SCHOOL *of* GRADUATE STUDIES
EAST TENNESSEE STATE UNIVERSITY

East Tennessee State University
Digital Commons @ East Tennessee State University

Electronic Theses and Dissertations

Student Works

5-2001

Leveraging Test Measurements into Proposing Additional Domain Tests.

Radhika Turlapati
East Tennessee State University

Follow this and additional works at: https://dc.etsu.edu/etd

Part of the Computer Sciences Commons

Recommended Citation
Turlapati, Radhika, "Leveraging Test Measurements into Proposing Additional Domain Tests." (2001). *Electronic Theses and Dissertations.* Paper 57. https://dc.etsu.edu/etd/57

This Thesis - Open Access is brought to you for free and open access by the Student Works at Digital Commons @ East Tennessee State University. It has been accepted for inclusion in Electronic Theses and Dissertations by an authorized administrator of Digital Commons @ East Tennessee State University. For more information, please contact digilib@etsu.edu.

Leveraging Test Measurements Into Proposing Additional Domain Tests

A Thesis
Presented to
The Faculty of the Department of Computer and Information Sciences
East Tennessee State University

In Partial Fulfillment
Of the Requirements for the Degree
Masters of Science in Information Science

By
Radhika Turlapati
May 2001

Dr. Martin Barrett, Chair
Dr. Don Bailes
Mr. John Chenoweth

Keywords: Test Suggestions, Domain Partitioning, Percentile Bucketing, Reliability Metrics, Test Types, and MATRIX$_x$® Automated Test Tool, Reliability Analysis Test Tool.

ABSTRACT

Leveraging Test Measurements Into Proposing
Additional Domain Tests
By
Radhika Turlapati

Accuracy and efficiency are extremely critical factors for large real-time control applications. A small oversight can cause catastrophic failure of a real-time system. Thus, these applications have to be tested meticulously to prevent any catastrophe that might occur. But, testing these applications exhaustively is not tractable, mainly due to the inherent complexity of the applications and also the huge amount of inputs and outputs that these applications involve. In order to save valuable amounts of time and resources, automated testing is imperative. Also, quantitative metrics have to be provided that assess the existing quality of the system and help increase the confidence in the user towards the software. However, to improve the overall quality of the software, additional focused testing needs to be done.

The work in this thesis involves providing specific test suggestions that help the user conduct thorough and precise domain tests based on the knowledge of the various parameters used in previous test runs. The information about the defective portions of the input domain is provided by dividing the input range into percentiles, which is referred to here as bucketing. The goal is to expose the exact inputs causing the defects and the range of inputs that have been lightly tested or left untested during previous tests. A Reliability Analysis Test Tool (RATT) was developed to implement these test suggestions.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my parents for giving me all the support and encouragement necessary to achieve my educational goals. Next, I express my most sincere gratitude to my thesis advisor, Dr. Joel Henry, for his valuable time, support and excellent direction throughout the project. I also gratefully acknowledge the help and support extended by my committee chair, Dr. Martin Barrett, for his valuable time and suggestions.

A special note of thanks goes out to my friend, Narendra Koneru, for his support and advice throughout the project.

CONTENTS

Chapter Page

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

This thesis defines and implements an approach to suggesting specific domain tests that help point out definite values in the input domain that cause defects in the system and also target future tests on sections of the input domain that have been lightly covered or left uncovered. These domain specific test suggestions use the results of previously conducted test runs and are based on the knowledge of sections of the input domain causing most defects and sections of the input range that have not been tested at all. In order to implement this strategy, a Reliability Analysis Test Tool (RATT) was developed in co-ordination with another student at ETSU, Mr.Koneru, who worked on providing reliability measures that assess the existing quality of the system. The reliability measures, based on dividing the input range into percentiles and referred to as bucketing, provide specific information about the effectiveness of previous tests to cover the domain of each input variable. This information is an important criterion to suggesting tests specific to that particular portion of the input domain.

Outline Of Thesis

The reminder of this chapter stresses the importance of thorough testing of real-time systems, some of the challenges involved in testing real-time systems, and the need for simulating the test environment.

Chapter 2 gives a brief overview of the background research done for this thesis. First, a brief description of MATRIX$_x$® and MATT are given, which are the two applications used for implementing the proposed strategy. Next is a brief discussion on how the information specified by a specific measurement, Domain Percentile Coverage, is an important criterion for suggesting additional tests. Also, other important parameters that influence further tests are discussed here.

Chapter 3 presents a statement of the problem. Specific problems in automated testing of real-time systems are addressed and the need for precise test suggestions that help focus future tests on erroneous inputs is stressed.

A solution to the problem is presented in Chapter 4. This essentially includes the mathematics involved in deriving the test suggestions.

Chapter 5 mainly covers the implementation and testing of the test suggestions within RATT. A comprehensive design is presented here and the use of RATT within NASA and other organizations is discussed.

Chapter 6 presents how the results obtained by RATT complement MATT and Chapter-7 discusses further work that can be done to improve the strategy proposed in this thesis.

Overview Of Testing Real-Time Systems

As computers become indispensable elements of complex systems, it becomes imperative to address the dependability of such systems, especially when they are increasingly used in safety critical environments. Examples of such large embedded real-time control systems are applications that control wind tunnels, the space shuttle, nuclear reactors, and missiles. The potential high cost associated with the erroneous operation of such systems has created a high demand for a comprehensive analysis of their reliability.

Reliability of a system is a measure of the error free behavior of the system over time. To improve reliability, the role played by testing of real-time systems before they are deployed becomes extremely important. However, testing of real-time systems poses challenging problems mainly attributed to the inherent complexity of these applications and also the combinatorial input and output domain space. For example, an input domain of 0 – 100 with an accuracy of 0.001 contains 1,000,000 possible values and in a system with just five input variables within this same domain would possible produce $1{,}000{,}000^5$ input value combinations [1]. Also, real-time applications interact with their environment through time-constrained inputs and outputs. Because the correct system functionality of a real-time system depends not only on its logical but also its temporal correctness, multiple executions of real-time software with same test cases might produce different test results.

The incorrect behavior of a real-time application caused by the breach or deviation of a time constraint makes testing real-time systems much more complex. However, in spite of the difficulties involved in real-time system testing, to gain the confidence of the user in the system and to prevent any catastrophe that might occur, effective testing is crucial. With reasonable assumptions about system behavior and careful analysis, well-designed test suites using automated test generation techniques can detect potential defects in the system.

Need For Simulation

Testing real-time applications on the actual target hardware is not always possible. Therefore, some degree of testing using simulation is needed mainly due to the risk of expensive hardware damage and the safety hazards associated with testing real-time systems. Because real-time systems sample large number of input values and output values in sometimes very short time intervals, a simulation typically requires a huge number of input values to be generated and a huge number of output values captured as a result of the simulation. Automated test generation tools for real-time systems can be used to construct test cases with effective inputs that essentially model the inputs of the target system and also perform a comprehensive analysis on the output of simulation.

However, simulation poses its own unique difficulties like error-prone output analysis, generation of potentially massive data sets, and dependability of the results of simulation when modeling life-critical applications [1]. Also, the impact of hardware faults on software processing cannot be considered during simulation. Simulation is thus used prior to hardware-software integration to achieve some degree of confidence in the software portion of the system.

CHAPTER 2

BACKGROUND

NASA's (National Aeronautics and Space Administration) Ames Research Center is using MATRIX$_x$®, a software product developed by Integrated Systems Inc., to provide comprehensive design and development solutions for its real-time embedded control systems. Examples include applications that control the unitary wind tunnel and control systems for the International Space Station. Exhaustive manual testing of such complex and important applications is overwhelming. MATT (MATRIX$_x$® Automated Test Tool) is an automated test tool developed by the Design Studio team at ETSU under the guidance of Dr. Joel Henry that provides an automated test environment to the users of MATRIXx® products (xMath and SystemBuild). But, the functionality of such a testing tool remains curtailed without providing provisions in it for ways to quantify the dependability of the system and provide additional testing suggestions that help focus future tests that have a high probability of discovering the defects. These additional tests are designed based on the knowledge of the various parameters used in the previous test run. This research uses MATRIX$_x$® and MATT and the information about the input domain provided by the reliability metric Percentile Domain Coverage for implementing the test suggestions.

Overview Of MATRIX$_x$®

Conventional real-time application development is usually a step-wise approach with separate tools for design, testing and integration. These tools work in tandem to aid engineers in accelerating the development. This allows a design to easily move from one step to the next, making it possible to create a working prototype very early in the design process. SystemBuild and xMath are tools that form the core of the MATRIXx® product family. SystemBuild is primarily a graphical tool that is used to represent and build graphical models of a control system. Complex control systems can be represented in a hierarchical fashion. Basic building blocks are grouped to form a super block. These superblocks can be placed in other superblocks to graphically represent a complex system. xMath is a design and analysis tool that operates with SystemBuild by acting as a working environment for simulating data and verifying SystemBuild models. The tools AutoCode and DocumentIt are used for automatic generation of high-level language code and industry standards' compliant documentation respectively. RealSim provides the hardware and software environment for rapid prototyping, data acquisition and testing [2].

Overview Of MATT

MATT (MATRIX$_x$® Automated Test Tool) is an automated test tool that offers a way of creating pertinent test inputs and reporting exceptions based on the generated output values. It functions in tandem with xMath and SystemBuild. The MATT application has to be started from xMath, which in turn operates with SystemBuild to load the models. When a super block selected from SystemBuild is loaded into MATT, all the parameters of the super block, including the inputs, outputs, and the data types, are directly loaded into MATT. Any change made to a model in SystemBuild is directly reflected in MATT [3]. The MATT test script includes the user selectable parameters like the test type, input minimum and maximum, output minimum and maximum, simulation time interval, the number of test steps, accuracy, and the desired exception types to report on simulation output. MATT automatically converts the test script into an input test matrix, which is in a format that can be directly read into MATT for simulation. The knowledge of the parameters used for simulation is very important for subsequent simulations to target the defects with much higher probability than the previous tests. One such important parameter is the test type used for simulation. Real-time systems often have to be tested for behavior under unstable conditions and also whether they maintain a safe state until the input values become stable. Also, many times there arise circumstances where the system is to significantly change behavior or maintain current behavior at critical values. These points have to be carefully scrutinized. Specific tests that hold the input at a constant value or test on boundary values where behavior changes significantly are very crucial [1]. MATT currently has 25 different test types grouped into 5 descriptive groups that are designed to accommodate the above-mentioned situations.

MATT Test Types

1. Critical Point Tests

For critical point tests, the constant is used to determine the value of generated test data. All values for the input test type are generated using a fixed value specified by a constant. Both the test minimum and test maximum will have an effect on the constant depending on the selected critical point test. Accuracy is ignored for these tests because the constant stated value becomes the generated test value for all individual test intervals or steps.

Floating Point Types: The constant value is set to (Test Max-Test Min)/2.

Integer Types: The constant is set to whole integer portion of (Test Maximum - Test Minimum)/2.

Logical Types: The constant value is set to 1.

Four different test types exist:

- CP@Con (Critical Point At Constant)
- CP@Min (Critical Point At Minimum)
- CP@Max (Critical Point At Maximum)
- CP@Zero (Critical Point At Zero)

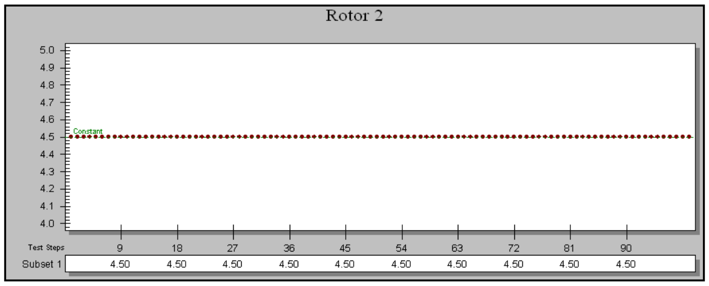


Figure 3.1: Example - Critical Point At Constant

2. Boundary Value Tests

For boundary value test data generation, the default increment between each consecutive test interval is calculated as 10 to the negative power of the accuracy. For integers accuracy is always ≤ 0, and for logical variables this test does not apply.

If the computed test values are outside the domain, they are set to the value of test minimum or test maximum depending on which test boundary is set. The values are set to test maximum when test minimum is the boundary and test minimum when the test maximum is the boundary. Users may set any valid value for the test minimum or test maximum based on the input type.

Eight different test types exist in this category.

- A2Max – Ascending to Maximum
- A2MaxX - Ascending to Maximum (Max Excluded)
- D2Min – Descending to Minimum
- D2MinX – Descending to Minimum (Min Excluded)
- DMax – Descending from Maximum
- DMaxX - Descending from Maximum (Max Excluded)
- AMin – Ascending from Minimum
- AMinX - Ascending from Minimum (Min Excluded)

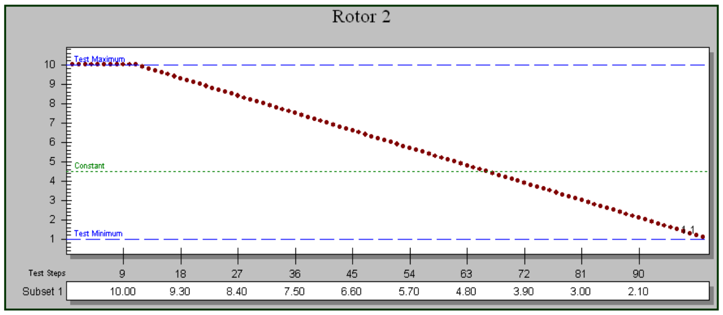


Figure 3.2: Example - Descending to Minimum

3. Linear Tests

For linear test data generation, the default increment between each consecutive test interval is calculated as (Test Max-Test Min)/(Number Of Tests). Hence, input values generated are equally spaced from each other. The boundaries for this test type are both the test minimum and the test maximum, and the accuracy setting for this test type is ignored. This test is not applicable to logical data types. For integer data types the increment is stripped of its fractional part.

Two test types exist here.

- Max2Min – Maximum To Minimum
- Min2Max – Minimum to Maximum

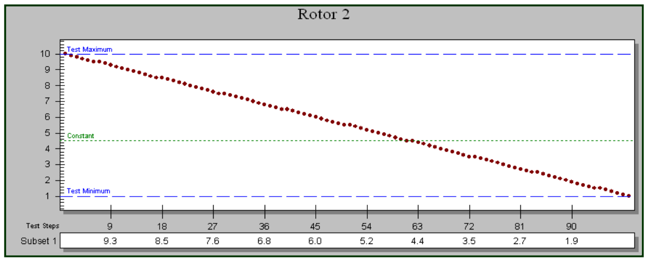


Figure 3.3: Example – Maximum Descending to Minimum

4. Random Value Tests

Random test data generation uses various randomizing techniques to produce a test value for each test step. Random tests may allow for either inclusion or exclusion of both the test minimum and maximum values. The accuracy setting for this test is ignored. Random testing is applicable to all data types, floating point, integer, and logical.

The following test types fall in this category.

- RASeg (Random Ascending Segmented)
- RDSeg (Random Descending Segmented)
- RA2Max (Random Ascending to Maximum)
- RD2Min (Random Descending to Minimum)
- RMM (Random between Minimum and Maximum)
- RMMX (Random between Minimum and Maximum, both excluded)

- RAMM (Random Ascending between Minimum and Maximum)
- RDMM (Random Descending between Minimum and Maximum)
- RAMin (Random Ascending from Minimum)
- RDMax (Random Descending from Maximum)

For the RASeg and RDSeg tests, random numbers are generated from a particular segment of the input domain for each time step where the width of the segment is calculated as (Test Max–Test Min)/(Number Of Tests).

For RA2Max, RD2Min, RAMin, RDMax, random numbers generated are focused within a cone that decreases in size with each test step from Test Min to (Test Max-Test Min)/2. For RMM, RMMX, RAMM, and RDMM, random numbers generated are distributed from the test minimum to the test maximum and may include either test minimum or maximum, or both.

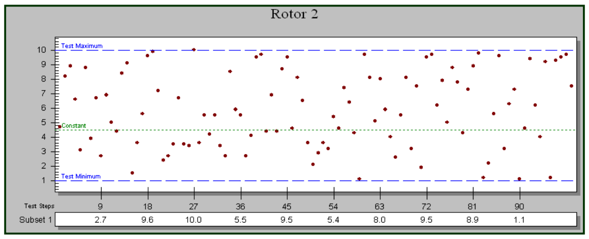


Figure 3.4: Example - RMM

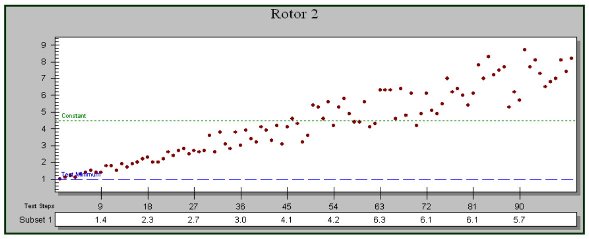


Figure 3.5: Example - RAMin

5. Sinusoidal Tests

Test data generated for this test type follows a sinusoidal waveform. The distribution of input values on the wave will vary based on the user selectable frequency, number of tests, and the values of test minimum and test maximum. For example, in Osc.25, the generated test values will form ¼ of a sine wave. Other test types that fall in this category are Osc.5, Osc, Osc2, Osc4, and Osc8.

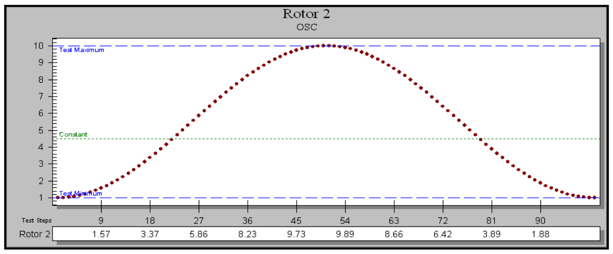


Figure 3.6: Example – Oscillate Value One

<u>Overview Of Percentile Bucketing</u>

Information about regions of the input domain producing defects in the output and input values where the system is not tested is very important in order for subsequent simulations to target those problematic sections. But as real-time systems involve combinatorial input and output domain space, it becomes necessary to reduce the problem space by adopting a strategy to partition the domain into sub-domains. In this research domain partitioning is done by dividing the entire input and output domains into percentiles, in effect 100 buckets of contiguous values. This approach is also termed Percentile Bucketing. Bucketing can be used to detect defect-prone inputs by saving the bucket each input value fell into when a defect is detected. Also, buckets that have no coverage or very little coverage can be saved for further testing. The data provided by percentile bucketing can be used target the tests primarily on input ranges that result in the highest number of defects and on input ranges that have not been tested at all.

In addition, if a combination of values from different input variables result in multiple defects, that combination can be identified. For example, if test values for bucket7 for input1 and bucket9 for input2 are associated with multiple defects, this combination of percentiles can be considered defect-prone and can be used to guide further testing [4]. The reliability measure, percentile domain coverage thereby reports input ranges to target further tests and also reports the failure intensity in those ranges. The test suggestions are then based on this data.

CHAPTER 3

PROBLEM STATEMENT

This chapter focuses on addressing specific problems in automated testing of complex real-time control applications. The primary function of such applications is mission and life support. The combination of temporal requirements, high reliability requirements, and need for testing in a simulated environment presents the system engineer with unique problems. Automated testing tools save valuable time and resources for developers by generating huge amounts of custom data, which can be used for software testing via simulation. Reliability metrics quantify software reliability and improve confidence of the user in the system. They provide a means of assessing the present quality of the software. Besides providing a testing and performance evaluation criteria, reliability metrics should improve the overall quality of the software by suggesting additional tests and supporting efforts to remove defects. Overall system quality can only be improved by probing the system for potential faults and subsequently debugging the system of these faults. However, arbitrary testing can waste the considerable testing effort as one might end up testing using similar test cases and leaving out more effective inputs that could potentially uncover more defects.

Goals of this Thesis

The primary goals of this thesis are to provide the user with techniques to improve the effectiveness of testing (i.e. find more defects) and improve debugging (by pinpointing the conditions generating defects). Hence, specific test suggestions have to be provided that guide the user in generating more precise domain tests that better expose the exact portions of the input domain causing the defects and suggest tests for the range of inputs that have been lightly tested or left untested during previous tests.

This thesis will attempt to answer the following questions:

1. How advantageous/disadvantageous is the suggested testing strategy?
2. How much better/ worse percentile input coverage is being achieved by following the suggested testing strategy?
3. How much more precisely is the user able to establish the exact inputs (or) input domains producing the defects?

A well-designed extensible application has to be developed with a simple, easy-to-use graphical interface that provides specific test suggestions for the user that guide the user to setting up a test wizard-like feedback. This application will also help verify the effectiveness of the suggested test strategy. The final goal is to document the application well enough for future extensions and user support.

CHAPTER 4
SOLUTION APPROACH

In spite of the great importance given to testing real-time systems, the main limitation of the development process still remains the lack of thorough testing during the verification stage. This is mainly attributed to the inefficiency of the current methods to accomplish the task. Testers often face the recurring question of exactly which tests to perform in the time typically allocated for testing. Also, the application of specific test types in a particular sequence can form an excellent test plan, if effective tests are sequenced properly. [1] Well-designed tests have a high probability of uncovering defects and a proper sequence of tests uncovers more defects than the tests run in isolation. Hence, a specific test followed by another might help uncover the latent defects of the system, which might remain undetected using a different combination of test types. Running such a sequence of tests will also require the user to fine-tune several other parameters on which the previously conducted test run was based.

This chapter presents a solution to the problem discussed in Chapter 3 in the form of specific test suggestions to the users of Matrix$_x^{®}$ Automated Test Tool that will help them set up precise domain tests based on the previous test runs.

First, the user needs clear understanding of the problem domain. The test suggestions will use the reliability metric ‘Percentile Domain Coverage’ discussed in Chapter 2 to get information about the problem domain. This metric reports:

1. Uncovered or lightly covered percentile domains
2. Input percentile domains producing most exceptions in the output

3. Input percentile combinations producing most exceptions

   The test suggestions will then be based on:

1. *Test Type:* The test type employed in the previous test run chosen from the 25 different test types discussed in Chapter 2.
2. *Number Of Test Runs:* This parameter suggests the number of test cases that should be run in a subsequent test run. The user can manipulate this value to change the number of input values generated, thereby changing the distribution of input values in the input domain.
3. *Accuracy:* This parameter controls the precision of each input variable and depends on the input data type. For integers, the accuracy is between 0 and –10, and for floating point data types, the accuracy ranges from 0 to 10. The accuracy settings are not applied to Boolean data types. The data type of an input variable cannot be manipulated but the input data type suggests the range of accuracy settings that can be applied to that particular input variable. The accuracy settings can also be manipulated to affect the input values generated.
4. *Test Boundaries:* This indicates the minimum and maximum values that each generated input value can take. The test boundaries can be set by the user to restrict the test set.

Test Suggestions – Employed Strategy

Because, the test suggestions primarily depend on the test type employed, the suggestions will remain common within each test type with slight variations in the formulae used for calculating the parameters for the test for each test type. However, the suggestions will take a different approach for each test type. Two distinct kinds of suggestions will be provided to the user.

1. Suggestions for percentiles with defects.
2. Suggestions for uncovered and lightly covered percentiles.

The following section presents an algorithm to suggesting tests for defective percentiles. The initial steps of the algorithm remain constant through all the test types. Through the reminder of this thesis, the word *percentile* and *bucket* will be used interchangeably and represent 1/100th of the input variable domain.

Percentiles With Defects – Basic Algorithm

Steps1, 2, and 3 remain same for all the tests regardless of the test type used. However, subsequent steps vary depending on the test type employed in the previous test run. These calculations will be explained in detail for each test type in the following sections. The basic algorithm for the percentiles with exceptions is as follows.

1. Get all the input buckets with exceptions and also the number of exceptions in each of those input buckets. These are the portions of the input domain that produce at least one exception in the output.
2. Build bucket blocks. Defective buckets that are less than 3 percentiles apart fall in a single block. For example, percentiles 2,5 fall in one block even if percentiles 3 or 4 do not have any defects. Similarly, 2,4,5,8 fall in one block. If bucket 11 has defects, it can be accommodated in the same block but if bucket 11 does not have any defects, then defective bucket 12 would be the first percentile in the next block and cannot be accommodated in the same block as (12 –8) > 3.

3. Set the test boundaries for this block. The test minimum is set to the input value corresponding to (first bucket in the block – *e*) and the test maximum is set to the input value corresponding to (last bucket in the block + *e*) where *e* is the bucket size and is calculated as (Test Maximum – Test Minimum)/100.
4. Set accuracy and the number of test runs. The values to be set for these parameters depend on the test type selected and will be discussed in detail for each test type in the subsequent sections.
5. Compare the number of defects in the first half of the block to the number of defects in the second half to decide on the test type to employ. After comparison, the decision of as to which test type to employ also depends on the test type employed in the previous test run.

Test Suggestions – Boundary Value Tests

The basic strategy employed for suggesting additional domain tests will remain fairly common for all boundary value test types. However, there will be slight variations in the formulae though the basic idea remains consistent. This section uses 'D2Min' (Descending to Minimum) to explain the strategy used to suggesting the tests for all boundary value test types.

Test Suggestions For Uncovered Percentiles:

In D2Min (Descending to Minimum), the input value generated in the ith test is calculated as *[Test Minimum + (Number of Tests – I) * Step] where Step =* $10^{-accuracy.}$ Hence, the first input value that is generated is [Test Minimum + (Number of Tests – 1) * Step] and the last input value that is generated is Test Minimum.

The following two cases have to be considered here.

Case 1: Because the first input value that is generated is not the test maximum (which would fall in the 100th bucket) but is a value that falls in some bucket x, where x < 100, all the percentiles from x - 100 will remain uncovered. For the user to achieve (x + n), percentile coverage, there are two alternatives. (1 < n < 99)

1. *Increase Number Of Tests:*

   The number of tests can be increased so that the first input value that is generated falls in (x + n) th percentile instead of the xth percentile. Because the increment in the constant here, this will ensure uniform coverage from the 1st percentile to the (x+ n) th percentile. Hence, the number of tests can be recalculated as

   Number of Tests = (Input Maximum (x + n) – Test Minimum) / Step where

   Input Maximum (x + n) is the maximum input value represented by the (x+ n) th bucket.

2. *Increase Step (or) Decrease Accuracy:*

   The accuracy value can be recalculated as

   Accuracy = Log$_{10}$ (1 / Step) where Step is calculated as

   Step = (Input Maximum (x + n) – Test Minimum) / Number Of Tests.

Case 2: Some buckets from the 1st bucket to the xth bucket will remain uncovered, as the increment used to generate the input values is greater than the bucket size. Hence, in order to avoid skipping buckets, the value of the increment specified (Step) should be set to less than or equal to the bucket size.

Therefore, if Step > Bucket Size, the value of accuracy is recalculated as *Log_{10} (1 / Bucket Size)* to ensure that each percentile holds at least one value. A value for accuracy less than Log_{10} (1 / Bucket Size) ensures more than one value in each percentile.

<u>Test Suggestions For Percentiles With Defects:</u>

In order to suggest tests for percentiles with defects, the first three steps of the algorithm described before are used to initially get the buckets with exceptions and the number of exceptions in each bucket and then build the bucket blocks and set the test boundaries for each of these blocks. The number of test runs, accuracy, and the test type have to be recalculated for the new test. These calculations depend on the test type employed and are as follows for D2Min.

a) First, if Step is greater than Bucket Size, then set Step equal to Bucket Size. Now, accuracy can be recalculated as Log_{10} (1 / Bucket Size). If Step is less than Bucket Size, the value of accuracy remains same as before.
b) Number of tests can then be calculated as (Test Maximum – Test Minimum) / Step calculated in step **a**.
c) For every block, if the number of exceptions in the first half is greater than the number of exceptions in the second half, run D2Min/AMin (that is, test more towards the test minimum) or else run DMax/A2Max, testing more towards the test maximum.

Because, in real-time applications, the value of the input generated in the i^{th} step might depend on the input or the output value generated in the $(i-1)^{th}$ step, thus by running AMin after D2Min (in that order), we are essentially covering the same inputs in the opposite order which might uncover some defects that running only D2Min might not.

Test Suggestions – Linear Value Tests

Here again, the approach will remain consistent for the Linear Value Tests. The following section uses 'Max2Min' (Maximum Descending to Minimum) to explain the strategy used in suggesting tests for the Linear Value Tests. Also, in case of Linear Value Tests the suggestions are the same for both defective percentiles and uncovered percentiles, which are as follows.

Test Suggestions For Uncovered Percentiles and Defective Percentiles:

In Max2Min (Maximum Descending to Minimum), the input value in the ith test is calculated as *[Test Minimum + (Test Maximum - Test Minimum) / (Number Of Tests - 1) * (i-1)]*

In Linear Value Tests, because the default increment between two successive intervals is constant and is calculated as (Test Max-Test Min)/(Number Of Tests), if there are any uncovered buckets in the input domain, it implies that the specified increment is greater than the bucket size. Here, two cases exist.

Case 1: Ensuring Full Coverage

To ensure full coverage, the increment should be less than or at least equal to bucket size where increment is calculated as (Test Maximum – Test Minimum)/Number Of Tests.

Therefore, (Test Maximum –Test Minimum)/Number Of Tests <= Bucket Size.

Hence, Number Of Tests should be greater than (Test Maximum – Test Minimum) / Bucket Size.

Case 2: Ensuring Equal Coverage

Continuing the argument from Case 1, if the Number Of Tests is an integral increment of (Test Maximum – Test Minimum) / Bucket Size, this will ensure equal number of values in each buckets thereby ensuring equal coverage. The integer value specifies the number of values that fall in each bucket. The user can thus specify how many values he wants in each bucket and the corresponding number of tests can be calculated for the test.

Thus repeat Max2Min and Min2Max (in that order) to target defective percentiles while ensuring either full or equal coverage. Ensuring complete coverage is important to target exceptions as some input values that have remained uncovered in the previous test runs might lead to exceptions in the current test run.

Test Suggestions – Random Value Tests

The following section uses 'RASeg' (Random Ascending Segmented) and 'RAMin' (Random Ascending from Minimum) to explain the two distinct strategies employed here in suggesting tests for the Random Value Test types. The test types RA2Max, RD2Min, RDMax, RAMM, and RDMM follow the strategy used for RAMin. RDSeg uses the approach followed RASeg.

Test Suggestions For Uncovered Percentiles - RASeg:

In RASeg (Random Ascending Segmented), the input value in the i^{th} test is calculated as

*Test Minimum + Segment * (i - 1) + [(Random 100) / 100] * Segment* where Segment = (Test Maximum – Test Minimum)/Number Of Tests.

To ensure that skipping of buckets is avoided, the value of the default increment specified between two successive intervals should be less than or at least equal to the bucket size. The value of the increment for ‘RASeg’ is calculated as *[Random (100)/100] * Segment.*

Thus, the maximum value that the increment can take is (Test Max – Test Min)/No. Of Tests as the value of Random (100)/100 is at most equal to 1, thereby implying that Segment should be less than or at least equal to the Bucket Size.

Therefore, to ensure skipping of buckets, Number Of Tests should be greater than (Test Maximum – Test Minimum) /Bucket Size.

<u>Test Suggestions For Defective Percentiles – RASeg:</u>

The following tests are suggested for defective percentiles in the following order.

1. Repeat ‘RASeg’ with the value for

   Number Of Tests > (Test Maximum – Test Minimum) / Bucket Size
2. Run ‘RDSeg’ with the value for

   Number Of Tests > (Test Maximum – Test Minimum) / Bucket Size. This test will essentially cover the same inputs specified by ‘RASeg’ in the opposite order for better defect exposure.
3. Run Min2Max with Number Of Tests > 100 for uniform coverage and Number Of Tests = n*100 for equal coverage.

The rationale for suggesting ‘RDSeg’ is that covering the same inputs in the opposite order might result in better defect exposure because in a real-time system the time step in which the input is generated is can be an important factor.

Running Min2Max, in essence covers the input domain in equal increments in an ascending fashion. Because ‘RASeg’ is a special case of linear value test with random linear increments instead of equal linear increments, running Min2Max might also improve the defect exposure as the user can dictate the number of inputs that have to fall in each input percentile.

Test Suggestions For Uncovered Percentiles and Defective Percentiles - RAMin:

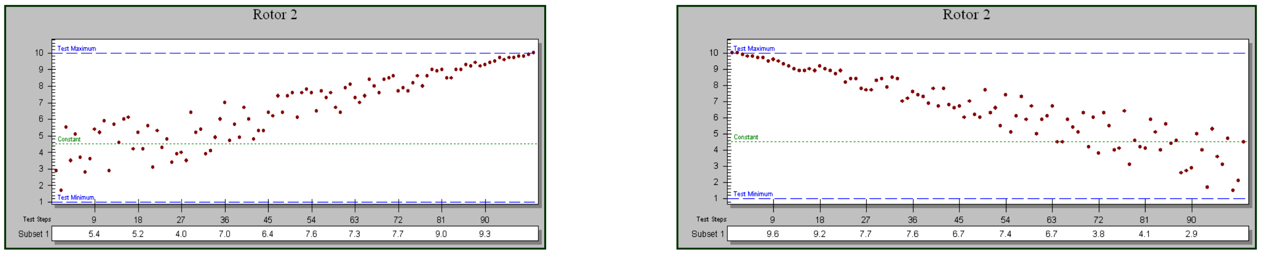


Figure 4.1: Test Suggestions - RAMin

These test types check the stabilizing/destabilizing conditions of a real-time system. Figure 1 illustrates the way the input domain is covered in case of ‘RAMin’. If we run an RDMax test immediately after RAMin, we are essentially covering the same inputs in the opposite order and interposed graph will form a pattern of a bow tie. (Please refer to Figure 4.1) This will be a helpful approach in uncovering more defects, as a real-time system may be stable and descend to instability or the system may be instable and approach a stable state. So, running a destabilizing from maximum test after a stabilizing to maximum is like running a mirror image test and checks for both the stabilizing and the instabilizing conditions near the maximum.

Test Suggestions – Critical Point Tests

The idea of percentile domains is not important in case of critical point tests as only one bucket is covered during every test run and the value of the bucket covered will depend on the critical point set for the test. So, the idea of covering the uncovered percentiles does not exist here. In case there are defects in percentiles covered, the best way would be to repeat the same test over again or run boundary value tests approaching the critical point.

Test Suggestions – Oscillate Value Tests

The oscillate value tests specify a series of input values along a sine wave beginning at the midpoint of the input variable range, ascending to input maximum, descending to input minimum and then returning to the midpoint. The input value generated in the ith test is calculated as *Test Value*$_i$ *= (Sine ((I – 1) * Frequency) * InputRange/2) + Midpoint* where

Frequency = 2*PI*X / (N – 1)

X = Number of oscillations per hundred tests (1/8, ¼, ½, 1, 2, 4, 8)

N = Number of test cases run in one test run

Input Range = Test Maximum – Test Minimum and

Midpoint = (Test Maximum – Test Minimum) /2

Test Suggestions For Uncovered Percentiles:

To suggest tests for oscillate value tests, one has to consider two cases.

Case 1: When the frequency factor is specified as 1/n, it means that the generated input values cover 1/nth of the sine wave for every hundred tests. Therefore, if the value of the number of tests is less than 100*n, some portion of the input domain will always remain uncovered. This case applies to only Osc1/8, Osc1/4, Osc1/2, and Osc1.For other oscillate-value test types, this case does not hold. Therefore, if the user wishes to cover the entire input domain, he should specify at least

- Number Of Tests = 800 for Osc1/8.
- Number Of Tests = 400 for Osc1/4.
- Number Of Tests = 200 for Osc1/2.
- Number Of Tests = 100 for Osc1.

Case 2: If the increment used between successive intervals to generate input values is greater than the bucket size will also leading to skipping buckets.

Hence, Increment < Bucket Size where

Increment = Sine (Frequency) * (Test Maximum – Test Minimum) / 2 and

Bucket Size = (Test Maximum – Test Minimum) / 100. Solving for the above equation implies that if 50*(Sine (2*PI*X)/(N - 1)) > 1 is true, then some buckets will surely be skipped.

Therefore, the value for Number Of Tests is recalculated as *N = 1 + [(2*PI*X)/(InvSin (1/50))]* by setting the value of the increment at least equal to the bucket size.

Test Suggestions For Defective Percentiles:

For suggesting tests for percentiles with exceptions, the algorithm described before is used to build the bucket blocks by grouping the buckets containing exceptions. The block boundaries have to be set depending on the concentration of the defects in the block. The test type to be used and the value of the number of tests have to be calculated. The test suggestions are as follows:

- If the number of defects in the first half of the block is greater than the number of defects in the second half of the block, then because the defects are more towards the test minimum, run an oscillate value test with the next higher frequency setting the test minimum to the minimum input value corresponding to the block and the test maximum to the midpoint of the block
- If the number of defects in the second half of the block is greater than the number of defects in the first half of the block, then test more towards the second half, that is set the test minimum to the midpoint and the test maximum to the maximum input value corresponding to the block.
- Check the value of the number of tests to confirm if the set value does not leave any buckets uncovered or else recalculate the value of number of tests as described above for uncovered percentiles for oscillate value tests.

CHAPTER 5

SOLUTION IMPLEMENTATION

The solution to the problem discussed in Chapter 3 will remain incomplete without an application that implements the proposed solution approach. The Reliability Analysis Test Tool (RATT) was developed to verify the feasibility and correctness of the test suggestions discussed in Chapter 4. RATT reads in MATT generated test case files, calculates Reliability and Test Coverage Metrics, and then provides specific test suggestions that guide the user to configuring additional domain tests. The test suggestions use MATT testing and interface terminology and guide the user to setting up specific tests using wizard-like feedback. In effect, RATT complements the Matrix$_x$®Automated Test Tool (MATT) by providing an analysis and supplementary testing environment. With a family of tools like Matrix$_x$®, MATT, and RATT, the user can build models, run simulations on those models, capture and analyze the output, and run additional tests based on the test suggestions. The user can also run multiple test runs on the model and then integrate the test results to analyze them. This will provide the user with a complete design, development, testing, and analysis environment. The Reliability Analysis Test Tool was developed in coordination with another student at ETSU, Mr.Narendra Koneru, who worked on providing the Reliability and Test Coverage Metrics.

A good design strategy is very important to develop any application well, especially if the application is complex and has to support future upgrades. Also, without good software engineering principles, it is very difficult to synchronize and finally integrate the software developed by two people attacking different aspects of the problem. MATT is now being ported to MatLab. RATT was thereby designed keeping in mind the future probability of being used with MatLab. In fact, RATT can be used with any application that can generate test case files in a format that can be read into RATT for analysis. Also, RATT was designed with a simple easy-to-use, user-friendly GUI. The bulk of the work involved developing the workhorses that actually calculated the reliability metrics and provided the test suggestions.

The reminder of this chapter discusses the design, implementation, and testing of the test suggestions within RATT. Some of the test cases used for testing RATT are from the NASA Wind Tunnel control software while others are generated using MATT.

RATT Design Strategy

RATT was developed using a top-down approach, wherein the entire application was divided into four distinct packages as shown in Figure 5.1. The four packages and the relationships described between them represent the skeleton architecture of the application. Each package encapsulates several classes, and these classes communicate with other classes using simple well-defined interfaces. The rationale for designing such architecture is to separate the user interface of the application from other implementation details, thus making the tool forward compatible and extensible. A different interface can be plugged into the application to make it fully functional on other platforms, such as Sun Solaris or Linux.

The following section is a brief description of the four modules of RATT and how they communicate with each other to make RATT a fully functional application.

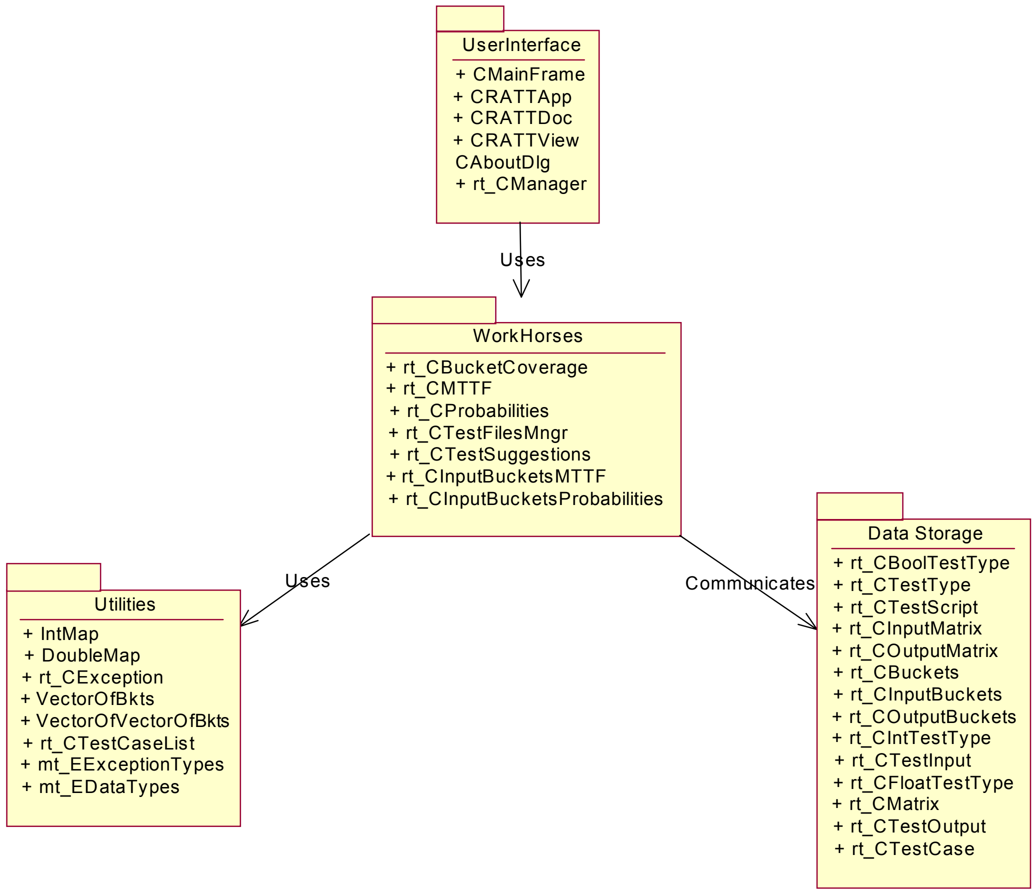


Figure 5.1: Skeleton Architecture of RATT

Modules Of RATT

1. User Interface

The User Interface package mainly consists of classes for developing the GUI of the application and will not be discussed in great detail in this thesis. When designing the User Interface, the goal was to keep it as simple as possible. However, the package also comprises of the *Manager class*. This class is the home of all the workhorse classes and contains methods used to manipulate the workhorse classes. All modifications to the workhorse classes are made through this class. Figure 5.2 demonstrates the various classes of the User interface package and the relationships between them.

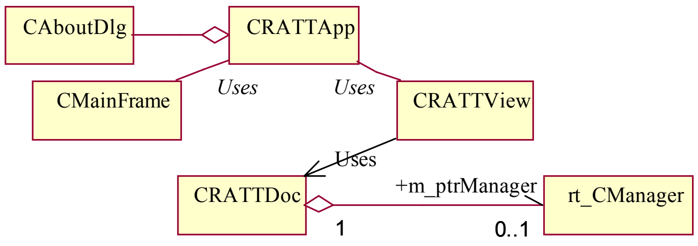


Figure 5.2: Class Diagram For User Interface Module

2. Utilities

Continuing the same argument from the User Interface package, the Utilities package consists of data structures that are independent of RATT and is not critical to the implementation of this thesis.

3. Data Storage

This package encapsulates the classes that store the massive number inputs read into RATT, outputs produced by RATT, and also the intermediate values that are generated as a result of analysis done on the inputs and outputs to produce test suggestions. The illustration in Figure 5.3 shows the classes in this package and the relationships between them.

The *Input Matrix* and the *Output Matrix* classes contain the huge MATT generated two-dimensional input value and output value arrays. These classes are inherited from the *Matrix* class and contain methods to read a matrix, access array values and clean up the memory used by the matrix when done. The *Test Script* class defines the user selected test script and contains the array of inputs, array of outputs, interval between successive test runs, and the number of test runs. The *Test Case* class is defined by the collection Input Matrix, Output Matrix, and the Test Script classes and contains pointers to each of them.

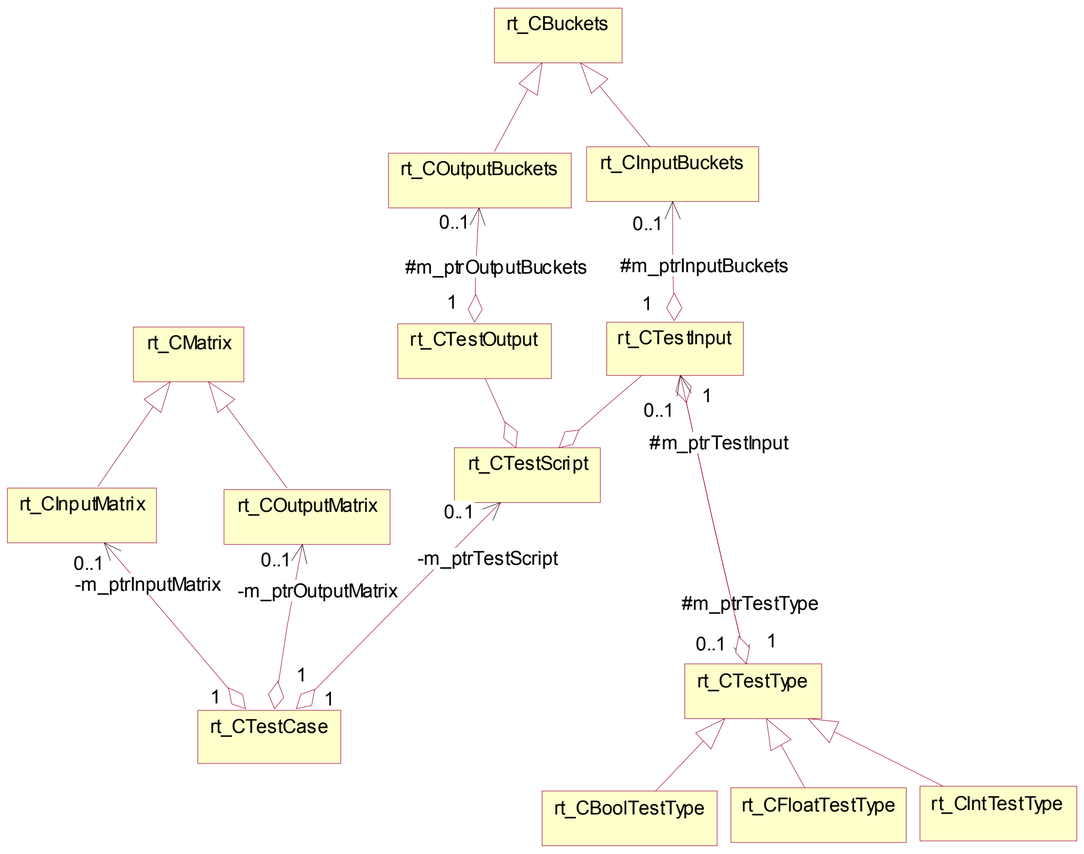


Figure 5.3: Class Diagram For Data Storage Module

The domain of each input variable and output variable is partitioned into 100 equal input buckets and output buckets respectively. For Boolean data types, the input and output variable range is partitioned into only two buckets, corresponding to 0 and 1 respectively. Thus, the *Test Input* class contains pointers to the *Input Buckets* and *Output Buckets* classes. These classes inherit from the *Buckets* class and contain an array of 100 buckets. They also contain methods to access buckets that correspond to specific input or output values, compute the total number of exceptions in a specific bucket, and also compute domain coverage values for each bucket. This information is very important both for deriving the test coverage metrics and for providing the test suggestions.

Each input variable is associated with a test type. The test suggestions actually reside in the *Test Type* class, which contains a pointer to the Test Input class. The *Boolean Test Type* class, *Float Test Type* class, and the *Integer Test Type* class inherit from the Test Type class. The test suggestions to be provided to the user depend both on the test type and the data type (an attribute of each input variable). Based on the input data type, the appropriate Test Type pointer is created when a test case file is loaded during initialization. Computations like grouping the buckets into bucket blocks and setting the test boundaries for each block after grouping are done in the Test Input class, as these computations do not need any test type information.

4. Workhorses

This package encapsulates all the classes that actually calculate the Reliability and Test Coverage Metrics and provide test suggestions to the user. The class diagram in Figure 5.4 illustrates the various classes in this package and the relationships between them.

The Manager class (from the User Interface package) is the home of all the above-illustrated Workhorse classes and contains methods to manipulate these workhorses. The *Bucket Coverage* class will calculate the test coverage values for all input and output variables. The *Probabilities* class and the *MTTF* class calculate the Reliability Metrics developed by Mr.Koneru. The *Test Files Manager* class is an important part of the workhorses' package and is responsible for maintaining the test case object and also maintaining correct values for the final test script object when ever more than one test case files are loaded into RATT for analysis. The Test Suggestions class contains a pointer to an array of test inputs and methods to retrieve the test suggestions. We retrieve the test suggestions in the Test Suggestions class using the test type pointer. Based on the input data type, the appropriate method in one of the derived Test Type classes will be invoked dynamically.

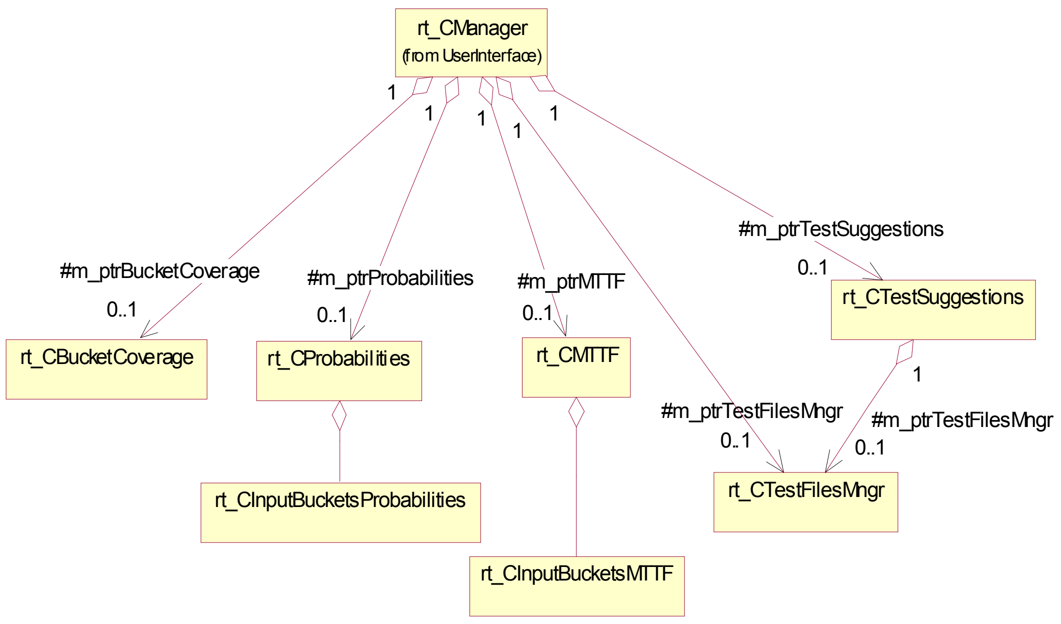


Figure 5.4: Class Diagram For Workhorses Module

Currently, no special test suggestions have been provided for the Boolean data types, but this class has been designed for any possible future extensions. However, the suggestions for the boundary value tests have been overridden in the Integer Test Type class. This is necessary as the accuracy settings are different for integer and floating-point data types giving rise to different test suggestions based on the input data type.

A complete class diagram that illustrates all the relationships between all the classes is given in the Appendix.

RATT Implementation and GUI

The Reliability Analysis Test Tool was developed using C++ and MFC in Microsoft Visual Studio environment. The main goal while developing RATT was to keep the tool as simple as possible while ensuring a high degree of accuracy and future extensibility. The User Interface of RATT is designed as a single document interface. The interface contains menus to load test case files, run the reliability metrics and the test suggestions and finally save the results obtained after analysis. The results are saved as simple ASCI text files. The results can also be displayed on the client area of the interface. Shortcuts buttons and accelerator keys have been provided for frequently used operations. The following section is a brief description of the working of RATT. The correct sequence of operations that should be followed for obtaining the results is given and some illustrations of RATT are provided for easy understanding.

Working Of RATT:

RATT is designed to run on Windows platforms. It does not need any special software to run. However, because it uses extensive MFC, some of the MFC libraries are required to support RATT. An installer has been built that automatically loads the required dlls and the RATT executable into the systems directory.

Figure 5.5 illustrates the main window of the RATT interface. The interface provides File, Edit, Run, Output, and Help menus. The user can use the mouse or the keyboard to select the various menu options provided by the interface.

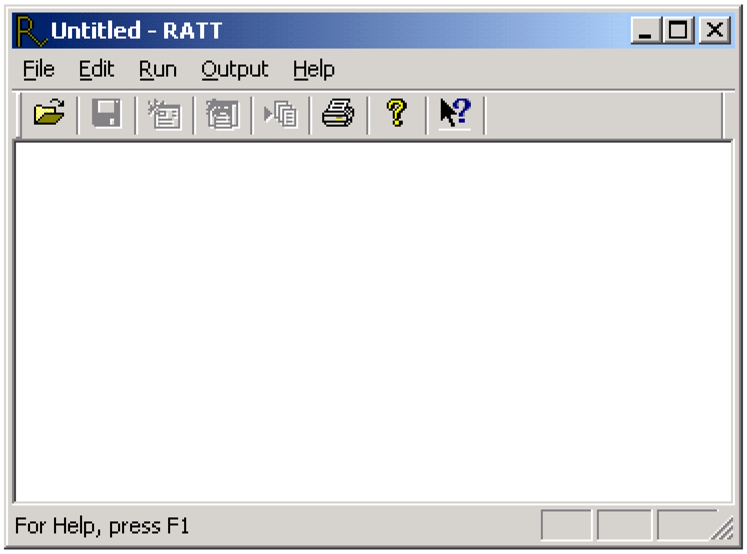


Figure 5.5: RATT User Interface

The following are the menus provided to the user.

File Menu:

The important sub-menu options of the File menu are Load, Add, and Save. Figure 5.6 illustrates the File Menu of RATT.

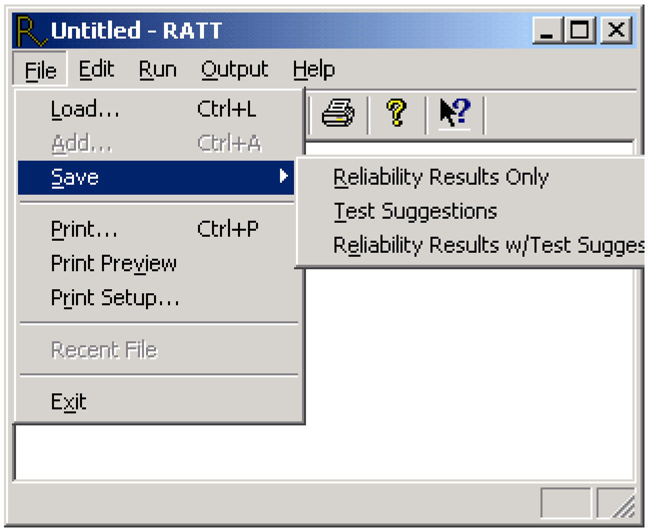


Figure 5.6: RATT File Menu

The *Load* menu enables the user to load a test case file for analysis. These test case files are CSV files that contain user selected test script information, the input matrix values and the output matrix values. RATT can analyze test case files that are in a specific format that can be read into RATT. The Load menu pops up an *Open dialog* box shown in Figure 5.7. The user can select the test case file to analyze from the file structure. Selecting the Load menu once again erases the data storage objects of the previously loaded test case file. Multiple files can also be read into RATT for analysis. But in order to add more files, the user has to select the *Add* menu. The Add menu is available only after loading a file. The Add menu also pops up the Open dialog from where the user can select additional test case files and these test case files should correspond to the same super block. The number of test case files that can be read into RATT can be set in the registry by manipulating the parameter MaxTestCases. This is done in order to prevent any memory overflows that might occur in case too many files are read into RATT for analysis. By default, the number of test case files that can be read into RATT is 10. The user, depending on the memory available on his machine can change this value.

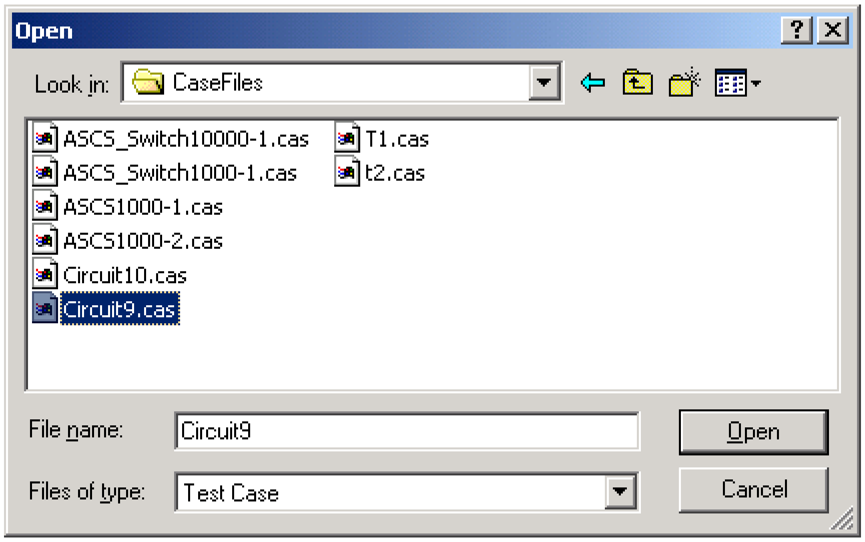


Figure 5.7: RATT Open Dialog

The *Save* menu option enables the user to save the RATT output files as CSV files. The reliability and test coverage metrics are saved as .rlr files, test suggestions as .sug files and a summary of the metrics with the test suggestions are saved as .rtt files. The File menu also provides the Print sub menus for printing and the Exit menu to quit the application.

Edit Menu:

The *Edit* menu provides the user with editing options like *Cut, Copy, Paste,* and *Clear Screen*. The Clear Screen menu is found especially useful when viewing results on the client area.

Run Menu:

The *Run* menu provides the user with three sub-menus, Get Reliability Metrics, Get Test Suggestions, and Get Test Suite Suggestions. After the user selects the test case files to analyze, the *Get Reliability Metrics* menu fetches the reliability and test coverage metrics for the user. The user can then either choose save the metrics or view them on the client area of the interface. The menus *Get Test Suggestions* and *Get Test Suite Suggestions* fetch the test suggestions to run additional domain tests. These menus are unavailable before the user obtains the reliability and coverage metrics. This is done because the test suggestions need test domain coverage information to run. The menu Test Suite Suggestions is available only when multiple test case files are loaded for analysis. This is done because the logic used for computing the test suggestions in case of multiple files is completely different from the logic used for computing the suggestions in case of a single test case file. Figure 5.8 shows the Run menu option selected on the interface.

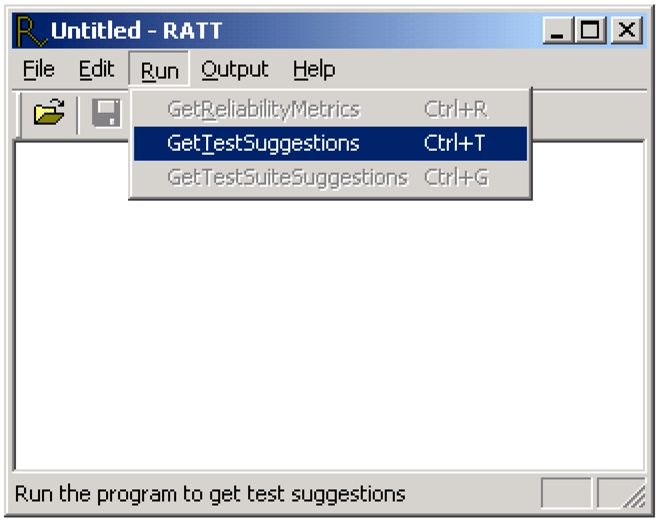
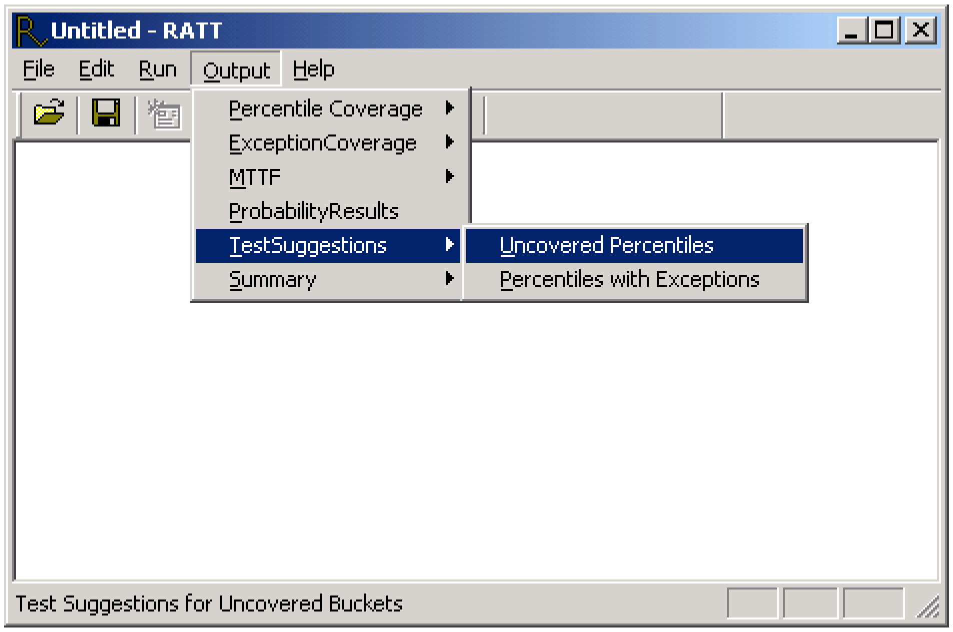

Untitled - RATT
File Edit Run Output Help
GetReliabilityMetrics Ctrl+R
GetTestSuggestions Ctrl+T
GetTestSuiteSuggestions Ctrl+G
Run the program to get test suggestions

Figure 5.8: RATT Run Menu

Output Menu:

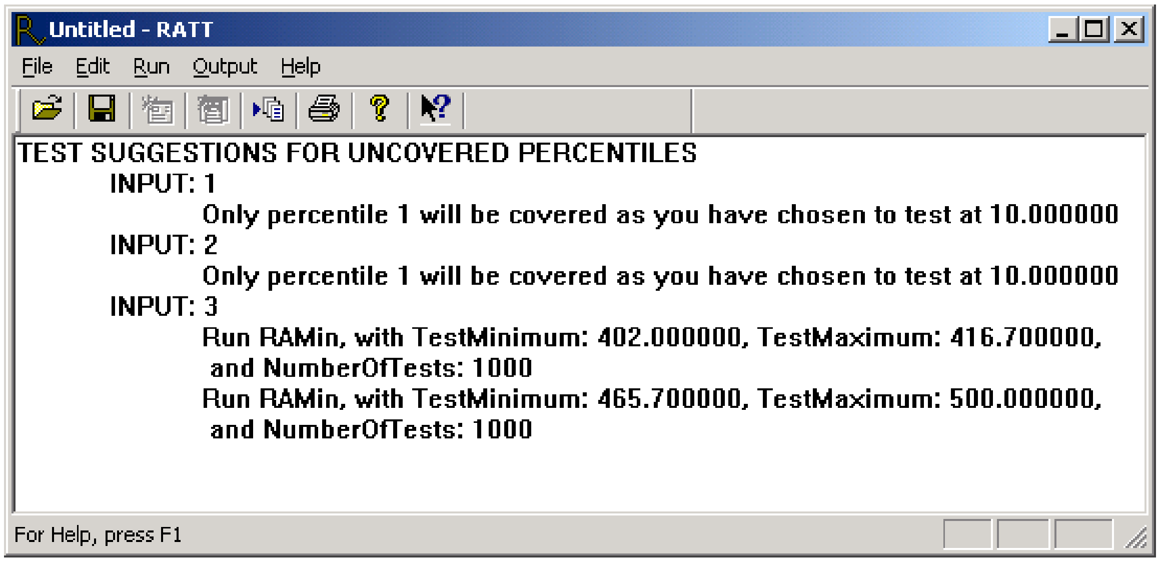
The *Output* menu provides the user with sub-menus to view the output on the client area of the interface, which is locked for editing. Figure 5.9 illustrates the Output menu selected on the interface.



Test Suggestions for Uncovered Buckets

Figure 5.9: RATT Output Menu

The *Percentile Coverage* menu provides the user with the input and output domain coverage. The *Exception Coverage* provides the user with the input percentiles producing the highest number of exceptions in the output and also the input percentile combinations producing highest exceptions. The *MTTF* and the *Probability Results* menus provide the user with the reliability metrics. The *Test Suggestions* menu provides the user with test suggestions both for uncovered percentiles and percentiles with exceptions. The *Summary* menu provides both the complete summary of the metrics and the test suggestions for the user. Figure 5.10 illustrates the test suggestions on the client area of the RATT user interface.



```
TEST SUGGESTIONS FOR UNCOVERED PERCENTILES
        INPUT: 1
                Only percentile 1 will be covered as you have chosen to test at 10.000000
        INPUT: 2
                Only percentile 1 will be covered as you have chosen to test at 10.000000
        INPUT: 3
                Run RAMin, with TestMinimum: 402.000000, TestMaximum: 416.700000,
                 and NumberOfTests: 1000
                Run RAMin, with TestMinimum: 465.700000, TestMaximum: 500.000000,
                 and NumberOfTests: 1000
```

Figure 5.10: RATT Client Area

Testing and Using RATT

Considerable emphasis was placed on testing RATT to obtain high degree of correctness. Many test cases were run to investigate and assess the correctness of the results. Most of the test case files used for testing were generated using MATT while others are from the Wind Tunnel control software. RATT was developed to be used within NASA and Boeing to guide the testing of independent groups of testers to help analyze the output obtained and guide them through setting up specific tests that will help them track the defects of the software much more effectively. RATT will also be used to help guide testing of Matrix$_x^®$ and MatLab developers.

CHAPTER 6

RESULTS

This chapter presents the implementation results of the proposed test suggestions using the Reliability Analysis Test Tool. In addition, an analysis of those results to understand the advantages and the limitations of the employed solution strategy is presented. The results produced after following the test suggestions are analyzed for their ability to:

1. Improve the effectiveness of testing by uncovering more defects.
2. Improve debugging by pinpointing the exact point in the input generating a defect.
3. Obtain a more complete coverage of test input domains.

Because the test suggestions largely depend on the test type used in the test run, tests were run against a sample model to verify the effectiveness of the suggestions provided for each test type.

However, in this chapter the results for one specific test type in each test type category are presented. Because, MATT has twenty-five different types grouped into five distinct test type categories, it would be overwhelming and not particularly useful to present the results for every single specific test type, especially when the logic used for suggesting additional tests remains consistent within each test type category with only some minor differences in the formulae used for the calculations. To be consistent, the test type used in each category for explaining the solution strategy in Chapter 4 would be used again in this chapter to demonstrate the results.

Testing Procedure

The Circuit Test model used for building NASA's Wind Tunnel control software was used for testing and verification. This is a simple but illustrative model with 3 inputs and 4 outputs. The model is loaded into MATRIX$_x$®, and MATT is used for simulation and testing to produce the test case files. The following are the values of the specific test case parameters used for generating the initial test case files.

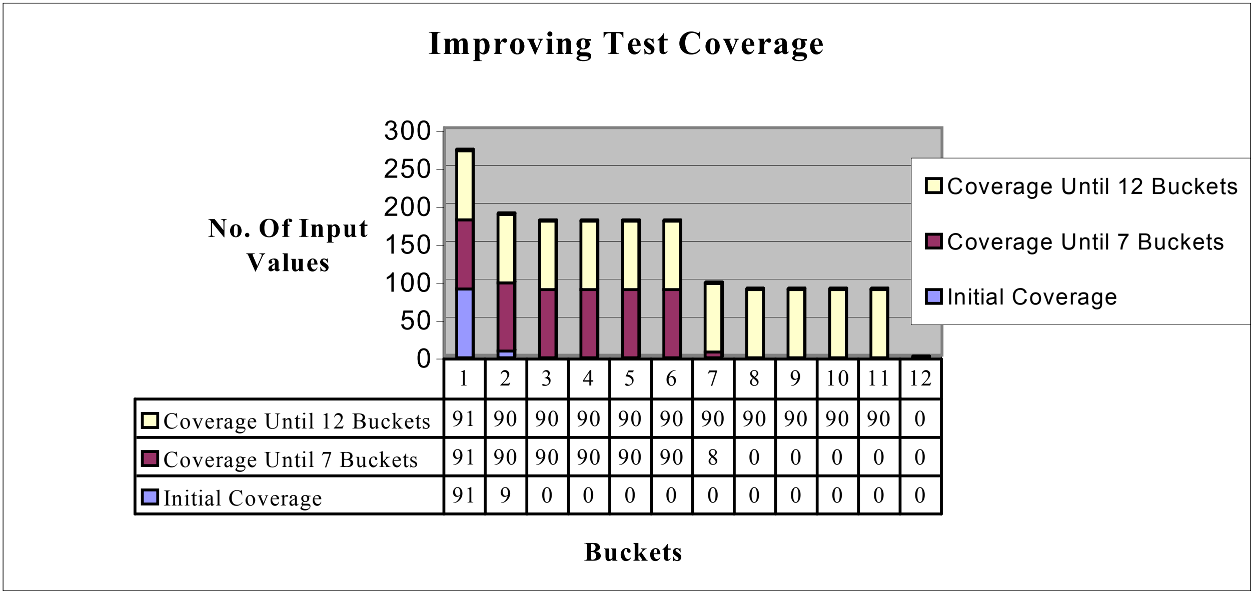
1. The value for the number of tests run is set at 100.
2. The accuracy of each input variable is set at 2.
3. The test minimum and the test maximum of all the three input variables are set to the actual input maximum and input maximum that the model inputs can take. For the first input variable, the test boundaries are 10 – 100. For the second input, the boundaries are set at 10 – 200 and for the third input they are set at 10 – 500.
4. The exception types for the four outputs are set as above limit, below limit, outside limits, and none respectively.
5. The results obtained by running D2Min, Min2Max, RASeg, RA2Max, Osc, and CP@Min against the three inputs are presented for analysis in the subsequent sections.

The test case files generated using MATT are read into RATT for analysis. The Domain Coverage metrics and the Test Suggestions obtained after the initial test run in RATT are saved for further comparison. The model is once again simulated in MATT with the new test script parameters specified by the test suggestions proposed in RATT. The test suggestions include manipulating one or more of the test script parameters such as the test boundaries, number of tests, accuracy, and the test type. The new test case file is again read into RATT and the results obtained are compared with the initially obtained domain coverage results. For comparison, both the original file and the file generated after the test suggestions are loaded into RATT at the same time and analyzed. This is done because the test boundaries for subsequent test case files can be less than the initial test boundaries in which case the buckets in the original file are not the same as the buckets in the subsequent files. This process can be iteratively continued until the engineer acquires sufficient knowledge of the defective input domain to debug the software of the defects. The following section describes the results obtained for each test type and an analysis of those results.

Results

Boundary Value Test Types:

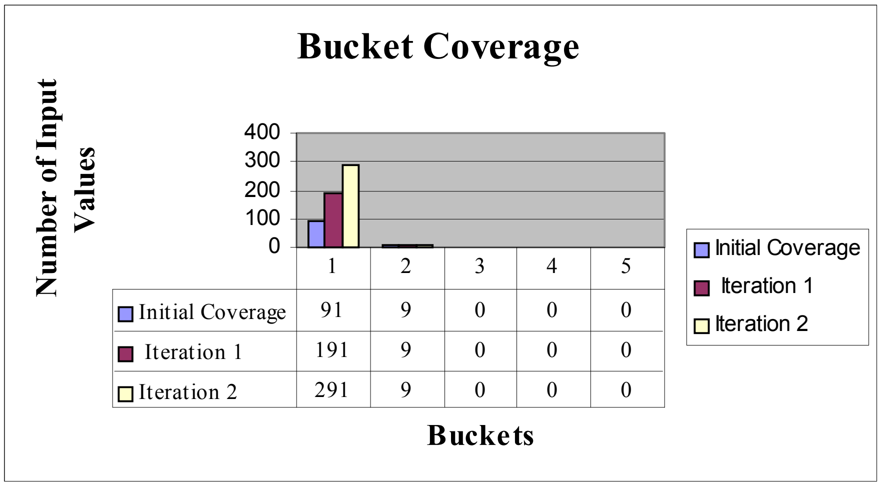
In all boundary value test types, uncovered buckets occur in two ways, both of which were discussed in detail in Chapter 4. Figure 6.1 illustrates the domain coverage values obtained after running the test suggestions proposed for the first case. In this case, all the buckets from [Test Minimum + (Number of Tests – 1) * Step] until 100 remain uncovered because the input value generated in a particular time step depends on the Test Minimum, Number of Tests and, Step. The graph in Figure 6.1 illustrates the test coverage values obtained after running D2Min on the first input variable. Initially, 91 inputs are covered in the first bucket and 9 inputs in the second bucket. All the buckets from 3 until 100 are uncovered. In a case where all 9 inputs in the second bucket are found defective, the user might want to test for more inputs in the second bucket and also in the adjacent buckets to determine if some of the inputs in those buckets might also generate defects. The suggestions enable the user to choose the number of buckets to cover and also provide the user with the coverage in each of those buckets. The graph in Figure 6.1 illustrates the bucket coverage results if the user chooses to cover buckets adjacent to the second bucket because there are a large number of defects in the second bucket. On the second iteration, the D2Min test is repeated with an increase in the value of number of tests to ensure coverage of the first seven buckets. The third iteration will ensure coverage of the first twelve buckets. The user can thus specify exactly the number of buckets to cover after the second bucket. This is especially helpful to uncover the latent defects in the neighboring buckets and because the user specifies the number of buckets to be covered, he can control the concentration of input values in those buckets.



	1	2	3	4	5	6	7	8	9	10	11	12
□ Coverage Until 12 Buckets	91	90	90	90	90	90	90	90	90	90	90	0
■ Coverage Until 7 Buckets	91	90	90	90	90	90	8	0	0	0	0	0
■ Initial Coverage	91	9	0	0	0	0	0	0	0	0	0	0

Figure 6.1: Bucket Coverage Results – Descending to Minimum

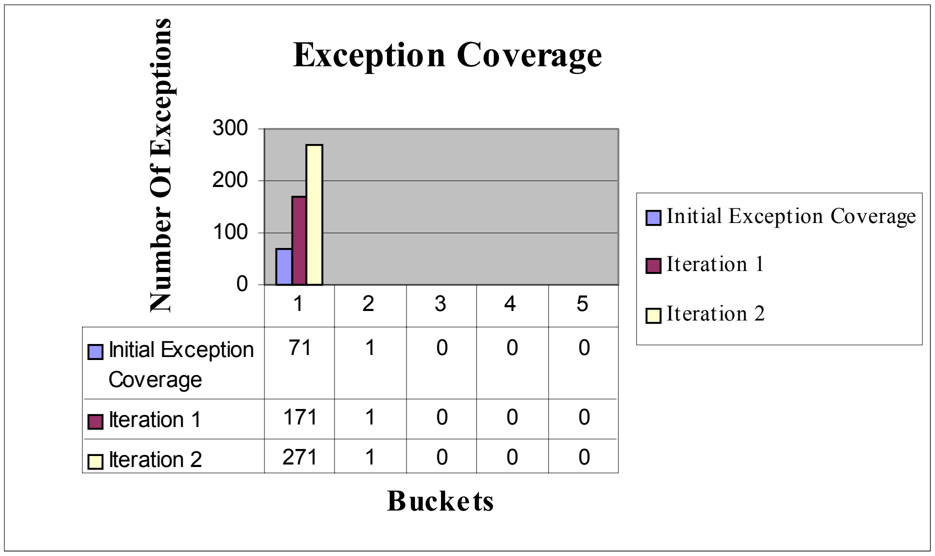
The graph in Figure 6.2 illustrates the number of input values covered in each bucket over successive iterations and the illustration in Figure 6.3 shows the number of input values in each bucket causing exceptions in the outputs. In the initial iteration, only the first two buckets are covered. Out of the 91 inputs covered in the first bucket, 71 inputs produce defects and out of the 7 inputs in second bucket, only one input produces a defect. The algorithm discussed in Chapter 4 is employed here to group the first 3 buckets into a single block and then repeat D2Min test, because the exceptions are more in the first half of the block than in the second half of the block. The new test boundaries are set at 10 and 12.7 respectively, and the value of accuracy is set at 3.



	1	2	3	4	5
■ Initial Coverage	91	9	0	0	0
■ Iteration 1	191	9	0	0	0
□ Iteration 2	291	9	0	0	0

Figure 6.2: Bucket Coverage Results – Descending to Minimum

The original test case file and the file obtained after the test suggestions are loaded into RATT at the same time and analyzed. In the third iteration, the domain was narrowed down further to 10 and 10.135 and the number of tests increased to 200. Here again, more defective inputs were found towards the test minimum. Either the accuracy settings or the number of tests run can be manipulated for successive test runs. Figure 6.2 and Figure 6.3 illustrate the results obtained over three successive iterations. It can be observed from the graph that on subsequent test runs all the inputs in the first bucket produce defects. After testing, the engineer should confidently be able to conclude that all the inputs in the first bucket are leading to defects so that he can check the software for the entire range. Because the accuracy settings or the number of tests are varied over subsequent test runs, the probability that the same inputs being counted twice over iterations is very low. However, this possibility is not completely eliminated.



	1	2	3	4	5
🟦 Initial Exception Coverage	71	1	0	0	0
🟥 Iteration 1	171	1	0	0	0
🟨 Iteration 2	271	1	0	0	0

Figure 6.3: Exception Coverage Results – Descending to Minimum

Let us suppose that the maximum precision of the first input variable is 5. Initially, on the first iteration, input values are generated from 10 to 100. The maximum number of values that can be tested in this domain are 90 * 10^5. On the second iteration, this number is significantly reduced to 270000 as the system is tested only between 10 and 12.7. Similarly, the maximum number of values that can be tested on the third iteration are 13500 as the domain is further narrowed down to 10 – 10.135. This is an order of magnitude reduction of the total number of input values that need be tested in a given domain. On subsequent iterations, this number can be further reduced to a number where the user can actually afford brute force to probe each input in the sub-domain for defects. Figure 6.4 illustrates the results for the first input variable over four test runs.

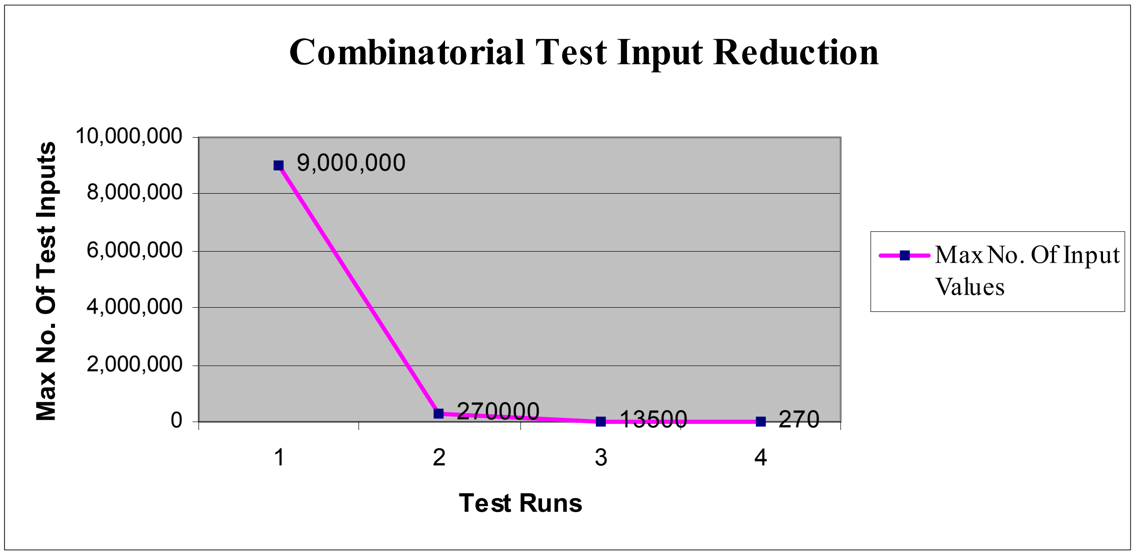


Figure 6.4: Combinatorial Input Domain Reduction – Descending to Minimum

<u>Linear Value Test Types:</u>

The graph in Figure 6.5 illustrates the results obtained by running a linear Min2Max test on the second input variable over three successive iterations. Initially, the test was run between test boundaries 10 – 200 with the value of number of tests set at 100. The bucket coverage is uniform with exactly one value in each bucket. All the buckets from 65 –100 reported exactly one defect, thereby implying that every input value covered in those buckets is leading to an exception.

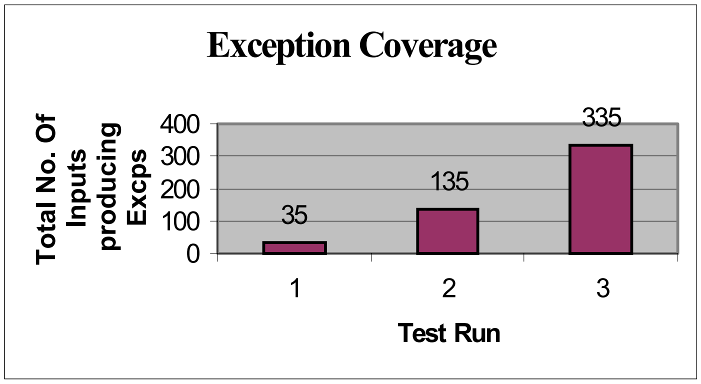


Figure 6.5: Exception Coverage Results – Minimum to Maximum

Therefore, in the initial test run 35 inputs in the bucket range 65 – 100 are producing defects. On the second iteration, all buckets from 64 – 100 are grouped into one block and the Min2Max test is repeated on this block increasing the number of test runs to 200. All the 200 inputs generated during the second iteration were found to produce exceptions increasing the total number of inputs producing exceptions in the buckets 65 - 100 to 235. On the third iteration, a mirror image Max2Min test was run on this domain increasing number of tests to 300. The rationale in running three iterations over the buckets 65 - 100 is to uncover more inputs in this sub-domain leading to defects, which will enable the engineer to confidently infer that all the inputs in this sub-domain are leading to exceptions and therefore test the software for the entire sub-domain. The engineer cannot reach such a conclusion without some degree of testing in the sub domain. Because, the increment used to generate inputs over an iteration is different from the increment used in the previous iteration, the probability of the same inputs being counted twice is unlikely but such a possibility is not completely eliminated.

Random Value Test Types:

This section presents the results obtained by running RASeg and RA2Max test types. These test types use two distinct strategies to generate test values. The RAMin, RD2Min, RDMax, RAMM, and RDMM tests follow the strategy used for RA2Max. RDSeg uses the approach followed RASeg. The graph in Figure 6.6 illustrates the results obtained by running RASeg on the third input variable of the Circuit Test model over four successive iterations. Initially, a RASeg test type was run between test boundaries 10 to 500 with the number of tests set at 100. All buckets from 1 to 11 generated defects.

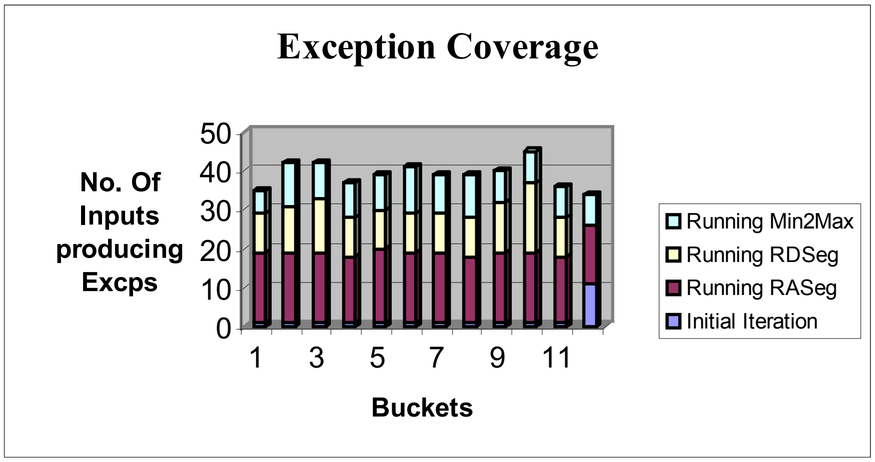


Figure 6.6: Exception Coverage Results – Random Ascending Segmented

The test suggestions proposed in Chapter 4 suggest the user run three different tests that might help uncover more inputs leading to defects in the sub-domain. Therefore, in the second test run all buckets from 1 to 12 are grouped into a single block and RASeg is repeated over this defective sub-domain and the number of tests is increased to 200. The graph in Figure 6.6 shows a more than significant increase in the number of inputs producing defects. On the third iteration, a mirror image random value test RDSeg was run keeping the number of tests constant at 200. RDSeg essentially covers the input sub-domain in the opposite order and might help uncover some inputs leading to exceptions mainly because the time step in which a particular input is generated is different from the previous iteration. On the fourth iteration, a linear Min2Max test was run with the value of number of tests set at 200. The results obtained over all four iterations are shown in Figure 6.6. One can observe from the graph that the number of defects uncovered by running a RASeg is much greater than the number of defects uncovered by running an RDSeg test or a Min2Max test. This suggests the engineer should execute a RASeg test to further test the sub-domains.

The graph in Figure 6.7 illustrates the results obtained by running a RA2Max test on the first input variable. After the initial test run, the first five buckets in the input domain generated defects. The first five buckets were grouped into a single block, and the RA2Max was repeated on the defective input domain keeping the other test case parameters constant. The results obtained are shown in Figure 6.7. On the subsequent iteration, a mirror image RDMax test was run on the defective input sub-domain. These test types mainly check for stabilizing and destabilizing conditions at the maximum. It can be observed from the graph in Figure 6.7 that in some of the buckets, the number of defects uncovered during the second iteration is significantly increased from the initial iteration. This might help the user in uncovering some latent defects, which might remain undetected, even after repeatedly running the RA2Max test. However, running RDMax might result in the same or perhaps less coverage than the previous test run in which case might not prove very useful.

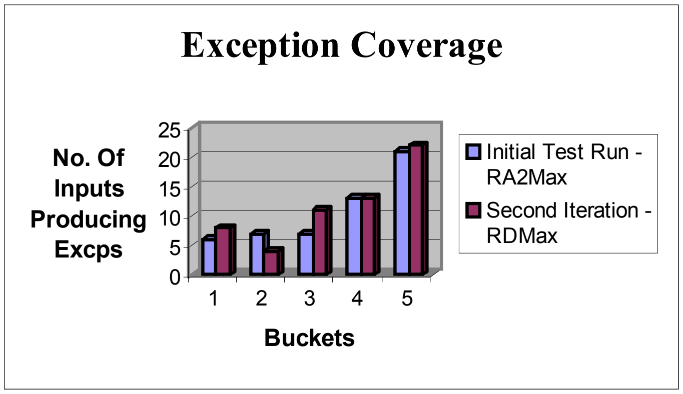


Figure 6.7: Exception Coverage Results – Random Ascending to Maximum

<u>Critical Value Test Types:</u>

The graph illustrated in Figure 6.8 shows the results obtained by running the CP@Min test on the first input variable of the Circuit Test model over three iterations. In the initial test run, all the input values generated are concentrated in the first bucket because the system is tested at the test minimum. Out of the 100 inputs covered in the first bucket, 82 generated defects. On the second iteration, a boundary value test approaching the minimum, D2Min was run on the defective domain with the number of tests set at 100.

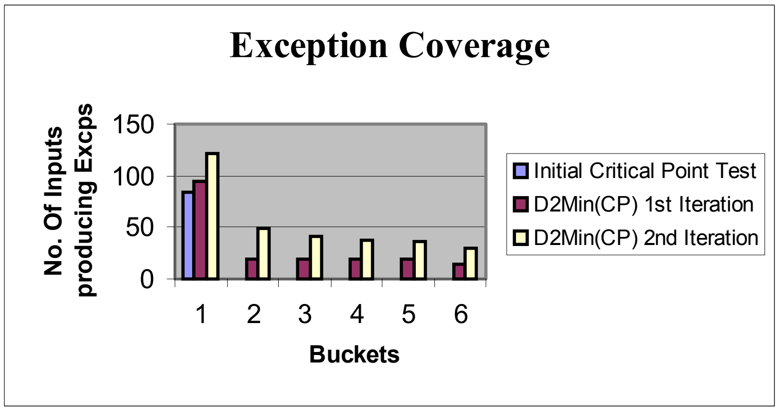


Figure 6.8: Exception Coverage Results – Critical Point At Minimum

The rationale for suggesting a D2Min test is to uncover more inputs in the first bucket that generated defects and at the same time also test for input values in the adjacent buckets producing defects. This is done mainly because a large number of inputs produce defects in first bucket that might lead to defects in subsequent buckets. Also, it might prove very crucial to probe the boundaries of the critical point for any defects, especially when a large number of inputs are producing defects at the critical point. The second iteration uncovered defects in buckets 2 to 6 are used. The third iteration is run again on the same test boundaries but increasing the number of tests to 200. Figure 6.8 shows the increase in the defects being uncovered over subsequent iterations.

<u>Oscillate Value Test Types:</u>

The graph in Figure 6.9 demonstrates the results obtained by running an Osc test on the second input variable over three iterations. Initially, an Osc test was run on the test boundaries 10 to 200 setting the number of tests at 100. The first five input buckets were found to produce most defects. On the second test run these buckets are grouped into one block and an Osc test with the next higher frequency, Osc2, is run the defective sub-domain. This will enable the user to uncover more inputs in the defective sub-domain keeping the other test parameters same as the original iteration. In the third iteration, an Osc4 test was run and the results generated over all the three iterations are shown in Figure 6.9.

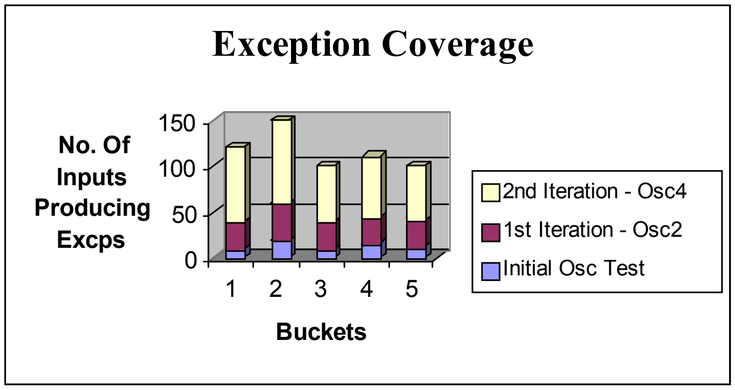


Figure 6.9: Exception Coverage Results – Oscillate Value One

The following conclusions can be drawn from the results obtained after running all the test cases:

1. The increase in the number of defects uncovered over successive test runs is greater than O (n). On subsequent iterations, the engineer will be able confidently infer from the results the exact input sub-domain that is defective. This will enable him to verify the software for the entire sub-domain.
2. The testing suggestions help in obtaining better domain coverage and also enable the user specify the coverage criteria in each input sub-domain.

Hence, the user can specify which buckets he wants covered and also the density of those buckets.

3. Because the test suggestions follow the test domain minimization approach, this significantly reduces the combinatorial number of the maximum input values that can be tested in a given domain. This will enable the user to impose brute force on all inputs in subsequent iterations to pinpoint the exact inputs producing the defects.
4. The tests also help the user to prefer one test to the other when multiple test suggestions are given.

CHAPTER 7

CONCLUSIONS

This chapter discusses some of the limitations of the proposed solution strategy and the future work that could be done to improve the test suggestions.

One of the major limitations of the proposed solution strategy is that the test suggestions depend on the test types used in MATT. If any additional test types are added to MATT, more suggestions for those test types should be added to RATT. However, MATT works seamlessly with $\text{Matrix}_x^{®}$ and is presently being ported to MatLab, and the test suggestions of RATT are designed to work with the MatLab version of MATT. Without question, an application independent solution strategy can be developed from this work to guide the user towards further domain testing in general. Currently, RATT throws an exception whenever the user selects a test type that it does not recognize.

Another limitation of this work is that only the parameters of a single previous test run are considered while suggesting subsequent tests. The test suggestions might be improved if a strategy could be developed that keeps track of all the test script parameters for an input variable over successive iterations and then suggest an appropriate test taking into account all the previous iterations run on this input variable.

The next important limitation of RATT is the test suggestions provided for multiple files. When multiple files are loaded into RATT for analysis, an input variable in these files can contain different test types. Currently, a linear Min2Max test has been proposed with the test minimum set to the minimum of the test minimums of the input variable and the test maximum set to the maximum of the test maximums of the input variable over all test case files. The value of the number of tests run is also chosen as the maximum of the value of the number of tests over all the files. More appropriate test suggestions could be provided by considering the various combinations of test types that could be chosen for a particular input variable. This problem presents a significant combinatorial and mathematical problem that may only be solvable through user guidance.

One final limitation of the test suggestions in RATT is that currently no suggestions have been provided for the input combinations producing defects. Pattern recognition or string matching techniques might be used to precisely determine the bucket combinations producing defects and suggest appropriate tests.

In spite of above-mentioned limitations, the suggestions help in guiding the user to a more effective degree of testing capability in setting up further domain tests. However, this strategy can be considered only as a step towards increasing the domain testing effectiveness.

BIBILIOGRAPHY

[1] Henry, J., and Patterson-Hine, A., “An Effective Strategy for Testing of Real Time Software”, International Symposium on Software Testing and Analysis, Portland, Oregon, August 21-24, 2000

[2] Integrated Systems Inc. MATRIX$_x$® [On-line]
URL: http://www.isi.com/products/matrixx/

[3] MATT User Guide [On-line]
URL: http://cscidbw.etsu.edu/matt/docs/matt_userguide/matt_userguide.htm

[4] Koneru, N., “Quantitative Analysis of Domain Testing Effectiveness”, ETSU Masters thesis, 2001

[5] Henry, J., Turlapati, R., Koneru, N., "Quantitative Evaluation of Domain Testing", Testing Computer Software, June 18-22, 2001, accepted for publication

[6] W. Eric Wong, Joseph R Horgan, Aditya P. Mathur, Alberto Pasquini, “Test Set Size Minimization and Fault Detection Effectiveness: A Case Study in a Space Application“, COMPSAC '97 - 21st International Computer Software Conference

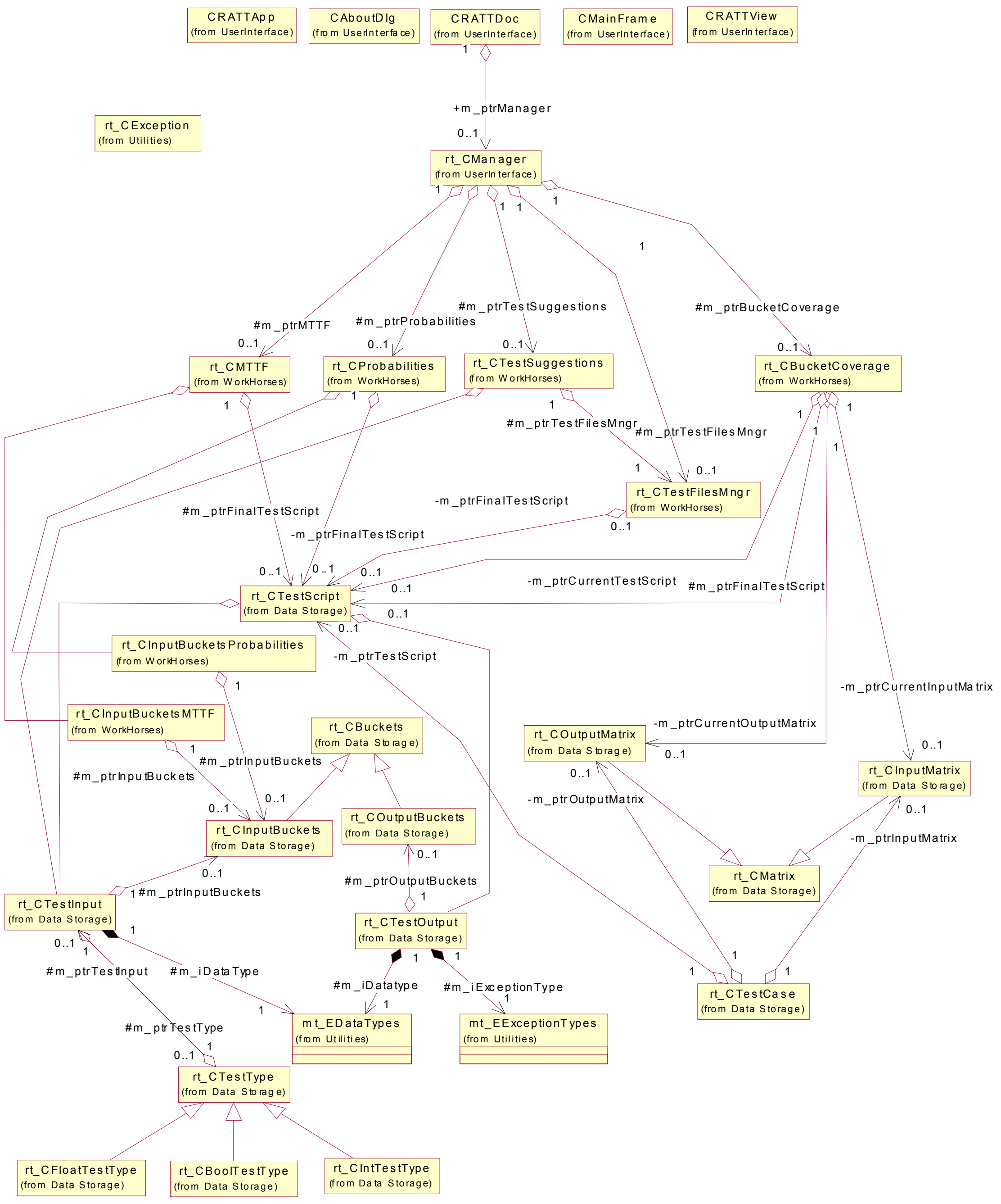
[7] “Formal Verification, Testing and checking of Real-time Systems”, ACM Computing Surveys, December 1996
URL: http://www.acm.org/pubs/articles/journals/surveys/1996-28-4es/a182-lee/a182-lee.html

[8] Abdeslam En-Nouaary, Ferhat Khendek, Rachida Dssouli, “Fault Coverage in Testing Real-Time Systems “, Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications

[9] J.D. Musa, A. Iannino, Kazuhira Okumoto, Software Reliability: Measurement, Prediction, Application. New York: McGraw-Hill, 1987.

APPENDIX

RATT Overall CLASS DIAGRAM



CRATTApp (from UserInterface)
CAboutDlg (from UserInterface)
CRATTDoc (from UserInterface)
CMainFrame (from UserInterface)
CRATTView (from UserInterface)
rt_CException (from Utilities)
+m_ptrManager
rt_CManager (from UserInterface)
#m_ptrMTTF
#m_ptrProbabilities
#m_ptrTestSuggestions
#m_ptrBucketCoverage
rt_CMTTF (from WorkHorses)
rt_CProbabilities (from WorkHorses)
rt_CTestSuggestions (from WorkHorses)
rt_CBucketCoverage (from WorkHorses)
#m_ptrTestFilesMngr
#m_ptrTestFilesMngr
rt_CTestFilesMngr (from WorkHorses)
#m_ptrFinalTestScript
-m_ptrFinalTestScript
-m_ptrFinalTestScript
-m_ptrCurrentTestScript
#m_ptrFinalTestScript
rt_CTestScript (from Data Storage)
-m_ptrTestScript
rt_CInputBucketsProbabilities (from WorkHorses)
rt_CInputBucketsMTTF (from WorkHorses)
rt_CBuckets (from Data Storage)
#m_ptrInputBuckets
#m_ptrInputBuckets
rt_CInputBuckets (from Data Storage)
rt_COutputBuckets (from Data Storage)
rt_COutputMatrix (from Data Storage)
-m_ptrCurrentOutputMatrix
-m_ptrCurrentInputMatrix
rt_CInputMatrix (from Data Storage)
-m_ptrOutputMatrix
-m_ptrInputMatrix
rt_CMatrix (from Data Storage)
#m_ptrOutputBuckets
#m_ptrInputBuckets
rt_CTestInput (from Data Storage)
rt_CTestOutput (from Data Storage)
rt_CTestCase (from Data Storage)
#m_ptrTestInput
#m_iDataType
#m_iDatatype
#m_iExceptionType
mt_EDataTypes (from Utilities)
mt_EExceptionTypes (from Utilities)
#m_ptrTestType
rt_CTestType (from Data Storage)
rt_CFloatTestType (from Data Storage)
rt_CBoolTestType (from Data Storage)
rt_CIntTestType (from Data Storage)

VITA

RADHIKA TURLAPATI

Personal Data:	Date of Birth:	April 9, 1978
	Place of Birth:	Hyderabad, INDIA

Education:	Atomic Energy Central School, Hyderabad, India Atomic Energy Junior College, Hyderabad, India VR Siddhartha Engineering College, Vijayawada, India; Computer Science, B.E., 1999 East Tennessee State University, Johnson City, Tennessee; Computer Science, M.S., 2001
Professional Experience:	Software Intern, Electronics Corporation of India Ltd., Hyderabad, India, 1998 Software Engineering Intern, Prithvi Information Solutions Inc., Pittsburgh, 2000 Graduate Assistant, East Tennessee State University, Johnson City, Tennessee, 1999-2001
Publications:	Henry, J., Turlapati, R., Koneru, N., "Quantitative Evaluation of Domain Testing", Testing Computer Software, June 18-22, 2001, accepted for publication
Honors and Awards	Outstanding Scholastic Achievement Award, East Tenessee State University, 1999 - 2000