicates. Predicates that refer to the final state or the output are output predicates. Intermediate variables are classified by the predicates it is involved with. If the variables inn these are consistent with the predicates being input predicates then it is assumed that this intermediate variable is involved in the preconditions and vice versa.

Input predicates : $(size = \#data), (size \leq Max), (size < Max), (size = Max), (size > 0), (size = 0)$

Output predicates : $(size' = \#data'), (size' \leq Max), (data' = item? \frown data), (size' = size + 1), (r! = OK), (r! = FULL), (item! = head\ data), (data' = tail\ data), (size' = size - 1), (r! = EMPTY)$

The input predicates were simple to identify. In the case of output predicates some could have been identified otherwise but since the left hand side of every predicate is an output or output state variable there should be no confusion.

4. Rewrite to the required form.

   The basic rewrite system used is the one described for a specification with two classes of predicate c1 and c2, with the final form desired

   $$\vee_{1 \leq i \leq n} (X_i \wedge Y_i)$$

   The set X_i and Y_i contains only input and output predicates respectively.

   First we combine predicates which are of the same class (i.e. c1 or c2). x defines a set of predicates that are of type input (c1) and y of type output (c2).

   $$x_1 : ((size = \#data) \wedge (size \leq Max))$$

   $$x_2 : (size < Max)$$

   $$x_3 : (size = Max)$$

   $$x_4 : (size > 0)$$

   $$x_5 : (size = 0)$$

   $$y_1 : ((size' = \#data') \wedge (size' \leq Max))$$

   $$y_2 : ((data' = item? \frown data) \wedge (size' = size + 1) \wedge (r! = OK))$$

   $$y_3 : (r! = FULL)$$

   $$y_4 : ((item! = head\ data) \wedge (data' = tail\ data) \wedge (size' = size - 1) \wedge (r! = OK))$$

   $$y_5 : (r! = EMPTY)$$

The expression () can be now written as

$$(x_1 \wedge ((x_2 \wedge y_2) \vee (x_3 \wedge y_3)) \wedge y_1) \bigvee (x_1 \wedge ((x_4 \wedge y_4) \vee (x_5 \wedge y_5)) \wedge y_1)$$

Applying rule 3.2 with x_1 we get

$$(((x_1 \wedge x_2 \wedge y_2) \vee (x_1 \wedge x_3 \wedge y_3)) \wedge y_1) \bigvee (((x_1 \wedge x_4 \wedge y_4) \vee (x_1 \wedge x_5 \wedge y_5)) \wedge y_1)$$

Applying rule 3.3 with y_1 we get

$$(((x_1 \wedge x_2 \wedge y_2 \wedge y_1) \vee (x_1 \wedge x_3 \wedge y_3 \wedge y_1))) \bigvee (((x_1 \wedge x_4 \wedge y_4 \wedge y_1) \vee (x_1 \wedge x_5 \wedge y_5 \wedge y_1)))$$

Reordering and keeping together as conjunctions predicates of the same class the specification is now in the form $\vee_{1 \leq i \leq n} (X_i \wedge Y_i)$

X_1 : $(size = \#data) \wedge (size \leq Max) \wedge (size < Max)$

Y_1 : $(data' = item? \frown data) \wedge (size' = size + 1) \wedge (r! = OK) \wedge (size' = \#data') \wedge (size' \leq Max)$

X_2 : $(size = \#data) \wedge (size \leq Max) \wedge (size = Max)$

Y_2 : $(r! = FULL) \wedge (size' = \#data') \wedge (size' \leq Max)$

X_3 : $(size = \#data) \wedge (size \leq Max) \wedge (size > 0)$

Y_3 : $(item! = head\ data) \wedge (data' = tail\ data) \wedge (size' = size - 1) \wedge (r! = OK)$

$\wedge (size' = \#data') \wedge (size' \leq Max)$

X_4 : $(size = \#data) \wedge (size \leq Max) \wedge (size = 0)$

Y_4 : $(r! = EMPTY) \wedge (size' = \#data') \wedge (size' \leq Max)$

5. Generate disjoint subdomains.

Before disjoint subdomains are generated, all possible subdomains are generated without considering whether they are disjoint or not. Not all input predicates (i.e. of class c1) are useful. Some of them involve (input) state variables. From these predicates only those that involve inequalities i.e. conditions are kept. All state variable predicates involving assignment are removed. This could be a problem if the specification is not written correctly, even otherwise the semantics of Z complicates matters. The predicate $(size = \#data)$ is an equality condition and not an assignment because the primed (after) state is not indicated.

After removing any unnecessary predicates the next step is the extraction of subdomains. Often a predicate has a coarse partition, one with not the best level of granularity. Thus each predicate could encompass several subdomains. One method of generating standard subdomains from the predicates is proposed by Stocks and Carrington (Stocks and Carrington 1996). They propose using the operators specified in the predicate. These are *function* subdomains which correspond to logical conditions in the software. For example the predicate $(size \leq Max)$ can be split up into $(size < Max), (size = Max)$.

To ensure that disjoint subdomains are generated, the same method of combining predicates with negations of other predicates used by Hierons is employed. This is used with predicates that contain the same variables. Simplifying, the resulting disjoint subdomains are :

$$(size = Max)$$
$$(size < Max) \wedge (size > 0)$$
$$(size = 0)$$

These subdomains have been identified explicitly from predicates in the specification. This thesis proposes another source of subdomains, termed *type* subdomains. These are subdomains that are extracted depending on the type of state or input variables involved in the predicate. In the above example the input variable $item?$ does not appear in any predicate. The type $\mathbb{N}$ provides legal and illegal subdomains.

$$(item? > 0) \wedge (item? < \mathbb{N}_{max}) \quad \mathbb{N}_{max} \textit{ is the largest legal value}$$
$$(item? = 0)$$
$$(-\mathbb{N}_{max} < item?) \wedge (item? < 0)$$
$$(item? \notin \mathbb{N})$$

It would seem that we should examine only the pre-conditions. However this thesis hypothesizes that the post-conditions often contain subdomain predicates as well. The post-conditions may test for boundary conditions that will be necessary to test. Hierons has not touched upon this subject, and this may be a limitation of his work. The interesting predicates in the post-condition are those involving state variables. Post-conditions that involve state variables indicate what the state of the system variables should be assuming that the operation was carried out. Thus test information regarding the state of system before the operation can be inferred. In the post-conditions all predicates that have inequalities involving state variables are examined and the variables are

unprimed. Since they are post-conditions, the pre-conditions that need to be tested are negations of these predicates. In this example there are no significant post-conditions that form additional disjoint partitions.

3.3.2 Defining the Encoding Scheme

Once the subdomains have been identified an encoding is now assigned. This process is the same as with normal checkpoint encoding. There is no fixed algorithm to follow since this process is not yet perfected manually. Codes are automatically assigned by applying heuristics that are known to work well from prior experience.

This thesis proposes an obvious heuristic. Assign one code to each case. More codes (say 2 or 3) could be assigned in case of cases that encompass a range of values rather than a single value as in the case on a equality condition. The number of codes assigned to disjoint subdomains of different parts of the domain i.e. involving different variables should sum to a power of 2. The number of codes is adjusted so that they satisfy this constraint. Thus for the *size* subdomain we have 3 cases and 4 codes. 2 bits (4 codes) could encode the *size* subdomain. Similarly for the *item*? subdomain we have 4 cases and 7 codes. This can be encoded using 3 bits (8 codes). An example encoding scheme is

Field	Bits	Value	Predicate
size	b1,b0	00	$(size = 0)$
		11	$(size = Max)$
		rest	$(size < Max) \wedge (size > 0)$
item?	b4,b3,b2	000	$(item? = 0)$
		rest	$(item? > 0) \wedge (item? < \mathbb{N}_{max})$
		001, 101	$(-\mathbb{N}_{max} < item?) \wedge (item? < 0)$
		111, 110	$(item? \notin \mathbb{N})$

This may not be the most perfect encoding scheme. There is a need to be able to classify legal and illegal cases. Also identification of interdependencies between subdomains and how that affects the encoding scheme is important. The subdomains generated have classified orthogonally. One major phase that has not been discussed is creation of a test script. Once the encoding is complete, test data needs to be instantiated given a binary vector. This is an area that this thesis does not go into. Some of these questions are dealt with later in this thesis.

3.4 Examples

To try and demonstrate the partitioning technique the proposed technique is compared to the methods used before. Examples from (Malaiya 1996) are used as a benchmark. These examples were used to test the efficacy of antirandom testing with checkpoint encoding. After comparing the checkpoint

encoding that the above process derives with the ones in (Malaiya 1996) this thesis hopes to point out some interesting observations. To simplify the comparison only the subdomains or partitions generated are compared. The encoding step is more or less the same.

- **The STRMAT program** . The STRMAT program is given as input a string of 0 to 80 characters long and a pattern of 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.

  A Z specification of the above problem is given below

  $[CHAR]$

  $STRING : \text{seq}\, CHAR$

  STRMAT

  $string?, pattern? : STRING$
  $pos! : \mathbb{Z}$

  $\#string? \leq 80$
  $\#pattern? \leq 3$
  $\exists j \in 1 \,..\, \#string?, \forall i \in 1 \,..\, \#pattern? \bullet (((string?(j+i)) = (pattern?i)) \land (pos! = j+1)) \lor (pos!$

  The partitions used in the checkpoint encoding from (Yin, Lebne-Dengel, and Malaiya 1997) for STRMAT are listed in table 3.1.

Field	Value
Text Length	0 80 $80 \leq$ len ≤ 100 $1 \cdots 79$
Pattern Positions	no pattern beginning end middle
Pattern Length	0 3 $3 \leq$ len ≤ 10 $1 \cdots 2$

Table 3.1: Standard checkpoint encoding for STRMAT

Using the proposed approach the specification is rewritten to

$((\#string? \leq 80) \land (\#pattern? \leq 3) \land (j \geq 1 \land j \leq \#string?) \land (i \geq 1 \land i \leq \#pattern?)$

$\land (((string?\ j+i) = (pattern?\ i)) \land (pos! = j)))$

$$\vee ((\#string? \leq 80) \wedge (\#pattern? \leq 3) \wedge (j \geq 1 \wedge j \leq \#string?) \wedge (i \geq 1 \wedge i \leq \#pattern?)$$

$$\wedge (((string?\ j + i) = (pattern?\ i)) \wedge (pos! = 0)))$$

The predicates are derived from the specifications directly. The case of $(pos! = 0))$ requires some expansion. The input predicates $(\#string? \leq 80)$ and $(\#pattern? \leq 3)$ give the "text length" and "pattern length" subdomains. Operator subdomains can provide partitions from these predicates. Using knowledge of the type we can derive finer partitions using illegal and legal values. The last partition of pattern position can be extracted from the last input predicate $(string?\ j + i) = (pattern?\ i)$. Operator subdomains provide two cases : the pattern exists $(=)$ or it does not exist $(\neq)$ in the text string. The position is determined from predicates involving j. $(j \geq 0) \wedge (j \leq \#string?)$ provides the partitions on the position of the string. The pattern position is not of interest to us, especially since it is specified using the $\forall$ qualifier. Thus the partitions that are generated with the proposed method are listed in table 3.2.

Field	Value
Text Length $\#string?$	0 80 > 80 $1 \cdots 79$
Pattern Positions j	no pattern 1 (beginning) $\#string?$ (end) $1 \cdots \#string?$ (middle)
Pattern Length $\#pattern?$	1 3 > 3 $1 \cdots 2$

Table 3.2: Proposed checkpoint encoding for STRMAT

- **The TRIANGLE program**. The TRIANGLE program classifies a triangle as legal or not. If its legal, there is a further classification as to whether it is isosceles, equilateral or scalene. The lengths of the sides of the triangle are input to the program. Any combination of input sides where the sum of the inputs of any two given sides is less than or equal to the third side is classified as an illegal triangle.

$$TRI\_TYPE ::= INVALID \mid ISOCELES \mid EQUILATERAL \mid SCALENE$$

TRIANGLE

$a?, b?, c? : \mathbb{N}$

$result! : TRI\_TYPE$

$(((a? + b? \leq c?) \lor (b? + c? \leq a?) \lor (c? + a? \leq b?)) \land (result! = INVALID))$
$\lor (((a? = b?) \lor (b? = c?) \lor (a? = c?)) \land (result! = ISOCELES))$
$\lor ((a? = b? = c?) \land (result! = EQUILATERAL))$
$\lor (result! = SCALENE)$

The standard checkpoint encoding (Yin, Lebne-Dengel, and Malaiya 1997) for the TRIANGLE program is listed in 3.3.

Field	Value
Not a Triangle	a+b $\leq$ c
	b+c $\leq$ a
	a+c $\leq$ b
	a+b = c
	b+c = a
	a+c = b
Legal triangle	a = b
	a = c
	b = c
	a = b = c
	scalene

Table 3.3: Standard checkpoint encoding for TRIANGLE

Using the approach we get

$$(a? + b? \leq c?) \land (result! = INVALID)$$

$$\lor (b? + c? \leq a?) \land (result! = INVALID)$$

$$\lor (c? + a? \leq b?) \land (result! = INVALID)$$

$$\lor (a? = b?) \land (result! = ISOCELES))$$

$$\lor (b? = c?) \land (result! = ISOCELES))$$

$$\lor (a? = c?) \land (result! = ISOCELES))$$

$$\lor (a? = b? = c?) \land (result! = EQUILATERAL))$$

$$\lor (result! = SCALENE)$$

This yields the same partitions as the standard scheme shown in 3.3. However using type subdomains another set of partitions based on values of $a?, b?, c?$. However the new partition has dependencies on the other partitions e.g. when values of a,b,c are zero or negative can we satisfy the other partition conditions or predicates. One way to get around this is to let one

partition overrule the other i.e. using whatever values of a,b,c we get from the first domain we try to satisfy the predicate of the other domains.

Field	Value
Values of sides	$a?, b?, c? = 0$
	$0 \leq a?, b?, c? \leq MaxNat$
	$a?, b?, c? < 0$
	$a?, b?, c? \notin \mathbb{N}$

Table 3.4: Proposed additions to the standard checkpoint encoding for TRIANGLE

- **The FIND program** The program takes an integer array B of size S ≥ 1 and index F. The program sorts the array elements 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 of F is $1 \leq \text{F} \leq \text{S}$.

FIND

$b?, out! : \text{seq}\,\mathbb{Z}$
$f? : \mathbb{Z}$

$\#b? \geq 1$
$(f? \geq 1) \land (f? \leq \#b?)$
$out! = b?$
$\forall\, i, j \in \mathbb{Z} \mid ((i > 0) \land (i < f?) \land (j \leq \#out!) \land (j > f?))$
$\bullet\ ((out!\, i) \leq (out!\, f)) \land ((out!\, f) \leq (out!\, j))$

The partitions generated for the standard checkpoint scheme (Yin, Lebne-Dengel, and Malaiya 1997) are listed in 3.5.

Field	Value
Array Size	1,2
	> 2
Array Status	already ordered
	reverse ordered
	all equal
	randomly ordered
Element Values	all positive
	all negative
	mixed
F points to	first element
	last element
	a middle element

Table 3.5: Standard checkpoint encoding for FIND

After rewriting the specification, the subdomains extracted from the pre-conditions are

$\#b? \geq 1$
$(f? \geq 1) \land (f? \leq \#b?)$

These provide the array size and index(F) partitions. The output predicates could be used to provide a subdomain, since $b?$ is assigned to $out!$ they are providing a post-condition on the array $b?$. Thus the following subdomains could be generated.

$(i > 0) \wedge (i < f?) \wedge (b?i \leq b?f?)$
$(j > f?) \wedge (j < \#b?) \wedge (b?j \geq b?f?)$

Operator subdomains can be generated using these predicates. However, by looking at the input type; a sequence of integer numbers ($\mathrm{seq}\,\mathbb{Z}$), type subdomains can be generated on the status of the sequence as well as element values. What is listed in 3.6 is an example of the partitions that could be generated. Depending upon heuristics used, an alternate set of partitions could well be generated.

Field	Value
Array Size	0
	1
	$1 \cdots MaxSequenceSize$
	$MaxSequenceSize$
F	0
	1
	$2 \cdots (\#b?$ - 1)
	$\#b?$ (Array Size)
Array Status	already ordered
	reverse ordered
	all equal
	randomly ordered
Element Values	$(\text{-}\mathbb{Z}_max) \cdots$ (-1)
	0
	$1 \cdots (\mathbb{Z}_max)$
	$(\text{-}\mathbb{Z}_max) \cdots (\mathbb{Z}_max)$

Table 3.6: Proposed checkpoint encoding for FIND

3.4.1 Observations

The partitions generated by the proposed method are extensive and quite complete. When comparing them with partitions generated by Yin *et al* (Yin, Lebne-Dengel, and Malaiya 1997) some of them seem non-intuitive. Some partitions do not make sense when we look at the programs they are meant to test. That is both an advantage as well as a disadvantage. The partitions generated may not as clever or efficient as the manually generated ones but they are more detailed. They may test more illegal conditions that may be thought of as too obvious by a manual tester. A surprising number of the partitions generated are type subdomains. This may indicate that the specifications are not as detailed as they should be or are too simple. If the specifications does not clearly spell out the conditions and characteristics of the data structures involved, the quality of the partitions

generated could decrease considerably. Also the need to use the output predicates or post-conditions is emphasized in the FIND and to some extent in the STRMAT specification example. Thus the procedure is successful in generating effective partitions. The actual encoding step, while not shown, is straightforward and can be carried out with little effort.

3.5 Conclusion

Based largely on the work Hierons (Hierons 1997b) a procedure for generating partitions and encoding them is proposed. The procedure works well when compared to that in (Yin, Lebne-Dengel, and Malaiya 1997) for several examples. The encoding process is still unclear due to insufficient understanding of what is the best scheme to be followed. As understanding of efficiency of different encoding rules is better understood this phase can also be implemented. Other limitations of this scheme are discussed.

The various approaches used in the generation of partitions from Z specifications have been explained in detail. They do not differ a lot in their style but in the formality of their approach. This approach seems to have largely stabilized with few drastic variations being proposed. This is the main reason that this thesis adopts this approach. This thesis does not claim that this is the best method. Far from it. The objective was to *prove* conclusively that the automation of anti-random testing is feasible. Also the Z formal specification langauage was chosen because it seemed to be the most popular language for standard application domains. The choices of formal language and method were purposely directed to show that there was enough work done to ensure that automation could be implemented. This is not intended to be the best or most efficient method just one that would work in the outside world.

3.5.1 Tool Support

Since the procedure for checkpoint encoding has been laid out the support for automation is now reviewed. Dick and Faivre (Dick and Faivre 1993) describe a tool (Dick and Faivre 1993) written in Prolog that implements their method albeit for small specifications. It hooks onto a VDM parser to extract a representation of the VDM specification. The Prolog code reduces the VDM specification to DNF form and has up to 200 inference rules. However this tool does not implement generation of test values.They are working on improving the tool to handle progressively larger specifications.‘

More interesting to this thesis is the test generation from Z with Isabelle (Helke, Neustupny, and Santen 1997; Kolyang, Santen, and Wolff 1996). The Isabelle system is a tactical theorem prover. At it’ heart is two powerful parameterized “tactics” : the classical reasoner and the simplifier. The

former simulates a proof system and is parameterized by sets of introduction and elimination rules. The simplifier implements conditional higher-order rewriting. While they implement a variant of Dick and Faivre's method, rules for the procedure described in the thesis could well be substituted. In citekolyang96, helke97 they prove that Z can be encoded in the higher order logic implemented in Isabelle while preserving the structure of specifications. The deduction support offered by Isabelle then allows reasoning about Z specifications. This is most important feature, allowing different sets of rules to be implemented. As the understanding of the procedure that this thesis proposes is better understood for complex specifications, the rules can be augmented to allow the elimination of unnecessary and unsatisfiable disjuncts.

Stepney (Stepney 1995) advocates systematically building abstractions of operation specifications for testing. This is based on ZEST (Cusack 1992), an object-oriented extension of Z. A tool to support these activities leaves the deduction/rewriting steps of simplification and weakening of predicates to the person setting up the the tests. It just uses a structured editor to support this process and cannot be classified as an automated testing tool.

An interesting idea addresses the test evaluation process i.e. running the tests using data generated. Mikk (Mikk 1995) describes a test evaluation tool which transforms into "executable" forms that are then compiled to Boolean valued C functions. This is not relevant to this thesis but may provide an answer to the problem of generating a test script to generate the data from the partitions.

3.5.2 Limitations

So far the advantages have been described, but what are the limitations to this procedure described ?

1. The use of Z is suspect. Z (and model based specifications in general) cannot be used to specify all kinds of systems. Those with concurrent or timing issues are specified using other langauages. Also Z is a cryptic language. For formal specifications to be used in the industry any specification technique must use graphical tools and other schemes to simplify how complex systems are specified. Thus another language might be a better choice.

2. While the structure of Z schemas describe pre and post conditions, this is not always the case. Some times they are implicit and have to be derived. This procedure depends on how the specifications is written. If the Z specification is written poorly then the test generation procedure could fail *i.e.* not all tests can be generated.

3. Removal of irrelevant predicates is difficult to do, so as to come up a minimal set of disjoint

partitions. Also interdependencies between partitions is difficult to extract. These are problems in automation that could be resolved with good automation tools.

The last two limitations are related. As the rules for rewriting and deriving disjoint partitions are standardized in an automated framework, such as Isabelle, these will be progressively improved. Also more complex specifications with complex operators and logic must be examined. One major phase that has not been discussed is creation of a test script. Once the encoding is complete, test data needs to be instantiated given a binary vector. This is outside the scope of this thesis.

Chapter 4

Sequencing of Tests

In most references to testing general software systems the issue of state is not often mentioned. Any reasonably complex software has several states. Whether the test adequacy criteria is coverage based or fault based, if the state of the software is not taken into account before generating test cases, the necessary adequacy criteria may not be satisfied . This is not to say that state is ignored, only that it indirectly resolved by the test generation method. The question is whether this is satisfactory or should state be made an integral part of testing. This question is pertinent in the case of anti-random testing.

Hardware testing has advanced much further than software testing. And while not all techniques can be transferred, most of testing theory stems from hardware testing. In hardware the circuits are classified into sequential and combinational circuits. Sequential circuits use the state of the circuit and the current input to determine the output. In combinational circuits the output is calculated based solely on the current input. The state is effectively a function of past inputs, so in sequential circuits previous inputs also affect how the current output will be like. While software is not classified as such, the same behaviour hold. Most software is sequential; using state to determine the output behaviour. The state, in software is the set of values of the internal variables. It could also be external in nature as in the case of database or control software. In any reasonably practical software the state is important.

Yet in software there are no special techniques as in hardware (Nachman, Saluja, Upadhyaya, and Reuse 1998) to test software from this perspective. The notable exception is testing mission-critical software or telecommunication protocol software(Poston 1996; v. Bochmann ad Alexandre Petrenko 1994). In Beizer (Beizer 1990) state testing is dealt with quite extensively. Finite state machine models are applicable to more general software systems, and these methods have the potential for wider use which has yet to be fully exploited (Laycock 1992).

In this chapter the issue of state based testing is closely examined.A possible solution discussed

in the context of extending checkpoint encoding to use state in generating tests. The rationale being that this improves the applicability of anti-random testing to more complex systems by using as much information as is possible. First the basics of state-based testing are reviewed. Then a technique to generate the FSM of a system from its Z specification is proposed. This is an extension of the work done with extraction of partitions from Z specifications. By building upon work already done and extracting a FSM, is there is a clear benefit of sequencing tests? There are 3 questions that this chapter answers. First, once a FSM is extracted, how can tests be sequenced with minimal effort? Secondly, how will checkpoint encoding apply to sequences of tests? Lastly, is there an increase in coverage when using this method? This is a proof of concept, attempting to show that checkpoint encoding can be extended to use state. For state intensive or mission critical systems that require stringent tests on the correctness of the state machine this is not applicable. The proposed techniques should be able to be applied to standard software systems therefore simplicity is a must.

4.0.3 Testing Based on Finite State Machines

The core of testing "sequential" software is modeling the state information in the program. The **Finite State Machine** (FSM) is a model of software structure(the FSM drives the control structure), software behaviour or specification of software behaviour. It is a functional testing tool (Beizer 1990). Consider Mealy machines. This is a deterministic machine that produce outputs on their state transitions after receiving inputs. The Moore machines are a variant, in which the output is only determined by the state, but these are not considered.

A finite state machine M is a quintuple

$$M = (I, O, S, \delta, \lambda)$$

where I, O, S are finite and nonempty sets of input symbols, output symbols and states respectively;
$\delta : S \times I \rightarrow S$ is the state transition function;
$\lambda : S \times I \rightarrow O$ is the output function.

A FSM can be represented by it's state transition diagram or a state table. A state transition diagram is a directed graph whose vertices correspond to the states of the machine and whose edges correspond to state transitions. Each edge is labeled with the input and output associated with the transition. Another, perhaps convenient way to represent a FSM is using a state table. Each row of the table corresponds to a state, each column to an input condition. The intersections of a row and column give the next state (transition) and the output.

Essentially in testing using a FSM, the FSM of the implementation (or the implementation itself) and the FSM of the specification are compared. This problem is well known in literature and

is referred to as the "checking sequence" or "machine verification" problem. Two kinds of errors that can be detected are : *output errors*, where the output produced for certain state transitions is wrong, and *transfer errors*, where the implementation's next state reached by a transition is faulty. Discussion of the fault coverage model of test methods is based on these two faults. Other errors include *unreachable states*; which are states that no input sequence can reach, *dead states*; which are states that once entered cannot be left, and *equivalent states*; in which every sequence of inputs starting from both states produce the same sequence of outputs. These errors are due to poor design of the system or poor modeling of the FSM.

A number of test selection methods have been developed for testing from a FSM modeled from the specification of the system. They are based on different test criteria(Kohavi 1978), including simply executing(traversing without regard for correctness of the before and after states) every transition, testing every transition, and producing a "checking" sequence. Coverage criteria could thus be simple: cover every state, transition or end to end paths, or complex i.e. % of faults covered. Regardless, given a test criterion, it is desirable to produce the shortest test suite that satisfies this criterion. The best known methods are Transition Tour(TT) (Naito and Tsunoyama 1981), W method (Chow 1978), Partial W-method (Fujiwara, Bochmann, Khendek, Amalou, and Ghedamsi 1991), Distinguishing Sequence (DS) method (Gonenc 1970), Unique-Input-Output (UIO) method (Sabani and Dahbura 1988). The W-method and DS method were originally developed in the context of software and hardware, respectively. The test suites developed by all these methods will find all output errors of the implementation FSM. However transfer errors are not always detected. When testing a transition it is necessary to check the final state. In order to do this one of the following approaches can be applied.

- a *distinguishing sequence* (DS): is a sequence that produces a different output for each state.
- an *unique input output sequence* (UIOs) : a UIO sequence is specific to a state, and can verify that state, but not necessarily any other state.
- a *characterizing set* (W set) : this is a set of input sequences which can distinguish between every pair of states.

By creating test suites using any one the above approaches one can test for faults. However it is not always possible to create a DS or an UIO for every state in a FSM even if they satisfy the criteria of minimality, complete specification and strong connection (Fujiwara, Bochmann, Khendek, Amalou, and Ghedamsi 1991) which not all do. Even if they do, often the algorithms needed to identify such sequences are complex and take too much time. The W-method and partial W-method which

use a characterizing sequence are easier to use and are effective. They are applicable to any FSM satisfying the assumptions of minimality, complete specification and reachability of all states from the initial one.

These methods make sense when testing FSMs in the traditional areas of protocols etc. However there are simpler methods like the transition tour (TT) (Naito and Tsunoyama 1981) and state tour (ST) (Kohavi 1978) methods. The TT method executes all transitions of the specification FSM atleast once, but does not make any effort to identify the target states. The ST method covers all states (but not necessarily all transitions) and does not identify all states reached. These have very poor fault-detection power as is expected. However they have very short test suites and are simple to calculate. This thesis hypothesizes that *they can be used to increase coverage, not of faults, but of the code itself.* This is an aspect of testing using an FSM that this thesis investigates.

4.1 Extraction of the Finite State Machine (FSM)

The idea of extracting state from specification is well accepted idea. However using state of the system as part of testing has been done with formal languages such as process algebra. In languages of this class the states and transitions are specified explicitly. In model based languages like Z this is more difficult.

4.1.1 Deriving a FSM from Z Specifications

In Dick and Faivre (Dick and Faivre 1993) the issue of controlling testing using a FSM is considered very important. This is done with a VDM-SL specification, hence there are some differences.

1. Perform partition analysis on all individual operations and initial state to obtain the set of sub-operations. These provide the set of transitions.

2. Extract from every sub-operation two sets of constraints, one describing the before state, the other describing the after state. This is done by quantifying all variables external to the state.

3. Reduce to DNF the disjunction of the constraints found using the earlier step. The partitions correspond the before state and after state. This step is just to partition the states cleanly.

4. Try to resolve the sub-operations with the states to generate the FSM. For every sub-operation OP and states S1, S2 satisfying $(S1, S2) \in rel - OP$, where $rel - OP$ is the relation on states defined by the constraints on OP, a transition labeled OP is created from S1 to S2.

5. If possible, simplify FSM.

They then proceed to propose a method to test using the FSM generated. It is a simple scheme, a variant of the TT method, in which paths traversing the FSM are selected such that all transitions are covered. This being done in VDM-SL is peculiar to that language. Also Dick and Faivre are not very precise about the entire process but this paper lays out the basic process.

In Hierons (Hierons 1997b) the FSM is extracted using the partitions generated. Assuming the partitions have been generated, the process begins as follows

- Create disjoint states.

  The set of disjoint partitions generated, P_X, are partitions on the space of input variables, input state variables and input intermediate variables. For an input state satisfying P_X the input-output (before-after state) relationship is

  $$\exists\, I' \bullet P_X \wedge (\vee_{i \in X}\ Q_i[Y_i/In_i])$$

  where I' represents the complete set of intermediate variables, Q_i represents the postcondition for the equivalent partition (or precondition) P_i, In_i is the intermediate variable (in Q_i) that has been renamed to Y_i.

  The set of P_X with the intermediate variables quantified, provides a partition on the space that is made up of input variables and the initial internal state variables. To generate the set of states for the FSM, Hierons partitions the initial internal state (before state) only and all other variables are hidden. Thus the description of input or before states are

  $$P'_X = (\wedge_{i \in X}\ \exists\, I, Y_i \bullet P_i[Y_i/In_i]) \wedge (\wedge_{i \notin X}\ \exists\, I, Y_i \bullet \neg P_i[Y_i/In_i])$$

  where I is the set of input variables. This yields the set of before states for the FSM.

  A modification to Hierons approach proposed in this report is to carry out a similar step with the output predicates. This would yield

  $$Q'_X = (\wedge_{i \in X}\ \exists\, O, Y_i \bullet Q_i[Y_i/Out_i]) \wedge (\wedge_{i \notin X}\ \exists\, O, Y_i \bullet \neg Q_i[Y_i/Out_i])$$

  the set of after states Q_X. This is inspired by the approach taken in Murray et al. Both these combine to form the set of states of the FSM. Hierons does not explicitly carry out this step.

- Determine the transitions.

  To determine whether a pair of states satisfies the specification, Hierons substitutes them as input and output state in the specification. If any operation can satisfy both input and output state for any input it provides the transition. This is similar to Dick and Faivre.

  Thus the operation can move the system from a state satisfying P'_X to a state satisfying P'_Y if and only if there exists a initial state state s_x satisfying P'_X, final state s_y satisfying P'_Y, input

$i?$, output $o!$ such that substituting these into the rewritten form of the specification gives a predicate that evaluates to true. Expressing the initial state as S, the final state as S', the input as I, the output as O, the set of intermediate variables as I', this predicate becomes

$$\exists\, k \in X, s_x, s_y, i?, o!, I' \bullet (P'_Y[S'/S] \wedge P_k \wedge Q_k)[s_x, s_y, i?, o!/S, S', I, O]$$

In Murray *et al* (Murray, Carrington, MacColl, MacDonald, and Strooper 1998) this technique is described less rigorously. They however make an important point; that both the test templates and oracles must be used to derive states of the system's finite state machine. The states from the test template only indicate the starting states which may not be the complete set of states. Hierons does not explicitly state this and this may depend more on the individual specification but without lack of evidence one can quite confidently state that Murray's approach seems to be more complete. This also dovetails with the thesis in emphasizing the importance of the output predicates or post-conditions.

The states are derived from the test templates of each operations test template hierarchy by using schema hiding to restrict the signatures of the templates to be the only state variables. Hiding involves removing the input variables and existentially quantifying them in the template's predicate. To derive states from the oracles the primed state variables are renamed to their unprimed equivalents. Any output variables are hidden. Thus templates are simplified to focus only on the state variables and state templates are derived.

Once this has been done, we consider whether the state templates are disjoint. This has to be satisfied to get a minimal partition of the system's state space. To do this we transform a disjunction into disjoint components (Dick and Faivre 1993). The equivalence of the form

$$A \wedge B \equiv A \wedge B \vee \neg\, A \wedge B \vee A \wedge \neg\, B$$

is used to obtain non-overlapping partitions. The Init schema (here Murray et al expect a schema of this name to contain the initialization operation for the system) is partitioned and these states are tagged to track the initial states of the system.

The transitions of the finite state machine are derived by considering each pair of states and checking whether the pair is related by an operation.

$$\exists\, IS_op;\ OS_op \bullet ST_1 \wedge Op_Schema \wedge ST_2$$

where IS, OS are the input and output spaces (variables) of the operation op and ST_1, ST_2 is the pair of states that are under consideration. Another important property considered is *reachability*. A state can be considered reachable from the initial state via any valid sequence of transitions. First

all possible paths present in the state machine is specified. Then all paths which start with an initial state is considered and all states in the path form the set of reachable states. These states and associated transitions are removed to form the final finite state machine.

4.1.2 ATM Specification Example

Certain examples of systems have become very popular in systems development *e.g.* cruise control system for a car, elevator controller. One such system is the **Automated Teller Machine (ATM)** system (Coleman, Arnold, Bodoff, Dollin, Gilchrist, Hayes, and Jeremaes 1994). This has also been cited as a good example for OO design and development.

4.1.2.1 Requirements of the ATM system

AN ATM is a machine through which the customers can perform several common financial transactions. The machine has a display screen, a bank card reader, a keypad accepting alphanumeric input with special function keys needed to speed up the operation, a money dispenser slot, and a receipt printer.

When the machine is idle, a greeting is displayed. As soon as a card is inserted the system is activated. The reader attempts to read the card. If the card cannot be read, the user is informed and the card is ejected. The system returns then to the idle state. If the card is readable (and is verified i.e. recognizable by the system database) the user is asked to enter a personal identification number (PIN). If the PIN is correct, the user is shown the menu with a list of transactions : deposit funds into an account, withdraw funds from an account, query balance for an account etc ... The customer has 3 attempts to enter the PIN. After that the card is kept by the ATM and the customer is informed.

For the deposit transaction, the customer must specify the account number and the amount to be deposited. The withdraw transaction requires the customer to specify the account and the amount to be withdrawn. This amount must not exceed the limit on the card or cause the account to be overdrawn beyond a limit. The query transaction requires that the customer enter an account for which the balance is displayed. After completing a transaction the menu is again displayed. To exit the system the user must press the Cancel key. The user's card is returned and the machine returns to the idle state. During a transaction if the customer makes a mistake then he/she can press the Cancel key to void the transaction and begin again. If the customer enters invalid data in the transaction, they are informed and the ATM displays the menu again.

4.1.2.2 Assumptions

We wish to focus on the extraction of state from this specification, the validity of the specifications is thus not particularly important to us. There may be some instances where the specification falls short or exceeds the above requirements.

The data base definition is one on which the requirements do not say much. We assume a simple one : the database (i.e. the bank) contains accounts. Each account has to be linked to one PIN. Several accounts may be linked to one PIN. Each PIN is paired with one card. Thus the database contains which cards can be read by the ATM. Each PIN has a limit on withdrawal. Each account contains the amount of money it contains and the overdraw limit.

The output or display system are not specified to any detail. The events are also assumed to be handled separately, they are modeled as if they were an input already processed. This is partly due to minimal experience lack of knowledge of how Z can be used. A transaction view of the system is taken looking at the sequence of operations for one transaction at a time.

4.1.2.3 Z specification

$$[CARD, PIN, ACC]$$

$$\begin{aligned} MENUCHOICE ::= & \; DEPOSIT \\ | & \; WITHDRAW \\ | & \; QUERY \\ | & \; CANCEL \end{aligned}$$

$$\begin{aligned} DISPLAY ::= & \; ENTERPIN \\ | & \; INVALIDCARD \\ | & \; ENTERMENUCHOICE \\ | & \; ENTERPIN - 2ndTRY \\ | & \; ENTERPIN - LASTTRY \\ | & \; INVALIDPIN \\ | & \; ENTERACCT\&DEPOSITAMT \\ | & \; ENTERACCT\&WITHDRAWAMT \\ | & \; ENTERACCT \\ | & \; AMTDEPOSITED \\ | & \; INVALIDACCT \\ | & \; AMTWITHDRAWN \\ | & \; OVERDRAWLIMITEXCEEDED \\ | & \; PINLIMITEXCEEDED \\ | & \; BALANCEDISP \\ | & \; CARDRETURN \end{aligned}$$

$$
\begin{array}{rcl}
STATUSVAL & ::= & ACCEPTCARD \\
& | & ACCEPTPIN \\
& | & PINFAIL1 \\
& | & PINFAIL2 \\
& | & MENU \\
& | & DEPOSIT \\
& | & WITHDRAW \\
& | & QUERY \\
& | & CANCEL
\end{array}
$$

Accounts

$$
\begin{array}{l}
accnos : \mathbb{P}\, ACC \\
Amt : ACC \pfun \mathbb{Z} \\
OLimit : ACC \pfun \mathbb{N} \\
\hline
\mathrm{dom}\, Amt = accnos \\
\mathrm{dom}\, OLimit = accnos
\end{array}
$$

ATMPins

$$
\begin{array}{l}
Accounts \\
atmpin : \mathbb{P}\, PIN \\
pinacct : ACC \pfun PIN \\
pinlimit : PIN \pfun \mathbb{N} \\
\hline
\mathrm{dom}\, pinacct = atmpin \\
\mathrm{ran}\, pinacct = accnos \\
\mathrm{dom}\, pinlimit = atmpin
\end{array}
$$

ATMCards

$$
\begin{array}{l}
ATMPins \\
atmcard : \mathbb{P}\, CARD \\
cardpin : CARD \rightarrowtail PIN \\
\hline
\mathrm{dom}\, cardpin = atmcard \\
\mathrm{ran}\, cardpin = atmpin
\end{array}
$$

ATM

$$
\begin{array}{l}
ATMCards \\
status : STATUSVAL \\
card : \mathbb{P}\, CARD \\
pin : \mathbb{P}\, PIN \\
\hline
\#\, card = 1 \\
\#\, pin = 1 \\
\forall\, c : card;\ p : pin \bullet (c, p) \in cardpin
\end{array}
$$

$$
\begin{array}{lcl}
ATMOp & \widehat{=} & ATMInit \fatsemi Transaction \\
Transaction & \widehat{=} & AcceptCard \fatsemi AcceptPin \fatsemi MenuOp \fatsemi Transaction \\
MenuOp & \widehat{=} & AcceptMenuChoice \fatsemi (Op \fatsemi MenuOp) \lor Cancel \\
Op & \widehat{=} & Deposit \lor Withdraw \lor Query
\end{array}
$$

ATMInit

ATM'
$initamt : \mathbb{Z}$

$\forall initamt \in \operatorname{ran} Amt' \bullet initamt \geq 0$
$status' = ACCEPTCARD$

AcceptCard

ΔATM
$custcard? : CARD$
$msg! : DISPLAY$

$(status = ACCEPTCARD \land custcard? \in atmcard \land status' = ACCEPTPIN \land card' = custcard?$
$\land\ msg! = ENTERPIN)$
$\bigvee (custcard? \notin atmcard \land status' = ACCEPTCARD \land msg! = INVALIDCARD)$

*AcceptPin*1

ΔATM
$custpin? : PIN$
$msg! : DISPLAY$

$(status = ACCEPTPIN \land custpin? \in cardpin(\!|\ card\ |\!) \land status' = MENU \land pin' = custpin?$
$\land\ msg! = ENTERMENUCHOICE)$
$\bigvee (status = ACCEPTPIN \land custpin? \notin cardpin(\!|\ card\ |\!) \land status' = PINFAIL1 \land$
$msg! = ENTERPIN - 2ndTRY)$

*AcceptPin*2

ΔATM
$custpin? : PIN$
$msg! : DISPLAY$

$(status = PINFAIL1 \land custpin? \in cardpin(\!|\ card\ |\!) \land status' = MENU \land pin' = custpin?$
$\land\ msg! = ENTERMENUCHOICE)$
$\bigvee (status = PINFAIL1 \land custpin? \notin cardpin(\!|\ card\ |\!) \land status' = PINFAIL2 \land$
$msg! = ENTERPIN - LASTTRY)$

*AcceptPin*3

ΔATM
$custpin? : PIN$
$msg! : DISPLAY$

$(status = PINFAIL2 \land custpin? \in cardpin(\!|\ card\ |\!) \land status' = MENU \land pin' = custpin?$
$\land\ msg! = ENTERMENUCHOICE)$
$\bigvee (status = PINFAIL2 \land custpin? \notin cardpin(\!|\ card\ |\!) \land status' = ACCEPTCARD \land$
$msg! = INVALIDPIN)$

AcceptMenuChoice

ΔATM
$custchoice? : MENUCHOICE$
$msg! : DISPLAY$

$(status = MENU \land custchoice? = DEPOSIT \land status' = DEPOSIT \land msg! = ENTERACCT\&DEPOS$
$\bigvee (status = MENU \land custchoice? = WITHDRAW \land status' = WITHDRAW \land msg! = ENTERACCT\&$
$\bigvee (status = MENU \land custchoice? = QUERY \land status' = QUERY \land msg! = ENTERACCT)$
$\bigvee (status = MENU \land custchoice? = CANCEL \land status' = CANCEL \land msg! = TRANSCANCEL)$

Deposit

ΔATM
$custacc? : ACC$
$amtdep? : \mathbb{N}$
$msg! : DISPLAY$

$status = DEPOSIT \land$
$((custacc? \in pinacct^{\sim}(\!| pin |\!) \land Amt' = Amt \oplus \{custacc? \mapsto (Amt\ custacc? + amtdep?)\} \land$
$status' = MENU \land msg! = AMTDEPOSITED)$
$\bigvee (custacc? \notin pinacct^{\sim}(\!| pin |\!) \land status' = MENU \land msg! = INVALIDACCT))$

Withdraw

ΔATM
$custacc? : ACC$
$amtwith? : \mathbb{N}$
$msg! : DISPLAY$

$status = WITHDRAW \land$
$((custacc? \in pinacct^{\sim}(\!| pin |\!) \land$
$\quad ((amtwith? \leq pinlimit\ pin \land$
$\quad\quad ((amtwith? \leq (Amt\ custacc? - amtwith?) \land msg! = AMTWITHDRAWN \land$
$\quad\quad Amt' = Amt \oplus \{custacc? \mapsto Amt\ custacc? + OLimit\ custacc?\} \land status' = MENU)$
$\quad\quad \bigvee (amtwith? > Amt\ custacc? + OLimit\ custacc? \land msg! = OVERDRAWLIMITEXCEEDED \land$
$\quad\quad status' = MENU)))$
$\quad \bigvee (amtwith? > pinlimit\ pin \land msg! = PINLIMITEXCEEDED \land status' = MENU)))$
$\bigvee (custacc? \notin pinacct^{\sim}(\!| pin |\!) \land status' = MENU \land msg! = INVALIDACCT))$

Query

ΔATM
$custacc? : ACC$
$amtbal! : \mathbb{N}$
$msg! : DISPLAY$

$status = QUERY \land$
$((custacc? \in pinacct^{\sim}(\!| pin |\!) \land amtbal! = Amt\ custacc? \land status' = MENU \land msg! = BALANCEDISP)$
$\bigvee (custacc? \notin pinacct^{\sim}(\!| pin |\!) \land status' = MENU \land msg! = INVALIDACCT))$

Cancel

ΔATM
$msg! : DISPLAY$

$status = CANCEL \land msg! = CARDRETURN \land status = ACCEPTCARD$

4.1.3 Proposed Method to Derive a FSM

The method proposed in this thesis is a simple one. It is largely based on Hierons' approach. However the output predicates are used as well. This follows the same reasoning as with the extraction of partitions. Murray *et al* also use the same idea in their technique for calculating the FSM. Instead of describing the step by step derivation process for the entire ATM specification we describe the steps for one operations and then show the final states derived.

Consider the *AcceptPin1* specification.

*AcceptPin*1

ΔATM
$custpin? : PIN$
$msg! : DISPLAY$

$(status = ACCEPTPIN \land \{card \mapsto custpin?\} \in cardpin \land status' = MENU \land pin' = custpin?$
$\land\ msg! = ENTERMENUCHOICE)$
$\bigvee (status = ACCEPTPIN \land \{card \mapsto custpin?\} \notin cardpin \land status' = PINFAIL1 \land$
$msg! = ENTERPIN - 2ndTRY)$

Step 1. Classify variables.

This is quite straightforward for this schema. All variables ending with a ? are input variables, with ! are output variables and the rest are input state or output state variables depending on whether they are primed or not.

- Input variables : custpin?
- Output variables : msg!
- Input state variables : status, card, cardpin
- Output state variables : pin', status'

Step 2. Rewrite the specification.

The specification is already in the required form.

Step 3. Categorize the predicates.

Based on the classification of variables the predicates are classified as either input or output predicates.

- Input predicates :

  $p_1 : status = ACCEPTPIN$

$p_2 : \{card \mapsto custpin?\} \in cardpin$

$p_3 : \{card \mapsto custpin?\} \notin cardpin$

- Output predicates :

  $q_1 : status' = MENU$

  $q_2 : pin' = custpin?$

  $q_3 : msg! = ENTERMENUCHOICE$

  $q_4 : status' = PINFAIL1$

  $q_5 : msg! = ENTERPIN - 2ndTRY$

Step 4. Rewrite specification to required form.

The required form is

$$\bigwedge_{1 \leq i \leq n} (X_i \bigvee Y_i)$$

where X_i, Y_i consist of input and output predicates only respectively. Thus we have $(p1 \wedge p2 \wedge q1 \wedge q2 \wedge q3) \bigvee (p1 \wedge p3 \wedge q4 \wedge q5)$

$P_1 : p_1 \wedge p_2$

$Q_1 : q_1 \wedge q_2 \wedge q_3$

$P_2 : p_1 \wedge p_3$

$Q_2 : q_4 \wedge q_5$

Step 5. Extract disjoint subdomains.

The input domains are already disjoint. They provide a partition on the space of input variables and input state variables.

Step 6. Generate the states and transitions.

To extract the states we hide all variables and partition only the space of internal state variables. This gives p_1 as the initial or before state. Using the output predicates similarly we derive after states as $q_1 \wedge q_2$ and q_4. This is done by hiding output variables. Also all primed variables are renamed to their unprimed values.

Whether q_2 should be a part of the output state is questionable since it contains a state variable. Here is where the rules get fuzzy and operator choice plays an instructive role. In Hieron's approach only the initial state (pre-conditions) are derived. In Murray et al the after states are also extracted. We have attempted to combine the two in the hope of improving the initial technique. This is not so easy since in Murray et al the test cases are derived in using strategies on the specification and not just by manipulating the specification.

This thesis proposes to extract both before and after states and then try to ensure that no duplicate or overlapping states occur between before and after states. Thus consider the above specification one of the after states is $status = MENU \wedge pin = custpin?$. After deriving states for the all other schemas we see that the only similar before state is $status = MENU$. This before state's partition of the state space is a superset of that of the after state and thus we can combine both of then into the before state.

Similarly states can be extracted from all operation schemas. The initial state is a default *start* that is the before state of *ATMInit* schema. The resultant set of states for the ATM specification is

1. $status = ACCEPTCARD$
2. $status = ACCEPTPIN$
3. $status = MENU$
4. $status = PINFAIL1$
5. $status = PINFAIL2$
6. $status = DEPOSIT$
7. $status = WITHDRAW$
8. $status = QUERY$
9. $status = CANCEL$

Verifying each pair of states with each operation we extract the transitions between the states. Each transition is, as a result of this method, an operation schema that changes the system state.

- $start \rightarrow ATMInit \rightarrow 1$
- $1 \rightarrow AcceptCard \rightarrow 2$
- $2 \rightarrow AcceptPin1 \rightarrow 3, 4$
- $4 \rightarrow AcceptPin2 \rightarrow 3, 5$
- $5 \rightarrow AcceptPin3 \rightarrow 3, 1$
- $3 \rightarrow AcceptMenuChoice \rightarrow 6, 7, 8, 9$
- $6 \rightarrow Deposit \rightarrow 3$

- 7 → *Withdraw* → 3
- 8 → *Query* → 3
- 9 → *Cancel* → 1

Using the states and transitions generate the FSM for the ATM system is shown in figure 4.1.

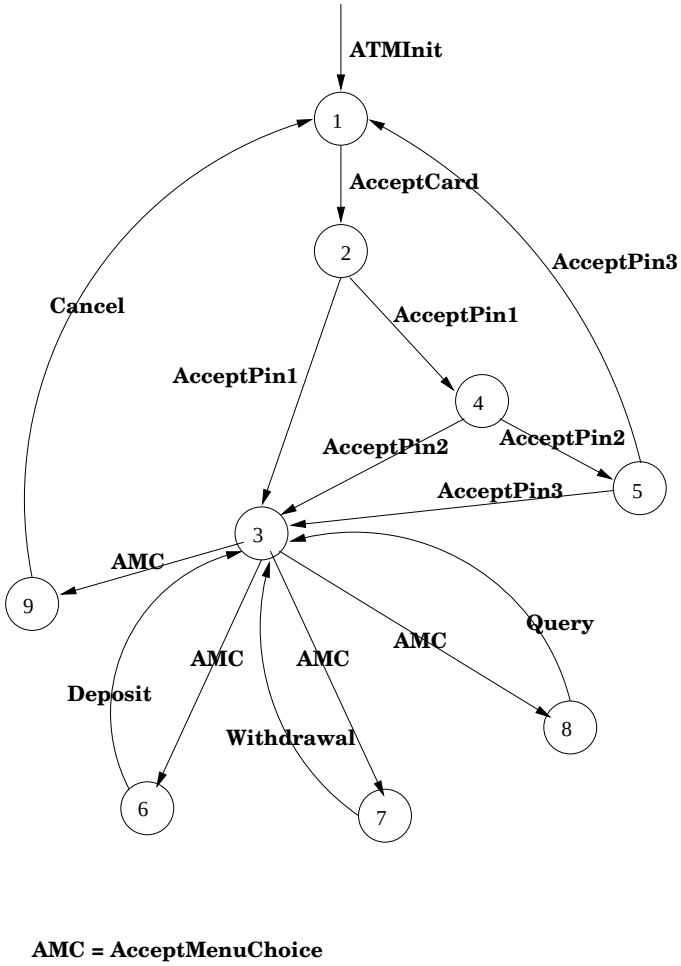


Figure 4.1: ATM Finite State Machine

4.1.4 Summary

The finite state machine of the specification is extracted using the technique explained. This technique is an extension of the extraction of partitions that was discussed in the earlier chapter. It is simple and can be automated relatively easily.

However there are some drawbacks. The outcome of this procedure depends on how the specifications has been written. This is not such a serious disadvantage as one might think. What is required is a specification which is clear and expressive as possible, stating every condition possible. Specifications that use complex or terse predicates would need to be simplified which could also be done automatically. Even if the FSM cannot be derived in one iteration, the process helps development of a better specification and brings out issues that could be hidden in a Z specification. This is so since in this process the states of the system are considered which Z does not always require to have been examined closely. Thus a different view of the system can improve understandability of the specification as it relates to the software.

Sometimes states or partitions of the state space cannot be derived directly from predicates in the schemas. One aspect of this problem is evident when looking at before and after states. Thus more work needs to be done on extracting the state that is "inherent" in a predicate i.e. if a predicate describes a operation even if the after state is not stated explicitly we should have some way of deriving what states the operation could cause.

Maybe the most serious drawback that one can point out is the very use of Z. Extraction of the FSM from Z is a relatively laborious process. It is not ideally suited to describe a system that is state-centric. Other languages like relational algebras or SDL (Telegraphy and Telephony 1989) are far suited to such systems. Extraction of the finite state machine would be much simpler, in many such languages the FSM forms the heart of the specification itself. Complex systems with attributes like concurrency are at best awkward to specify using Z. But this does not void the premise for using the approach in this thesis. Not all systems are large, complex or have time dependent properties. Yet they have states and treating them as monolithic in this respect would decrease the test effectiveness of any test set. Even a approximate representation of the finite state machine enables sequencing of test cases and can improve test effectiveness for some systems. This simple approach can be very useful in such cases. In the next section this thesis attempts to prove this simple point.

4.2 Using Checkpoint Encoding with Test Sequences

Most methods for testing state based systems use an adequacy criterion, whether it be fault based, transition coverage or state coverage. However all these criteria are based on coverage of some aspect of the FSM. What this thesis aims for is *structural* coverage. These are satsified by the test selection methods described before like the W method and the DS method. But is it possible using a much simpler technique to improve coverage ?

To improve the efficacy of testing state intensive software systems the first step is restate the raison d'etre of anti-random testing. The philosophy of the anti-random testing (Malaiya 1996) scheme is to explore the input space more completely and in a more systematic manner than can be done with random testing. Also it does not require much more effort or computation than random test generation. It is, in principle, a black box testing scheme.

For state intensive software partition and exploration of the input domain as carried out in the "standard" anti-random testing scheme ignores the whole aspect that state can control how an input is processed. Thus the input domain we are interested in has another dimension : **state**. Anti-random testing, for such systems, must explore the state space. This is conceptually difficult to grasp. The main impediment is the checkpoint encoding phase. This would have to partition and encode the state space of the software. Anti-random testing would then have any meaning in such a context. This is also difficult to implement since how should the FSM be partitioned ?. What would it mean to test one part of the state space over another ? However this is the most effective interpretation of anti-random testing for such software.

A simpler proposition is this : the input domain is now an input sequence domain, with each input not just an individual independent value but a sequence. Each element of this domain is a different sequence which may consist of *none*, *one* or *many* individual input data values. Thus instead of exploring the state space directly, a different abstraction of the input space the *input sequence space* could be explored. The rationale is that each input, by itself may or may not have any effect. However an input as part of a sequence will explore different states. The first step is to encode the input sequence space. This is the crucial step, since the encoding is the understanding of the input domain. If this is not encoded effectively then anti-random testing does not have any meaning since we cannot exploit the philosophy of antirandom testing. Also this encoding should be simple, otherwise the complexity would be a disadvantage. More mathematically rigorous state space exploration techniques could be used instead.

An input sequence is one that causes an end-to-end traversal of the FSM. There are many such sequences depending upon the complexity of the FSM under test. Simple state enumeration would generate all such "paths" through the FSM but how could this be encoded. This thesis proposes two simple different encoding schemes for state intensive software systems : **Positional Encoding** and **Hierarchical encoding**. The basis for these encoding schemes is

- The length of the sequence is an important parameter and must be controlled.
- If the length is large enough, greater than the depth of the state machine (i.e. the shortest

input sequence that can traverse the state machine) then for most systems this would guarantee exploration of some if not all states. This provides the minimum length of the input sequence that should be generated.

- For complex systems, the length parameter cannot be the sole variable. What is suggested is to identify interesting input subsequences from the FSM. They could be very crude as long as they have one basic property : they cause a transition from a state or set of states to another. This is termed in this thesis as a "transfer" subsequence. These transfer subsequences could be added as normal input partitions to a checkpoint encoding scheme that is already built up from the specification.

- At the very least, if all possible subsequence of inputs cannot be found or encoded, then from the FSM information as to which certain inputs occur before others can be extracted. This information can be added to the checkpoint encoding scheme with the inputs that have been encoded already.

1. **Positional Encoding**.

   This is the simplest extension of the current checkpoint encoding scheme. First sequences of input values have to be generated so an extra field for length of sequence is added to the standard check-pointing scheme of partitioning each input operation. Then a position field is added for certain inputs. This field contains information as to what position in the sequence should the input occur with the most frequency. Information extracted from the FSM as to whether a certain input occurs more in states closer to the start state, the later states, somewhere in the middle or is spread uniformly throughout the states. Thus in the OTCS model we can specify that *TankerArrive* input operation occurs more frequently in the beginning of a sequence.

   Thus by indicating frequency/probability of occurrence of an input in a certain location for all inputs we can bias the sequence construction towards a general ordering that we observe/extract from the state description. Anti-random testing can now test different orderings by generating appropriate vectors.

2. **Hierarchical Encoding**.

   This is a more complex encoding scheme. First, as in positional encoding, an extra field for length of sequence is added to the standard checkpointing scheme of partitioning each input operation. Then from the state machine of the model all transfer subsequences that can be

extracted are identified. This may be difficult for a large state space but if the FSM can be sufficiently abstracted.An atomic occurrence of an input operation is already handled by the standard checkpoint encoding scheme.

A *subsequence* for this scheme means one or more than one consecutive occurrence of similar or exact inputs. Often a subsequence is needed to cause a transition from one state to another. The intent is to force such sequences to occur rather than hoping they would occur in a sequence of sufficient length. This may well occur (and does) but is not efficient.

Several fields can be added for each subsequence (which also includes atomic operations). Each field describes frequency of occurrence of that subsequence in different locations within the sequence, length of a subsequence (if it is not atomic) and, possibly, relative positioning with respect to other inputs/input subsequences. This is called hierarchical encoding since encoding is carried out at two levels of abstraction : encoding sequences and also inputs within the sequence (the standard partitions based on the inputs themselves).

There are several advantages to carrying out a simple procedure.

- It is not dependent on size of FSM. There is no exponential state space explosion.
- The FSM need not have to satisfy strict requirement like strong connection, complete specification etc. to be used. It would improve the efficacy of the method but it is not necessary.
- It's simplicity ensures a small test set.

To reduce the size of test set even further one step can be taken to minimize the FSM. One simple method is to partition the FSM based on equivalence of states. This could be done based on partition of inputs that is already done as part of the standard checkpoint encoding scheme.

4.2.1 Oil Tanker Control System(OTCS)

This is a simple example of a state based system (Wordsworth 1992). This is a simulation of a simple system to control the allocation of berths to tankers in an oil terminal. An oil terminal has a number of berths to tankers can discharge their cargoes. When an approaching tanker asks for permission to dock, the controller will ask the system to allocate a berth for it to use. If no berth is free the tanker is queued in the approach to the terminal. When a tanker finishes unloading it informs the system and frees the berth. As berths are made free tankers from the head of the queue move into the terminal. The system has enquiry facilities about which tankers are queuing and which berths are occupied and by which tankers.

The OTCS is simulated using a software program that accepts a sequence of any of the following inputs.

- Tanker Arrive : A#
- Tanker Leave : L#
- Query Berth : B
- Query Queue : T
- Exit OTCS : E
- Tanker Number # : 1-99

The output is one of several flags indicating what action has been taken. The various actions are

- Tanker Arrive operation is successful : OK
- Tanker Arrive attempted with tanker already in OTCS : KNOWN_TANKER
- Tanker Arrive operation with berths full : WAIT
- Tanker Leave operation is successful : OK
- Tanker Leave attempted with tanker not at berth : NOT_AT_BERTH
- Tanker Leave operation successful, tanker moves from queue to berth : MOVE_TANKER

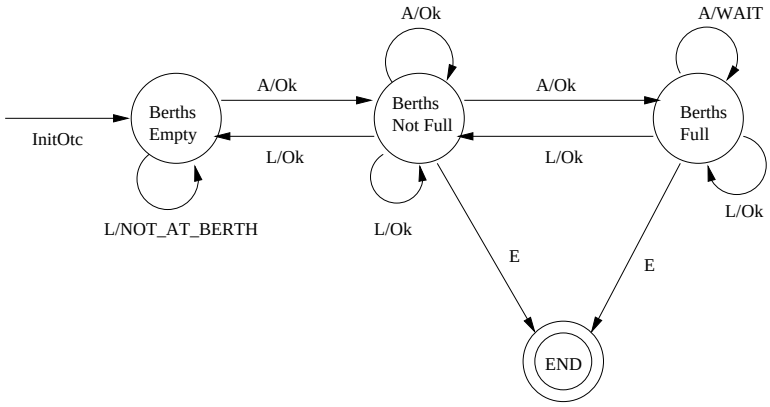
For the purposes of simulation several factors are maintained at constant value; the number of berths at 5, the queue length at 100 and the maximum length of the input sequence is 100.

4.2.1.1 OTCS Finite State Machine

The entire Z specification of the OTCS and the calculation of the FSM from the Z specification is given in Appendix I. A simplified FSM of the OTCS focusing on just the Arrive and Leave operations is shown below 4.2. *InitOtc* is the operation that initializes the OTCS system to the (initial) state *BerthsEmpty* where the berth and the queue are empty. At this state any L (tanker leave) operation results in a NOT_AT_BERTH output since there are no tankers in the berth. After a single Arrive Tanker operation the system moves to the next state *BerthsNotFull*. In this state all Tanker Arrive operations result in tankers moving into a berth. The last Tanker Arrive operation, which fills all the berths triggers a transfer to the next state *BerthsFull*. In this state all tankers that arrive go into the queue. Any tanker that leaves results in a tanker moving from the queue to replace it at

its berth. When all tanker have left from the queue, the next Tanker Leave operation transfers the system back to the earlier state *BerthsNotFull*. The system continues to operate until the Exit operation causes the OTCS to stop.

This is a simplified FSM since it deals only with Tanker Arrive and Tanker Leave operations. Also the tanker number is assumed to be valid e.g. for a Tanker Arrive operation the tanker is not already in the berth or the queue.



A : Arrive Tanker
L : Leave Tanker
E : Exit OTCS
InitOtc : sets up the OTCS in the initial state

NOTE : No Query operations are shown; all operations have a tanker number which is assumed to be correct.

Figure 4.2: Simplified FSM of the OTCS

4.2.1.2 Testing the OTCS

The base method in this case is the checkpoint encoding scheme that can be calculated using the method proposed in this thesis, discussed in the previous chapter. One assumption made is that the tanker number will always be generated correctly. This is because in modeling the OTCS it is not relevant to test the tanker numbers per se. Also the test sequence is created as per the specification; there are no incorrect operations or sequences with no operations etc. This affects the amount of coverage to some extent but since this is uniformly implemented it is not relevant to the comparison.

The base checkpoint encoding scheme (Table 4.1) has two categories based on the operations and the tanker number used in the operations. It is a simple scheme but is not exactly combinational.

Certain cases for the tanker number involve knowledge of past tanker numbers, this is a property of sequential systems. Thus checkpoint encoding per se does not rule out exploiting the sequential character of the system under test. Each test for the OTCS system is a sequence of operations. The base checkpoint encoding scheme generates each operation individually. So a test, consisting of a sequence of length n, is implemented by sequencing n separate tests using the base checkpoint encoding scheme.

Table 4.1: Base checkpoint encoding scheme

Field	Bits	Value	Significance
Operation	b1,b0	00	Tanker Arrive
		01	Query Berth
		10	Query Queue
		11	Tanker Leave
Tanker	b3,b2	01	Known Tanker from Berth
		10	Known Tanker from Queue
		00	New Tanker
		11	Illegal Tanker

The proposed sequence checkpoint encoding scheme uses a hierarchical encoding mechanism. Examining the FSM generated from the specification and the state predicates, certain transition subsequences are first identified.

- To move from *BerthsEmpty* to *BerthsNotFull* requires one Tanker Arrive operation.
- To move from *BerthsNotFull* to *BerthsFull*, the transition subsequence is a sequence of *TankerArrive* operations of length $\#berth - 1$ i.e. one less than the number of berths which is kept constant at 5.
- To move from *BerthsFull* to *BerthsNotFull* is more difficult to identify. It requires a sequence of *TankerLeave* operations. The length of this sequence cannot be determined easily so it is made symmetric to the first sequence.
- To move from *BerthsNotFull* to *BerthsEmpty* requires a sequence of Tanker Leave operations of length $\#berth - 1$.

These subsequences are now encoded. There are three partitions for each subsequence, the operations itself, the length of the subsequence and position of the subsequence. The other operations i.e. the query operations do not cause any transitions so as such they are not treated as subsequences. However additional partitions are added for these operations including a position encoding partition. For the OTCS system the length of the subsequences are made symmetrical for lack of information. The depth of this state machine is atleast 5, since we need at least that many *TankerArrive* operations

to fill the berths and cover all states. There is no field for the length of the entire sequence. Usually a field would be present with a maximum length several times the depth of the state machine. In the current experiment the length is controlled externally, forcing a fixed length sequence to be generated.

Table 4.2: Sequence checkpoint encoding scheme

Field	Bits	Value	Significance
Tanker Arrive	b1,b0	00	New tanker
		01	Known Tanker from Berth
		10	Known Tanker from Queue
		11	Invalid Tanker
Length	b3,b2	00	1
		01,10	$2 \cdots 4$
		11	5
Position	b5,b4	00	Head
		01,10	Middle
		11	Tail
Tanker Leave	b7,b6	00	New tanker
		01	Known Tanker from Berth
		10	Known Tanker from Queue
		11	Invalid Tanker
Length	b9,b8	00	1
		01,10	$2 \cdots 4$
		11	5
Position	b11,b10	00	Head
		01,10	Middle
		11	Tail
Query	b12	0	Query Queue
		1	Query Berth
Position	b14,b13	00	Head
		01,10	Middle
		11	Tail

This encoding is quite simple; it does not attempt to bias the position or length of any operation towards any specific values. If there was any information from the state machine then this could be done. For example in the OTCS system it would make sense to bias the position of the *TankerArrive* operation. The proposed encoding is listed in Table 4.2.

4.2.2 Design of the experiment

The OTCS simulation is tested using 3 different testing methods as used in Yin (Yin, Lebne-Dengel, and Malaiya 1997). All methods generate a sequence of operations of fixed length. Random (or Pure Random) testing generates input test sequences by randomly generating any of the legal operations. The tanker number is also generated randomly. Checkpoint Encoded Random testing uses randomly generated binary vectors as test cases which are then translated using both the checkpoint encoding

schemes. Anti-random testing uses binary vectors but these are generated using the anti-random generation technique (Yin, Lebne-Dengel, and Malaiya 1997; Malaiya 1996). Both the checkpoint encoding schemes are implemented for each method. The first scheme is referred to as the *standard* or *base* checkpoint encoding scheme and the second as the *sequence* checkpoint encoding scheme.

The graphs display the cumulative % block coverage vs the number of test sequences applied. The sequence length is kept fixed at 5,15, and 50 input operations. In all tests a total of 100 sequences are applied. Each sequence is applied separately. The OTCS application exits completely after each sequence is applied. There are no sequences in the base checkpoint encoding scheme. Whereas in the sequence encoding scheme as mentioned before the scheme generates a sequence of a certain length containing all the various subsequences described in 4.2. This length for the purpose of this experiment is set externally to the experimental values. In the base scheme a sequence is generated by concatenating all the individually generated tests to the specified length.

The OTCS is simulated using a C program of 140 lines of code and 59 branches. It follows the Z specification completely.

4.2.3 Results

The graphs show the the percentage of code coverage or block coverage achieved vs the number of tests. The results for branch coverage were also measured but matched closely with block coverage. To simplify the display of results the graphs focus only on block coverage.

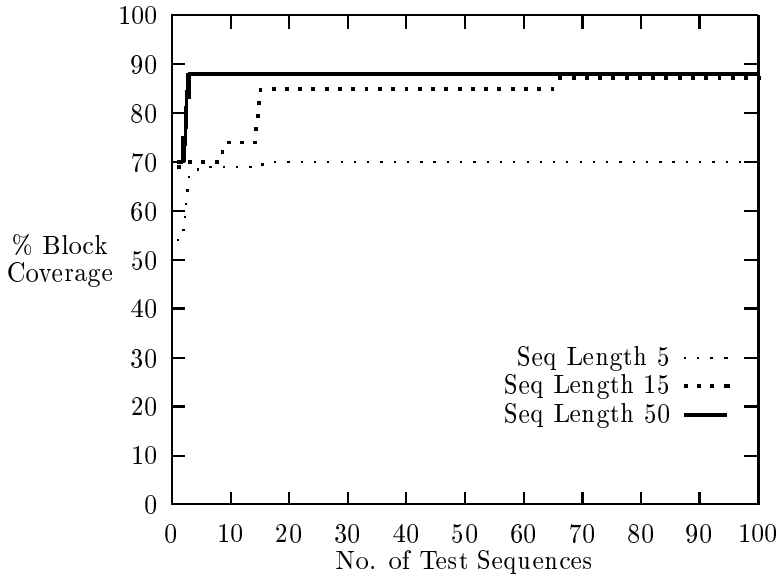


Figure 4.3: OTCS Random Testing

In the random testing method (Figure 4.3) sequences are generated with *TankerArrive*,

TankerLeave and *Query* operations having equal probabilities of occurring. No incorrect or exceptional cases are used. The random sequences generated for both the schemes are similar and the results are approximately the same. With the sequence length length 5 both schemes perform poorly. They reach the peak early and do not improve. What is observed is as the length of the sequence increases the coverage increases.

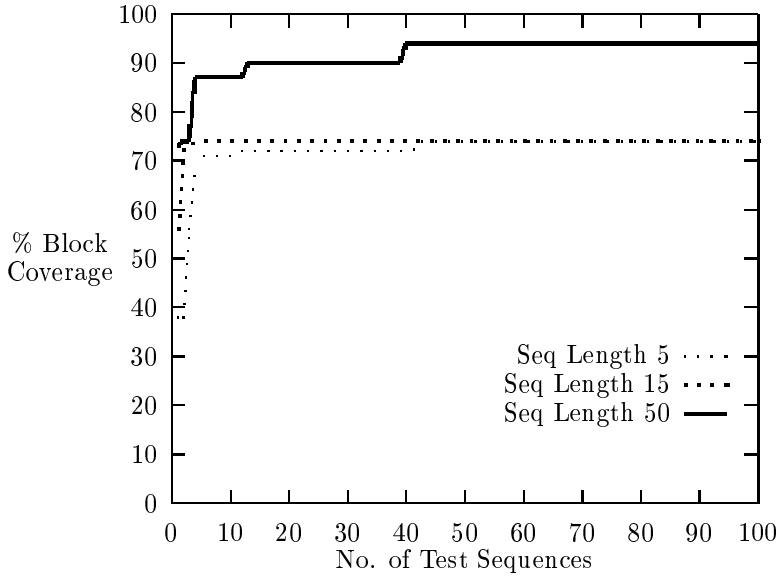


Figure 4.4: OTCS Checkpoint Encoded Random Testing : Base encoding scheme

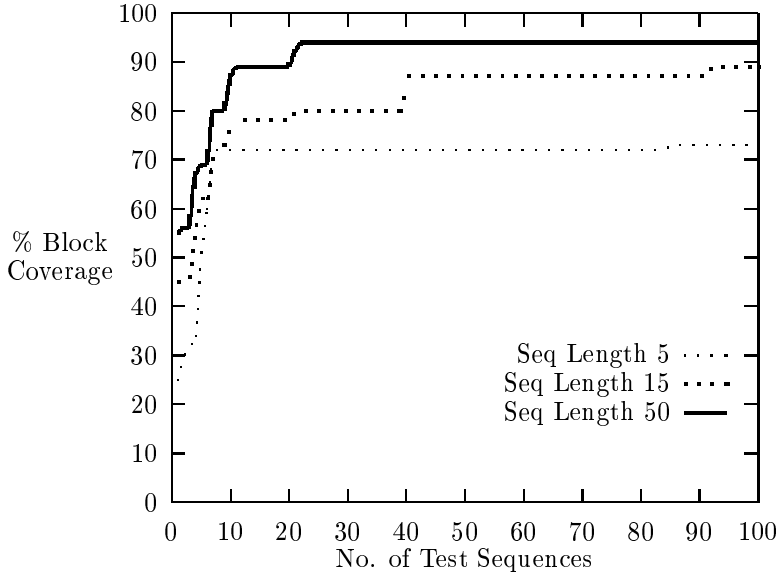


Figure 4.5: OTCS Checkpoint Encoded Random Testing : Sequence encoding scheme

The checkpoint encoded random testing(Figures 4.4 & 4.5) uses both schemes. For the sequence length of 5 there is no difference in the coverage. Both schemes with a sequence length of 5 perform

poorly. The obvious conclusion being that the sequence length is not enough. For a length of 15 the base scheme performs the same as with length 5. Again it would seem that the length is not enough. However there is a dramatic improvement in the sequence encoded scheme. For the length of 50 also there is little difference between both schemes. The sequence encoded scheme is effective with fewer number of test sequences, though the final result is nearly the same. Except for the base scheme with sequence length 5 checkpoint encoding in general has better coverage than the random testing scheme. Comparable results are also achieved earlier.

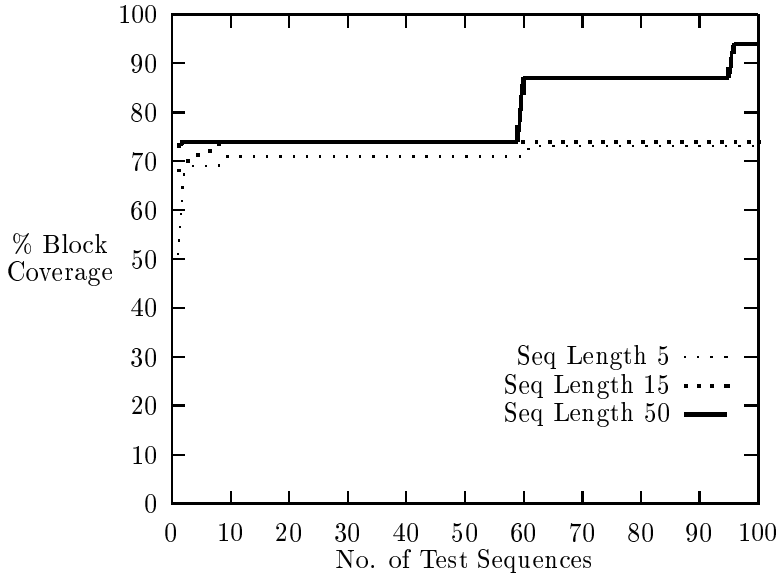


Figure 4.6: OTCS Anti-Random Testing : Base encoding scheme

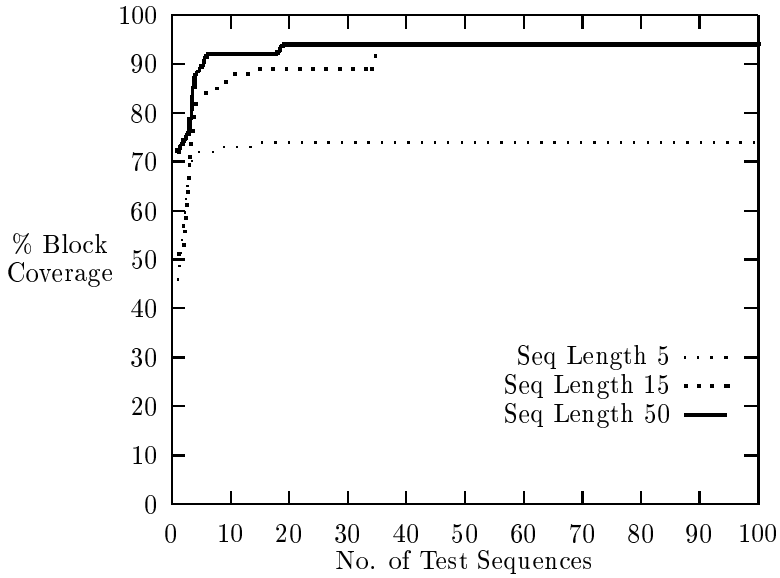


Figure 4.7: OTCS Anti-Random Testing : Sequence encoding scheme

The results for anti-random testing (Figures 4.6 & 4.7) show a similar trend to that of checkpoint encoded random testing. The only difference between the coverage results of both encoding schemes is observable with the sequence length of 15. Again the sequence encoded scheme is effective with fewer number of tests.

The results in numerical form are as follows.

Table 4.3: Comparison of base and sequence checkpoint encoding schemes

Testing Method	Seq Length 5				Seq Length 15				Seq Length 50			
	Base		Sequence		Base		Sequence		Base		Sequence	
Random	70	15	-	-	87	66	-	-	88	3	-	-
Checkpoint Encoded	74	42	73	85	74	4	89	92	94	40	94	22
Anti-random	73	61	74	15	74	8	94	35	94	96	94	18

Consider random testing. The use of a sequence even in the base encoding method does make a difference. It is not just the overall number of tests but the length of the sequence that affects the coverage. Consider in the case of sequence length 5. 15 sequences yield 75 individual OTCS operations (i.e. tests) and a maximum coverage of 70%. For the sequence length 15 70% coverage is achieved in 2 sequences (30 operations). The overall coverage shows it too with 70% for sequence length 5 and 87% for sequence length 15. It would seem that the coverage maximizes at this point since increasing the sequence length to 50 does not improve the coverage but improves the efficiency dramatically reaching it within 3 sequences (150 operations) vs 66 operations for sequence length 15 (990 operations). The reason why the coverage does not improve too much is that each individual operation is generated randomly so some exception handling code may not get covered easily or not at all.

In checkpoint encoded random testing the importance of the length of the sequence is emphasized. With length 5 there is no difference between the base and sequence encoded techniques. When the length is 15 the both techniques achieve the previous coverage of approximately 74% in about 4 sequences. But the sequence encoded technique achieves higher coverage which the base technique is unable to achieve. This is because with length 15 it possible to sample the right subsequences within the sequence, not possible in a sequence of length 5. Again with length 50 both techniques achieve the same coverage but the sequence encoded technique is slightly better since it covers quicker than the base technique.

Anti-random testing has similar observations as the checkpoint encoded random test results. The sequence encoded technique however is clearly better this time. With a sequence length of 15 both techniques achieve 74% coverage in 8 sequences but the sequence encoded technique achieves the highest coverage in 35 tests.

The conclusions drawn are

- The length of the sequence is important. Even for random testing 100 sequences of 5 input operations results in far less coverage than 10 sequences of 50 input operations. Given a sequence of enough length (10 times the depth of the state machine at 5) many if not all states will get covered.
- There is only minor difference between the results for checkpoint encoded random and anti-random testing. Anti-random testing gives better results but the checkpoint encoding scheme does not make use of the properties of anti-random testing. One reason could be that the program under test has a very simple state structure.
- The checkpoint encoding of subsequences is effective. It could result in shorter test suites. Brute force testing using sequences of large length and lots of test cases could achieve the same results using even random test generation techniques. That is not feasible for complex state machines. The results indicates that *sequence checkpoint encoded test generation gives higher coverage, with fewer sequences of shorter sequence length.*

4.3 Summary

The first step in this chapter is a demonstration that a FSM can be extracted from a Z specification. This has proven to be cumbersome but as a proof of concept it serves it's purpose. If model-based specifications become the preferred specification technique for simple general purpose software systems, they could be used to extract the FSM for that system. This can be done with little effort, especially considering that the previous steps are used to extract input partitions. If other specification techniques are used that are better suited to specifying the state of a system then the process of extracting the FSM should be more efficient.

The need to use the state of the system to generate tests is not questioned. But the question that this thesis attempts to answer is : what is the adequacy criteria that should be used? Consequently what methods should be used for test selection? This thesis proposes that we can use structural coverage criteria i.e. state or transition coverage. This indicates that a simple criterion can be used which applies to every kind of software, and that test selection based on the FSM is an important technique to satisfying traditional criteria.

The method that is proposed is to reinterpret checkpoint encoding for state. A simple method is to obtain the transfer sequences from the FSM and encode it into a larger scheme using position of inputs and the depth of the state machine. This scheme is at best an intermediate one. The final

goal is to be able to carry out checkpoint encoding on the state machine itself. Currently while state coverage criteria are used to generate the sbsequences the results are measured using code coverage criteria.

Sometimes it may not be possible to derive states or partitions of the state space directly from predicates in the schemas. This is where Murray et al (Murray, Carrington, MacColl, MacDonald, and Strooper 1998) is more capable. One aspect of this problem is looking at before and after states. Thus more work needs to be done on extracting the state that is "inherent" in a predicate i.e. if a predicate describes a operation even if the after state is not stated explicitly we should have some way of deriving what states the operation could cause. However even a approximate representation of the finite state machine enables sequencing of test cases and can improve test effectiveness for some systems. The simple checkpoint encoding approach proposed can be very useful in such cases.

Chapter 5

Effectiveness of Checkpoint Encoding Schemes

5.1 Background

Yin (Yin, Lebne-Dengel, and Malaiya 1997) makes the case for further experimentation into the effectiveness of different distributions for checkpoint encodings. In each checkpoint encoding scheme there are two questions that must be solved. These are the same issues that arise during partition testing. The first is the *partitioning* scheme; how should the input domain be partitioned into subdomains. The second is the *test case allocation* scheme; how to allocate the number of test cases to be selected from the subdomains. How to partition the input domain is well researched (Beizer 1990; Chan, Chen, and Tse 1997; Weyukar and Ostrand 1980; Weyukar and Jeng 1991; Richardson and Clarke 1985). In this document the relevant segments are chapters 3 and 4.

Test allocation schemes have been the subject of few studies. The earliest was studies by Duran and Ntafos(Duran and Ntafos 1984) and Hamlet and Taylor (Hamlet and Taylor 1990). Both these used similar experiments and concluded that there was only a marginal difference in effectiveness between the two methods. With partitioning costs being high for some programs random testing might be more cost-effective. Weyukar and Jeng (Weyukar and Jeng 1991) used the "P-measure" as an effectiveness metric for testing. It is defined as the probability of detecting at least one failure. They revealed that if all subdomains were of equal size and the same number of test cases were selected from each subdomain then the P-measure of partition testing is never worse than random testing. Based on the same model Chen and Yu (Chen and Yu 1994; Chen and Yu 1996a; Chen and Yu 1996b) generalized the results of Weyukar and Jeng. They found several conditions when partition testing has an equal or higher P-measure than random testing, one of which is the proportional sampling strategy, where the number of test cases selected from each subdomain is proportional to the size of the subdomain. They also defined another measure called an "E-measure",

defined as the expected number of failures detected.

Chan et al (Chan, Chen, and Tse 1997) makes a comparison of test allocation schemes rather than between partition and random testing. They prove that if a partial ordering of subdomains (or partitions) based on the failure rates can be calculated, and test cases are allocated mostly to those subdomains with a higher failure rate then such a test allocation scheme has a better chance of detecting failures based on either the P-measure or the E-measure. Two problems arise : how can these failure rates be calculated, and what prevents allocation of all test cases to the subdomain with the highest failure rate to get the highest value of the P-measure/E-measure. Failure rates cannot easily calculated. However the authors suggest a heuristic: partition subdomains such that they correspond to different functions of the software system. Then if a particular function is very complex then it is reasonable to assume that it would have a higher failure rate. The second problem is not really relevant since it violates a fundamental assumption that all subdomains should be sampled at least once. Also this may not find all program faults. Lastly the P-masure/E-measure may not be the adequacy criterion of choice when partition testing.

In this chapter this thesis examines the problem of test allocation in checkpoint encoding. The difference between how this thesis attempts to compare test allocation schemes and other work in the area of partition testing is an important one. The test adequacy criterion used is structural coverage with an emphasis on branch coverage. Unlike the other work, there is no use of failure detection based metrics. The notion of detecting failures is addressed by considering the fact that there is a need to cover hard to detect (or hard to cover) branches. Since they are less explored there is a probability of failures lying undetected that are caused by those branches. This is discussed in more detail in the next section.

5.2 Understanding Effectiveness of Checkpoint Encoding

Most experimental results of testing methods including anti-random testing focus on a single final goal. This goal can take a variety of forms; for structural coverage it is coverage of as many of the programs artifacts under examination as is possible. For example: branch coverage criteria require close to 100 % branch coverage with the least amount of test cases. Thus a method is deemed to be effective relative to another when the final coverage is higher taking into account the number of test cases required. What is needed are better measures to understand how a testing method covers branches. Based on measures proposed an experiment can then be set to compare effectiveness of different checkpoint encoding schemes.

5.2.1 Relation between Testability and Coverage

There are different ways of looking at the branch coverage. 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. These are termed as *hard to detect* branches. In this section, we explain how covering the hard to detect branches improves the testability of software programs.

The IEEE Standard Glossary of Software Engineering Terminology (1990) (Committee 1990) defines testability as

"(1) the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met,
and (2) the degree to which a requirement is stated in terms that permit establishment of test criteria and performance of tests to determine whether those criteria have been met".

Consider the IEEE standard definition, testability of software is a measure of how easy is it to satisfy some criterion for a piece of software. For the branch coverage criterion, considering the IEEE definition, covering hard to detect branches is *improving* the testability of the software. Thus testability of software is improved with respect to a particular test data generation strategy.

Voas and Miller (Voas and Miller 1995) 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.* Voas' definition of software testability is different from the one suggested by the IEEE Standard. The IEEE Standard definition has it's flaws in that satisfying a criterion does not mean that faults are detected which is the purpose of testing. The definition proposed by Voas is a better definition. Even this has it's limitations as pointed out by Bertolina and Strigini (Bertolina and Strigini 1995), who extend Voas' definition to correct the deficiencies. However the two definitions can be considered to be sufficiently similar for the purpose of this thesis.

While testability of code is a complex issue, it is usually considered as testability of a module as a whole. Voas et al (Voas 1992) give a method to calculate the testability of a module based on the PIE model. Further, Voas et al (Voas and Miller 1992) define a metric, the *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 calculation is also done 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(Howden and Huang 1995). 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 a change in the output.

Probabilities for each stage have to 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. Executing program locations more often 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 a modified definition of the branch coverage criterion coud be defined as '*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. In this chapter the terms hard to detect, hard to test and hard to cover have the same meaning.

5.2.2 Proposed Branch Coverage Measures

Using coverage tools (Horgan and London 1992) data on which branches were covered and how many times is collected after testing a program. The coverage criteria used is *decision coverage* i.e. each predicate in a conditional statement that controls the branch has to be exercised as true and false. This is a stricter coverage criteria than branch coverage. Each predicate or decision is treated as a branch and has both true and false values or paths. There is no other difference between the two criteria in terms of the data collected. However throughout we use the term branch coverage rather than decision coverage and branches instead of decisions for ease of understanding.

The goal is to define a measure that shows how well a test or set of tests exercises hard to test branches. But there is no clear definition of what is a hard to test branch. Thus instead of coming up a with a single numeric measure what is proposed is a graphical measure showsing the distribution of branches covered. If a program was tested with a total of N test vectors then a coverage tool can extract a list of which branches were covered $0, 1, ...N$ times. From this is extracted three different

measures: the number of branches *exactly covered* n times(ECN), the number of branches *not covered* n times(NCN), and the number of branches *at least covered* n times(LCN) where $n = 0, 1, ...N$. Each measure is plotted as a bar graph showing the number of branches on the Y axis v/s the number of tests on the X axis.

The measure, *exactly covered* n times (ECN), clearly displays the number of branches that are covered exactly n times. It shows precisely how easy was it for the tests to cover the branches in a program. If most of the taller bars occur at lower values of n then most of the branches were each covered by few tests out of the total N tests. If most of the taller bars are clustered at higher values of n, then most of the branches were each covered by most of the tests out of the total N tests. Uniform height of the bars means that the distribution of branches covered amongst the tests varies widely with no clear trend. The number of branches covered exactly 0 times gives the number of uncovered branches i.e. they were not covered at all. The branches covered exactly n times for smaller values of n are the hard to cover branches.

The measure, *not covered* n times (NCN), indicates the number of branches in the program not covered. This is a measure of how easy was it to cover the branches. Since it is a cumulative display i.e. the number of branches not covered cannot decrease as n increases, it resembles a histogram. If the histogram is like a plateau it indicates that most of the branches are easily covered. Most of the branches have been covered by nearly every test case used. The height of the histogram would determine how difficult it was to cover the branches. If the histogram peaks at the tail (i.e. higher values of n) then it indicates that there are some new branches have been covered by every test case. This is the normal situation indicating that a small set of branches, represented by the increase in the tail, have been covered by many test cases. These are the easy to test branches. The branches not covered n times for smaller values of n are the hard to detect or hard to cover branches. The branches not covered 0 times give the total number of branches covered (i.e. covered at least once). The branches not covered once ($n = 1$) give the number of uncovered branches.

The ECN gives a better idea of which branches are easy to cover i.e. branches exactly covered n times where n is close to N (the total number of tests). It gives more precise values to the number of hard to cover branches. The NCN measure is more visually informative, giving a better understanding of the overall behaviour of the program. The LCN measure is the similar to the NCN measure but with a different perspective.

5.2.3 Design of Experiment

In Yin's thesis (Yin, Lebne-Dengel, and Malaiya 1997) an experiment is carried out to compare the effectiveness of anti-random testing. Three testing methods are used.

1. Pseudo-random testing (PR) is used as a base method for comparison. This is the "standard" random test data generation method. The program's specification is examined manually and ranges set for the generation of program input variables.

2. Checkpoint Encoded Random (CER) test data generation which is intermediate between random and anti-random testing. A checkpoint encoding scheme is proposed based on the specifications. Binary vectors are randomly generated. They are then translated into actual input values (test cases) using the checkpoint encoding scheme defined.

3. Anti-random (AR) testing generates the binary vectors using a specific method (Yin, Lebne-Dengel, and Malaiya 1997) to ensure that they are well distributed throughout the input domain. These are then translated into the actual test cases using the checkpoint encoding scheme.

The programs to be tested are instrumented by compiling with a code coverage tool ATAC (Horgan and London 1992). The test cases generated are fed to the instrumented programs and structural coverage is measured.

The sample programs used in the testing experiment have been used by (Wong 1993; Yin, Lebne-Dengel, and Malaiya 1997) to investigate test coverage issues.

- The FINDNO program. This program (see Appendix A) takes an integer array B of size $S \geq 1$ and index F. The program sorts the array elements 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 $1 \leq F \leq S$.

- The STRMAT program. he 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.

- The TRIANGLE program. This triangle example (see Appendix A) is used by Jorgenson (Jorgensen 1995) and (North 1990). Demillo (Demillo and Offutt 1993) 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".

The complete code listing for these programs are in Appendix A **??**.

Program	Number of lines	Number of branches
FINDNO	95	26
STRMAT	99	20
TRIANGLE	90	20

Table 5.1: Some details of the programs used in the experiment.

For each program and each method structural coverage metrics including data flow metrics were measured. Briefly the conclusions drawn were as follows; anti-random testing gives better overall coverage. This ability is clearly demonstrated for some programs but not all. The checkpoint encoded random testing is generally better than random or pseudo-random testing. However, the coverage for random testing varies (due to seed used) and occasionally is better than checkpoint encoded random testing.

This experiment is duplicated as a first step. This is referred to as the **Standard testing scheme**. Essentially what is duplicated is the testing methodology and the checkpoint encoding schemes used. The pseudo random generation methods used are the same as in the original experiment. However this is where the similarity ends. The current experiment uses the same 3 programs and the same 3 methods and focuses only on branch coverage. The ECM measure is calulated and plotted with varying test suite sizes (N). These three methods are examined using the new measure. One of the most important differences is that hard to test branches are closely examined in this experiment.

In this thesis hard to cover branches are defined as those branches that are covered by at most $1/3^{rd}$ or approximately 30 % of the total number of test cases. If this condition is not satisfactory (either too many or too few branches) then approximately 25 % of the branches that are covered by the fewest number of tests are selected. There no reason other than conventional wisdom for such a loose definition. Also there are no conventions that can be followed in this aspect. Hard to cover branches, as a default, are defined for random testing since these branches are hard to cover because of the program's properties rather than being influenced by the testing method.

The second step in the experiment is to experiment with different checkpoint encoding schemes. To simplify the experiment two different checkpoint encoing schemes were selected. The first scheme, termed the **Legal testing scheme**, encodes partitions of the input domain that are legal. The

second which is conversely termed the **Illegal testing scheme** focuses on partitions that encompass incorrect or illegal input subdomains i.e. these will not work if the program is correct as per the specifications. Also boundary cases i.e. those elements of the input domain that form the boundary between the acceptable and unacceptable regions of the input domain are also encoded. The first step is repeated using all 3 programs but only the latter 2 methods since they use the new checkpoint encoding schemes. Otherwise nothing else changes. The ECN measure is calculated for the second step.

Branches (or predicates) are examined closely for all three testing methods. While the measure is based on the number of tests and shows program level behaviour each branch is measured for the number of times it is covered. Lastly the findings are summarized and compared numerically.

Each branch is displayed with the predicates highlighted differently indicating that it's **true**, *false* or *both* branches were not covered. Both branches not being covered occurs sometimes and is more a due to branches nested above it not being covered rather than any inherent testability.

Rationale of the Design of the Experiment.

The programs are selected for a couple of reasons. It was used in the original experiment so the previous checkpoint encoding schemes can be used. They are standard exmaples from literature and display different behaviour i.e. have different input domains and different program logic.

The ECN measure is used as it displays in a single graph the testability behaviour of the branches of the program. The branch is the artifact selected for examination since the testability of the code is strongly linked to the branches in the program. They influence the control flow of the program. Thus branches provide a level of abstraction while allowing the observations made to be equally applicable to the program as a whole.

The first step provides a base set of findings. Behaviour of the program is understood (from the random testing results) and also we see how the Standard testing scheme performs in both CER and AR methods. Then the performance of the two other schemes are examined both for differences between the schemes as well as how the two testing methods; checkpoint encoded random and anti-random fare in these new schemes. The two schemes represent opposite ends of the checkpoint encoding spectrum. We hope to gain insight into what makes a good checkpoint encoding scheme. Possible the reasons behind it may also become clear. One way of finding the reasons is to look at each branch more carefully as is done at the end of each method. There can be many variations in the checkpoint encoding process but by looking at the two extremes it could point the way for further experiments.

5.2.4 Results of Standard Checkpoint Encoding

5.2.4.1 FINDNO Program

The FINDNO program is inherently testable with a high coverage being obtained with few test cases. It is tested in exactly the same manner as in (Yin, Lebne-Dengel, and Malaiya 1997). The checkpoint encoding used is also the same. The branch coverage comparing the coverage obtained by all three methods is shown in 5.1. The number of tests used is decided based on this graph.

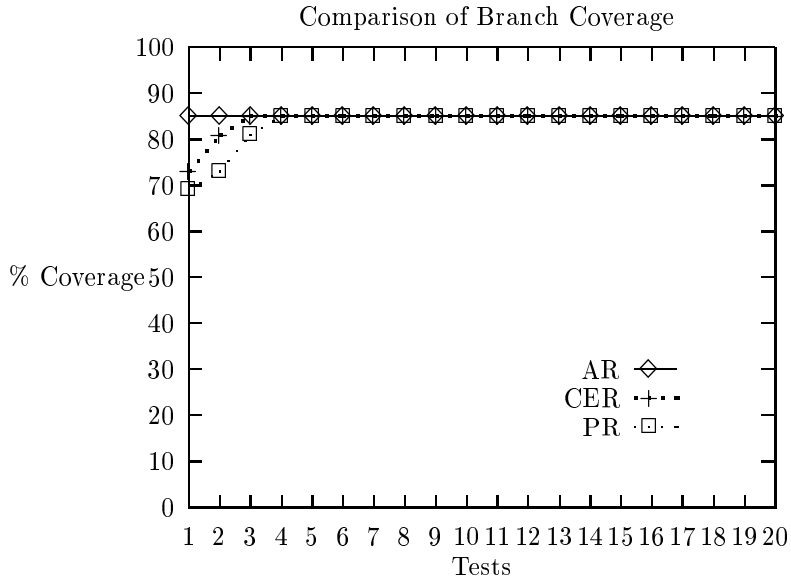


Figure 5.1: FINDNO program

The ECN measure is calculated for this program for 5 and 15 tests. For the checkpoint encoding scheme used in checkpoint encoded random and anti-random testing the standard scheme (Yin, Lebne-Dengel, and Malaiya 1997) is used. The input parameters are the size of the array(N), the index of the element pointed to (F) and the array values. Array Size : $1 \leq N \leq 64$; Index :$1 \leq F \leq N$; Element Values : $-256 \leq V \leq 255$.

1. Random Testing.

   The input parameters; the size of the array, the index of the element pointed to and the array values itself are generated randomly within the allowable ranges.

   The ECN graphs 5.2 shows clearly that the FINDNO program is testable. Almost all the branches are clustered towards the end of the graph i.e. they are covered by almost all the tests. With the increasing number of tests the shape of the plot using the exact metric remains

Table 5.2: Standard checkpoint encoding scheme for FINDNO.

Field	Bits	Value	Significance
Array Size	b1,b0	01	1,2
		rest	$2 \cdots 64$
Array Status	b4,b3,b2	110	sorted order
		100	reverse order
		011	all equal
		rest	random order
Element Values	b7,b6,b5	010	all positive
		101	all negative
		rest	mixed
Index	b9,b8	10	first element
		01	last element
		rest	middle element

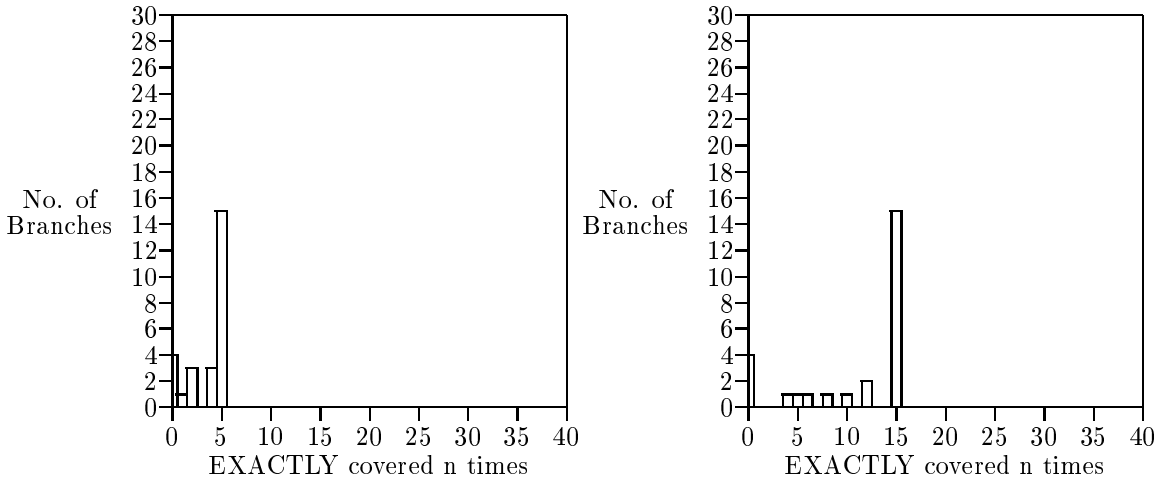


Figure 5.2: FINDNO program : Random Testing: 5 & 15 Tests

the same. Aside from 4 branches that are not covered most of the branches are covered by a large number of tests.

Table 5.3: Coverage of each branch in FINDNO using random testing.

Line No.	Branch	5 Tests		15 Tests	
		True	False	True	False
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
24	`for(i=1;i ≤ n;i++)`	5	5	15	15
31	`if (x == z)`	5	0	15	0
43	`for(i=1;i ≤ n;i++)`	5	-	15	-
56	`while ((m < ns) || b)`	5	5	15	15
56	`while ((m < ns) || b)`	0	-	0	-
58	`if (!b)`	5	5	15	15
65	`if (i >j)`	5	5	15	15
67	`if (f > j)`	4	2	12	6
69	`if (i > f)`	4	1	10	4
79	`while (a[i] < a[f])`	2	5	8	15
81	`while (a[f] < a[j])`	4	5	12	15
83	`if (i ≤ j)`	5	2	15	5

The hard to cover branches identified are

- `if ((f ≤ 0) || (f > n))`
- `if ((f ≤ 0) || (f > n))`
- `if` *(x == z)*
- `while ((m < ns) || b)`
- `if` *(i > f)*

The first 4 branches were not covered at all. By examining the program code it can be seen that the first 3 branches which were not covered were conditions that checked for exceptions. The fourth branch was not covered due to the logic of the program and also since in condition joined by a logical OR the later predicates are not always checked if any of the prior predicates is true. The last branch is the false branch of the predicate which occurs at high level of nesting. This affects how many times it is covered.

Thus most of the branches are easy to cover as is shown by the coverage results of the random testing.

2. Checkpoint Encoded Random Testing.

In checkpoint encoded random testing, binary vectors are generated randomly and translated into actual test cases using the standard checkpoint encoding scheme 5.2. The branch coverage graph 5.1 shows that checkpoint encoded random testing exploits the testability properties of the FINDNO program. It hits the maximum possible coverage within a few tests.

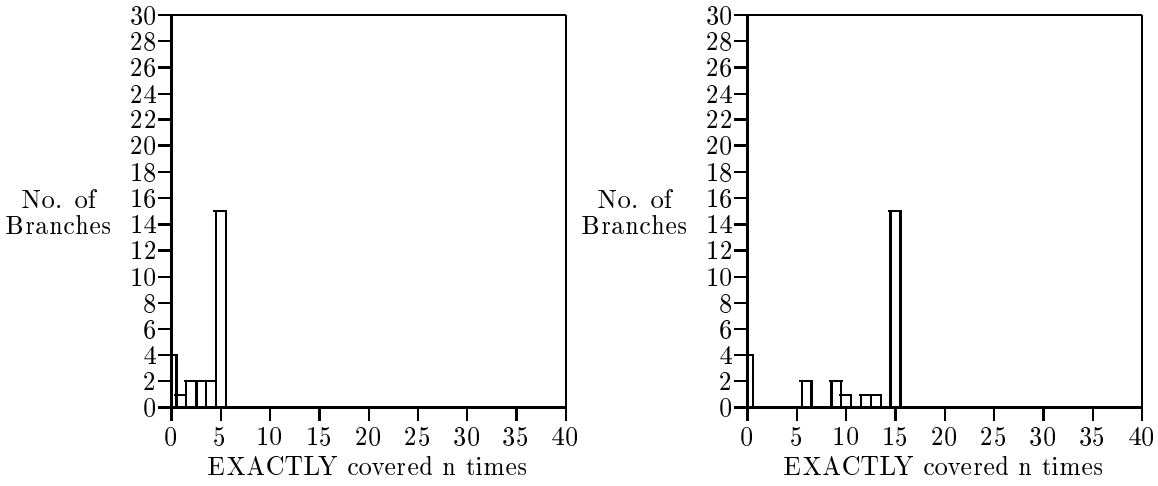


Figure 5.3: FINDNO program : Checkpoint Encoded Random Testing : 5 & 15 Tests

The graphs show the same shape as with random testing but the tail of the graph is emphasized. More branches are covered a large number of times i.e. more branches are covered by nearly every test case. However unlike random testing there seems to be a clear division between the frequectly covered and rarely covered branches. This is due to the checkpoint encoding scheme. Depending on the partitions and the allocation of test cases, it could get biased towards certain branches more than others. This does not occur in random testing.

The hard to cover branches with checkpoint encoded random testing identified are

- `if ((f ≤ 0) || (f > n))`
- `if ((f ≤ 0) || (f > n))`
- `if` *(x == z)*
- `while ((m < ns) || b)`

These 4 branches were not covered. This means that the checkpoint encoding scheme was not effective. This is the same branches not covered in random testing. They are exception conditions and therefore take special effort to cover. During normal operations they would not be triggered at all. Thus the normal operational profile would also neglect to test these. The checkpoint encoding makes no provisions to cover these conditions.

Table 5.4: Coverage of each branch in FINDNO using checkpoint encoded random testing.

Line No.	Branch	5 Tests		15 Tests	
		True	False	True	False
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
24	`for(i=1;i ≤ n;i++)`	5	5	15	15
31	`if (x == z)`	5	0	15	0
43	`for(i=1;i ≤ n;i++)`	5	-	15	-
56	`while ((m < ns) || b)`	5	5	15	15
56	`while ((m < ns) || b)`	0	-	0	-
58	`if (!b)`	5	5	15	15
65	`if (i >j)`	5	5	15	15
67	`if (f > j)`	3	2	10	9
69	`if (i > f)`	1	2	6	6
79	`while (a[i] < a[f])`	3	5	9	15
81	`while (a[f] < a[j])`	4	5	13	15
83	`if (i ≤ j)`	5	4	15	12

Thus the checkpoint encoded random testing does not offer any advantages over that of random testing. For easily testable programs this is to be expected.

3. Anti-random Testing

Anti-random testing has the same results as the other testing methods 5.1. It is slightly quicker to get to its maximum coverage.

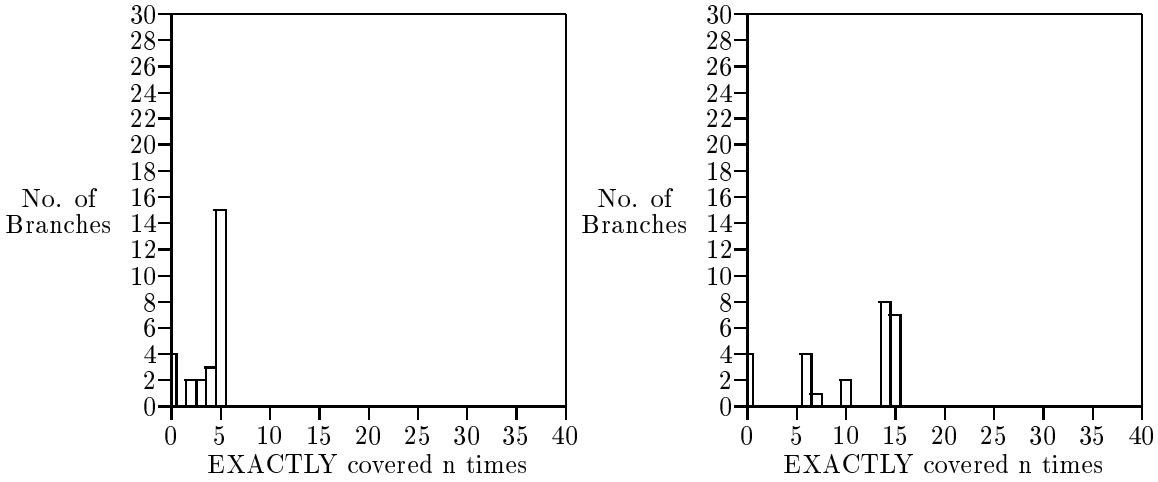


Figure 5.4: FINDNO program : Anti Random Testing : 5 & 15 Tests

The ECN measure shows that anti-random testing differs from checkpoint encoded random testing in only one crucial aspect; the distribution of branches is a little more even. There not as much of a division between the most covered and least covered branches as in checkpoint

encoded random testing.

Table 5.5: Coverage of each branch in FINDNO using anti-random testing.

Line No.	Branch	5 Tests		15 Tests	
		True	False	True	False
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
24	`for(i=1;i ≤ n;i++)`	5	5	15	15
31	`if (x == z)`	5	0	15	0
43	`for(i=1;i ≤ n;i++)`	5	-	15	-
56	`while ((m < ns) || b)`	5	5	14	15
56	`while ((m < ns) || b)`	0	-	0	-
58	`if (!b)`	5	5	14	14
65	`if (i >j)`	5	5	14	14
67	`if (f > j)`	4	3	10	6
69	`if (i > f)`	4	2	7	6
79	`while (a[i] < a[f])`	3	5	6	14
81	`while (a[f] < a[j])`	4	5	10	14
83	`if (i ≤ j)`	5	2	14	6

Hard to cover branches with anti-random testing identified are

- `if ((f ≤ 0) || (f > n))`
- `if ((f ≤ 0) || (f > n))`
- `if` *(x == z)*
- `while ((m < ns) || b)`

The branches which remain uncovered in CER testing remains uncovered. This is to be expected since anti-random testing is limited by the checkpoint encoding scheme. However the test vectors generated have better properties. They focus on covering fewer branches unlike checkpoint encoding random testing when tests seemed to cover more easy to cover branches. Anti-random vectors are thus well distributed in the input domain. This confirms previous results (Yin, Lebne-Dengel, and Malaiya 1997) that anti-random testing is better than the other two methods.

5.2.4.2 STRMAT Program

The STRMAT program is again relatively testable with random testing yielding good coverage. The comparison of branch coverage of the three methods for the STRMAT program is shown 5.5.

The number of tests used in the ECN measure is 10 and 30 tests. The standard checkpoint encoding scheme is used 5.6. The input parameters are the text string and the pattern substring whose position in the text string is returned if found. Text length : $0 \leq T \leq 80$; Pattern position : 1,T,1 $\cdots$ T,outside; Pattern length : $0 \leq P \leq 3$.

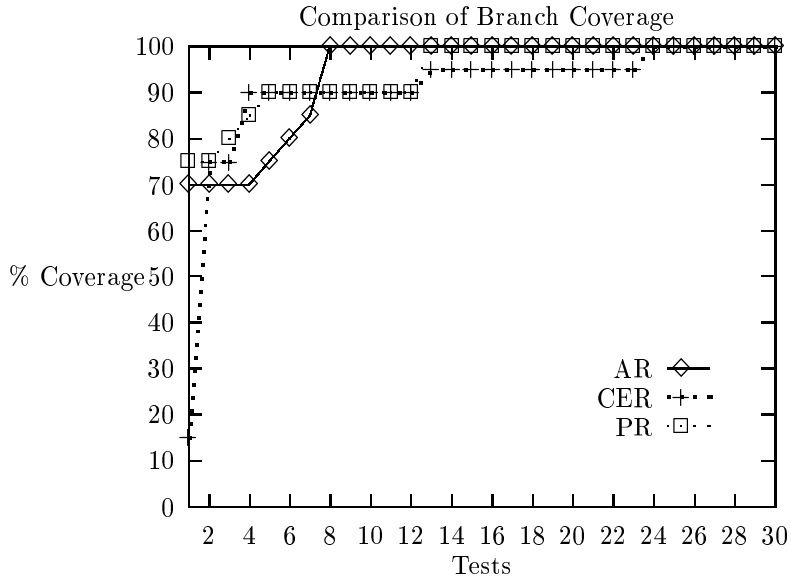


Figure 5.5: STRMAT program

Table 5.6: Standard checkpoint encoding scheme for STRMAT.

Field	Bits	Value	Significance
Text Length	b2,b1,b0	110	0
		010	80
		011	$80 < < 100$
		rest	$1 \cdots 79$
Pattern Position	b5,b4,b3	110	no pattern
		010	beginning
		011	end
		rest	middle
Pattern Length	b8,b7,b6	110	0
		010	3
		011	$3 < < 10$
		rest	1,2

1. Random Testing.

Random testing works well for STRMAT achieving 100% branch coverage. This inidcates that STRMAT should be an easily testable program.

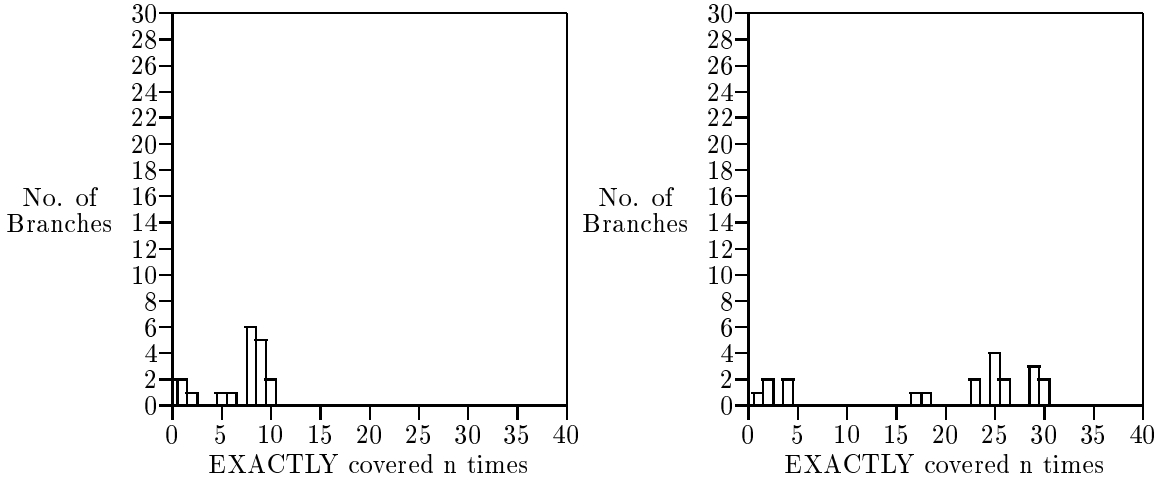


Figure 5.6: Strmat program : Random Testing : 10 & 30 Tests

As can be seen from figure 5.5 the STRMAT program is easily tested. It gives 100 % branch coverage for all methods within 30 test vectors. The ECN measure also reflects this. Most of the bars are clustered towards the end of the graph. Thus a large majority of the branches are covered by most of the tests with very few hard to cover branches. This behaviour is consistent even after increasing the number of tests.

Table 5.7: Coverage of each branch in STRMAT using random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
18	`while (c != '\n')`	9	10	29	30
20	`if (i ≤ tmax)`	9	2	29	4
39	`while (c != '\n')`	9	10	26	30
41	`if (i ≤ pmax)`	9	6	26	18
71	`if (textlen == 0)`	1	9	1	29
76	`if (patlen == 0)`	1	8	4	25
81	`while ( (patpos ≤ patlen) && (textpos ≤ textlen))`	8	0	25	2
81	`while ((patpos ≤ patlen) && (textpos ≤ textlen))`	8	8	25	23
83	`if (pattern[patpos] == text[textpos])`	5	8	17	25
95	`if (patpos > patlen)`	0	8	2	23

The hard to cover branches are listed below.

- `if (textlen == 0)`

- `while (`*(patpos <= patlen)* `&& (textpos <= textlen))`
- `if` **(patpos > patlen)**
- `if` **(patlen == 0)**
- `if` *(i<=tmax)*

For 10 tests the second and third branches were not covered. The other branches were covered by less than 1/3rd of the total number of tests. The first, fourth and fifth conditions are error checking conditions which explains why they are hard to cover. The second and third conditions are related. These branches would be triggered if a pattern match occurs. However in pseudo-random testing where both the text string and the pattern are randomly generated, the chances of a match occuring is minimal. Thus it is rarely covered.

2. Checkpoint Encoded Random Testing.

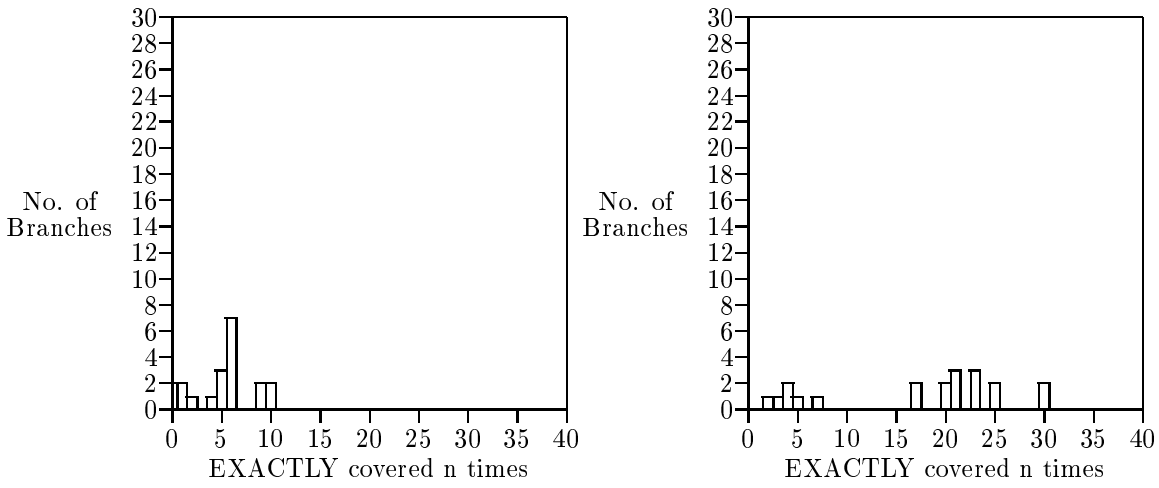


Figure 5.7: Strmat program : Checkpoint Encoded Random Testing : 10 & 30 Tests

The ECN measure shows a minor change with respect to that shown with random testing. While hard to cover branches indicated by random testing become easier to cover the overall coverage middles out. This is to be expected since checkpoint encoding forces tests to be distributed over the entire input domain. Fewer branches get covered by all (or almost all) of the tests. Even so the ECN measure shows that this distribution of coverage of the branches is not as "fair" as it should be, with the measure being weighted towards the total number of tests. THis could be because the checkpoint encoding scheme is not as good as it should be but this is actually influenced by the program behaviour.

Table 5.8: Coverage of each branch in STRMAT using checkpoint encoded random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
18	`while (c != '\n')`	6	10	23	30
20	`if (i ≤ tmax)`	6	0	23	3
39	`while (c != '\n')`	9	10	25	30
41	`if (i ≤ pmax)`	9	2	25	5
71	`if (textlen == 0)`	4	6	7	23
76	`if (patlen == 0)`	0	6	2	21
81	`while ( (patpos ≤ patlen) && (textpos ≤ textlen))`	6	5	21	17
81	`while ((patpos ≤ patlen) && (textpos ≤ textlen))`	6	1	21	4
83	`if (pattern[patpos] == text[textpos])`	5	6	20	20
95	`if (patpos > patlen)`	5	1	17	4

Hard to test branches with checkpoint encoded random testing are listed below.

- `if (patlen == 0)`
- `if` *(i<=tmax)*
- `while ( (patpos <= patlen) &&` *(textpos <= textlen)*)
- `if` *(patpos > patlen)*
- `if (textlen == 0)`
- `if` *(i<=pmax)*

There is a difference as compared to the hard to cover branches from random testing. They are covered more often and there is a change in the order. The second and third branchers are the opposite of what was hard to cover in random testing. As explained the earlier branches were triggered if the pattern occurs in the text string. This is effectively tested using the checkpoint encoding scheme. These those branches get covered and the other case, that of the patteern *not* occuring in the text string is rarely tested. Also while text string length 0 is tested, pattern length 0 is not tested in the standard checkpoint encoding scheme. This explains the reordering with respect to the error checking branches.

3. Anti-random Testing.

The ECN measure for anti-random testing with 10 tests shows a division between hardly covered and more frequently covered branches. This is not at all encouraging but what is to be realized is that anti-random testing does not show good behaviour with fewer number of tests. As the number of tests increase the distribution of branches is much better. In line

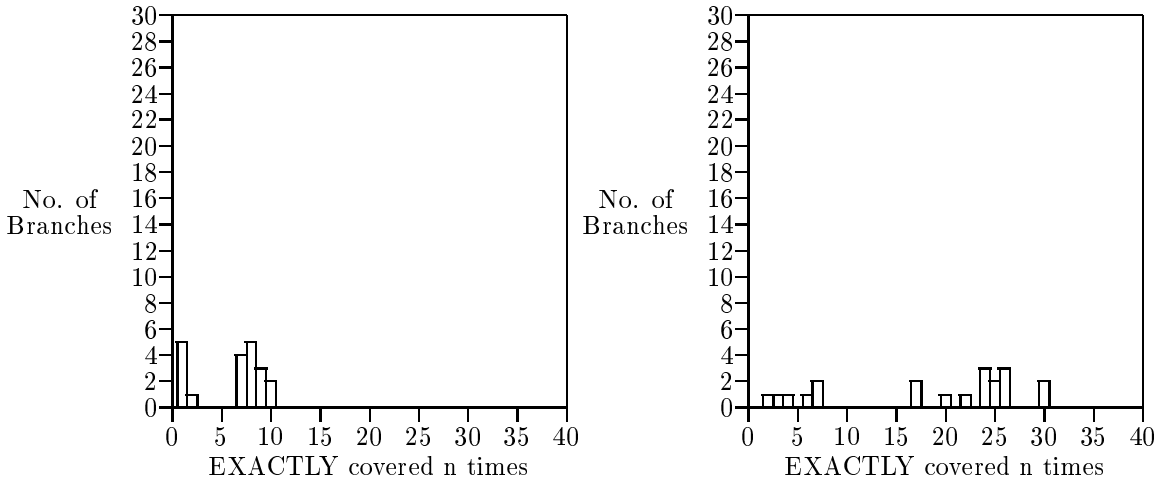


Figure 5.8: Strmat program : Anti Random Testing : 10 & 30 Tests

with expectations the distribution improves over that shown for 30 tests in checkpoint encoded random testing. It gets more uniform but only marginally so. The reason for the difficulty in improving the coverage distribution is assumed to be the program behaviour.

Table 5.9: Coverage of each branch in STRMAT using anti-random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
18	`while (c != '\n')`	9	10	26	30
20	`if (i ≤ tmax)`	9	1	26	3
39	`while (c != '\n')`	8	10	25	30
41	`if (i ≤ pmax)`	8	2	25	6
71	`if (textlen == 0)`	1	9	4	26
76	`if (patlen == 0)`	1	8	2	24
81	`while ( (patpos ≤ patlen) && (textpos ≤ textlen))`	8	7	24	17
81	`while ((patpos ≤ patlen) && (textpos ≤ textlen))`	8	1	24	7
83	`if (pattern[patpos] == text[textpos])`	7	7	20	22
95	`if (patpos > patlen)`	7	1	17	7

The hard to cover branches with anti-random testing are shown below.

- `if (patlen == 0)`
- `if` *(i<=tmax)*
- `if (textlen == 0)`
- `if` *(i<=pmax)*
- `while ( (patpos <= patlen) &&` *(textpos <= textlen)*)
- `if` *(patpos > patlen)*

The number of hard to test branches and the order of how many times each was covered is quite similar to checkpoint encoded random testing. They are covered a few more times however. One important difference is the fifth and sixth branches. These are covered more often than in checkpoint encoded random testing. This is because of the fact that the tests vectors are generated using anti-random methods improving the distribution of tests. So while the case of the pattern not occuring in the text substring is less tested as the case of the patern occuring in the text string, there is not such an imbalance beyond that enforced by the checkpoint encoding scheme. Thus the antirandom testing method is more faithful to the checkpoint encoding scheme, reflecting its inadequacies just as well.

5.2.4.3 TRIANGLE Program

The TRIANGLE program is not so testable as comapred to the previous two programs with random testing yielding poor coverage. The comparison of branch coverage of the three methods for the TRIANGLE program is shown 5.9.

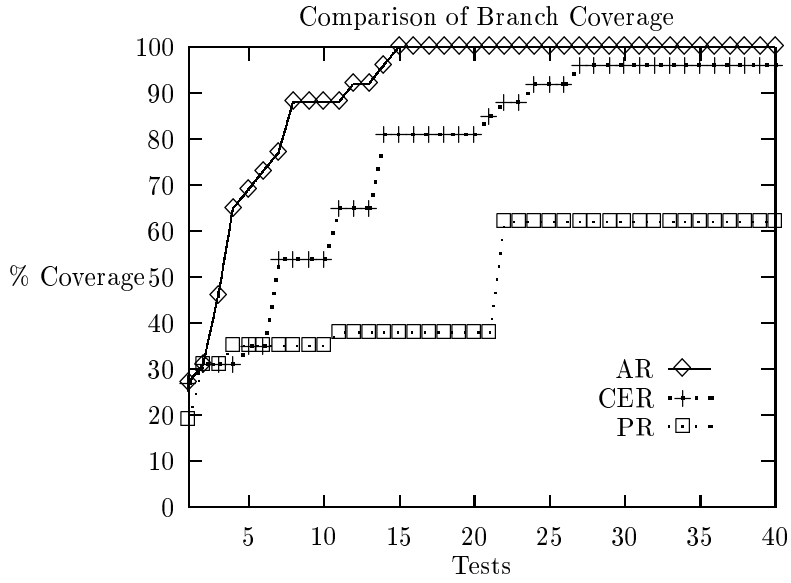


Figure 5.9: TRIANGLE program

Pseudo-random testing fares the worst of all three methods. This shows that the TRIANGLE program is hard to test. Anti-random testing performs clearly much better than the other methods. This is in contrast to the other programs where there does not seem to be a clear advantage to using the anti-random testing method. The number of tests used in the ECN measure is 10 and 30 tests. The standard checkpoint encoding scheme is used 5.4

1. Random Testing.

Table 5.10: Standard checkpoint encoding scheme for TRIANGLE.

Field	Bits	Value	Significance
Triangle Type	b4,b3,b2,b1,b0	11111	a+b $\leq$ c, a != b (Not a triangle)
		01111	a+b $\leq$ c, a = b
		11001	b+c $\leq$ a, b != c
		01001	b+c $\leq$ a, b = c
		00011	a+c $\leq$ b, a != c
		10011	a+c $\leq$ b, a = c
		00100	a+b = c, a != b
		10100	a+b = c, a = b
		00101	b+c = a, b != c
		10101	b+c = a, b = c
		01100	a+c = b, a != c
		11100	a+c = b, a = c
		01010	a = b (Legal triangle)
		11010	a = c
		00110	b = c
		10110	a = b = c
		rest	scalene

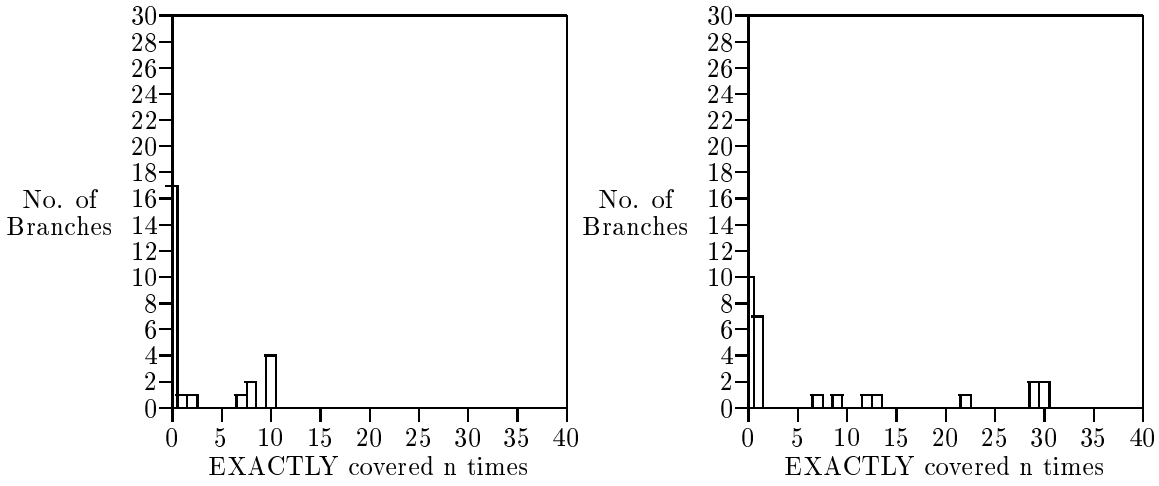


Figure 5.10: Triangle program : Random Testing : 10 & 30 Tests

The ECN measure shows how poorly the pseudo-random testing method performs. Almost all the branches are not covered or covered just once. The few branches that are covered are covered by almost all the tests. This imbalance is a little too stark and is better explained by analysing the hard to cover branches in the program.

The hard to cover branches with random testing is shown below

- `if (a==b)`
- `if (a==c)`

Table 5.11: Coverage of each branch in TRIANGLE using random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
13	`if (a == b)`	0	10	0	30
15	`if (a == c)`	0	10	0	30
17	`if (b == c)`	0	10	1	29
23	`if (match == 0)`	10	0	29	1
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	2	8	7	22
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	8	9	13
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	1	7	1	12
30	`else if (match == 1)`	0	0	0	1
33	`if ((a+b) ≤ c)`	0	0	0	0
38	`else if (match == 2)`	0	0	0	1
40	`if ((a+c) ≤ b)`	0	0	0	0
45	`else if (match == 3)`	0	0	1	0
47	`if ((b+c) ≤ a)`	0	0	0	1

- `if (b==c)`
- `if` *(match == 0)*
- `if ((a+b) <= c) || ((b+c) <= a) || ((a+c) <= b))`
- `else if` *(match == 1)*
- `if` *((a+b) <= c)*
- `else if` *(match == 2)*
- `if` *((a+c) <= b)*
- `else if` *(match == 3)*
- `if` *((b+c) <= a)*

Almost all the predicates highlighted above were not covered. Again in the case of branches that could not be covered for a predicate i.e. if both true and false branches of a predicate did not get covered, it is usually because other branches affect it. Either in a nested structure parent branches do not get covered so the child branches can't be either. Or as is the case below in a if-else if structure if the first predicate's false branch is never taken then other branches never get covered as a result.

The sixth, eighth and tenth conditions false branches are not triggered due to the conventions used in ATAC. On examination of the code for the TRIANGLE program it can be seen that the fourth conditions true branch is always taken. Thus the other conditions are never tested. Thus the false branches are implicitly covered but this is not shown in ATAC.

Many branches are not covered in pseudo-random testing because the conditions under which they do are very specific. The test cases generated are the lengths of the sides (3 numbers). The specific combination of values needed occur very rarely at random. These situations are ideal for the use of a checkpoint encoding scheme where specific cases have to be targeted.

2. Checkpoint Encoded Random Testing.

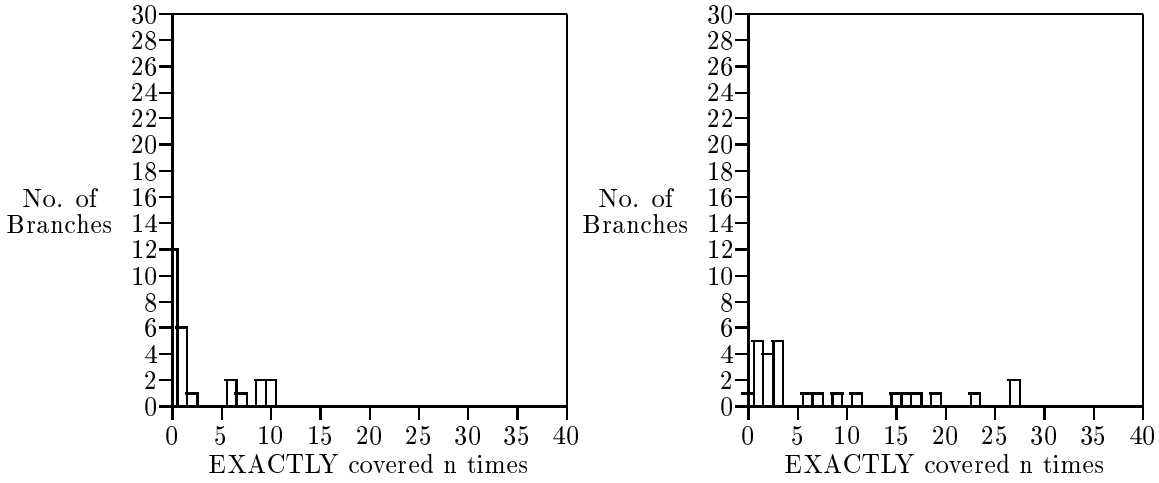


Figure 5.11: Triangle program : Checkpoint Encoded Random Testing : 10 & 30 Tests

The ECN measure for checkpoint encoded random testing shows an improvement as compared to pseudo-random testing. For a lower number of tests like 10 test cases it is not able to cover all the branches. More branches are covered especially with 30 test cases. Only one branch is not covered. Even so a majority of the branches are clustered at the beginning of the graph inidcating that most branches are still hard to cover.

Hard to test branches with checkpoint encoded random testing are listed below.

- `if` *((a+b) <= c)*
- `if` **(a==b)**
- `if ( ((a+b) <= c) || ((b+c) <= a) || ((a+c) <= b))`
- `else if` **(match == 1)**
- `if` *((a+c) <= b)*
- `else if` *(match == 3)*
- `if` *((b+c) <= a)*

Table 5.12: Coverage of each branch in TRIANGLE using checkpoint encoded random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
13	if (a == b)	0	10	3	27
15	if (a == c)	1	9	7	23
17	if (b == c)	0	10	3	27
23	if (match == 0)	9	1	19	11
25	if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))	2	7	2	17
25	if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))	1	7	1	16
25	if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))	0	7	1	15
30	else if (match == 1)	0	1	2	9
33	if ((a+b) ≤ c)	0	0	2	0
38	else if (match == 2)	1	0	6	3
40	if ((a+c) ≤ b)	0	1	3	3
45	else if (match == 3)	0	0	2	1
47	if ((b+c) ≤ a)	0	0	1	1

- if (a==c)
- if (((a+b) <= c) || ((b+c) <= a) || ((a+c) <= b))
- else if (match == 2)
- if ((a+c) <= b)
- if (b==c)
- else if (match == 3)
- if ((b+c) <= a)
- if *(match == 0)*

While all the hard to cover branches are largely the same as with random testing, the difference is that other than one branch (the false branch in the fourth condition) all other branches are covered and more times than with random testing. The false branch of predicate in line 33 is mostly uncovered because the random nature of the binary vector generation does not sample the whole encoding scheme. Thus the checkpoint encoding scheme helps in improving coverage but does not take complete advantage of the checkpoint encoding scheme. For that anti-random testing is better.

3. Anti-random Testing.

Anti-random testing shows a clear advantage over the other methods in this hard to test program. It covers 100% of the branches faster than checkpoint encoded random testing 5.9 exploring the entire domain space faster.

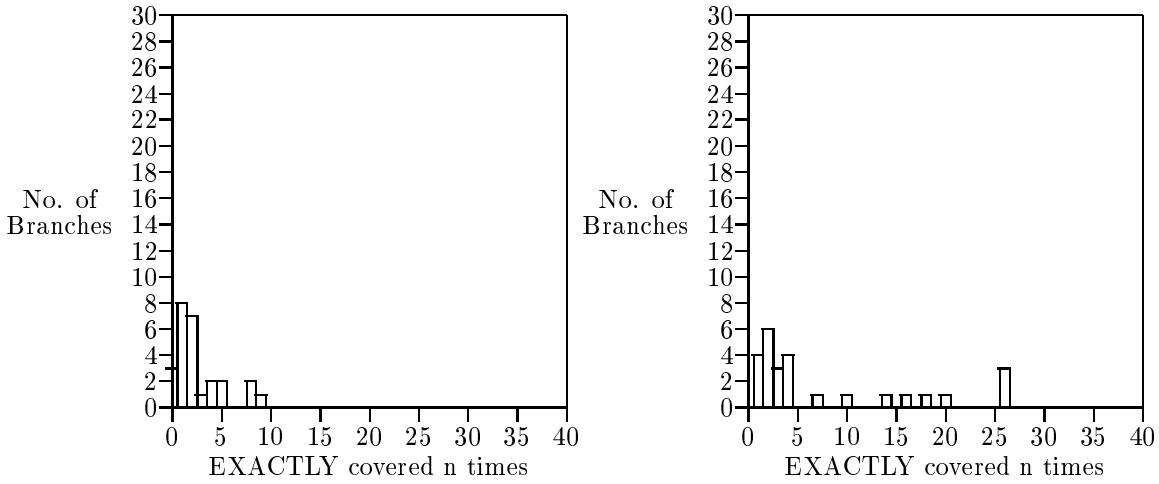


Figure 5.12: Triangle program : Anti Random Testing : 10 & 30 Tests

There is an improvement over the results from checkpoint encoded random testing shown in the ECN measure. This is more evident in the ECN measure using just 10 tests. Fewer branches remain uncovered and the graph indicates a shift to left i.e. branches are covered more often. This trend is present in the ECN measure with 30 tests but is not so evident.

Table 5.13: Coverage of each branch in TRIANGLE using anti-random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
13	`if (a == b)`	1	9	4	26
15	`if (a == c)`	2	8	4	26
17	`if (b == c)`	2	8	4	26
23	`if (match == 0)`	5	5	20	10
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	1	4	2	18
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	1	3	2	16
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	2	1	2	14
30	`else if (match == 1)`	1	4	3	7
33	`if ((a+b) ≤ c)`	0	1	2	1
38	`else if (match == 2)`	2	2	3	4
40	`if ((a+c) ≤ b)`	2	0	2	1
45	`else if (match == 3)`	2	0	3	1
47	`if ((b+c) ≤ a)`	1	1	2	1

Hard to cover branches with anti-random testing is shown below.

- `if ((a+b) <= c)`
- `if` *((a+c) <= b)*
- `else if` *(match == 3)*
- `if (a==b)`

- `if (((a+b) <= c) || ((b+c) <= a) || ((a+c) <= b))`
- `else if (match == 1)`
- `if ((a+b) <= c)`
- `if ((b+c) <= a)`
- `if (a==c)`
- `if (b==c)`
- `if ( ((a+b) <= c) || ((b+c) <= a) || ((a+c) <= b))`

Anti-random testing manages to cover all the branches unlike checkpoint encoded random testing. The lone branch (false branch, line 33) that was not covered by checkpoint encoded random testing was targeted in the checkpoint encoding scheme but was sample only in anti-random testing. Again the branches still remain classified as hard to cover but have been covered more often. This was a hard to test program with nested branching constructs; the number of branches were disproportionately large compared to the program size. Anti-random testing 5.9 gets better branch coverage faster and tests the branches more uniformly. The advantages of using anti-random testing become clear for a program of this nature i.e. with many hard to test branches.

5.2.4.4 Summary

The result of of using the standard checkpoint encoding scheme for all 3 programs are summarized in table 5.14. These results are in comparison with random testing. Thus the "Not Covered" column indicates the number of branches that did not get covered. The "Hard to Cover" column gives the number of branches that were covered but are classified as "hard to cover" branches since they were covered by less than 30% of the tests. The "Previously Not Covered" and "Previously Hard to Cover" columns give the number of covered branches that were previously not covered; and easy to cover branches that were previously hard to cover, respectively, as compared to random testing. That is, those branches that were not covered or were considered hard to cover for that program using random testing. These results contrast the effectiveness of all three testing methods using the standard checkpoint encoding scheme.

Both the FINDNO and STRMAT programs are relatively testable. In FINDNO almost all the branches are covered and very few of them are hard to test. The few branches that remain uncovered are error checking conditions that are hard to cover in normal operational testing. Even the checkpoint encoding scheme fails to target them. This indicates deficiencies in the procedure

Table 5.14: Summary of results using the standard checkpoint encoding scheme.

Program	Total # Branches	# Not Covered	# Hard to Cover	# Previously Not Covered	# Previously Hard to Cover
		Random Testing			
FINDNO	26	4	1	-	-
STRMAT	20	0	5	-	-
TRIANGLE	26	10	8	-	-
		Checkpoint Encoded Random Testing			
FINDNO	26	4	1	0	1
STRMAT	20	0	6	0	1
TRIANGLE	26	1	17	9	0
		Anti-Random Testing			
FINDNO	26	4	0	0	1
STRMAT	20	0	6	0	1
TRIANGLE	26	0	18	10	0

to build a checkpoint encoding scheme and will be discussed in the next section. The TRIANGLE program is hard to test with random testing unable to cover a lot of branches and the most of the covered branches turn out to be hard to cover. The checkpoint encoding scheme is able to target these branches well. Even though they do not become easy to cover (i.e. covered by 30% or more of the tests), they are covered more often than in random testing. Anti-random testing, even though it uses the same checkpoint encoding scheme as checkpoint encoded random testing, is more effective vs hard to cover or hard to test branches. The standard checkpoint encoding scheme also indicates certain trends. It is harder for the error conditions and special cases to get covered. High levels of nesting are also another problem. Thus there is a need to focus on improving the checkpoint encoding scheme to cover these types of branches.

5.3 Different Checkpoint Encoding Schemes

The previous section examined in detail the three programs, how the three different testing methods compared in terms of overall branch coverage as well as their effectiveness v/s hard to cover branches. This thesis proposes a method of checkpoint encoding to improve effectiveness with respect to hard to cover branches.

As explained before two different checkpoint encoding schemes are experimented with. The first looks at checkpoint encoding *legal* partitions i.e. partitions of the input domain containing valid inputs that do not trigger errors in normal operation of the program. The second scheme focuses on checkpoint encoding *illegal* partitions i.e. containing inputs that not part of the programs normal operational profile. The illegal checkpoint encoding scheme also includes boundary cases; those inputs that lie on the junction of valid and invalid domains. This rationale is taken from partition testing theory, where this thesis experiments with different kinds of partitions at a very basic level instead of considering the problem of hard to test branches at a higher level of abstraction. This is also based on results from the standard checkpoint encoding scheme.

The question this thesis attempts to answer is whether such a basic technique yields benefits. If the checkpoint encoding scheme focuses on illegal partitions and boundary cases then would be more effective? This also helps in improving the understanding of the effectiveness of the checkpoint encoding scheme and in formulating a procedure for checkpoint encoding.

5.3.1 Checkpoint Encoding of Legal Partitions

5.3.1.1 Findno Program

The legal checkpoint encoding scheme for the FINDNO program focuses on valid inputs and avoids boundary cases. Thus the array size range is divided to focus on divisions within it. The array status is either sorted or randomly ordered. The index of the element pointed to is split amongst the range of the array size avoiding boundary values.

The graph of coverage using the legal checkpoint encoding scheme is similar to that of standard checkpoint encoding 5.1. One conclusion can be easily drawn from this. The standard checkpoint encoding scheme for the FINDNO program focused largely on valid inputs. These valid domains cover the easy to test branches. Since a high level of coverage is achieved perhaps the conclusion can also be drawn that the program is easy to test.

1. Checkpoint Encoded Random Testing

Table 5.15: Legal checkpoint encoding scheme for FINDNO.

Field	Bits	Value	Significance
Array Size(N)	b1,b0	00	2 ··· 21
		rest	22 ··· 42
		11	43 ··· 63
Array Status	b2	0	sorted order
		1	random order
Element Values	b4,b3	00	all positive
		11	all negative
		rest	mixed
Index	b9,b8	00	2 ··· N/3
		01	N/3 ··· 2N/3
		rest	2N/3 ··· N-1

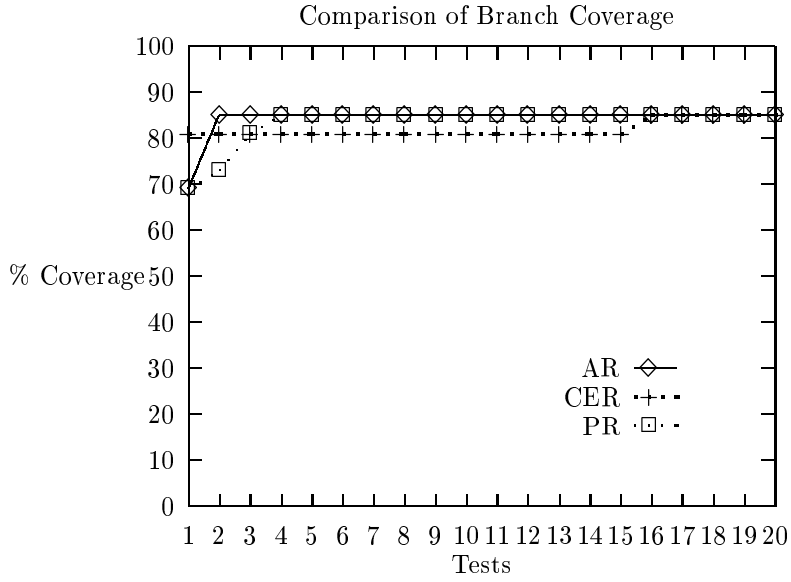


Figure 5.13: FINDNO program

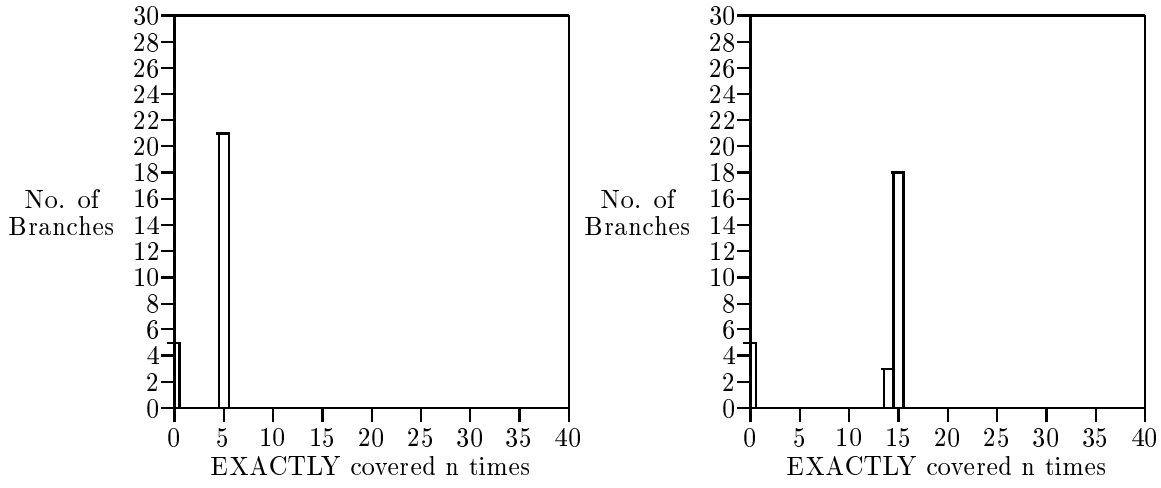


Figure 5.14: FINDNO program : Checkpoint Encoded Random Testing : 5 & 15 Tests

The ECN graph shows clearly the structure of the program. All the easy to cover branches are covered by almost all the tests. The ECN graph makes one thing clear : the legal test cases cover the easy to test branches repeatedly. Alternately it could be said that the branches encapsulating code that contains the logoic to handle "normal" inputs can be considered to be easy to test. The few tests that are not covered are the ones that were identified as hard to cover or hard to test during random testing.

Table 5.16: Coverage of each branch in FINDNO using checkpoint encoded random testing.

Line No.	Branch	5 Tests		15 Tests	
		True	False	True	False
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
24	`for(i=1;i ≤ n;i++)`	5	5	15	15
31	`if (x == z)`	5	0	15	0
43	`for(i=1;i ≤ n;i++)`	5	-	15	-
56	`while ((m < ns) || b)`	5	5	15	15
56	`while ((m < ns) || b)`	0	-	0	-
58	`if (!b)`	5	5	15	15
65	`if (i >j)`	5	5	15	15
67	`if (f > j)`	5	5	14	15
69	`if (i > f)`	0	5	0	14
79	`while (a[i] < a[f])`	5	5	15	15
81	`while (a[f] < a[j])`	5	5	14	15
83	`if (i ≤ j)`	5	5	15	15

The hard to cover branches identified with checkpoint encoded testing are

- `if ((f ≤ 0) || (f > n))`
- `if ((f ≤ 0) || (f > n))`
- `if` *(x == z)*
- `while ((m < ns) || b)`
- `if (i > f)`

These are almost all error or exception checking conditions. The last condition still does not get covered due to it being deeply nested. Thus the checkpoint encoding scheme using the legal checkpoint encoding scheme targets the easy to cover branches very effectively.

2. Anti-Random Testing

The ECN graphs for anti-random testing show similar structure as with the ECN graphs for checkpoint encoded random testing. However as observed before the clear separation between the

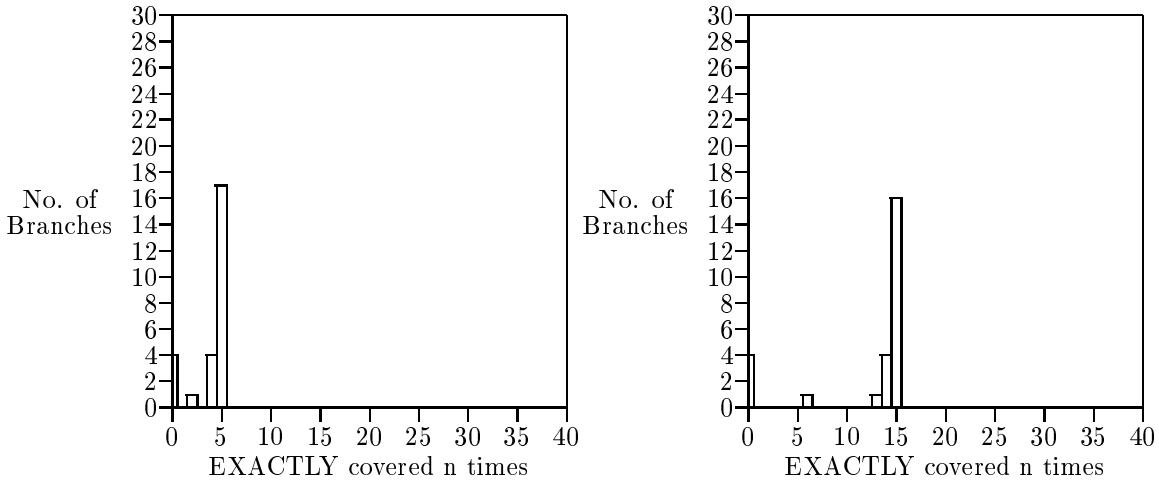


Figure 5.15: FINDNO program : Anti Random Testing : 5 & 15 Tests

easy and hard to cover branches is not as stark in anti-random testing as it is in checkpoint encoded random testing.

Table 5.17: Coverage of each branch in FINDNO using anti-random testing.

Line No.	Branch	5 Tests		15 Tests	
		True	False	True	False
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
18	`if ((f ≤ 0) || (f > n))`	0	5	0	15
24	`for(i=1;i ≤ n;i++)`	5	5	15	15
31	`if (x == z)`	5	0	15	0
43	`for(i=1;i ≤ n;i++)`	5	-	15	-
56	`while ((m < ns) || b)`	5	5	15	15
56	`while ((m < ns) || b)`	0	-	0	-
58	`if (!b)`	5	5	15	15
65	`if (i >j)`	5	5	15	15
67	`if (f > j)`	5	4	15	14
69	`if (i > f)`	2	4	6	13
79	`while (a[i] < a[f])`	4	5	14	15
81	`while (a[f] < a[j])`	5	5	14	15
83	`if (i ≤ j)`	5	4	15	14

The hard to cover branches identified are

- `if ((f ≤ 0) || (f > n))`
- `if ((f ≤ 0) || (f > n))`
- `if` *(x == z)*
- `while ((m < ns) || b)`

It has one less branch than checkpoint encoded random testing. The branches was previously not getting covered enough times since it was nested deeply. With anti-random testing since tests are distributed more evenly that branch does get covered.

5.3.1.2 STRMAT Program

The domains identified and encoded are the same as with the standard testing scheme. As explained before all boundary and error conditions are avoided. The domain ranges are split to sample them in further detail. There is no particular division scheme: it is just divided into equal thirds. The pattern length has a maximum value of 3 which is not encoded since it is a boundary value. The other boundary value is taken as 0 not 1. So even that is not encoded. The text length varies from 2 to 79. The varies from 2 to the text length chosen (T).

Table 5.18: Legal checkpoint encoding scheme for STRMAT.

Field	Bits	Value	Significance
Text Length(T)	b1,b0	00	2 ··· 10
		rest	11 ··· 70
		11	71 ··· 79
Pattern Position	b3,b2	00	2 ··· T/3
		rest	T/3 ··· 2T/3
		11	2T/3 ··· T-1
Pattern Length	b4	0	1
		1	2

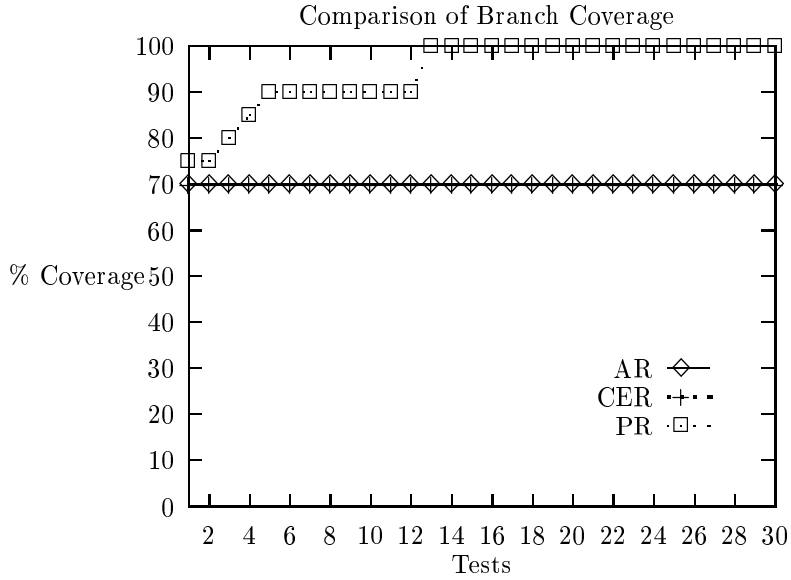


Figure 5.16: STRMAT program

Comparing the graph of the coverage with legal test cases with the standard result 5.5 both the checkpoint encoded random testing and anti-random testing reach the same initial branch coverage

(70%) and remain constant after that. Thus the domains are homogenous with none of the tests covering any new branches.

The graphs for the *Strmat* program are shown below

1. Checkpoint Encoded Random Testing.

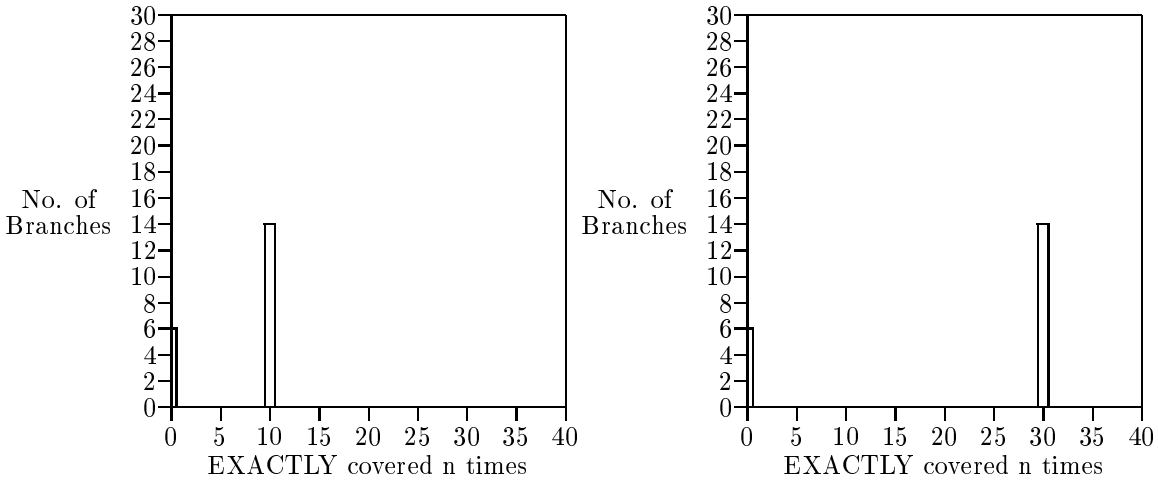


Figure 5.17: Strmat program : Checkpoint Encoded Random Testing : 10 & 30 Tests

As stated before the domains that make up the STRMAT legal partitions are homogenous. Thus all tests from that domain either cover all branches or none at all. This is observed clearly. 6 branches are not covered at all and 14 branches are covered by all tests. This remains the case even with increasing number of test cases.

Table 5.19: Coverage of each branch in STRMAT using checkpoint encoded random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
18	`while (c != '\n')`	10	10	30	30
20	`if (i ≤ tmax)`	10	0	30	0
39	`while (c != '\n')`	10	10	30	30
41	`if (i ≤ pmax)`	10	0	30	0
71	`if (textlen == 0)`	0	10	0	30
76	`if (patlen == 0)`	0	10	0	30
81	`while ( (patpos ≤ patlen) && (textpos ≤ textlen))`	10	10	30	30
81	`while ((patpos ≤ patlen) && (textpos ≤ textlen))`	10	0	30	0
83	`if (pattern[patpos] == text[textpos])`	10	10	30	30
95	`if (patpos > patlen)`	10	0	30	0

The hard to cover branches identified are

- `if` *(i ≤ tmax)*

- `if` *(i ≤ pmax)*
- `if` **(textlen == 0)**
- `if` **(patlen == 0)**
- `while ((patpos ≤ patlen) &&` *(textpos ≤ textlen)*)
- `if` *(patpos > patlen)*

All the conditions check for error conditions. Since this coverage is measured only with legal test cases this is to be expected. The fifth condition is never false since the pattern aslways exists in the text, with length 1 or 2. Thus the results match with what is expected.

2. Anti Random Testing.

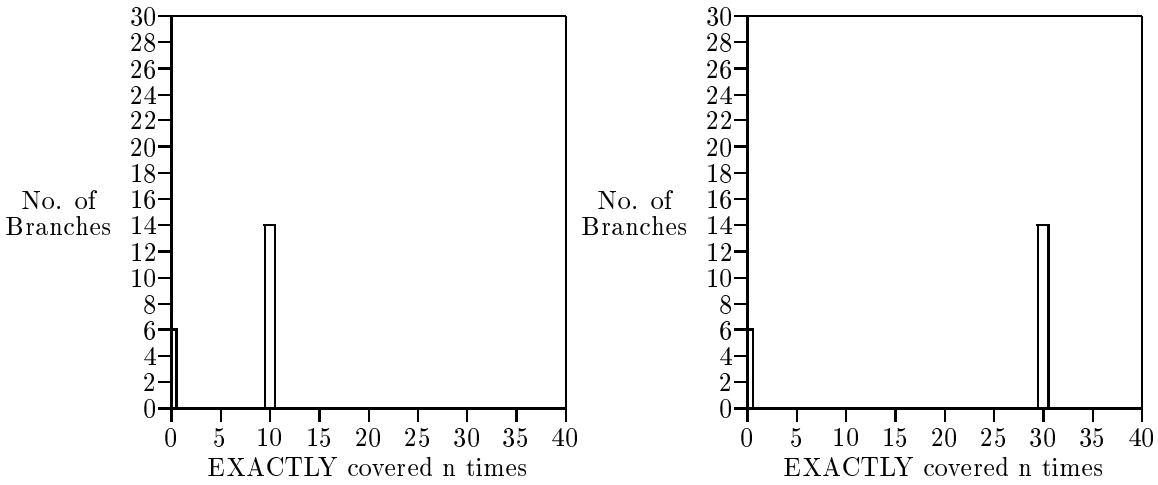


Figure 5.18: Strmat program : Anti Random Testing : 10 & 30 Tests

Since the domains are homogenous anti-random testing should not yield any different results. It does not. Since the domains are homogenous it does not matter how well the domains are sampled. The ECN graphs for anti-random testing are the same as checkpoint encoded random testing graphs. 6 branches do not get covered and 14 branches get covered by all the tests, whether it is 10 or 30 tests.

The hard to test branches for anti-random testing are the same as with checkpoint encoded testing. The 6 branches that are not covered are error checking conditions.

Table 5.20: Coverage of each branch in STRMAT using anti-random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
18	`while (c != '\n')`	10	10	30	30
20	`if (i ≤ tmax)`	10	0	30	0
39	`while (c != '\n')`	10	10	30	30
41	`if (i ≤ pmax)`	10	0	30	0
71	`if (textlen == 0)`	0	10	0	30
76	`if (patlen == 0)`	0	10	0	30
81	`while ( (patpos ≤ patlen) && (textpos ≤ textlen))`	10	10	30	30
81	`while ((patpos ≤ patlen) && (textpos ≤ textlen))`	10	0	30	0
83	`if (pattern[patpos] == text[textpos])`	10	10	30	30
95	`if (patpos > patlen)`	10	0	30	0

5.3.1.3 TRIANGLE Program

The *legal* checkpoint encoding scheme for TRIANGLE is different than the standard checkpoint encoding scheme. No cases involving illegal triangles are used. All valid input cases are used with equal emphasis given to all 3 kinds of triangles : equilateral, isosceles ad scalene. The sides of the triangle also were partitioned in the legal domain. No boundary cases were used.

Table 5.21: Legal checkpoint encoding scheme for TRIANGLE.

Field	Bits	Value	Significance
Triangle Type (legal triangle)	b2,b1,b0	010	a = b (Isoceles)
		100	a = c (Isoceles)
		101	b = c (Isoceles)
		000,001	a = b = c (Equilateral)
		rest	a != b != c (Scalene)
Sides of triangle (a,b,c)	b5,b4,b3	000,001	2 - (Max/3 - 1)
		111,110	2Max/3 - Max-1
		rest	Max/3 - (2Max/3 - 1)

The graph of total branch coverage of the TRIANGLE program for legal checkpoint encoding is shown in figure 5.19.

Compared to the standard checkpoint encoding scheme 5.9 the checkpoint encoded random and anti-random results have the same initial results. The legal checkpoint scheme then plateaus at about 80%. Anti-random reaches the maximum coverage faster than checkpoint encoded random testing for the legal scheme.

The graphs for the *Triangle* programs are shown below

1. Checkpoint Encoded Random Testing.

The ECN graphs for checkpoint encoded random testing using the legal checkpoint encoding

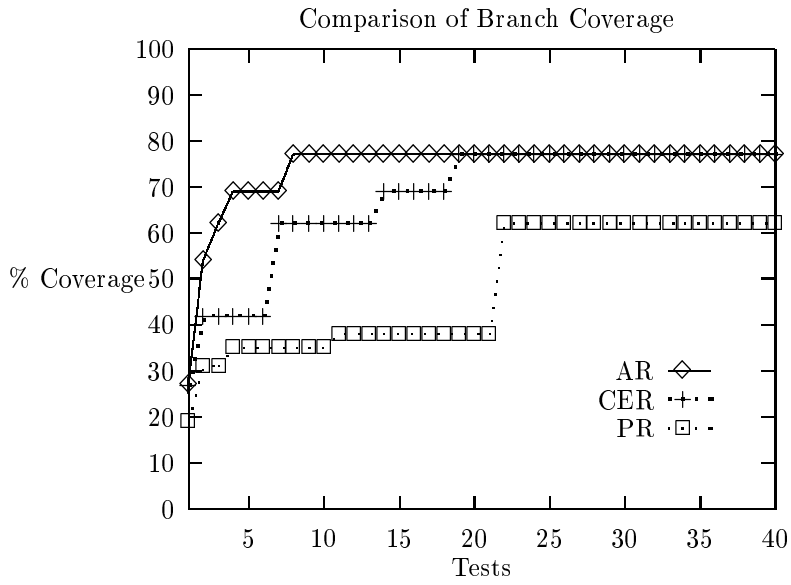


Figure 5.19: TRIANGLE program

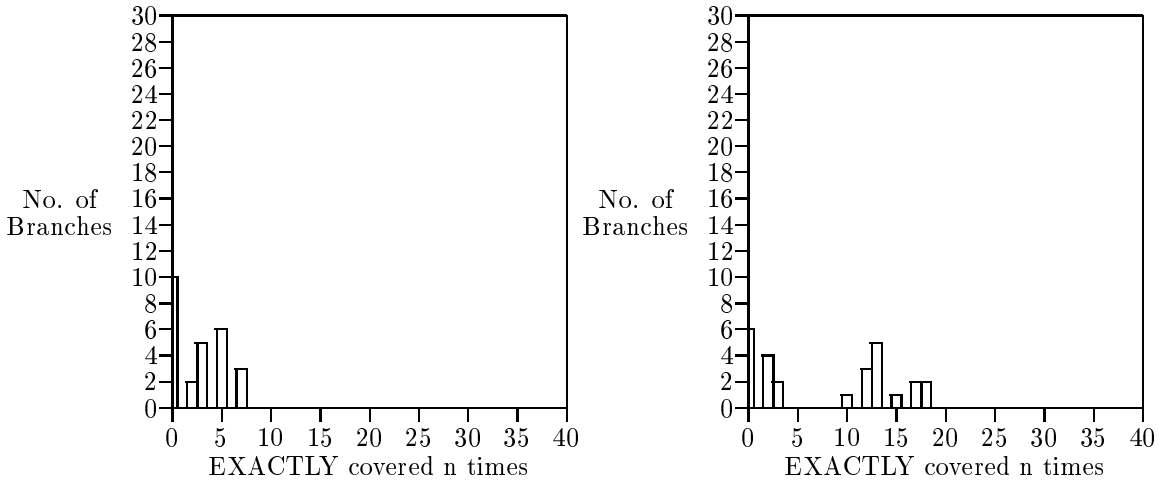


Figure 5.20: Triangle program : Checkpoint Encoded Random Testing : 10 & 30 Tests

scheme are shown in 5.20 for 10 and 30 tests. What is observed is that several branches do not get covered. The branches that do get covered are spread over the spectrum of tests indicating a non-uniform behaviour. For 10 tests 10 branches do not get covered. The rest of the branches are clustered around the half way mark indicating that the TRIANGLE program behaviour is not uniform. The branches are neither very testable not hard to test. With the number of tests increasing to 30 the number of undetected branches reduce to 6. The branches that do get covered show a decided shift to the tail of the graph. Thus most of the branches get covered and quite often though not by every other test.

Table 5.22: Coverage of each branch in TRIANGLE using checkpoint encoded random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
13	`if (a == b)`	5	5	12	18
15	`if (a == c)`	7	3	13	17
17	`if (b == c)`	5	5	12	18
23	`if (match == 0)`	3	7	13	17
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	3	0	13
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	3	0	13
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	3	0	13
30	`else if (match == 1)`	0	7	2	15
33	`if ((a+b) ≤ c)`	0	0	0	2
38	`else if (match == 2)`	2	5	3	12
40	`if ((a+c) ≤ b)`	0	2	0	3
45	`else if (match == 3)`	0	5	2	10
47	`if ((b+c) ≤ a)`	0	0	0	2

The hard to test branches identified are

- `if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`
- `if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`
- `if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`
- `else if (match == 1)`
- `if` $((a+b) \leq c)$
- `else if (match == 2)`
- `if` $((a+c) \leq b)$
- `else if (match == 3)`
- `if` $((b+c) \leq a)$

Most of the branches not covered are error checking conditions. This is to be expected since legal partitions do not include those input cases. However the TRIANGLE program is quite

complex with nesting of conditions, and very specific ones at that. Thus only with many test cases do those branches get covered. The hard to cover branches that are listed are extracted from the results of using 30 test cases not 10. Even so legal partitions are not able to make branches any easier to cover.

2. Anti Random Testing.

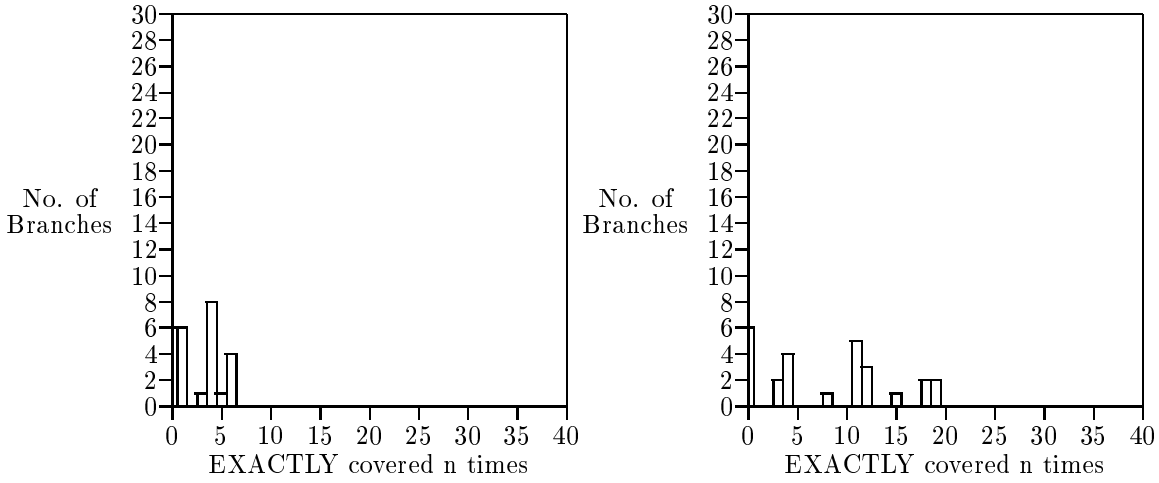


Figure 5.21: Triangle program : Anti Random Testing : 10 & 30 Tests

The ECN graphs for anti-random testing using the legal checkpoint encoding scheme are shown in figure 5.21 for 10 and 30 tests. The ECN graphs are similar to that shown in figure 5.20 for checkpoint encoded random testing. However the trend shifts slightly to the head of the graph. The branches are covered but less frequently as compared to checkpoint encoded random testing.

This does follow the behaviour of anti-random testing to distribute its tests more evenly. More branches are tested but less frequently. For 10 tests the number of uncovered branches is 6 but they are covered less frequently. For 30 tests there is no clear difference.

The hard to test branches identified are

- `if (((a+b)` $\leq$ `c) || ((b+c)` $\leq$ `a) || ((a+c)` $\leq$ `b))`
- `if (((a+b)` $\leq$ `c) || ((b+c)` $\leq$ `a) || ((a+c)` $\leq$ `b))`
- `if (((a+b)` $\leq$ `c) || ((b+c)` $\leq$ `a) || ((a+c)` $\leq$ `b))`
- `else if (match == 1)`

Table 5.23: Coverage of each branch in TRIANGLE using anti-random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
13	`if (a == b)`	4	6	12	18
15	`if (a == c)`	4	6	11	19
17	`if (b == c)`	4	6	12	18
23	`if (match == 0)`	4	6	11	19
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	4	0	11
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	4	0	11
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	4	0	11
30	`else if (match == 1)`	1	5	4	15
33	`if ((a+b) ≤ c)`	0	1	0	4
38	`else if (match == 2)`	1	4	3	12
40	`if ((a+c) ≤ b)`	0	1	0	3
45	`else if (match == 3)`	1	3	4	8
47	`if ((b+c) ≤ a)`	0	1	0	4

- `if` *((a+b) ≤ c)*
- `else if` **(match == 2)**
- `if` *((a+c) ≤ b)*
- `else if` **(match == 3)**
- `if` *((b+c) ≤ a)*

Again the same overall trend as with checkpoint encoding. Branches are overall covered a little more often but the program behaviour limits effectiveness of anti-random testing to make branches any easier to cover.

5.3.1.4 Summary

The result of of using the legal checkpoint encoding scheme for all 3 programs are summarized in table 5.24. These results are in comparison with random testing with standard checkpoint encoding. Thus the "Not Covered" column indicates the number of branches that did not get covered using legal checkpoint encoding scheme. The "Hard to Cover" column gives the number of branches that were covered but are classified as "hard to cover" branches since they were covered by less than 30% of the tests. The "Previously Not Covered" and "Previously Hard to Cover" columns give the number of covered branches that were previously not covered; and easy to cover branches that were previously hard to cover, respectively, as compared to random testing with the standard checkpoint encoding scheme. That is, those branches that were not covered or were considered hard to cover for that program using random testing. Indirectly these results help contrast the results of using checkpoint encoding with legal partitions instead of standard checkpoint encoding.

Table 5.24: Summary of results using the legal checkpoint encoding scheme.

Program	Total # Branches	# Not Covered	# Hard to Cover	# Previously Not Covered	# Previously Hard to Cover
		Checkpoint Encoded Random Testing			
FINDNO	26	5	0	0	1
STRMAT	20	6	0	0	2
TRIANGLE	26	6	6	7	4
		Anti-Random Testing			
FINDNO	26	4	0	0	1
STRMAT	20	6	0	0	2
TRIANGLE	26	6	7	7	4

The first two columns are describe the effect of using legal checkpoint encoding schemes with checkpoint encoded random and anti-random testing. There is not much difference in the two test generation methods: checkpoint encoded random and anti-random testing, though anti-random testing proves to be slightly better than checkpoint encoded random testing especially considering fewer number of test cases. Both yield about the same number of uncovered and hard to test branches. Comparing each testing method's results with the results obtained with standard checkpoint encoding the differences are similar. For both methods the number of uncovered branches increase but the number of hard to cover branches decrease sharply.

Relative to random testing the same observation is made about both methods. With legal checkpoint encoding fewer branches are covered that did not get covered by random testing as compared to standard checkpoint encoding. However more branches that were considered hard to cover with random testing are easy to cover with legal checkpoint encoding schemes.

Thus the overall observation for the set of programs under test is as follows : anti-random testing is better than checkpoint encoded random testing but not by much. Legal checkpoint encoding does not yield as high a coverage as standard checkpoint encoding but the number of branches that are covered but are classified as hard to cover is lower than with standard checkpoint encoding.

5.3.2 Checkpoint Encoding of Illegal Partitions

The checkpoint encoding scheme for all the programs focus on illegal partitions and boundary cases. This is similar in intent to Chan's (Chan, Chen, and Tse 1997) suggestion that partitions with higher probability of failure should provide most of the test cases. Thus what is targeted with these partitions are high risk areas in the code, portions of the code that have a higher probability of containing faults. This is a well known result from partition testing/domain testing. However the objective of this chapter is to understand how this affects checkpoint encoding and test generation methods using checkpoint encoding.

5.3.2.1 FINDNO Program

The partitions for the FINDNO program are not precise. The array status partitions do not focus solely on illegal values since that sometimes makes it impractical to run any kind of tests. Some partitions focus almost exclusively on illegal and boundary partitions.

Table 5.25: Illegal checkpoint encoding scheme for FINDNO.

Field	Bits	Value	Significance
Array Size	b1,b0	00	negative, NaN
		01	0
		11	1,64
		10	$64 < < 100$
Array Status	b4,b3,b2	110	sorted order
		100	reverse order
		011	all equal
		rest	random order
Element Values	b7,b6,b5	000	NaN
		010	all positive
		101	all negative
		110	all zeros
		rest	mixed
F points to	b9,b8	00	negative, NaN
		01	0
		11	first or last element
		10	no element

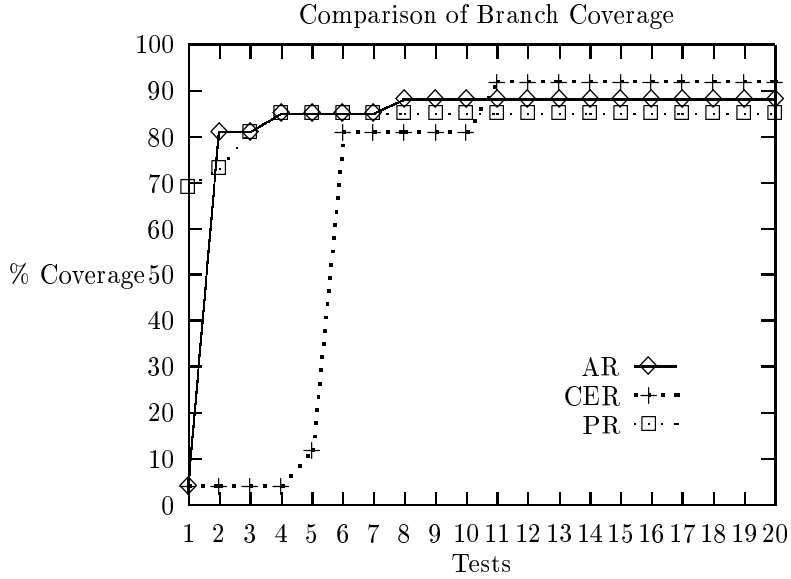


Figure 5.22: FINDNO program

The branch coverage (shown in figure 5.22) demonstrates the effectiveness of the above checkpoint encoding scheme. Compared to the use of standard checkpoint encoding (shown in figure 5.1) the

current results show higher coverage. Checkpoint encoded random testing takes some time but finally shows a higher coverage than anti-random testing.

1. Checkpoint Encoded Random Testing

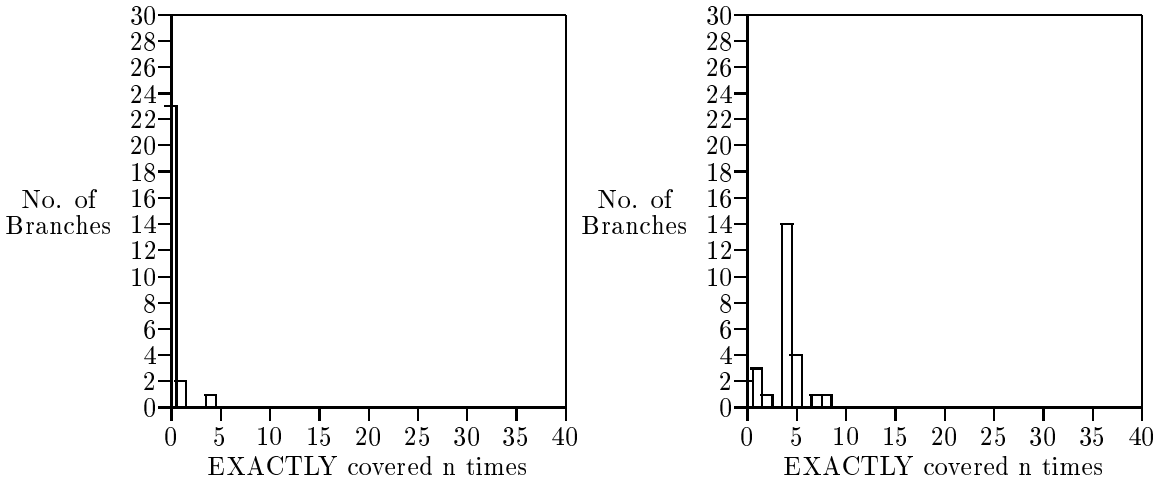


Figure 5.23: FINDNO program : Checkpoint Encoded Random Testing : 5 & 15 Tests

The lag shown by checkpoint encoded random testing can be seen in 5.23 with the results for 5 tests being very poor : almost all the branches are not covered. With 10 tests the results are much better. However even then most branches just about come above the hard to cover threshold. This is to be expected with illegal checkpoint encoding schemes, especially with the FINDNO program. The FINDNO program has many easy to cover branches which do not get covered by illegal test cases. These are covered by boundary cases but these cases take many tests to be sampled repeatedly.

2. Anti-Random Testing

Anti-random testing (shown in figure 5.24) shows better results than checkpoint encoded random testing especially with 5 tests. With 10 tests checkpoint encoded random testing is better than anti-random testing with much fewer branches being hard to cover. This may actually be a result of anti-random tests being better distributed across the partitions. In checkpoint encoded random testing the distribution of tests may have been skewed towards boundary rather than illegal cases which could lead to better coverage.

Table 5.26: Coverage of each branch in FINDNO using checkpoint encoded random testing.

Line No.	Branch	5 Tests		15 Tests	
		True	False	True	False
18	`if ((f ≤ 0) || (f > n))`	4	1	8	7
18	`if ((f ≤ 0) || (f > n))`	1	0	2	5
24	`for(i=1;i ≤ n;i++)`	0	0	5	5
31	`if (x == z)`	0	0	5	0
43	`for(i=1;i ≤ n;i++)`	0	-	4	-
56	`while ((m < ns) || b)`	0	0	4	4
56	`while ((m < ns) || b)`	0	-	0	-
58	`if (!b)`	0	0	4	4
65	`if (i >j)`	0	0	4	4
67	`if (f > j)`	0	0	4	1
69	`if (i > f)`	0	0	1	4
79	`while (a[i] < a[f])`	0	0	4	4
81	`while (a[f] < a[j])`	0	0	1	4
83	`if (i ≤ j)`	0	0	4	4

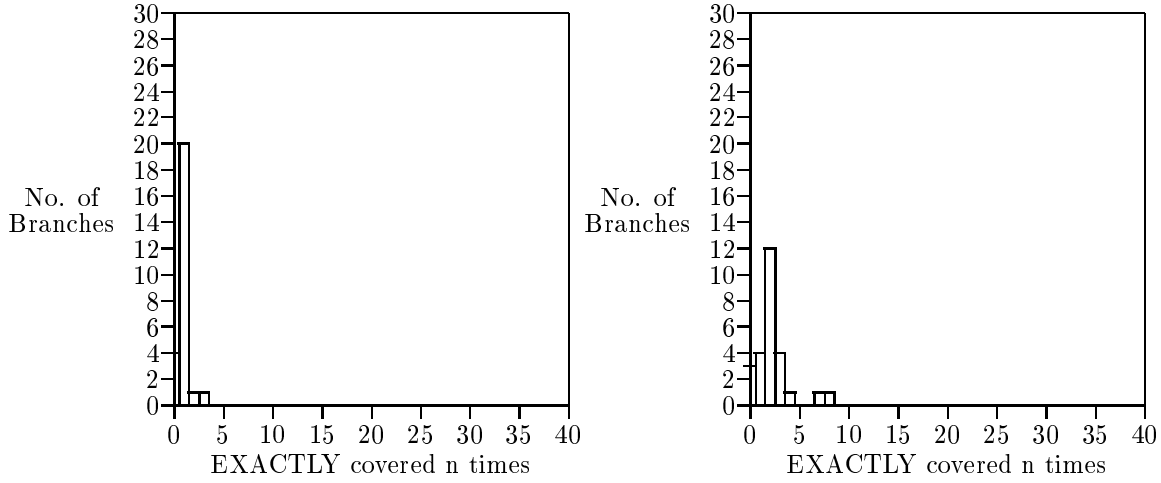


Figure 5.24: FINDNO program : Anti Random Testing : 5 & 15 Tests

Table 5.27: Coverage of each branch in FINDNO using anti-random testing.

Line No.	Branch	5 Tests		15 Tests	
		True	False	True	False
18	`if ((f ≤ 0) || (f > n))`	3	2	8	7
18	`if ((f ≤ 0) || (f > n))`	1	1	4	3
24	`for(i=1;i ≤ n;i++)`	1	1	3	3
31	`if (x == z)`	1	0	3	0
43	`for(i=1;i ≤ n;i++)`	0	-	1	-
56	`while ((m < ns) || b)`	1	1	2	2
56	`while ((m < ns) || b)`	0	-	0	-
58	`if (!b)`	1	1	2	2
65	`if (i >j)`	1	1	2	2
67	`if (f > j)`	1	0	1	1
69	`if (i > f)`	0	1	0	1
79	`while (a[i] < a[f])`	1	1	2	2
81	`while (a[f] < a[j])`	1	1	2	2
83	`if (i ≤ j)`	1	1	2	2

5.3.2.2 STRMAT Program

The partitions being encoded in the illegal checkpoint encoding scheme for STRMAT is shown in table 5.28. The variables are the same as in the standard 5.6 and legal 5.18 checkpoint encoding schemes but the partitions only include boundary and illegal partitions of those variables. In the case of STRMAT the standard checkpoint encoding scheme is similar to the illegal checkpoint encoding scheme except that it includes legal partitions as well. By sampling equally both illegal and boundary cases only an attempt is made to examine the effect of an illegal checkpoint encoding strategy.

Table 5.28: Illegal checkpoint encoding scheme for STRMAT.

Field	Bits	Value	Significance
Text Length	b2,b1,b0	000,001	0
		111,110	80
		rest	80 < < 100
Pattern Position	b5,b4,b3	000,001	beginning
		111,110	end
		rest	no pattern
Pattern Length	b8,b7,b6	000,001	0
		111,110	3
		rest	3 < < 10

The overall branch coverage of the STRMAT program is shown in 5.25. This shows much better results than the legal checkpoint encoding scheme (5.16) and is quite similar to the coverage graph for standard checkpoint encoding (5.5). Thus a preliminary conclusion we can draw is that while the STRMAT program is relatively testable with legal checkpoint encoding giving a 70% coverage from the start, the rest of the coverage increase is provided by the illegal input test cases which are similar in both standard and illegal checkpoint encoding schemes. Also this indicates that the illegal

checkpoint encoding scheme is able to cover all the branches that the legal checkpoint encoding scheme can i.e. the partitions are homogenous, the boundary cases and legal cases cover the same branches.

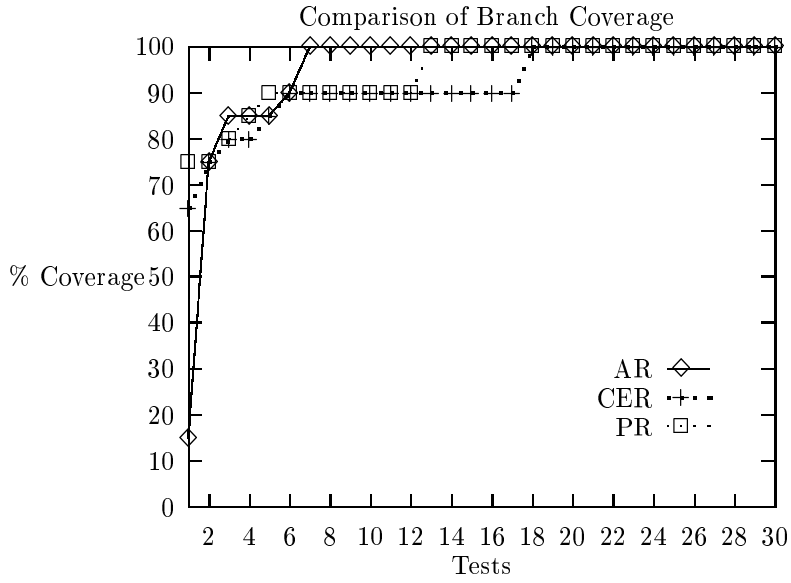


Figure 5.25: STRMAT program

Anti-random testing is a bit better than checkpoint encoded random testing, reaching 100% coverage before it. Overall as compared to the standard checkpoint encoding, illegal checkpoint encoding achieves 100% slightly before it. The difference admittedly is marginal.

The graphs for the *Strmat* programs are shown below

1. Checkpoint Encoded Random Testing.

   The ECN graphs for checkpoint encoded random testing with the illegal checkpoint encoding scheme are shown in Figure 5.26. They show a similar trend to that of standard checkpoint encoding (Figure 5.7). Most of the branches are still hard to cover with both 10 & 30 tests. With 10 tests 2 branches do not get covered, which get covered with 30 tests. Overall the coverage is fairly balanced, with most branches coverage being near half the total number of tests.

   The hard to test branches for checkpoint encoded random testing with illegal checkpoint encoding (30 tests) are

   - `if` **(textlen == 0)**
   - `if` **(patlen == 0)**

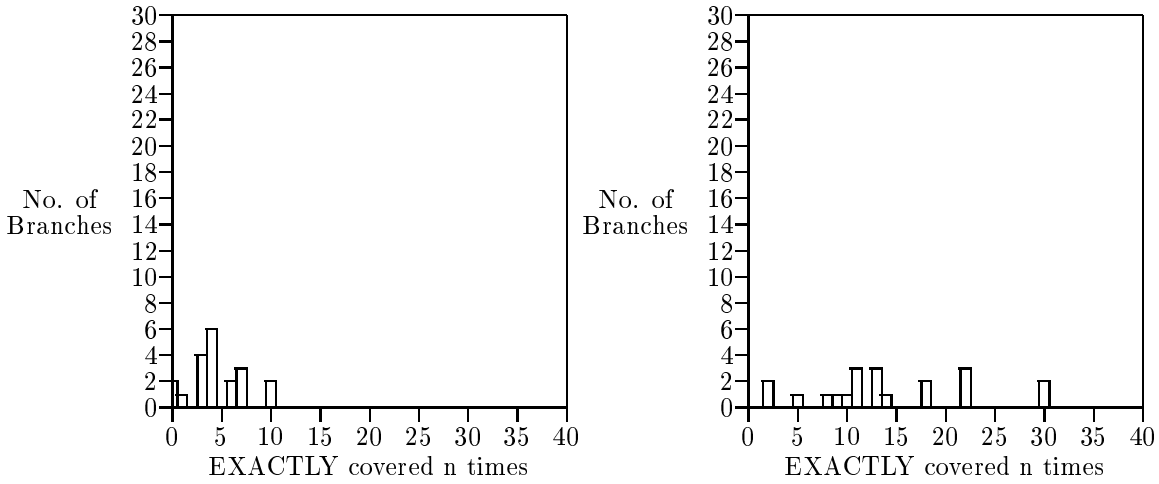


Figure 5.26: Strmat program : Checkpoint Encoded Random Testing : 10 & 30 Tests

Table 5.29: Coverage of each branch in STRMAT using checkpoint encoded random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
18	`while (c != '\n')`	7	10	22	30
20	`if (i ≤ tmax)`	7	3	22	14
39	`while (c != '\n')`	6	10	18	30
41	`if (i ≤ pmax)`	6	3	18	10
71	`if (textlen == 0)`	3	7	8	22
76	`if (patlen == 0)`	3	4	9	13
81	`while ( (patpos ≤ patlen) && (textpos ≤ textlen))`	4	0	13	2
81	`while ((patpos ≤ patlen) && (textpos ≤ textlen))`	4	4	13	11
83	`if (pattern[patpos] == text[textpos])`	1	4	5	11
95	`if (patpos > patlen)`	0	4	2	11

- `while ( (patpos ≤ patlen) && (textpos ≤ textlen))`
- `if (pattern[patpos] == text[textpos])`
- `if (patpos > patlen)`

Both the conditions in the predicates of the first two branches(true), check for a the text length or pattern length being zero. This is a boundary condition (it could also be considered an error condition) that is not sampled as many times, therefore the branch does not covered by or above 30% of the total number of tests. The third and fifth branches are triggered only if the pattern occurs completely in the text which is more a legal case than an illegal case, thus they are also hard to cover. The fourth branch is covered whenever any part of the pattern occurs in the text string (even if its not completely present) but with more emphasis given to the pattern not being present (illegal case) then this is also rarely covered but more often than the third and fifth conditions.

2. Anti Random Testing.

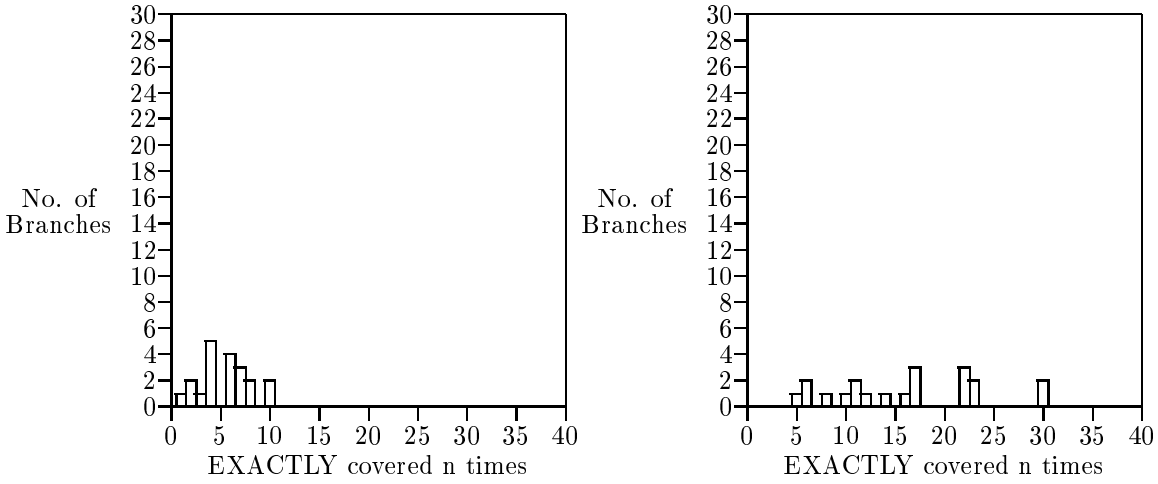


Figure 5.27: Strmat program : Anti Random Testing : 10 & 30 Tests

The ECN graphs for anti-random testing with the illegal checkpoint encoding scheme are shown in Figure 5.27. Compared to the graphs for standard checkpoint encoding in Figure 5.8 the illegal checkpoint encoding scheme has a more balanced coverage with branches getting covered more often and no real division between the number of branches getting covered more often and less often. It is also more balanced than that of checkpoint encoded random testing with no uncovered branches for 10 tests with anti-random testing.

Table 5.30: Coverage of each branch in STRMAT using anti-random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
18	`while` **(c != '\n')**	7	10	22	30
20	`if` **(i $\leq$ tmax)**	7	4	22	14
39	`while` **(c != '\n')**	8	10	18	30
41	`if` **(i $\leq$ pmax)**	8	6	23	16
71	`if` **(textlen == 0)**	3	7	8	22
76	`if` **(patlen == 0)**	1	6	5	17
81	`while (` **(patpos $\leq$ patlen)** `&&` **(textpos $\leq$ textlen))**	6	2	17	6
81	`while ((patpos` $\leq$ `patlen) &&` **(textpos $\leq$ textlen))**	6	4	17	11
83	`if` **(pattern[patpos] == text[textpos])**	4	4	10	12
95	`if` **(patpos > patlen)**	2	4	6	11

The hard to test branches for anti-random testing with illegal checkpoint encoding (30 tests) are

- `if` **(textlen == 0)**

- `if` **(patlen == 0)**
- `while (` *(patpos* $\leq$ *patlen)* `&& (textpos` $\leq$ `textlen))`
- `if` **(patpos > patlen)**

The hard to test branches are mostly the same as with checkpoint encoded random testing. However the branches are covered more often indicating that anti-random testing coverage is more balanced. The branch on line 83 is on the borderline of hard to test for anti-random testing, it is much improved over checkpoint encoded random testing precisely since anti-random testing samples the partitions in a more uniform manner. Thus the pattern does occur in the text string more often even if not completely.

5.3.2.3 TRIANGLE Program

The checkpoint encoding of illegal partitions in the TRIANGLE program is simpler. All the cases in which the sides of a triangle that are input, do not form a triangle are encoded. The basic condition for an illegal triangle is when any two sides are less than or equal to the third. This is the triangle type partition. The "Scalene" triangle case acts as a sort of boundary case. Another partition is the value of the sides themselves. This partition is equally represented by incorrect values (i.e. negative, 0, Not A Number etc.) and valid range of values. Without the valid range the illegal triangle cases cannot be tested. The illegal checkpoint encoding proposed for the TRIANGLE program is given in Table 5.31.

Table 5.31: Illegal checkpoint encoding scheme for TRIANGLE.

Field	Bits	Value	Significance
Triangle Type	b3,b2,b1,b0	1111	a+b $\leq$ c, a != b (Not a triangle)
		0111	a+b $\leq$ c, a = b
		1101	b+c $\leq$ a, b != c
		0101	b+c $\leq$ a, b = c
		0011	a+c $\leq$ b, a != c
		1011	a+c $\leq$ b, a = c
		0100	a+b = c, a != b
		1100	a+b = c, a = b
		0010	b+c = a, b != c
		0011	b+c = a, b = c
		0110	a+c = b, a != c
		1110	a+c = b, a = c
		rest	a != b !=c (Scalene)
Sides of triangle (a,b,c)	b7,b6,b5,b4	0000	0
		1111	NaN
		1110,0111	(float) [-Max, +Max]
		0001,0010	(negative) [-Max, -1]
		0100,1000	(negative) [-Max, -1]
		rest	[1, Max]

The overall branch coverage graph of TRIANGLE with the illegal checkpoint encoding scheme is shown in Figure 5.28. Compared to the branch coverage graph of the legal checkpoint encoding scheme (Figure 5.19) the illegal checkpoint encoding results in higher coverage. This is not, however as high as the standard checkpoint encoding achieves (Figure 5.9). Checkpoint encoded random testing performs as well as anti-random testing with illegal checkpoint encoding. The difference in performance between the two techniques is much smaller in the results for illegal checkpoint encoding as compared to the other two checkpoint encoding schemes.

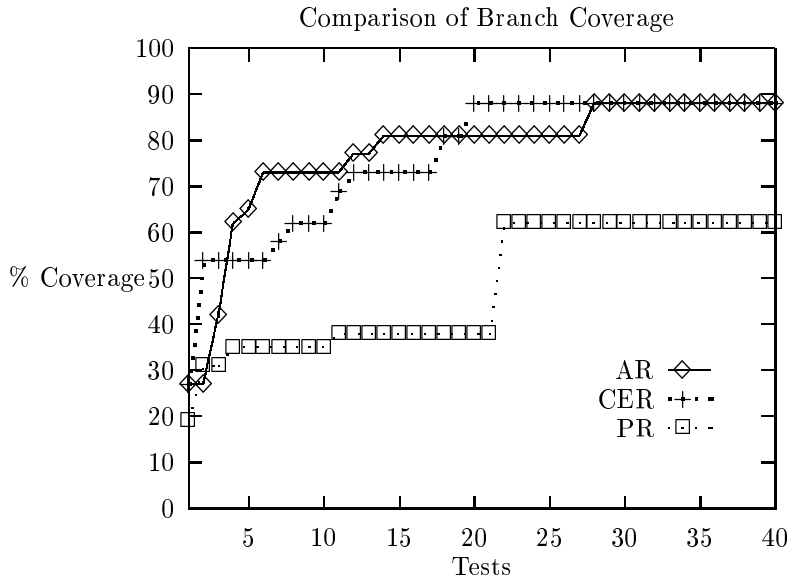


Figure 5.28: TRIANGLE program

The ECN graphs for the *Triangle* programs are shown below

1. Checkpoint Encoded Random Testing.

The ECN graphs for checkpoint encoded random testing with the illegal checkpoint encoding scheme are shown in Figure 5.29. For 10 tests checkpoint encoded random testing cannot cover 10 branches. A cluster of branches are covered 5 times. No branch is covered by more than 5 tests. Apart from the uncovered branches there are few hard to cover branches. With 30 tests, the results differ a lot. Only 3 branches are uncovered but a majority of the branches are hard to cover. This would seem to indicate that with more tests the branches that were not covered were converted to the hard to cover state. Few branches are covered by a large proportion of the test suite. This is as much due to the nature of the TRIANGLE program as

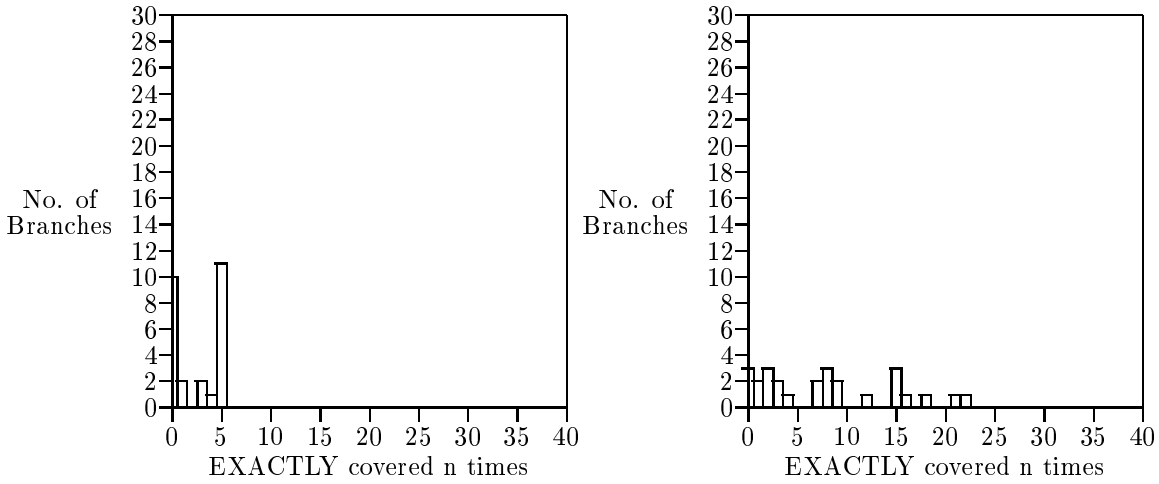


Figure 5.29: Triangle program : Checkpoint Encoded Random Testing : 10 & 30 Tests

it is due to the checkpoint encoding scheme used.

Table 5.32: Coverage of each branch in TRIANGLE using checkpoint encoded random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
13	`if (a == b)`	5	5	9	21
15	`if (a == c)`	5	5	8	22
17	`if (b == c)`	5	5	15	15
23	`if (match == 0)`	5	5	12	18
25	`if ((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	1	4	3	9
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	1	3	3	7
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	3	3	4
30	`else if (match == 1)`	0	5	2	16
33	`if ((a+b) ≤ c)`	0	0	2	0
38	`else if (match == 2)`	0	5	1	15
40	`if ((a+c) ≤ b)`	0	0	1	0
45	`else if (match == 3)`	0	5	8	7
47	`if ((b+c) ≤ a)`	0	0	8	0

Of the total 26 branches, 19 of them are hard to cover. They are to numerous to list. Basically all the branches covered less than 10 times (for 30 tests) are hard to cover. Most of them are the branches that are nested and occur deeper in the program logic. As discussed before the nesting level in the TRIANGLE program makes it hard to cover branches deep with in the program (both in terms of nesting level and program logic) many times. Also with illegal checkpoint encoding the valid branches are not triggered

2. Anti Random Testing.

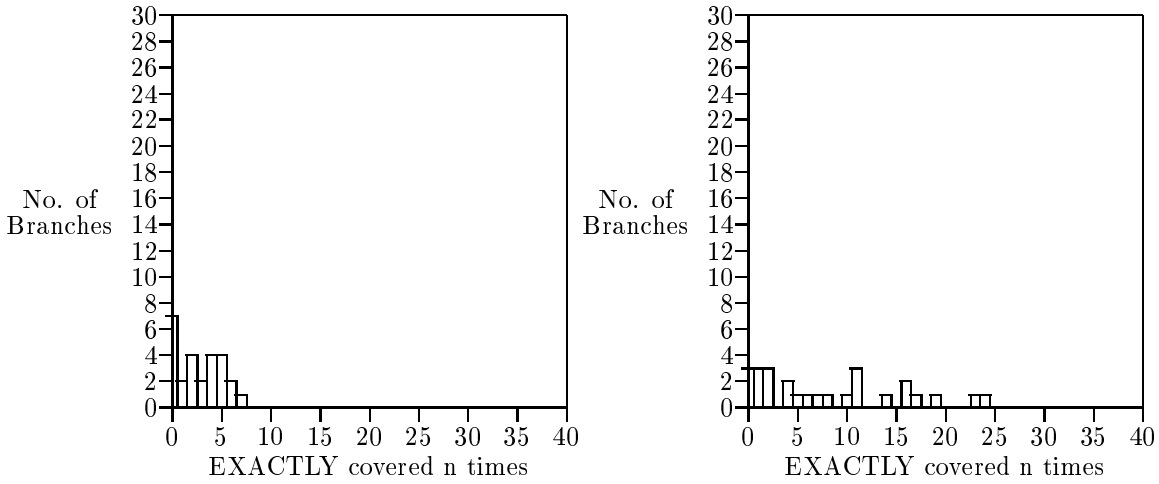


Figure 5.30: Triangle program : Anti Random Testing : 10 & 30 Tests

The ECN graphs for anti-random testing with the illegal checkpoint encoding are shown in Figure 5.30. The behaviour is similar to that shown by checkpoint encoded random testing but is more uniform. Anti-random testing has only 7 uncovered branches with 10 tests as compared to 10 for checkpoint encoded random testing. However it gas more hard to cover branches. With 30 tests the number of uncovered branches is 3, same as that of checkpoint encoded random testing. The number of hard to cover branches is less. Thus the results for anti-random testing are more stable than checkpoint encoded random testing.

Table 5.33: Coverage of each branch in TRIANGLE using anti-random testing.

Line No.	Branch	10 Tests		30 Tests	
		True	False	True	False
13	`if (a == b)`	4	6	7	23
15	`if (a == c)`	3	7	6	24
17	`if (b == c)`	5	5	16	14
23	`if (match == 0)`	4	6	11	19
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	4	1	10
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	0	4	2	8
25	`if (((a+b) ≤ c) || ((b+c) ≤ a) || ((a+c) ≤ b))`	2	2	4	4
30	`else if (match == 1)`	1	5	2	17
33	`if ((a+b) ≤ c)`	1	0	2	0
38	`else if (match == 2)`	0	5	1	16
40	`if ((a+c) ≤ b)`	0	0	1	0
45	`else if (match == 3)`	2	3	11	5
47	`if ((b+c) ≤ a)`	2	0	11	0

Again there are too many hard to cover branches to list separately. Of 26 total branches, 15 branches are hard to cover (for 30 tests). Anti-random testing coverage pattern is much the same as with checkpoint encoded random testing but is quantitatively better. With fewer tests

it has better coverage and always has fewer hard to cover branches than checkpoint encoded random testing.

5.3.2.4 Summary

The result of of using the illegal checkpoint encoding scheme for all 3 programs are summarized in table 5.34. These results are in comparison with random testing with standard checkpoint encoding. Thus the "Not Covered" column indicates the number of branches that did not get covered using illegal checkpoint encoding scheme. The "Hard to Cover" column gives the number of branches that were covered but are classified as "hard to cover" branches since they were covered by less than 30% of the tests. The "Previously Not Covered" and "Previously Hard to Cover" columns give the number of covered branches that were previously not covered; and easy to cover branches that were previously hard to cover, respectively, as compared to random testing with the standard checkpoint encoding scheme. That is, those branches that were not covered or were considered hard to cover for that program using random testing. Indirectly these results help contrast the results of using checkpoint encoding with illegal partitions instead of standard checkpoint encoding.

Table 5.34: Summary of results using the illegal checkpoint encoding scheme.

Program	Total # Branches	# Not Covered	# Hard to Cover	# Previously Not Covered	# Previously Hard to Cover
		Checkpoint Encoded Random Testing			
FINDNO	26	2	18	2	0
STRMAT	20	0	5	0	1
TRIANGLE	26	3	15	8	4
		Anti-Random Testing			
FINDNO	26	2	18	2	0
STRMAT	20	0	5	0	1
TRIANGLE	26	3	15	8	4

First comparing the two test generation methods, there is no quantitative difference between checkpoint encoded random and anti-random testing. They are virtually the same. Comparing with random testing results both methods have fewer uncovered branches. hey manage to cover branches that were not covered by random testing (Covered). The number of hard to cover branches increase a lot. Still a few branches that were hard to cover in random testing were made easy to cover with illegal checkpoint encoding (Easy).

Comparing the three checkpoint encoding schemes, illegal checkpoint encoding is (as expected) the opposite of legal checkpoint encoding (Table 5.24). The number of hard to detect branches are higher but the number of uncovered branches decrease. Thus standard checkpoint encoding seems to form the middle ground with legal and illegal checkpoint encoding forming the two ends of the

spectrum.

5.4 Implications for Checkpoint Encoding

5.4.1 Analysis of results

The experiment to compare different checkpoint encoding methods used two different checkpoint encoding schemes (*legal* and *illegal*) with two different test data generation methods (checkpoint encoded random and anti-random) for three programs. The reference point for the experiment was the standard checkpoint encoding scheme used for these programs (Yin, Lebne-Dengel, and Malaiya 1997) and the test data generation method was random testing. Random testing is important since it allows a reasonable conclusion to be drawn about the effectiveness of these methods in general instead of just versus other checkpoint encoding methods. The ECN graphs show clearly how often the branches are covered and gives a visual idea of the hard to test coverage of a program. The tables (5.14, 5.24, 5.34) summarize the results.

First compare the different checkpoint encoding schemes. This is an inexact comparison:the programs are too small and too few. Also the standard checkpoint encoding is not the result of a specific partitioning strategy but was evolved to maximize coverage. Therefore the coverage achieved is less important than any trends that can be observed. The ECN graphs show that for all programs the legal checkpoint encoding schemes cover the easily testable branches and cover them repeatedly. The ECN graphs have all the branches clustered at the beginning and the end of the graph i.e. either the branch is not covered or it is covered by a majority of the test cases. The only programs where this differs is the TRIANGLE program which is not as easy to cover as the other two. looking t the overall coverage the illegal checkpoint encoding schemes have better coverage than the legal checkpoint encoding schemes but the initial rate of increase of branch coverage is not as high as with the legal checkpoint encoding schemes.

Comparing with the standard checkpoint encoding scheme there are two aspects. On the coverage behaviour the legal checkpoint encoding scheme has more in common with the standard checkpoint encoding scheme. This is to be expected since the standard checkpoint encoding schemes used followed a traditional approach of allocating more test cases towards legal partitions. While with overall branch coverage illegal checkpoint encoding has results similar to the standard checkpoint encoding scheme. Again this is because while standard checkpoint encoding schemes do not explore the illegal input partitions as much, they sample them enough to cover certain exception conditions i.e. they do not ignore them completely as in legal checkpoint encoding. To conclude: illegal checkpoint encoding scheme gives higher branch coverage but the number of hard to cover branches

are higher than with legal checkpoint encoding.

Now compare the different test data generation methods. Random or pseudo-random testing still works out well covering the easy to test branches. Also since it is a different test generation mechanism than either of the two using the checkpoint encoding scheme, it is useful in covering aspects of the program that might otherwise be missed out by the checkpoint encoding scheme. Random testing performs an important function; with minimal effort and cost it can cover the simpler branches and isolate the hard to test branches. The coverage behaviour of random testing is relatively similar to that of checkpoint encoded random testing rather than anti-random testing. This is to be expected. It is also similar to the ECN graph behaviour of the legal checkpoint encoding scheme in that there is division between branches that are hard to cover and those that are easily covered. Though the the division between the two sets of branches is not so stark as with legal checkpoint encoding.

The anti-random testing method is better than the checkpoint encoded random method with respect to improving the "testability" of the program i.e. the coverage behaviour (ECN graphs) of anti-random testing is better than checkpoint encoded random testing. By sampling the partitions encoded by the checkpoint encoding scheme uniformly, the ECN measure shows that it covers branches more uniformly than the checkpoint encoded random method. It may not cover some branches many times but does try to cover more branches. This effect is however more prominent given more number of vectors. With smaller number of vectors used anti-random testing performs poorly. This effect is visible more in illegal and standard checkpoint encoding schemes than in legal checkpoint encoding schemes. In overall branch coverage anti-random testing achieves as good coverage as checkpoint encoded random testing. More often it achieves higher coverage and in fewer number of test cases. Again the conclusion is that anti-random testing is better than checkpoint encoded testing especially for hard to test programs and with illegal checkpoint encoding (i.e. when trying to cover hard to test branches).

5.4.2 A Proposed Checkpoint Encoding Procedure

The results of the experiment show that with the illegal checkpoint encoding scheme, while the hard to cover branches get covered and in some cases they are covered more frequently than is usual there is one flaw. These branches are not covered or detected by enough number of tests. Most of the branches are covered by far fewer test cases than the total number of test cases used. Even the branches that enclose code that is more frequently exercised by the usual operational profile (corresponding to the "legal" partitions). This hold grave implications for the testability aspects of

the checkpoint encoding scheme. This point is also reiterated by Chan (Chan, Chen, and Tse 1997) in an different context of failure detection.

Both the different encoding schemes represent different ends of the testing spectrum. While the goal is to cover all branches, the almost implicit requirement of modeling the operational requirements should not be ignored. If branches (and therefore code) that get exercised more often during standard operations are not covered in tests more frequently, the chances of detecting failures that have a high probability of occurrence decrease. While failures would occur in error-checking and other rarely used and/or complex code more than in simpler code segments, those segments are rarely used in actual practice. Those failures must be detected but those that occur in more frequently exercised code should logically take higher priority. What is needed is a balanced checkpoint encoding scheme with more test cases being allocated towards illegal partitions and boundary cases. Perhaps the highest proportion of test cases should be allocated to testing the boundaries between partitions as in domain testing (Zeil, Afifi, and White 1992). The actual proportions are difficult to calculate. They could be calculated for different application domains.

This is a simple proposal for calculating a checkpoint encoding scheme for a program.

1. Use specifications of the program to identify the input variables and extract its subdomains.

2. Divide the subdomains into 3 categories : legal partitions, illegal partitions and boundary cases that lie at the junction of the legal and illegal partitions.

3. Rules for test allocation

   - Sample each category at least once
   - If there are several discrete conditions sample each one.
   - If the partition is a range, divide it and sample each segment.
   - Sample several boundary points both on and off the boundary.
   - Calculate the total size of the subdomain as number of legal, illegal and boundary sample points. Calculate the proportion between the three partitions and adjust the sampling to satisfy three constraints.

     - Test cases sampled from each subdomain should be done in a proportion simila to that of it's size vs the size of the entire input domain.

     - Within the subdomain the sampling for the different kinds of partitions should follow the partial order; boundary $\leq$ illegal $\leq$ legal.

- The total number of sample points for each orthogonal subdomain should be a power of 2.

4. Assign binary codes to each sample point. Thus if a partition is sampled several times then each input sample needs to be assigned a binary code. Assign codes having minimum distance to sample points from the same partition.

5. Calculate the number of bits for each subdomain.

6. Repeat for each subdomain.

The rationale for the above proposed method is as follows. The need to use the program specifications to extract the needed partitions has been discussed in earlier chapters. However specifications could mean several options. Here informal specifications could be used. This is as is carried out currently for checkpoint encoding. A better option would be to extract subdomains from formal specifications as proposed in the third chapter. Dividing the subdomains into partitions is actually done as part of step 1. However as part of step 2 they are classified as legal(or valid) partitions, illegal(or invalid) partitions and boundary partitions or cases. This is important because they map onto different parts of the program and should be treated differently. This is classic partition testing theory.

Step 3 gives some heuristics for allocating test cases. They indicate how to sample partitions when they are ranges, discrete values or just a single value. This is based on experience of this thesis on encoding various programs. The sampling of a subdomain is based on different literature as well as the results of the experiment. The proportional sampling rule comes from an extensive body of work in partition testing which includes Weyukar. They prove that effectiveness of partition testing schemes improve with this kind of proportional sampling. The partial order sampling rule is mentioned in Chan (Chan, Chen, and Tse 1997) in a different context. The experimental results also support such an idea. While Chan has a partial order based on the risk of a partition, the same can be extended. Boundary cases have a higher chance of failure than other values (Beizer 1990). From the experiment we know that boundary and illegal cases cover the exception conditions and provide higher coverage. Thus to both cover more branches quickly and to improve chances of detecting failures this scheme is suggested. Legal partitions are also sampled to cover the easy to test branches.

The rest of the procedure is in line with the binary representation of the checkpoint encoding scheme.

Alternatively three different phases of testing could occur (using three different checkpoint encoding schemes) starting with testing legal partitions and ending with testing illegal and exceptional cases. Each phase could end when the coverage plateaus out with no fewer branches being detected. The next phase could then begin where the previous phase left off. The only obstacle to keep in mind is that there should be a method to ensure that branches are covered enough times. The limit could be set the same for all branches or could vary as per a certain operational profile. Yet another method is to test with different methods in sequence. In sequence random, checkpoint encoded random and anti-random testing could be used. Each method could be used after the coverage achieved by the previous one plateaus

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The first part of the thesis proves that checkpoint encoding process can be automated. This automation can best be carried out by starting from a specification. It is simple and this thesis demonstrates the automation of checkpoint encoding by extracting partitions of the input domain of a software system from its Z specification. The use of Z specification is not incidental. For automation to exist an unambiguous representation of the program specification must also exist. The solution lies in a formal specification. Z is chosen as an example due to its widespread use and understanding as a general purpose specification technique. Any other specification technique could be substituted, assuming its suitability for that software, without any loss of applicability. The only other issue in the automation process is the test allocation problem i.e. the number of bits to be associated with each orthogonal partition and the distribution with the partition of correct and incorrect input test data. A simple heuristic is proposed since the understanding of how a good checkpoint encoding scheme is constructed, prior to this thesis, was poor. A later chapter deals with this issue at length. For automation this is not an issue; as long as there is a set of rules the automation process will implement it.

There are also other advantages of generating a checkpoint encoding scheme from specifications. By implementing the procedure proposed in this thesis it is found that checkpoint encoding schemes generated are more detailed and probe all parts of the input domain more thoroughly and systematically than is done manually. This is consistent with the common experience with automation of testing; what is a tedious process of considering every possible case when generating the checkpoint encoding scheme manually, is now done far better by a automated process. Since the checkpoint encoding scheme is the first phase in anti-random testing, its automation would mean that the anti-random testing method can now be automated which gives it advantages over other testing

methods.

The second part of the thesis deals with the applicability of the checkpoint encoding scheme to complex software systems. Complexity in the thesis is discussed in terms of a large finite state machine. Using formal specifications allows the state machine of a system to be derived. Since the state machine can be extracted a modified checkpoint encoding scheme is proposed to utilize this state machine information to improve structural coverage. This scheme aims to use less computational requirements and a small test suite by encoding subsequences. The results show it is effective in both aspects.

Lastly this thesis looks at how to improve the checkpoint encoding with respect to hard to cover branches. This goal would also imply that failures would be better detected by any improvement. An experiment is carried out to improve the understanding of how checkpoint encoding schemes work. First the the effectiveness of the standard checkpoint encoding scheme is examined using a new graphical metric i.e. exactly how many tests cover or detect each branch. Then two different schemes are experimented with. The schemes emphasize encoding of either legal or valid and illegal or invalid partitions. The behaviour of each scheme vis-a-vis ability of cover new branches and to make hard to cover branches easy to cover is used to propose a new checkpoint encoding scheme. This was a simple experiment. The aim is not to extract detailed recommendations but to provide a basis on which further experiments could be carried out. But the results allow formulation of guidelines for an effective checkpoint encoding scheme.

This thesis had a simple objective. To improve the understanding and effectiveness of the checkpoint encoding process. It would follow then that any method using the checkpoint encoding scheme would also be improved. The understanding of how the checkpoint encoding process can achieve better coverage is improved by experimenting with different encoding schemes. The effectiveness is improved by an automation framework that uses formal specifications. Formal specifications allow checkpoint encoding to be extended to use state information, allowing more effective application of checkpoint encoding in testing complex systems. The three ideas were studied separately to achieve the common objective. More work needs to be done to combine the lessons learnt from these steps to improve checkpoint encoding.

6.2 Future Work

There remains more work to be done in automation of checkpoint encoding. more experiments with automating checkpoint encoding from Z using different systems of increasing complexity. This automation technique has been shown manually. It should be used to implement an automated tool.

Some tools exist that could help including Isabelle. Other specifications languages (e.g. SDL) also hold promise for automation.

The need to use other specification languages or techniques is more important considering that an extension is to extract the state machine from the specification. Extracting state machines is cumbersome at best using Z since it is not ideally suited for this purpose. Once the state machine is extracted then checkpoint encoding can use this information. Currently the thesis proposes a simple method as a first step. Other methods could be tried for checkpoint encoding using state machines.

The ECN graphical metric is one way of understanding how effective a testing scheme is in understanding the coverage behaviour of a testing method. Other proposed graphical metrics could be used. The experiment with different checkpoint encoding schemes needs to be repeated with a larger variety of programs and different checkpoint encoding schemes. Detailed analysis of which scheme is effective against different programs and different kinds of branches could improve the understanding of hard to test branches. Checkpoint encoding with the new recommendations should be used in an automated framework.

Appendix A

Program Listings

A.1 FIND program

```
1   #include <stdio.h>
2   #define max    64
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    void find(int, int);
13
14    mystr = (char *)malloc (80);
15    scanf("%d", &n);
16    gets(mystr);
17    scanf("%d", &f);
18    if ((f<=0) || (f>n))
19    {
20      printf("f cannot be negative or 0; here f=%d\n", f);
21      exit(0);
22    }
23    gets(mystr);
24    for(i=1;i<=n;i++)
25    {
26      float z;
27      int x;
28
29      scanf("%f", &z);
30      x = (int)z;
31      if (x==z)
```

```
32          a[i] = x;
33        else {
34          printf("Array value is illegal. Choose integer array values!\n");
35          exit(0);
36        }
37        gets(mystr);
38      }
39      find(n, f);
40      printf("%5d\n", n);
41      printf("%5d\n", f);
42
43      for(i=1;i<=n;i++)
44       printf("%5d\n", a[i]);
45    }
46
47    find(n, f)
48    int n;
49    int f;
50    {
51      int m, ns, i, j, w;
52      BOOLEAN b;
53      b = false;
54      m = 1;
55      ns = n;
56      while ((m < ns) || b)
57      {
58         if (!b)
59         {
60           i = m;
61           j = ns;
62         }
63         else
64           b = false;
```

```
65        if (i>j)
66        {
67           if (f>j)
68           {
69              if (i>f)
70                m = ns;
71              else
72                m = i;
73           }
74          else
75            ns = j;
76        }
77        else
78        {
79          while (a[i] < a[f])
80               i = i+1;
81          while (a[f] < a[j])
82               j = j-1;
83          if (i <= j)
84          {
85            w = a[i];
86            a[i] = a[j];
87            a[j] = w;
88            i = i+1;
89            j = j-1;
90          };
91          b = true;
92        }
93      }
94    }
```

A.2 STRMAT program

```
1   #include <stdio.h>
```

```
2  #define tmax 80
3  #define pmax 3
4
5  extern char text[];
6  extern char pattern[];
7
8  main()
9  {
10    int c,i;
11    int textlen,patlen;
12    int result;
13    char text[tmax];
14    char pattern[pmax];
15    textlen = 0;
16    i=1;
17    c=fgetc(stdin);
18    while(c != '\n')
19    {
20     if(i<=tmax)
21     {
22      text[i]=c;
23      i++;
24      c=fgetc(stdin);
25      textlen = i-1;
26     }
27     else
28     {
29       i++;
30       c=fgetc(stdin);
31      }
32    };
33    printf("\n");
34    printf("textlen=%d\n", textlen);
```

```
35
36      patlen=0;
37      i=1;
38      c=fgetc(stdin);
39      while(c != '\n')
40      {
41        if(i<=pmax)
42        {
43          pattern[i]=c;
44          printf("%c",c);
45          i++;
46          c=fgetc(stdin);
47          patlen = i-1;
48        }
49        else
50        {
51          i++;
52          c=fgetc(stdin);
53        }
54      };
55      printf("\n");
56      printf("patlen=%d\n", patlen);
57      result = stringmatch2(pattern, text, patlen, textlen);
58      printf("%d\n", result);
59  }
60
61  int stringmatch2(pattern, text, patlen, textlen)
62  char pattern[];
63  char text[];
64  int patlen, textlen;
65  {
66    int patpos, textpos;
67
```

```
68    patpos = 1;
69    textpos = 1;
70
71    if(textlen == 0)
72      return (0);
73    else
74      ;
75
76    if(patlen == 0)
77      return(1);
78    else
79      ;
80
81    while ((patpos <= patlen) && (textpos <= textlen))
82    {
83      if(pattern[patpos] == text[textpos])
84        {
85          textpos = textpos + 1;
86          patpos = patpos + 1;
87        }
88      else
89        {
90          textpos = (textpos-patpos) + 2;
91          patpos = 1;
92        }
93     };
94
95     if(patpos > patlen)
96       return(textpos-patlen);
97     else
98       return(0);
99  }
```

A.3 TRIANGLE program

```
1  #include "stdio.h"
2  #include "math.h"
3
4  main()
5  {
6    int a, b, c, match;
7
8    printf("Please input three integers separated by Comma for sides of triangle.\n");
9    scanf("%d, %d, %d", &a, &b, &c);
10
11   match = 0;
12
13   if (a==b)
14      match = match+1;
15   if (a==c)
16      match = match+2;
17   if (b==c)
18      match = match+3;
19
20   printf("a=%d, b=%d, c=%d\n", a, b, c);
21   printf("match=%d\n", match);
22
23   if (match == 0)
24   {
25     if (((a+b) <= c) || ((b+c) <= a) || ((a+c) <= b))
26       printf("Not a triangle\n");
27     else
28       printf("Triangle is Scalene\n");
29   }
30   else if (match == 1)
31   {
```

```
32
33        if ((a+b) <= c)
34         printf("Not a triangle\n");
35        else
36         printf("Triangle is Isosceles\n");
37     }
38     else if (match == 2)
39     {
40        if ((a+c) <= b)
41         printf("Not a triangle\n");
42        else
43         printf("Triangle is Isosceles\n");
44     }
45     else if (match == 3)
46     {
47        if ((b+c) <= a)
48          printf("Not a triangle\n");
49        else
50          printf("Triangle is Isosceles\n");
51     }
52     else
53        printf("Triangle is Equilateral\n");
54   }
55
```

Appendix B

ATM Z Specification

B.1 ATM Z specification

$$[CARD, PIN, ACC]$$

$$\begin{aligned} MENUCHOICE ::= & \; DEPOSIT \\ | & \; WITHDRAW \\ | & \; QUERY \\ | & \; CANCEL \end{aligned}$$

$$\begin{aligned} DISPLAY ::= & \; ENTERPIN \\ | & \; INVALIDCARD \\ | & \; ENTERMENUCHOICE \\ | & \; ENTERPIN - 2ndTRY \\ | & \; ENTERPIN - LASTTRY \\ | & \; INVALIDPIN \\ | & \; ENTERACCT\&DEPOSITAMT \\ | & \; ENTERACCT\&WITHDRAWAMT \\ | & \; ENTERACCT \\ | & \; AMTDEPOSITED \\ | & \; INVALIDACCT \\ | & \; AMTWITHDRAWN \\ | & \; OVERDRAWLIMITEXCEEDED \\ | & \; PINLIMITEXCEEDED \\ | & \; BALANCEDISP \\ | & \; CARDRETURN \end{aligned}$$

$$\begin{aligned} STATUSVAL ::= & \; ACCEPTCARD \\ | & \; ACCEPTPIN \\ | & \; PINFAIL1 \\ | & \; PINFAIL2 \\ | & \; MENU \\ | & \; DEPOSIT \\ | & \; WITHDRAW \\ | & \; QUERY \\ | & \; CANCEL \end{aligned}$$

Accounts

$$accnos : \mathbb{P}\, ACC$$
$$Amt : ACC \pfun \mathbb{Z}$$
$$OLimit : ACC \pfun \mathbb{N}$$

$$\mathrm{dom}\, Amt = accnos$$
$$\mathrm{dom}\, OLimit = accnos$$

ATMPins

$Accounts$
$atmpin : \mathbb{P}\, PIN$
$pinacct : PIN \leftrightarrow ACC$
$pinlimit : PIN \pfun \mathbb{N}$

$\mathrm{dom}\, pinacct = atmpin$
$\mathrm{ran}\, pinacct = accnos$
$\mathrm{dom}\, pinlimit = atmpin$
$\forall p \in \mathrm{dom}\, pinacct \bullet \#\, \mathrm{ran}\{p \lhd pinacct\} \geq 1$
$\forall a \in \mathrm{ran}\, pinacct \bullet \#\, \mathrm{dom}\{pinacct \rhd a\} \leq 1$

ATMCards

$ATMPins$
$atmcard : \mathbb{P}\, CARD$
$cardpin : CARD \pfun PIN$

$\mathrm{dom}\, cardpin = atmcard$
$\mathrm{ran}\, cardpin = atmpin$

ATM

$Accounts$
$ATMPins$
$ATMCards$
$status : STATUSVAL$
$card : CARD$
$pin : PIN$

$\{card \mapsto pin\} \in cardpin$

$ATMOp \mathrel{\widehat=} ATMInit \fatsemi Transaction$
$Transaction \mathrel{\widehat=} AcceptCard \fatsemi AcceptPin \fatsemi MenuOp \fatsemi Transaction$
$MenuOp \mathrel{\widehat=} AcceptMenuChoice \fatsemi (Op \fatsemi MenuOp) \lor Cancel$
$Op \mathrel{\widehat=} Deposit \lor Withdraw \lor Query$

ATMInit

ATM'

$\forall a \in \mathrm{ran}\, Amt \bullet a' \geq 0$
$status' = ACCEPTCARD$
$card' = \varnothing$
$pin' = \varnothing$

AcceptCard

ΔATM
$custcard? : CARD$
$msg! : DISPLAY$

$(status = ACCEPTCARD \land custcard? \in atmcard \land status' = ACCEPTPIN \land card' = custcard?$
$\land\ msg! = ENTERPIN)$
$\bigvee (custcard? \notin atmcard \land status' = ACCEPTCARD \land msg! = INVALIDCARD)$

*AcceptPin*1

ΔATM
$custpin? : PIN$
$msg! : DISPLAY$

$(status = ACCEPTPIN \land \{card \mapsto custpin?\} \in cardpin \land status' = MENU \land pin' = custpin?$
$\land\ msg! = ENTERMENUCHOICE)$
$\bigvee (status = ACCEPTPIN \land \{card \mapsto custpin?\} \notin cardpin \land status' = PINFAIL1 \land$
$msg! = ENTERPIN - 2ndTRY)$

*AcceptPin*2

ΔATM
$custpin? : PIN$
$msg! : DISPLAY$

$(status = PINFAIL1 \land \{card \mapsto custpin?\} \in cardpin \land status' = MENU \land pin' = custpin?$
$\land\ msg! = ENTERMENUCHOICE)$
$\bigvee (status = PINFAIL1 \land \{card \mapsto custpin?\} \notin cardpin \land status' = PINFAIL2 \land$
$msg! = ENTERPIN - LASTTRY)$

*AcceptPin*3

ΔATM
$custpin? : PIN$
$msg! : DISPLAY$

$(status = PINFAIL2 \land \{card \mapsto custpin?\} \in cardpin \land status' = MENU \land pin' = custpin?$
$\land\ msg! = ENTERMENUCHOICE)$
$\bigvee (status = PINFAIL2 \land \{card \mapsto custpin?\} \notin cardpin \land status' = ACCEPTCARD \land$
$msg! = INVALIDPIN)$

AcceptMenuChoice

ΔATM
$custchoice? : MENUCHOICE$
$msg! : DISPLAY$

$(status = MENU \land custchoice? = DEPOSIT \land status' = DEPOSIT \land msg! = ENTERACCT\&DEPOS$
$\bigvee (status = MENU \land custchoice? = WITHDRAW \land status' = WITHDRAW \land msg! = ENTERACCT\&$
$\bigvee (status = MENU \land custchoice? = QUERY \land status' = QUERY \land msg! = ENTERACCT)$
$\bigvee (status = MENU \land custchoice? = CANCEL \land status' = CANCEL \land msg! = TRANSCANCEL)$

Deposit

ΔATM
$custacc? : ACC$
$amtdep? : \mathbb{N}$
$msg! : DISPLAY$

$status = DEPOSIT \land$
$((custacc? \in \mathrm{ran}\{pin \lhd pinacct\} \land Amt' = Amt \oplus \{custacc? \mapsto (Amt\ custacc? + amtdep?)\} \land$
$status' = MENU \land msg! = AMTDEPOSITED)$
$\bigvee (custacc? \notin \mathrm{ran}\{pin \lhd pinacct\} \land status' = MENU \land msg! = INVALIDACCT))$

Withdraw

ΔATM
$custacc? : ACC$
$amtwith? : \mathbb{N}$
$msg! : DISPLAY$

$status = WITHDRAW \land$
$((custacc? \in \text{ran}\{pin \lhd pinacct\} \land$
$\quad ((amtwith? \leq pinlimit\ pin \land$
$\quad\quad ((amtwith? \leq (Amt\ custacc? + OLimit\ custacc?) \land msg! = AMTWITHDRAWN \land$
$\quad\quad Amt' = Amt \oplus \{custacc? \mapsto Amt\ custacc? + OLimit\ custacc?\} \land status' = MENU)$
$\quad\quad \bigvee (amtwith? > Amt\ custacc? + OLimit\ custacc? \land msg! = OVERDRAWLIMITEXCEEDED \land$
$\quad\quad status' = MENU)))$
$\quad \bigvee (amtwith? > pinlimit\ pin \land msg! = PINLIMITEXCEEDED \land status' = MENU)))$
$\bigvee (custacc? \in \text{ran}\{pin \lhd pinacct\} \land status' = MENU \land msg! = INVALIDACCT))$

Query

ΔATM
$custacc? : ACC$
$amtbal! : \mathbb{N}$
$msg! : DISPLAY$

$status = QUERY \land$
$((custacc? \in \text{ran}\{pin \lhd pinacct\} \land amtbal! = Amt\ custacc? \land status' = MENU \land msg! = BALANCEDIS$
$\bigvee (custacc? \notin \text{ran}\{pin \lhd pinacct\} \land status' = MENU \land msg! = INVALIDACCT))$

Cancel

ΔATM
$msg! : DISPLAY$

$status = CANCEL \land msg! = CARDRETURN \land status = ACCEPTCARD \land card' = \varnothing \land pin' = \varnothing$

Appendix C

OTCS Specification and Program Listing

C.1 OTCS Z Specification

$[TANKER, BERTH]$

$OTCSResp ::= ok \mid wait \mid move_t\, anker \mid known_t\, anker \mid not_a\, t_b\, erth$

$berths : \mathbb{P}\, BERTH$

Otcsys

$waiting : \text{seq}\ TANKER$
$docked : TANKER \rightarrowtail BERTH$
$known : \mathbb{P}\, TANKER$

$\text{ran}\ waiting \cap \text{dom}\ docked = \varnothing$
$\#waiting = \#(\text{ran}\ waiting)$
$\#waiting > 0 \Rightarrow \text{ran}\ docked = berths$
$\text{ran}\ docked \subseteq berthsknown = \text{ran}\ waiting \cup \text{dom}\ docked$

Initotc

$Otcsys'$

$waiting' = \langle\rangle$
$docked' = \varnothing$

$Arriveok == Arriveq \lor Arrivenoq$
$Leaveok == Leaveq \lor Leavenoq$
$Arrive == Arriveok \lor Knownt$
$Leave == Leaveok \lor Notatberth$
$Query == Queryb \lor Queryq$
$OTCSOp == Initotc;\ Arrive;\ (Arrive \lor Leave \lor Query)$

*Arrive*0

$\Delta Otcsys$
$a? : TANKER$

$a? \notin known$

Arrivenoq

$Arrive0$
$b! : BERTH$
$r! : OTCSResp$

$\text{ran}\ docked \neq berths$
$r! = ok$
$b! \in berths \ \land docked$
$docked' = docked \oplus a? \mapsto b!$
$waiting' = waiting$

Arriveq

$Arrive0$

$r! : OTCSResp$

$\text{ran}\ docked = berths$

$r! = wait$

$docked' = docked$

$waiting' = waiting \frown a?$

Leave0

$\Delta Otcsys$

$a? : TANKER$

$a? \in \text{dom}\ docked$

Leavenoq

$Leave0$

$r! : OTCSResp$

$waiting = \langle\rangle$

$docked' = a? \lhd docked$

$waiting' = waiting$

$r! = ok$

Leaveq

$Leave0$

$c! : TANKER$

$b! : BERTH$

$r! : OTCSResp$

$waiting \neq \langle\rangle$

$c! = head\ waiting$

$b! = dockeda?$

$docked' = (a? \lhd docked) \oplus c! \mapsto b!$

$waiting' = tail\ waiting$

$r! = move_t anker$

Queryq

$\Xi Otcsys$

$q! : \text{seq}\ TANKER$

$q! = waiting$

Queryb

$\Xi Otcsys$

$f! : TANKER \rightarrowtail BERTH$

$f! = docked$

Knownt

$\Xi Otcsys$
$a? : TANKER$
$r! : OTCSResp$

$a? \in known$
$r! = known_t anker$

Notatberth

$\Xi Otcsys$
$a? : TANKER$
$r! : OTCSResp$

$a? \notin \mathrm{dom}\ docked$
$r! = not_a t_b erth$

C.2 OTCS program

```
1   # include <stdio.h>
2   # include <stdlib.h>
3   # define BERTHS  5
4   # define TANKERS 99
5
6   int dock[BERTHS];
7   int queue[TANKERS+1];
8   int docked = 0; /* actual no of berths in use */
9   int numqueued = 0; /* no of tankers queued */
10  int arrive_tanker();
11  void leave_tanker();
12
13  main(argc, argv)
14  int argc;
15  char *argv[];
16  {
17   char option = ' ';
18   int tanker = 0, known = 0, i = 0, berth = 0, count = 0;
19
20   if (argc < 2) {
21   printf ("otcs [param] [param] .... Q, \n where params = A[tanker number] | L[tanker numbe
```

```
22    printf(" A = tanker arrive, L = tanker leave, T = query tankers, B = query berths, Q = Qu
23    exit(-1); }
24
25    /* set docked, queue clear */
26    for (i=0; i < BERTHS; i++) dock[i] = 0;
27    for (i=0; i < TANKERS+1; i++) queue[i] = 0;
28
29    count = 1;
30    option = argv[count++][0];
31    while ((option != 'Q') && (count < argc))
32    {
33           switch (option)
34           {
35           case 'A' : tanker = atoi(argv[count-1]+1);
36                      known = 0;
37                      if ( (tanker > 0) && (tanker <= TANKERS) )
38                         { for (i=0; i < docked; i++)
39                                   if (dock[i] == tanker) known = 1;
40                           for (i=0; i < numqueued; i++)
41                                   if (queue[i] == tanker) known = 1;
42                           if (known)
43                               printf("KNOWN_TANKER ");
44                           else
45                               arrive_tanker(tanker);
46                         }
47                      else
48                           printf("ERROR ");
49                      break;
50
51           case 'L' : tanker = atoi(argv[count-1]+1);
52                      known = 0;
53                      if ( (tanker > 0) && (tanker <= TANKERS) )
54                         { for (i=0; i < docked; i++)
```

```
55                                  if (dock[i] == tanker)
56                                     { known = 1; berth = i; }
57
58                          if (known)
59                              leave_tanker(berth);
60                          else
61                              printf("NOT_AT_BERTH ");
62                        }
63                  else
64                          printf("ERROR ");
65                  break;
66
67          case 'T' : /* print out queue */
68                  if (queue[0]) printf ("%0.2d", queue[0]);
69                  else { printf("0 "); break; }
70                  for (i=1; i < TANKERS+1; i++)
71                          if (queue[i]) printf (",%0.2d", queue[i]);
72                          else { break; }
73                  printf(" ");
74                  break;
75
76          case 'B' :/* print out dock */
77                  if (dock[0]) printf ("%0.2d", dock[0]);
78                  else   printf ("0");
79                  for (i=1; i < BERTHS; i++)
80                          if (dock[i]) printf (",%0.2d", dock[i]);
81                                  else   printf (",0");
82                  printf(" ");
83                  break;
84
85          default  : if (option != 'Q') printf ("ERROR ");
86                  break;
87          }
```

```
88          option = argv[count++][0];
89   }
90  }
91
92  int arrive_tanker(tanker)
93  int tanker;
94  {
95     if (docked < BERTHS)
96     {
97          dock[docked] = tanker;
98          docked++;
99          printf("OK ");
100         return 1;
101    }
102    else
103    {
104         queue[numqueued] = tanker;
105         numqueued++;
106         printf("WAIT ");
107         return 0;
108    }
109 }
110
111 void leave_tanker(berth)
112 int berth;
113 {
114   int i;
115
116   for (i=berth; i < BERTHS-1; i++)
117     dock[i] = dock[i+1];
118    if (numqueued <= 0)
119    {
120         printf("OK ");
```

```
121             dock[--docked] = 0;
122         }
123         else
124         {
125             dock[BERTHS-1] = queue[0];
126             for (i=0; i < numqueued-1; i++)
127                 queue[i] = queue[i+1];
128             queue[--numqueued] = 0;
129             printf("MOVE_TANKER ");
130         }
131     }
132
```

REFERENCES

Amla, N. and P. Ammann (1992, June). Using z specifications in category partition testing. In Seventh Annual Conference on Computer Assurance (COMPASS 92), Gaithersburg, MD, pp. 3–10.

Ammann, P. and J. Offutt (1994, June). Using formal methods to derive test frames in category-partition testing. In Ninth Annual Conference on Computer Assurance (COMPASS 94), COMPASS, Gaithersburg, MD, pp. 69–80.

Arkko, J., V. Hirvisalo, J. Kuusela, and E. Nuutila (1990). Supporting testing of specifications and implementations. Microprocessing And Microprogramming 30, 297–302.

Balcer, M., W. Hasling, and T. Ostrand (1989). Automatic generation of test scripts from formal specifications. In ACM Symposium on Software Testing, Analysis, and Verification, SIGSOFT 89/TAV3, Software Engineering Notes, Key West, FL, pp. 210–218.

Beizer, B. (1990). Software Testing Techniques (2nd ed.). New York, NY: Van Nostrand Reinhold.

Bernot, G., M. Gaudel, and B. Marre (1991, November). Software testing based on formal specifications : a theory and a tool. Software Engineering Journal, 387–405.

Bertolina, A. and L. Strigini (1995). Using testability measures for dependability assessment. In International Conference on Software Engineering, Number 17 in ICSE, pp. 61 – 70.

Bouge, L., N. Choquet, L. Fribourg, and M.-C. Gaudel (1986, November). Test sets generation from algebraic specifications using logic programming. Journal Of Systems And Software 6(4), 343 – 360.

Bowen, J. P. Formal methods website. http://www.comlab.ox.ac.uk/archive/formal-methods.html.

Bowen, J. P. and M. G. Hinchey (1994, October). Seven more myths of formal methods: Dispelling industrial prejudices. In M. Naftalin, T. Denvir, and M. B. Eds. (Eds.), Industrial Benefits Of Formal Methods, Second International Symposium On Formal Methods In Europe, FME, Barcelona, Spain, pp. 105–117. Springer-Verlag.

Buttle, D. (1995, March). Specification based testing. Master's thesis, Depratment of Computer Science. The University of York.

Chan, F., T. Chen, and T. Tse (1997, October). On the effectiveness of test case allocation schemes in partition testing. Information and Software Technology 39(10), 719–726.

Chen, T. Y. and Y. T. Yu (1994, December). On the relationship between partition and random testing. IEEE Transactions on Software Engineering 20(12), 977–980. Concise papers.

Chen, T. Y. and Y. T. Yu (1996a). Constrainst for safe partition testing strategies. The Computer Journal 39(7), 619–625.

Chen, T. Y. and Y. T. Yu (1996b). A more general sufficient condition for partition testing to be better than random testing. In Information Processing Letters, Volume 57, pp. 145–149. Elsevier.

Chow, T. S. (1978, March). Testing design modeled by finite-state machines. IEEE Transaction on Software Engineering 4(3), 178–186.

Clarke, L., J. Hassell, and D. Richardson (1982, July). A close look at domain testing. IEEE Transactions on Software Engineering 8(4), 380–390.

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

Coleman, D., P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes (1994). Object-Oriented Development: The Fusion Method (1 ed.). Prentice Hall Object Oriented Serie. Englewood Cliffs, NJ: Prentice-Hall.

Committee, S. E. S. (1990). IEEE Std 610.12-1990, Glossary of Software Engineering Terminology. IEEE.

Cusack, E. (1992). Object oriented modelling in z for open distributed systems. In Proc. International Workshop on ODP 1991, Springer-Verlag Workshops In Computing, Berlin. Elsevier Science Publishers (North-Holland).

Demillo, R. and J. Offutt (1993, April). Experimental results from an automatic test case generator. ACM Transactions on software engineering and methodology 2(2), 109–127.

DeMillo, R. A. (1989, May). Test adequacy and program mutation. In 11th Annual Conference On Software Engineering, Volume 11, pp. 355–356. ACM Press.

Dick, J. and A. Faivre (1993, April). Automating the generation and sequencing of test cases from model-based specifications. In FME 93, Industrial Strength Formal Methods, Intl. Symposium of Formal Methods Europe, Volume 670, Odense, Denmark, pp. 268–284.

Duran, J. W. and S. C. Ntafos (1984, April). An evaluation of random testing. IEEE Transactions on Software Engineering 10(4), 438–444.

Elmstrom, R., P. G. Larsen, and P. B. Lassen (1994, September). The ifad vdm-sl toolbox: A practical approach to formal specifications. ACM SIGPLAN Notices 29(9), 77–80.

Fujiwara, S., G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi (1991, June). Test selection based on finite state models. IEEE Transactions on Software Engineering 17(6), 591–603.

Gonenc, G. (1970, June). A method for the design of fault-detection experiments. IEEE Transaction on Computers C-19, 551–558.

Goodenough, J. and S. Gerhart (1975, June). Toward a theory of test data selection. IEEE Transaction on Software Engineering 1(2), 156–173.

Guttag, J. V., J. J. Horning, and J. M. Wing (1985). Larch in five easy pieces. Technical Report 5, DIGITAL Systems Research Center.

Hall, P. (1990, September). Seven myths of formal methods. IEEE Computer 7(5), 11–19.

Hall, P. and R. Hierons (1991). Formal methods and testing. Technical Report 91/16, Computing Department, The Open University.

Hamlet, D. and R. Taylor (1990, December). Partition testing does not inspire confidence. IEEE Transactions on Software Engineering 16(12), 1402–1411.

Hayes, I. (1986, January). Specification directed module testing. IEEE Transactions on software engineering 12(1), 124–133.

Helke, S., T. Neustupny, and T. Santen (1997). Automating test case generation from z specification with isabelle. In International Conference of Z Users (ZUM 97), Volume LNCS 1212, pp. 52–71. Springer-Verlag.

Hierons, R. (1997a). Testing from a finite-state machine : extending invertibility to sequences. The Computer Journal 40(4), 220–230.

Hierons, R. (1997b). Testing from a z specifcation. Software Testing, Verification, and Reliability 7, 19–33.

Horcher, H. and J. Peleska (1995). Using formal specifications to support software testing. Software Quality Journal 4, 309–327.

Horgan, J. and S. A. London (1992). A data flow coverage testing tool for c.

Howden, W. E. and Y. Huang (1995, January). Software trustability analysis. ACM Transactions on Software Engineering and Methodology 4(1), 36 – 64.

Interconnection, I. P. S. O. S. (1989a). ISO/IEC 8807 LOTOS - A Formal Description Technique Based On Temporal Ordering Of Observational Behaviour. Geneva: International Organisation For Standards (ISO).

Interconnection, I. P. S. O. S. (1989b). ISO/IEC 9074 Estelle - A Formal Description Technique Based On An Extended State Transition Model. Geneva: International Organisation For Standards (ISO).

Jackson, M. A. (1975). Principles of program design. In APIC Studies In Data Processing, Volume 12. Boston, MA: Academic Press.

Jones, C. B. (1986). Systematic Software Development Using VDM (1 ed.). Prentice Hall.

Jorgensen, P. C. (1995). Software Testing: A Craftsman's Approach (1 ed.). New York: CRC Press.

Kohavi, Z. (1978). Switching and finite state automata (2 ed.). McGraw Hill Computer Science Series. McGraw Hill.

Kolyang, T., T. Santen, and B. Wolff (1996). A structure preserving encoding of z in isabelle/hol. In Lecture Notes in Computer Science, Volume 1125, pp. 283–298. Springer-Verlag.

Kung-Kiu, L., V. J. Bush, and P. J. Jinks (1994, March). Towards an introductory formal programming course. In 25th SIGCSE Technical Symposium On Computer Science Education, Volume 26, pp. 121–125.

Laycock, G. (1992). Formal specification and testing : a case study. Software Testing, Verification and Reliability 2(1), 7–23.

Logica. Formal methods tools and services. http://public.logica.com/ formaliser/.

Ltd, Y. S. E. (1994). The cadiaetutorial - manual version 2.02t. Technical report, York Software Engineering Limited. http://www.cs.york.edu/ ian/cadiz.

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

Meyer, B. (1985, January). On formalism in specifications. IEEE Software 2(1), 6–26.

Mikk, E. (1995, September). Compilation of z specifications into c for automatic test evaluation. In International Conference of Z Users, Volume 967 of ZUM, Limerick, Ireland, pp. 167–180.

Murray, L., D. Carrington, I. MacColl, J. MacDonald, and P. Strooper (1998, September). Formal derivation of finite state machines for class testing. In International Conference of Z Users (ZUM 98), pp. 42–59.

Murray, L., D. Carrington, I. MacColl, and P. Strooper (1997). Extending test templates with inheritance. In P. Bailes (Ed.), Australian Software Engineering Conference ASWEC'97, pp. 80–87. IEEE Computer Society.

Myers, G. (1979). The Art of Software Testing. New York, NY: John Wiley and Sons.

Nachman, L., K. Saluja, S. Upadhyaya, and R. Reuse (1998, January). A novel approach to random pattern testing for testing of sequential circuits. IEEE Transactions on Computers 47(1), 129–134.

Naito, S. and M. Tsunoyama (1981). Fault detection for sequential machines by transition-tours. In Proc Fault Tolerant Computing Systems, pp. 238–243.

North, N. D. (1990). Automatic test generation for the triangle problem. Technical Report NPL Report DITC 161/90, National Physics Laboratory, Teddington, UK.

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

Ostrand, T. J. (1986). The use of formal specifications in program testing. In The 3rd International Workshop On Software Specification, London, pp. 253–255.

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

Poston, R. M. (1996). Automating specification-based software testing. IEEE Computer Society Press.

Richardson, D. and L. Clarke (1985, December). Partition analysis : a method combining testing and verification. IEEE Transactions on Software Engineering 11(12), 1477–1490.

Richardson, D. J. and M. C. Thompson (1986). Relay: A new model of error detection. Technical Report 86-64, Computer And Information Science, University Of Massachusetts, Amherst, MA.

Sabani, K. and A. T. Dahbura (1988). A protocol testing procedure. Computer Networks and ISDN Systems 15(4), 285–297.

Spivey, J. (1992). The Z notation : a reference manual (2 ed.). Prentice-Hall. http://spivey.oriel.ox.ac.uk/ mike/zrm.

Stepney, S. (1995). Testing as abstraction. In J. P. Bowen and M. G. Hinchey (Eds.), Proc. 9th International Conference of Z Users, Volume 967 of LNCS, pp. 137–151. Springer-Verlag.

Stocks, P. and D. Carrington (1993, May). Test templates : a specification-based testing framework. In 15th International Conference on Software Engineering (ICSE 93), Baltimore, MD, pp. 405–414.

Stocks, P. and D. Carrington (1996, November). A framework for specification-based testing. IEEE Transactions on Software Engineering 22(11), 777–793.

Telegraphy, I. C. C. O. and Telephony (1989). Specification And Description Language(SDL) CCITT Recommendation Z.100. Geneva: International Consultative Committee On Telegraphy And Telephony.

Toyn, I. and J. Hall (1994). Proving conjectures using z. Technical report, Department Of Computer Science, University Of York, York, UK.

v. Bochmann ad Alexandre Petrenko, G. (1994). Protocol testing: Review of methods and relevance for software testing. In ACM International Symposium on Software Testing, Analysis, and Verification, ISSTA '94, Software Engineering Notes, Seattle, WA, pp. 109–124.

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

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

Voas, J. M. and K. W. Miller (1995, May). Software testability: The new verification. IEEE Software.

Weyukar, E. and B. Jeng (1991, July). Analyzing partition testing strategies. IEEE Transactions on Software Engineering 17(7), 703–711.

Weyukar, E. J. (1988, June). The evaluation of program-based software test data adequacy criteria. Communications Of The ACM 31(6), 668 – 675.

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

Wong, W. E. (1993). On mutation and dataflow. Ph. D. thesis, Computer Science Department, Purdue University.

Woodcock, J. C. P. and S. M. Brien (1992). W: A logic for z. In J. E. N. Ed. (Ed.), Z User Workshop, Springer-Verlag Workshops In Computing, York, U.K. Springer-Verlag.

Wordsworth, J. (1992). Software development with Z (1 ed.). International Computer Science Series. Addison-Wesley.

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

Zeil, S., F. Afifi, and L. White (1992, October). Detection of linear erros via domain testing. ACM Transactions on Software Engineering and Methodology 1(4), 423–451.

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