5-2001

Quantitative Analysis of Domain Testing Effectiveness.

Narendra Koneru
East Tennessee State University

Quantitative Analysis of Domain Testing Effectiveness

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 Sciences

---

by
Narendra Koneru
May 2001

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

Key Words: Real-time Systems, Simulation, Software Testing, Domain and Fault Coverage, Software Reliability, Reliability growth models.

ABSTRACT

Quantitative Analysis Of Domain Testing Effectiveness

By

Narendra Koneru

The criticality of the applications modeled by the real-time software places stringent requirements on software quality before deploying into real use. Though automated test tools can be used to run a large number of tests efficiently, the functionality of any test tool is not complete without providing a means for analyzing the test results to determine potential problem sub-domains and sub-domains that need to be covered, and estimating the reliability of the modeled system.

This thesis outlines a solution strategy and implementation of that strategy for deriving quantitative metrics from domain testing of real-time control software tested via simulation. The key portion of this thesis addresses the combinatorial problems involved with effective evaluation of test coverage and provides the developer with reliability metrics from testing of the software to gain confidence in the test phase of development. The two approaches for reliability analysis- time domain approach and input domain approaches are studied and a hybrid approach that combines the strengths of both these approaches is proposed. A Reliability analysis Test Tool (RATT) has been developed to implement the proposed strategies. The results show that the metrics are practically feasible to compute and can be applied to most real-time software.

ACKNOWLEDGEMENTS

Firstly, I would like to thank my parents and my sister for giving me the support and encouragement necessary to achieve my educational goals. Next, I express my sincere gratitude to my thesis advisor, Dr. Joel Henry, for his valuable time, support, excellent direction, and motivation throughout the project. I must acknowledge NASA for the monitory support provided. I also gratefully acknowledge the help and support extended by my committee chair, Dr. Martin Barrett, for his valuable time and suggestions. Finally a special thanks to my friend, Radhika Turlapati, for her support throughout the project.

CONTENTS

Chapter		Page
ABSTRACT. ……		2
ACKNOWLEDGEMENTS. ……		3
LIST OF ILLUSTRATIONS……		7
1.	INTRODUCTION…..……..	8
	Importance of Real Time Software Testing...……	8
	Outline of Thesis……...…..……	9
2.	PROBLEM STATEMENT……	11
	Terminology and Definitions.……	11
	Testing Real-Time Systems.…….….	12
	Goals of Testing…..……...…….	13
	Steps in Real-Time Testing……..	14
	Simulating the System's Operating Environment……	14
	Selecting the Test Cases……	15
	Run the Test Cases……	15
	Analyze the Test Results……...	16
	Problems and Challenges in Testing Real-Time Software……..	16
	Huge Domain Space……..	16
	Faulty Domain Combinations……	17
	Need for Quantitative Metrics……	18
	Goals of the Thesis……..	19

Chapter Page

3. QUANTITATIVE METRICS FOR REAL TIME SOFTWARE…… 20

Software Reliability……. 20

Reliability Metrics…… 21

Factors Influencing Software Reliability…….. 23

Reliability vs. Cost……... 23

Reliability vs. Time…….. 24

Reliability vs. Efficiency…….. 25

Reliability vs. Operational Profile…… 25

Improving Reliability…… 25

Time Domain Models…….. 27

Input Domain Models…….. 29

4. SOLUTION APPROACH..……. 31

Domain Percentile Bucketing…… 32

Probability Metrics…….. ……. 35

Reliability Metrics Describing Output Variable Correctness.. 35

Reliability Metrics Based on Valid Input States…….. 36

MTTF Results……... 38

5. SOLUTION IMPLEMENTATION…….……. 40

Overview of MATRIX$_x$®and MATT……... 40

RATT Design…….……...…… 42

Data Storage Component……. 43

Work Horses Component……. 45

Chapter Page

RATT Implementation and GUI……... 46

File Menu……. 47

Edit Menu……. 49

Run Menu……. 49

Output Menu..……. 50

6. RESULTS……..…… 52

Domain Percentile Bucketing…….…….. 53

Percentile Coverage……. 53

Exception Coverage……. 54

Input Bucket Combinations……. 56

Quantitative Metrics……. 57

Reliability Results Based on Output Variable Correctness….. 57

Reliability Results Based on Input Variable Correctness…… 59

MTTF Results.……. 60

7. LIMITATIONS AND FUTURE WORK…….…….…… 63

BIBILIOGRAPHY……..…….…….. 65

APPENDIX……. 68

VITA……..…… 70

LIST OF ILLUSTRATIONS

Figure		Page
3.1	Reliability vs. Cost……...	23
3.2	Time vs. Failure Rate……...	24
5.1	Package Diagram for RATT……	43
5.2	Class Diagram for RATT Data Storage Component……	44
5.3	Class Diagram for RATT Work Horses Component……..	45
5.4	RATT Main Window……..	47
5.5	Open Dialog Box……..	48
5.6	RATT Window with Results……	51
6.1	Bucket Coverage……..	53
6.2	Exception Coverage…….……....	55
6.3	Input Bucket Combinations……..	56
6.4	Probability of an Exception in any One Time-Step……	57
6.5	Probability of an Exception in an Output Variable in any Time-Step	58
6.6	Probability of System Success Over all Time Steps (4 Test Runs) …	59
6.7	Probability of System Success Over all Time Steps (5 Test Runs) …	60
6.8	System Mean Time To Failure (MTTF)…….	61

CHAPTER 1

INTRODUCTION

This thesis outlines a solution strategy and implementation of that strategy for deriving quantitative metrics from domain testing of real-time control software tested via simulation. The key portion of this thesis addresses the combinatorial problems involved with effective evaluation of test coverage and provides the developer with reliability metrics from testing of the software to gain confidence in the test phase of development. A Reliability Analysis Test Tool (RATT) has been developed to accomplish these tasks. RATT evaluates the test results produced from testing the software and provides reliability metrics along with quantitative test coverage measurements for input and output domains of the system.

Importance of Real Time Software Testing

The increased flexibility and functionality provided by digital computers had led to an ever-increasing demand for use in many different applications. The tremendous amount of computing power provided by these machines, at exceptionally low cost, has enabled them to be used in building more complex and larger real -time control systems. Many of these systems are used to manage safety critical tasks. Examples of these tasks include controlling of wind tunnels, piloting of vehicles, and monitoring and coordinating missiles for control systems. Failures in such computer systems may cause great financial loss, an environmental disaster, or even loss of human lives. The criticality of the tasks involved and the potential heavy losses associated with failures of control systems place

stringent requirements on the quality of real-time software. There is a great need for this control software to be tested thoroughly before it is deployed for actual use. This in turn forces the developer to measure software testing quantitatively and attain maximum quality in the system before its final release. There are many challenges involved in testing real-time software, including the unavailability of target hardware, huge input and output domain spaces, and the time constraints of real-time systems. Merely determining the correctness of outputs is also a major challenge. These problems make assessing the quality of software testing and achieving the required quality all the more difficult. There is a need for systematic testing of the control software using automated test tools and analyzing the test results to measure overall system reliability.

This research is focused on reducing the overwhelming combinatorial domain space problem and analyzing the test results for reliability predictions of large real time software control systems. The reliability metrics are aimed to guide the test process and improve the confidence of the developer in the system as the test phase progresses.

Outline of Thesis

A precise statement of the problems addressed by the thesis is defined in Chapter 2. An overview of testing and quantifying real time software is given in Chapter 3.

A solution for the problem is proposed in Chapter 4. A brief description of the two tools used for analyzing the proposed metrics, MATRIX$_x$® and MATT**,** is given to introduce the concept of testing real time systems using simulation.

Chapter 5 covers the design and implementation approach used for RATT. It also discusses the use of RATT within NASA and other organizations.

Chapter 6 deals with the results obtained from RATT. Ideas and areas that can be explored further to improve the metrics are given in chapter 7.

Formulas used for obtaining reliability metrics are presented as well as software reliability growth models. These topics lead to discussion of ways to improve reliability and are described here as well.

CHAPTER 2

PROBLEM STATEMENT

This research focuses on addressing the problems and challenges involved in testing and improving the quality of real-time software. The aim of the study is to provide the software developer with quantitative reliability measures of the developed real-time control software. The remainder of this chapter introduces real-time testing and describes the problems and challenges involved in testing real-time systems.

Terminology and Definitions

This section introduces definitions that will be used throughout the thesis [9]. *Reliability* is defined as the probability that a system functions according to its specification for a specified time and in a specified environment. It gives a measure of confidence in the system for the user. A *failure* occurs when the run-time system behavior does not match the specifications. A *fault* is a static component of the software that causes the failure. A fault causes failure only when that part of the code is executed, and hence, not all faults result in failures. An *error* is a programmer action or omission and results in a fault. *Mean time to Failure (MTTF)* is defined as the time between successive failures. Reliability is a statistical study of the failures, and MTTF is one of the measures of reliability. A *domain space* for the variable is defined as the set of all possible legal values that the variable can assume during system operation.

To achieve high system reliability, extensive testing is required, and, hence, some terms used in testing are defined next. A *Test Case* is defined as a single value mapping

for each input of the program that enables a single execution of the software system. *Test Script* is a set of conditions specified on each input variable for guiding the automatic generation of test cases. A *Test Suite or a Test Run* is a set of test cases that are executed sequentially.

Testing Real-Time Systems

Real-time systems interact with the environment in a time constrained input/output event model. Real-time systems sample external devices, execute a loop in the program, and then refresh the outputs, typically within milliseconds of the input of data or stimuli. Input and output data are sampled many times per second and this process is continued over possibly long periods of time. The precision of an output depends both on the logical and temporal correctness of values. These factors make real time systems inherently complex, which in turn make testing highly difficult. As many real-time software systems control mission critical applications, it is very important that these systems are highly reliable and the only way to assure that the system is reliable is by exhaustive testing. In spite of employment of the best available software development techniques and the participation of highly talented personnel, there have been serious software errors, some of which led to unbelievable failures like the destruction of a French meteorological satellite [8]. One of the causes of this failure was attributed to non-execution of a valid input combination during the test phase. Hence, thorough testing of the accuracy of the outputs and response times is necessary before the software can be deployed in the real environment.

The rest of this chapter gives a description of the goals and steps involved in real time testing and also discuss the challenges involved.

Goals of Testing

Testing is a validation process that determines the conformance of the software's implementation to its specification. It is an important phase in software development life cycle and is even more important in real time systems.

There are two main goals of testing real-time software. On one hand, testing can be viewed as a means of achieving required quality of the system. The main aim here is to probe the software for defects and fix them. This is also called *debug testing* and is assumed to be very effective in uncovering potential faults. On the other hand, testing can also be viewed as a means of assessing the existing quality of the system and provide fault coverage measurement, which consists of studying the potential faults generated by a given test suite. This method is called *operational testing* and is proved to be effective in predicting the future reliability of the system.

In debug testing, a systematic approach is followed for selecting test cases based on situations likely to produce the most number of errors. The drawback of this approach is that it may uncover failures with negligible rates of occurrence and risk, and the test effort may not worth the reliability improvement achieved. Another drawback is that it does not provide complete mathematical and technical validity of the reliability assessments.

In operational testing, a test case is selected based on the probability of its occurrence in the real operating environment. Hence, this method is likely to uncover

failures with highest probability of occurrence first and provide accurate assessment of current reliability. This is done after the debugging phase, and the objective is to assert that the system is reliable and gain confidence for the final release of the system. The method is not very effective in improving the reliability of the system, and at the same time it is very difficult to generate an accurate operational profile.

The main goal of this thesis is to assist the developer to improve the quality of the real-time control models by accurately predicting the domain areas of weakness based on test results. A blend of debug testing assisted by the domain knowledge of the final users is used in this thesis to improve reliability.

Steps in Real-Time Testing

Real-time software works in association with external environment and hence validating the system against the functional specifications is important. This form of testing is called Black-Box Testing and is the process in which test cases are selected based on the requirements and external design specifications, rather than the knowledge of internal details of the program. Testing real-time systems involves four phases and each phase is described below [10].

Simulating the System's Operating Environment

Testing real-time software cannot always be done on the target hardware and in the target environment. This occurs due to the costs and risks associated with the potential damage of the sophisticated target hardware environment. The tester's main aim is to break the software system that in turn may lead to the rupture of the operating environment. Also, the target hardware may not be available for testing during the

maintenance phase and, in aerospace applications, target hardware may not even exist during the test phase. The expensive hardware can wear out during testing and may not be useful by the time the product is deployed. Hence, simulation is used by many organizations to test functions that cannot be tested on the target hardware. Also, simulation is cheaper and faster than testing on the real hardware. With simulation, more tests can be executed without hardware degradation. In view of these advantages, simulation is used in this thesis for testing and analyzing the test results.

Selecting the Test Cases

The input and output domain space for a complex system is typically very large and completely impractical for manual testing of each combination of the input values within the domain space. An automated test tool for generating test cases based on user-specified criteria is needed to cover as much of the test domain as possible. Automated testing also allows the many more tests to be run than manually testing and permits tests to be easily run over and over again. This repeating of previous tests is an important step in testing and is referred to as regression testing. Regression testing attempts to assure that software failure correction does not introduce additional failures. In chapter 4 of this thesis a brief description of an automated test tool is given that generates large numbers of test cases based on some user selected test criteria.

Run the Test Cases

Once the test suite is selected, it must be run in the simulated environment sequentially and the results must be captured. Any failures generated also have to be captured and reported to the user. This involves integrating the automated test tool with the simulated environment.

Analyze the Test Results

This is the most important part of testing and should be done very carefully. The test results are used as a mechanism for identifying the defects in the software or the model and should be used as a mechanism to quantify the software reliability. The test results have to be analyzed to assess the quality of the software and to determine potential problem areas that require more testing and to identify portions of the input domains that have not been tested. An important consideration here is to leverage measurements against multiple independent test runs. A decision regarding the end of the test phase is to be made based on all test runs.

These are the four essential steps that are to be performed when testing real-time software, and the importance of the last step can never be realized until it is done. This thesis mainly focuses on analyzing the test results from multiple independent test runs and providing measurements indicating the overall reliability of the system.

Problems and Challenges in Testing Real-Time Software

Although there are many issues involved with simulation and automated test case generation, much work has already been done in this area and there are many software packages available that provide this functionality. An area to be explored further is the analysis of the test results. The rest of this section discusses the problems involved in analyzing the test results and quantifying the software.

Huge Domain Space

Complete testing of real-time software presents some interesting challenges that are not encountered in other software application areas. The input and output domain

space of a real-time system can be very large and a quantitative evaluation of domain testing faces some difficult combinatorial problems. For example, an input domain of 0.0 to 99.9999 with an accuracy of just four decimal points gives us 10^6 (1,000,000) possible values. Consider a simple real-time system with two inputs with the same range; the total number of possible input combinations would be 10^{12}. This is true for the output domain as well if two outputs exist with ranges of 10^6. Soon these figures become unmanageably large even for relatively small real-time systems. Given the combinatorial complexity, we need to find a way of validating the software with a reasonable and tractable number of test cases and determine if a test effort is sufficient. An effective solution to this problem has to be found that makes the combinatorial problem tractable and determines if sufficiently large numbers of test cases are executed to satisfy a minimal portion of both input and output domain values and combinations.

Faulty Domain Combinations

Another important concern in testing real-time systems, which is not involved in non-real-time software, is the complexity involved in determining the combination of input subsets producing a significant number of faults. For example, when we are testing a word processor application for its correctness in saving the file with a given name, it does not make sense to test for all combinations of input characters for the name of the file. However, this is not same as testing a missile control system with two hardware valves controlling the speed and direction of the missile. Values of 10.1, 20.3 for these valves may be valid but values of 10.2, 20.3 may increase the speed to unacceptable levels causing the missile to miss its target. The combinations of speed and direction valves causing miss hits are to be determined so that further testing in those areas can be

taken to pinpoint the fault. Hence, it is important to determine the combination of input domain subsets producing highest number of errors.

The inherent complexity and huge domain space of real-time systems make tracking the test process more difficult. As testing is a continuous process, test results from multiple test runs have to be analyzed together and measurements computed based on these results. An effective solution addressing this problem has to be found.

Need for Quantitative Metrics

The reliability of real-time software should be as high as possible. Reliability of the system is not a measure of the test effort but rather a measure of the probability of the system's correct operation given its specifications. Software metrics increase the confidence of the developer in the system and determine if the software is ready for release. These metrics also help the developer to actually track reliability changes over time and assess the test process for improvement. Reliability is an important and measurable aspect of software quality and is a measure of how well the system meets its specifications. Reliability metrics should depend on the attributes of the system and should change accordingly. As response times are an important concern in real-time systems, reliability metrics should not only predict the probability of system failure for all possible input states but also should predict the time interval between successive failures. At the same time, metrics also should provide the dependability or probability of successful operation for each output variable. This is important in real-time systems as each output variable may control an external device and the probability requirements should match the priorities of the external devices. For example, the probability of failure

for hardware valve controlling the in-flow of plutonium into a nuclear reactor should be much lower than another valve controlling the in-flow of water.

Goals of the Thesis

Software reliability measurements are evolving and studies have shown that there is no exact method of measuring the reliability of software. The goals of this thesis are to provide measurements for the domain analysis of the software control systems, address the software quantification problem, and estimate reliability. The final goal of the thesis is to produce a well-designed and extensible software product with a minimal interface that assists the user in analyzing the test results and calculates the reliability metrics. A standard ASCII text file should be generated with the all the reports. It is also necessary to have proper design documentation for the software to support future extensions.

CHAPTER 3

QUANTITATIVE METRICS FOR REAL TIME SOFTWARE

Before a solution can be proposed for the problems stated in the previous chapter, a thorough understanding of factors affecting software reliability and existing reliability growth models is required. This chapter introduces software reliability metrics and reliability growth models and describes techniques for applying reliability metrics to real-time software.

Software Reliability

Software reliability is an important measurable aspect of software quality. According to the American National Standards Institute (ANSI), Software Reliability is defined as:

> "The probability of failure free operation software operation for a specified period of time in a specified environment" [5].

Informally, Software Reliability can be defined as a measure of how well the users think the system operates according to its specifications. Software Reliability cannot be defined objectively and it means different things in different contexts. It can be fairly assumed that software does not wear out over time like a physical device, and, hence, software reliability is not linearly dependent on time and is always a probabilistic function of time.

Software reliability estimation is a difficult problem not suitable for the standard models used in other engineering disciplines for estimating reliability. Software failures are different from hardware failures. Hardware failures occur due to physical faults

resulting from malfunctioning of one or more physical components. On the other hand, software failures are always design faults, occurring from the misinterpretation of specifications or incorrect implementation of those specifications. Software failures can be closely related to fuzzy human errors and are usually difficult to visualize and correct. A simple replacement of a component causing a failure, as is typically done in hardware engineering, may not always remove faults and hence may not always result in reliability growth. Hence, the standard models of hardware reliability cannot be applied for estimating reliability metrics for software.

Software reliability models are also complicated by the fact that not all software failures are equally catastrophic. Removing *X*% faults may not always increase the reliability by a corresponding *X*%. Reliability is more often improved when faults present in most frequently used parts of software are removed. It is often necessary to identify different failure categories and specify different measurements for each of them. The main objective in reliability improvement is to remove most frequently occurring faults [9].

Reliability Metrics

While hardware reliability metrics are dependent on component failures and the need to replace a component, this approach cannot always be applied to software. Several metrics proposed in the literature closely relate to software reliability and some of them are described next [5].

Mean time to failure (MTTF) is defined as the time interval between successive failures. An MTTF of 100 indicates that the average failure interval is 100 time units. The

time units are totally dependent on the system and it can even be specified in the number of transactions, as is the case of database query systems. For time-based systems, to ensure that there is no failure within a transaction, MTTF should be at least greater than response time [5].

System Availability is a measure of the time during which the system is available over long time periods. It takes into account the repair and restart times for the system. An availability of 0.99 indicates the system is available 99 out of 100 time units [5].

Probability of failure on demand (POFOD) is defined as the probability that the system will fail when a service is requested. A POFOD of 0.1 indicates that at least one out of 10 service requests will fail. POFOD is an important measure for safety critical systems and should be kept as low as possible [5].

Rate of failure occurrence (ROCOF) is the number of failures occurring in unit time interval. A ROCOF of 0.02 means 2 failures will occur for every 100 operational time unit steps [5].

A measure of the number of values tested and the number of inputs causing failures in each input sub-domain, gives an indication of the confidence for that input variable. This is particularly important for software with a large number of inputs and outputs each covering a large range of values [2].

Because real-time software is used to control hardware events and these events are prioritized, an important metric would be to find the *probability of failure for a given output variable* controlling the event. A lower probability of failure is desired for events with high priority.

Factors Influencing Software Reliability

There are many factors influencing software reliability growth. Increase in reliability typically results from a change in one or more of these factors. The factors affecting reliability and their influence are discussed next.

Reliability vs. Cost

Reliability is linearly dependent on cost. Testing software systems to achieve more reliability is a costly process and a tradeoff has to be chosen between the costs involved in finding and removing the failures and the costs involved in repairing the system when the failure occurs in the operating environment. The following graph demonstrates the relationship between reliability and cost [9].

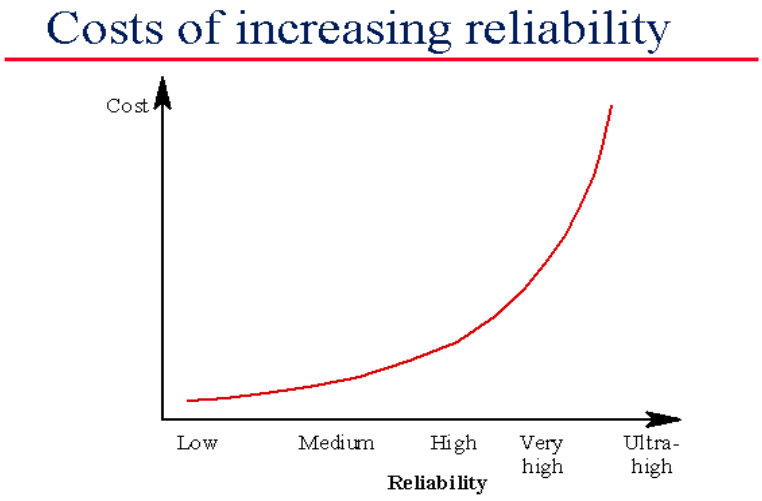


Figure 3.1: Reliability vs. Cost.

As the graph shows, a higher level of desired reliability requires more testing effort and consequently costs more. For normal business applications, modest reliability may be adequate and the failure removal costs may be more than the actual costs of the failures. However, in real-time software the costs of failures in the operating environment

may be very high and hence it is desirable to remove as many faults as possible before the system is deployed.

Reliability vs. Time

Reliability changes only when the software is modified either during the development or maintenance phases, and, hence, the time required to achieve the desired reliability level is proportional to the actual reliability of the software. The following graph shows the relationship between time and reliability [8].

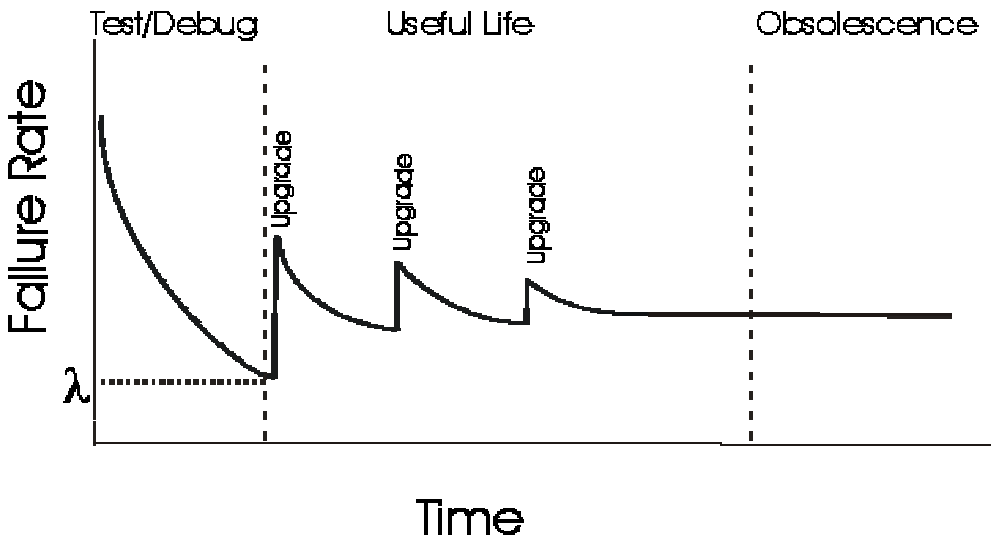


Figure 3.2: Time vs. Failure Rate.

Software reliability is influenced by fault introduction resulting from new or modified code, fault removal that occurs during debugging, and the environment in which the software is used. The failure intensity decreases rapidly during initial test phases, and this can be attributed to the removal of failures with highest probability of occurrence. Software does not wear out with time and the quality of software will not change once it is deployed. Hence, software reliability is not dependent on operational time and the straight horizontal line at the end of the graph indicates this.

Reliability vs. Efficiency

Reliability is often increased by making the systems redundant and by additional run time checks, thus decreasing the overall use of the system resources. Hence, efficiency of the system is decreased with increase in reliability. However, in real-time control software, failure costs are much higher than the total system costs. Because hardware costs have come down rapidly, sometimes the need to use a resource's capacity completely is not as vital.

Reliability vs. Operational Profile

Reliability improvement is largely dependent on the availability of operational profile that statistically models the patterns in which the system is more likely to be used in the operating environment. An accurate operational profile tends to bring out most frequently occurring failures first. However, generating an operational profile is a non-trivial process and is especially difficult for real-time software.

All the above factors make reliability analysis and growth in real-time software a difficult process. As reliability is an important issue in real-time systems, an attempt should be made to increase reliability, which in turn requires the allocations of sufficient resources.

Improving Reliability

A large number of reliability models are proposed in the literature for improving the reliability of the software, but there is no single model that can be used in all situations. Every model is dependent on a set of constraints and assumptions, and not all of them are valid for every system. The rest of this section gives an overview of some

main reliability models that are applicable for quantifying reliability of real time software.

One of the standard ways for improving software reliability is to use formal methods for the software development where the implementation of the software can be traced back to the requirements specification. The entire process can be automated and the number of remaining faults in the software can be determined directly. The development of formal specification forces a detailed analysis of the system and leads in better understanding of the requirements. However, the inherent complexity of formal methods makes them difficult to use in practice, even for trivial real time systems. Formal methods may hide problems that users do not understand, and at the same time formal program proofs can contain errors, resulting in generation of false passes.

Hence other models are proposed in the literature to increase the reliability of real-time software, each based on some assumptions about the operating environment of the software. These models can be classified using several criteria.

Based on the time at which reliability assessment is done, these models are classified into the following two criteria: *Reliability growth models and Reliability models. Reliability Growth Models* are applied during the debug phase of the software development life cycle and are aimed at improving the reliability of the system. They model repeated testing, failure detection, and correction. These models calculate MTTF based on the observed failure patterns and help the managers to decide when to release the final product. *Reliability models* on the other hand are used during the operational stage after debugging of the software development and are aimed at *assessing* the reliability of the software. An accurate operational profile is required and these models

predict MTTF that the users can expect from the system by fitting the failure patterns to statistical models.

Reliability growth models use feedback data to predict the reliability of the system, and the predictions are very accurate in most cases. Conversely, Reliability models use statistical models to predict the future reliability of the system, and, hence, these models need to be very accurate. The correctness of reliability models depends on the accuracy of operational profile, which is a difficult task. More courage is needed to trust the reliability models than the reliability growth models. As generation of an accurate profile is very difficult in real-time software, in this research reliability growth models are used to model the characteristics of reliability improvement.

Based on the data analyzed for predicting reliability, reliability improvement models can again be classified into two categories- *time domain models and input domain models* [3]. *Time domain models* assess software reliability by calculating the ability of the software to function properly over time. *Input domain models*, on the other hand, assess reliability by calculating the ability of the software to function properly for different sets of input. Time domain models stress and predict the overall reliability of the system where as the input domain models provide information relating the input states to the reliability. Input domain models can be used to thoroughly test the software before its final release.

Time Domain Models

Reliability in time domain models is defined as the "probability of failure free operation for a specified amount of time and in a specified environment" [6]. The predictions here depend on the failure arrival rate, which is assumed to be a random

variable in the statistical validation process. The observed failure data are fitted to one of the statistical models, based on the assumptions of the system's operating environment, and the final predictions of software or system reliability are made. These models treat either the failure interval or the number of failures in the given interval as a random variable and fit the observed data accordingly. For example, in the Jelinski-Moranda [6] model, the failure interval is treated as random variable and the failure rate for i^{th} failure is given by

$$\varphi_i = \mu(N-(i-1), \text{ where } \mu, N \text{ are constants.}$$

In this model, the current failure rate is assumed to be directly proportional to the number of remaining faults in the system.

The three key elements of reliability measurements in time domain models are failure rate, time, and testing environment. The definitions of failure and testing environment are consistently defined to reflect the final customer-operating environment. However, the time factor may vary and it needs to be defined based on the environment. There are three ways in which time can be specified for a reliability model- *calendar time, execution time, and logical time.*

Calendar time is recorded as the actual failure time with timestamps loosely related to the calendar dates. However, this measurement cannot be applied to the systems where there are large variations between successive test steps. Execution time takes into account only the actual amount of CPU execution time for modeling the reliability. Execution time cannot be applied to all systems and is not effective when the majority of the system's operations are user driven with little CPU execution. Hence, execution time based reliability models can be applied only to specific subsystems with

excessive CPU usage. Logical time is applied to transaction-based systems or to systems where the tests are time ordered in a logical sequence. As many real-time systems are event-triggered and operate in a cycle, with each cycle having a number of time steps with a fixed interval, the focus of this research is on applying calendar-time in reliability modeling

Though the time domain models have proven to be effective in some practical applications, they are based on assumptions some of which may be not valid in all applications. These assumptions include considering time as the basis for determining failure rate that demands an accurate measurement of time related data, assuming that reliability is directly dependent upon the remaining faults in the system, assuming that faults are equally distributed throughout the system, and also assuming that testing is randomized without any coverage criteria for detecting failure intervals. The time domain models do not take valid input states into consideration, which are important to determine when to stop testing.

Input Domain Models

Reliability in input domain models is informally defined as "the probability of failure free operation for specific input states" [3]. The valid input and output states are defined and the overall reliability is assessed based on the coverage achieved for the domains. In Nelson's Input domain model, system reliability is given by

$$R = 1 - f/n;$$

Here, 'f' is the number of inputs producing failures and 'n' is the total number of sampled input states. The input values for each test run can be selected based on a coverage criterion and techniques like sampling input domains into sub-domains, and random

selection to mimic operational profile can be used. The probability of failure for each sub-domain can be determined pointing out areas requiring additional tests.

Although input domain models provide valuable information for the test process, they are based on the assumption of random sampling without error correction. For most systems, this is not a valid assumption as faults are located and removed as soon as a failure is detected and the system reliability changes once the software is modified. Input domain models do not take this change into consideration.

Implicit assumptions of this research are that all detected defects are removed and higher coverage of input domains leads higher confidence of the system. Also, as exhaustive testing of all the valid input and output states is not practical for large applications, it is imperative to divide the domain space into sub-domains and apply the metrics to these sub domains separately. As real-time software is characterized by large domain space with time constraints, the strengths of both time domain and input domain approaches can be combined and used for reliability prediction.

CHAPTER 4

SOLUTION APPROACH

From the researcher's point of view, one of the greatest limitations in assessing the reliability of real-time software is a detailed analysis of the test results. After a careful study of the existing reliability improvement models, a solution to the problems stated in Chapter 2 is proposed here.

As most real-time software is hierarchically organized into different modules, from here on, the words software and module will be used interchangeably. The solution approach makes certain assumptions about a software module and the test environment. They are as follows.

- Every module has some input variables and output variables. Typically in real-time software, input variables are linked to sensors while output variables control physical devices and act as hardware triggers.
- Each input/output variable has a data type. Although the number of distinct data types supported by the system varies, these can be broadly classified into three types: Integer, Floating Point, and Boolean.
- The domain and accuracy for each variable is specified. This includes the minimum and maximum values for the variable, and accuracy in case of floating point variables. The range and accuracy of the variable can be different for different test runs allowing incremental testing
- The number of test steps for each test run is specified and the software remains unchanged between successive steps in a test run.

- Input values are generated, output values captured, and criteria for determining exceptions (faults) are specified.
- The response time and time interval between successive test steps, in case of a simulated environment are specified.

Our solution approach uses a hybrid scheme that combines time domain and input domain reliability models, and measures software reliability as a function of the correctness of input states calculated over time. The domain knowledge of the system developer is taken into consideration for deriving the metrics. Several metrics modeling reliability, as defined in Chapter 3, are provided. The proposed metrics are broadly classified into the following three categories: 1) Domain Percentile bucketing, 2) Probability results, 3) MTTF results. All other aspects of the system are merely additional features and are means of efficiently accomplishing these functions.

Domain Percentile Bucketing

Quantitative evaluation of domain testing in real-time systems faces especially difficult combinatorial problems, as noted in Chapter 2 [2]. One solution to this problem is to partition the domain space for each input and output variable into percentile sub-domains, in effect 100 buckets of contiguous values. The partitioning scheme presented here is termed *Percentile Bucketing*. Tests can then be executed and evaluated to determine which input buckets for each input variable are sampled. Similarly, the output domain can be evaluated to determine if testing produced values covering all output buckets for each output variable. The domain knowledge of the systems engineer is used to specify minimum coverage requirement for each input percentile prior to testing,

giving higher coverage requirements for error-prone and boundary percentiles. Achieving the coverage requirements for a bucket is then equivalent to testing for all the possible values in that bucket and the engineer can concentrate on remaining buckets.

Another related and important measure is buckets with *higher percentage of defects*. For each test step, defects are identified based on the exception criteria and are mapped to the corresponding input buckets. Most of the time, exceptions occur when an output value falls out of the given range, and, hence, this value cannot be mapped to any of the output buckets. The coverage values and the error values for each percentile can be used to infer the behavior of the software for all possible inputs in that percentile.

Bucketing also reduces the number of possible combinations. For the example given in Chapter 2, real-time software with two input variables will now have 100^2 (10^4) combinations instead of 10^{12}. Although this is a significant reduction in the problem space and makes the job of developer much easier, for large systems with hundreds of inputs with large domains, bucketing cannot reduce the combinatorial problem sufficiently to make testing all combinations tractable. However, bucketing can be used to identify defect-prone combinations by saving the bucket value for each input value when a defect is discovered and associating a counter with each such combination. For example, if test values from bucket 10 for input 1, bucket 20 from input 2, and bucket 45 for input 3 are associated with multiple defects, then this combination could be considered error prone. This combination can be used to guide further testing to discover the point of error causing defects and leveraged when retesting to determine if the error has been corrected [2].

Results from multiple test runs can be leveraged to obtain the percentile coverage values. However, as the specified domain space for a variable can be different for different test runs, a partitioning scheme that leverages all the values from multiple runs uniformly is designed. For every input or output variable, the domain is determined by calculating the minimum as the minimum of all test run minimums, and the maximum as the maximum of all test run maximums. In case of floating point values, accuracy is determined as the maximum of all the test accuracies. This gives the engineer the ability to execute several independent test runs and then integrate and analyze the test results only once. The engineer will also be able to see the growth in percentile coverage. The decrease in the number of error-prone percentiles indicates that defects are being removed and software reliability increasing.

The formulae for calculating the bucket size and bucket participation are given next. For every input variable, the size for each bucket is calculated as

Bucket Size = (Test Max – Test Min) / 100.

Given the bucket size, the range of values for each bucket is calculated as follows.

Min of Bucket I = Test Min + (I-1) * Bucket size,

Max of Bucket I = Test Min + I * Bucket size, where 1 <= I <= 100.

The minimum value is included for bucket 1 and excluded for all other buckets, whereas maximum value is included for all buckets. Given a value x, the bucket that contains x is calculated as follows.

Bucket for x = If x is not equal to Test Min

ceil ((x – min) / Bucket Size),

Otherwise

= 1.

Although percentile bucketing, in its most general sense, can be applied to integer and floating-point variables, it cannot be applied to Boolean variables. The minimum and maximum values for every variable are 0 and 1 respectively and hence there are at most two buckets for all Boolean variables. Also, though input values are controlled, software can generate out of bound values for output variables and hence two additional buckets 0 and 101 are added for every output variable to signify values falling below minimum and above maximum respectively.

Probability Metrics

Analysis of test results is not complete without providing quantitative metrics for the software. This section gives a description of the metrics that have been designed to describe various aspects of software.

Reliability Results Describing Output Variable Correctness

An output value is considered as an exception if the value does not satisfy the specified correctness conditions for that output variable. A test step is considered as a failure if there exists an exception in at least one output variable. Based on these assumptions, the following metrics are proposed that describe the software based on the behavior of output variables.

Probability of an exception in any one test step is the ratio of the number of test steps producing at least one exception in any output variable to the total number of test

steps. This value signifies the correctness of the system for a given number cycles over a period of time.

Probability of an exception in exactly one output variable in any one test step is the ratio of total number of exceptions in all output variables to the product of number of output variables and the number of time steps. This value signifies the confidence in the system based on all output variables over a period of time.

Probability of an exception in an output variable O_i in any one test step is the ratio of the number test steps producing exceptions in O_i to the total number of time steps. This value signifies the reliability of the system based on the reliability of a single output variable and is important in real time software, where each output value acts as a trigger for some system operation. Output variables are prioritized and this metric should be close to zero for outputs with higher priorities.

Reliability Results Based on Valid Input States

All three metrics proposed in the previous section take only valid output states into consideration and predict the probability of correct system operation based on a given output state and are completely independent of the input states and input sub-domains. However, gaining confidence in input sub-domains is equally important, given the huge domain space of real-time software. The following section details a technique for computing the reliability of the system based on input percentile bucketing.

This approach is an enhancement of the basic reliability assessment done in input domain models and involves calculating the probability of selecting each bucket, determining the failure probability of each bucket, and then calculating the reliability of the entire system. The steps in this approach are outlined below.

For every input variable I,

1. For each bucket b_j,
    - Assign the probability that bucket b_j is selected as

      $P(b_j)$ = No. Of values selected in bucket b_j / Total number of test steps.
    - Assign the probability that bucket bi is a failure as

      $\theta(b_j)$ = No. Of values in bucket b_j producing exceptions in at least one output variable / No. Of values selected for bucket b_j for the current test run.
    - Calculate the probability that a bucket b_j is selected and is a failure as $P(b_j) * \theta(b_j)$.
2. The probability that the system fails for input variable I, in any one test step, is the sum of the failure probabilities for each bucket calculated as
    - $\phi_i = \sum P(b_j) * \theta(b_j)$.
3. The probability that the system succeeds for this input variable in N test steps can be calculated as
    - $(1 - \phi_i)^N$
4. The total reliability of the system considering all the input variables is given by
    - $(1 - \phi_1)^N (1 - \phi_2)^N (1 - \phi_3)^N \ldots\ldots$

This metric gives the overall reliability of the system based on valid input buckets and is a close measure of the Probability of Failure On Demand (POFOD) defined in Chapter 3. The term ‘demand’ here reflects the validity of an input state. The intermediary results, probability of failure associated with a bucket ($\theta(b_j)$) can also be used to identify defect-prone percentiles and can be used to guide further tests.

MTTF Results

As time is an important factor in real-time systems, metrics are necessary to predict Mean Time To Failure (MTTF), which is defined as the time interval between successive failures. The failure rate should in turn be dependent on input or output states. The solution proposed extends the basic idea in time-domain reliability models by leveraging the correctness of output states and also predicts the failure arrival rate when input values are selected from a bucket.

MTTF is calculated by considering the fact that test cases are run sequentially in a single test run. The testing time required for a test run with N_1 time steps and a time interval T_1 between each test step is N_1*T_1. The execution time of real time software is negligible when compared to time interval between successive steps, and, hence, is ignored. As the model remains unchanged during the testing phase, the failure rate can be fairly assumed to be uniform through out successive steps in a test run. If we have m independent test runs for the same module, then the total testing time is calculated as

$$N_1*T_1 + N_2*T_2 + N_3*T_3 +\ldots + N_m*T_m$$

Let the number of time steps with at least one exception in any output variable for each test run be E_1, E_2, $E_3\ldots$ E_m respectively. Considering the validity of the output states throughout the testing time, the total number of failures is given by

$$E_1 + E_2 + E_3 +\ldots E_m.$$

Fitting this data into the time domain approach, MTTF is the ratio of the total time to the number of failures occurring in that time and is given by the formula below,

$$MTTF = (N_1*T_1 + N_2*T_2 + N_3*T_3 +\ldots + N_m*T_m) / (E_1 + E_2 + E_3 +\ldots E_m)$$

For a single test run, this formula turns out to be

$MTTF = (N_1 * T_1) / E_1$.

In case of a single test run, each input bucket is also given a MTTF value which enables the user to determine the approximate time in which the system fails when an input value in that bucket is selected. The formula for MTTF for a bucket I for any input variable is given by,

MTTF for a bucket I = (No of values falling in bucket I * Time Interval) / (No Of values in bucket I producing exceptions in at least one output variable);

This value cannot be computed in the case of multiple test runs, as all the values falling in a given bucket may not come from the same test run and the time interval may be different for different test runs.

CHAPTER 5

SOLUTION IMPLEMENTATION

Theoretical work is rarely of practical use until transitioned to commercial environments. In the case of this research, theoretical results need be transitioned to the commercial environment via a tool that provides the capability to compute and verify the theoretical results. The Reliability Analysis Test Tool (RATT) has been developed to verify the practicality of the results proposed in Chapter 4. RATT analyzes the results produced from testing real-time software via simulation and provides metrics to assess domain-testing effectiveness. In addition to this, RATT also provides test suggestions, based on the domain coverage metrics, to guide further testing. The latter part was done by another graduate student, Ms. Radhika Turlapati, as part of her thesis. This chapter provides an overview of the RATT design, the problems encountered during the development process, and the RATT user interface. A brief description of the tools used for developing and testing real-time software, $MATRIX_x$® and MATT is given**.** An introduction to the concept of testing real-time systems using simulation is also provided.

Overview of $MATRIX_x$® and MATT

The environment for this work included $MATRIX_x$®**,** a real-time software-modeling package developed by a Sunnyvale based company called Integrated Systems Inc., and MATT, a real-time software-testing tool. A brief description of each product is given next.

MATRIX$_x$® is a product family consisting of the following tools- *Xmath*, *SystemBuild*, *AutoCode*, *DocumentIt,* and *RealSim*. *SystemBuild* is used to graphically visualize, model, and simulate hierarchically structured dynamic systems. *Xmath* is a mathematical analysis tool used to verify SystemBuild models using simulation. *DocumentIt* automatically generates specification documents for SystemBuild models. *AutoCode* is used to generate the C or Ada source code from the *SystemBuild* model for the target hardware. *RealSim* is used for rapid prototyping of the generated models and allows generated code to be run as a simulation before deploying on target system. These tools allow the control engineer to create graphical models of the system, simulate the models at various levels, and automatically generate source code implementing the modeled system. The resulting source code is compiled, linked, and deployed on the target system [1].

Though MATRIX$_x$® product family provides comprehensive environment for developing real-time systems, it lacks the environment for automated testing of SystemBuild models. Testing tools can generate and run thousands of tests and automatically detect defects. One such tool used to test real-time software via simulation is the MATRIX$_x$® Automated Test Tool (MATT). MATT developed under a NASA grant at ETSU, implements effective test strategies defined by Dr. Joel Henry and validated on NASA Wind Tunnel Systems, the X-33, and the International Space Station [2].

MATT provides a tab-based user interface for configuring test scripts. MATT currently provides 25 test types to generate test values for the input variables. The user can set the test min, test max values, and the test type required for each input variable.

MATT also allows specifying the number of test steps and the accuracy required for each input variable. The range and the type of exceptions required for each output variable can be specified. MATT then generates the input matrix and runs simulation in tandem with SystemBuild and generates the output matrix. MATT then detects and lists the exceptions, which are identified as defects. The test scripts and input and output matrices can be saved as comma separated *test case* files and can be used later for further analysis. RATT analyses these test case files and computes the metrics proposed in Chapter 4.

RATT Design

A top down design approach has been followed during RATT development process. Figure 5.1 gives the overall system architecture with the major system components and the classes contained in each of these components.

The system is divided into four different components communicating with each other using well-defined interfaces. The 'User Interface' component contains all the GUI related classes. The 'Work Horses' contains classes used to compute the actual metrics and the 'Data Storage' contains classes used to store all the massive data read from the files, and also the intermediate data computed using 'Work Horse' classes. 'Utilities' component contains general classes and data structures that are independent of RATT. The 'Work Horses' and 'Data Storage' classes are written in C++ and are designed to be platform independent. The 'User Interface' classes are implemented using MFC and communicate with the interface provided by 'Work Horses'. The decision to make 'Work Horses' and 'Data Storage' classes independent of 'User Interface' enables RATT to be extended to other platforms by simply plugging in the new 'User Interface' component.

As the focus of RATT is on computing and saving the proposed metrics, only 'Work Horses' and 'Data Storage' classes are described here. The complete class diagram showing all the classes with relationships and the class diagram for 'User Interface' package is given in appendix A.

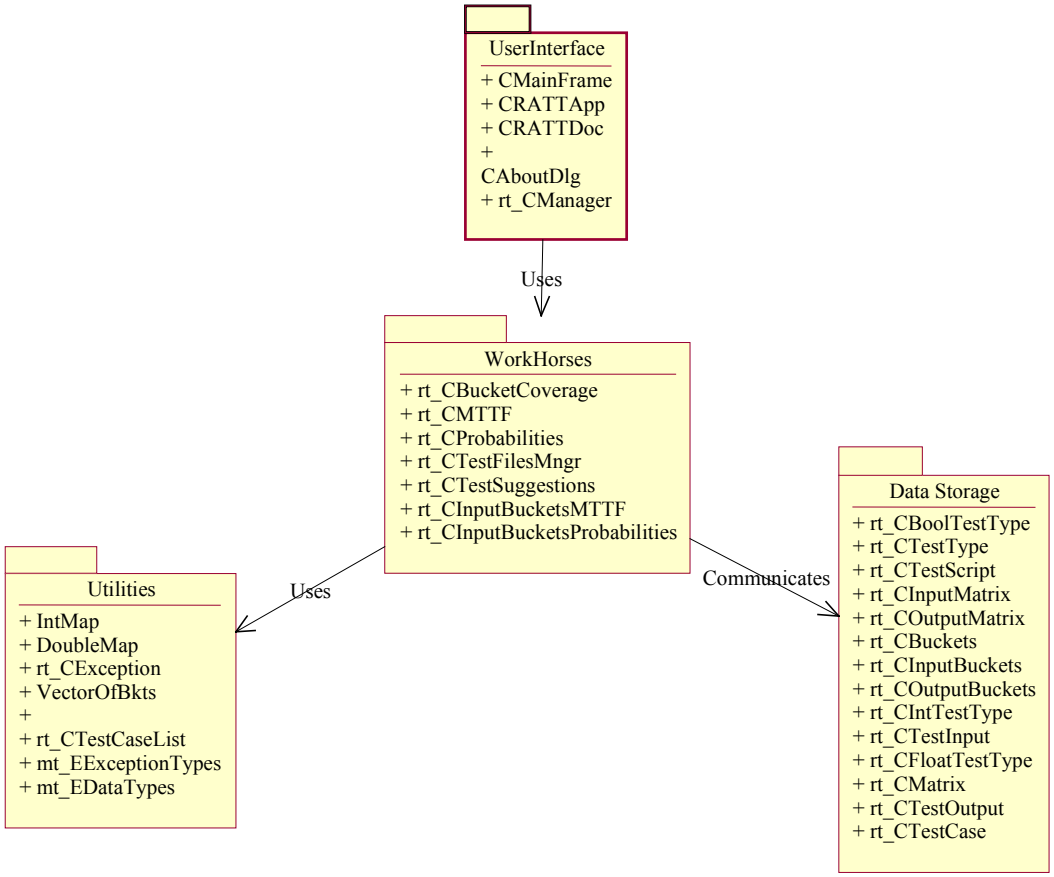


Figure 5.1: Package Diagram for RATT

Data Storage Component

The classes involved in 'Data Storage' component and the relationships between them are shown in figure 5.2. A top-down approach is followed for the design of data storage classes and the requirements originate from the format of test case files.

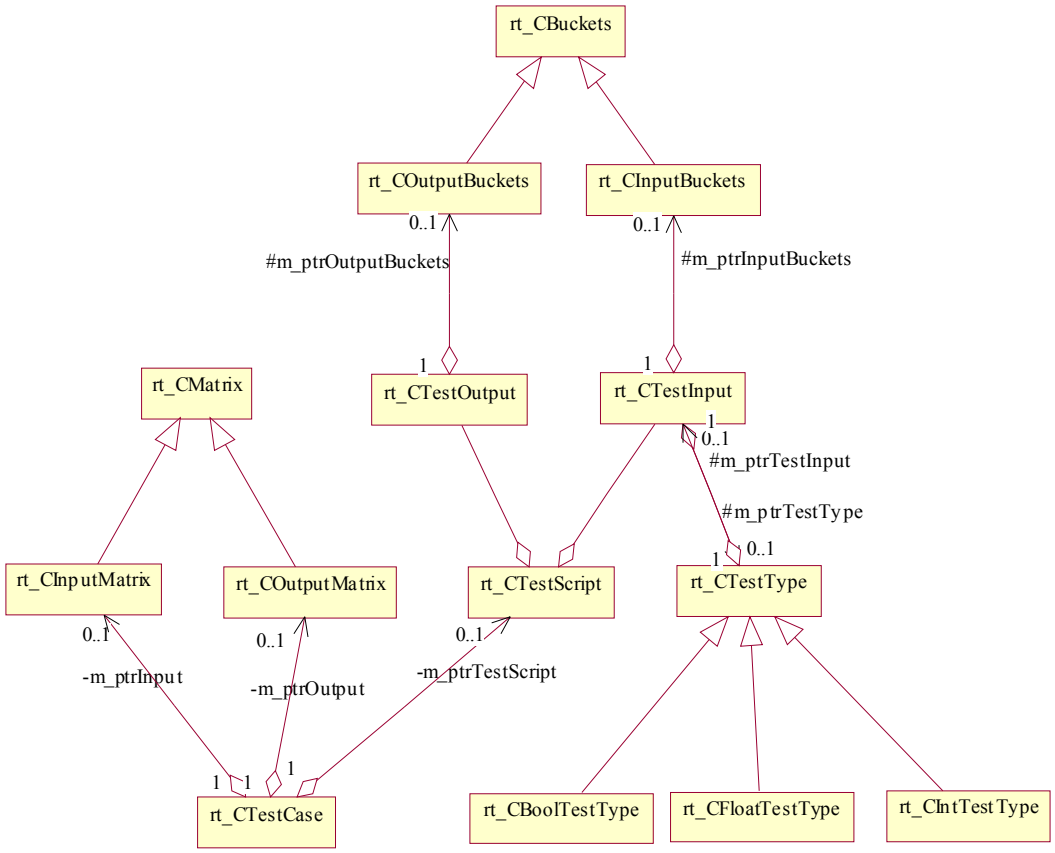


Figure 5.2: Class Diagram for RATT Data Storage Component.

A test case in MATT is defined as a grouping of the user defined test script, the MATT generated input matrix, and the output matrix generated by testing the model. A test script in turn contains the number of tests, the test interval, and an array of the input and output objects. Each of the inputs and outputs will contain a buckets object, apart from its properties. Every bucket object will have an array of 100 integer values, each reflecting the number of test values covered in that bucket. In addition, the input buckets will also have 100 integer values representing the number of exceptions in each bucket. The test input also contains the corresponding MATT test type object, and this object is used to calculate further test suggestions. Access to all the data values in all classes is

provided through inspector and mutator methods. Methods are also provided to read from the test case files and populate the required classes.

Work Horses Component

The class diagram with relationships for 'Work Horses' component is shown in figure 5.3.

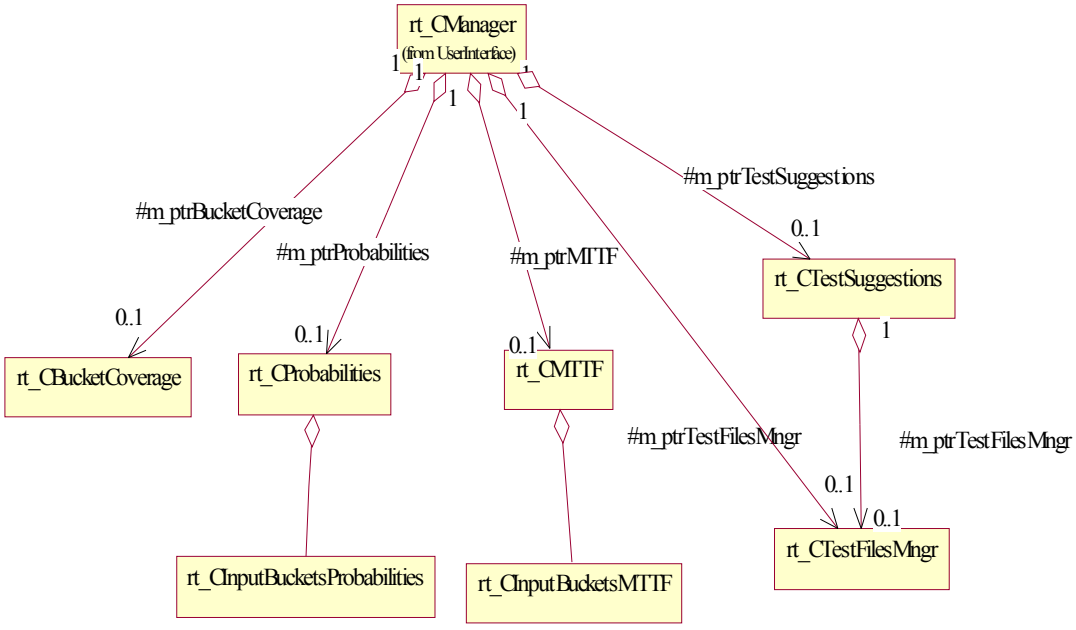


Figure 5.3: Class Diagram for RATT Work Horses Component.

There is a centralized manager class that controls all the communication between the user interface and 'Work Horse' classes. The distinct functionalities of RATT proposed in Chapter 4, the domain coverage results, the probabilities results, and the MTTF results, are each supported by a class as shown in the diagram. There is one additional test suggestion class that guides the user to carry out further tests based on the metrics and is managed by Ms. Radhika. Every Probability and MTTF object will have an array of pointers to the bucket objects, one each for every input variable of the system. The type of this bucket object is determined dynamically using Run Time Type

Information (RTTI) to obtain bucket coverage, and exception coverage values. As it is important to read and analyze data from multiple files, a test files manager class is created that isolates all the operations required for managing file i/o. A Test Case object is created within the test files manager class every time a file is loaded and a single test script object is maintained in the test files manager class that leverages the information gathered from multiple test scripts.

RATT Implementation and GUI

RATT is implemented in Microsoft Visual Studio 6.0 as a single document application using Microsoft Foundation Classes (MFC) application wizard. The user interface for RATT is implemented using MFC resource editors. However, a header and a source file are created for every class in 'Work Horse' and 'Data Storage' component to separate interface from implementation details, and also to reuse these classes easily. Compile time dependencies are removed using forward declarations and this enables these classes to be used directly by including their header files. Good software engineering principles are followed wherever applicable to prevent illegal access to data within classes.

The user interface of RATT uses both mouse and keyboard input for carrying out the menu options. Shortcut buttons and accelerator keys are also provided for frequently used operations. The main window is shown in figure 5.4. The client area of the main window is used only to display the results and is locked for editing by user. A brief description and a sample illustration of the major menu options are given below.

File Menu

The following selections are available from the file menu: Load, Add, Save, Print, Print Preview, Print Setup, and Exit. Each selection involves calling the file manipulation functions within RATT.

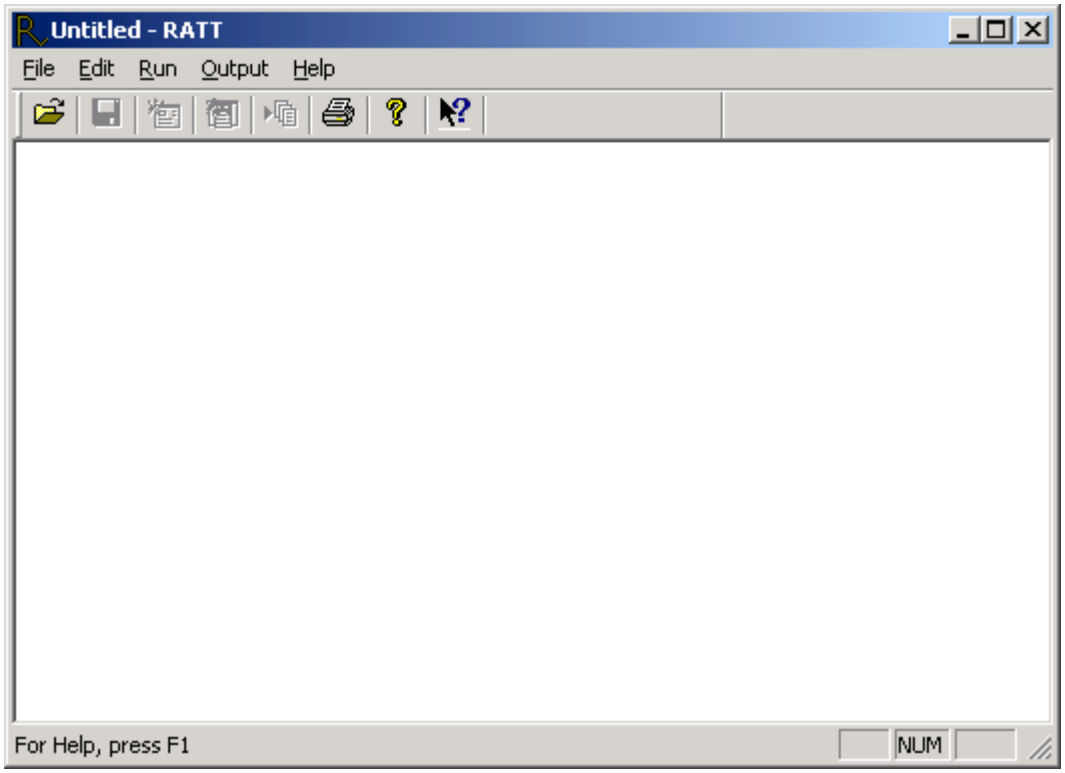


Figure 5.4: RATT Main Window.

Load: The Load option pulls up a dialog box as displayed in Figure 5.5. This dialog box gives the user the ability to select a comma separated Test Case file from the windows file structure and load into the system. The selected file should be in the same format as a MATT generated test case file. Load option will remove any of the previous test case files from the system and eventually clean any data storage objects created for the

previous analysis. The loaded file is scanned and the data storage objects are re-initialized.

Add: The Add option also displays a dialog box similar to the one shown in Figure 5.5 and gives the user the ability to add another test case file to the existing one and get comprehensive results. The added file is scanned and the data storage objects are updated. The Add option is not available to the user unless there is at least one file already loaded into the system.

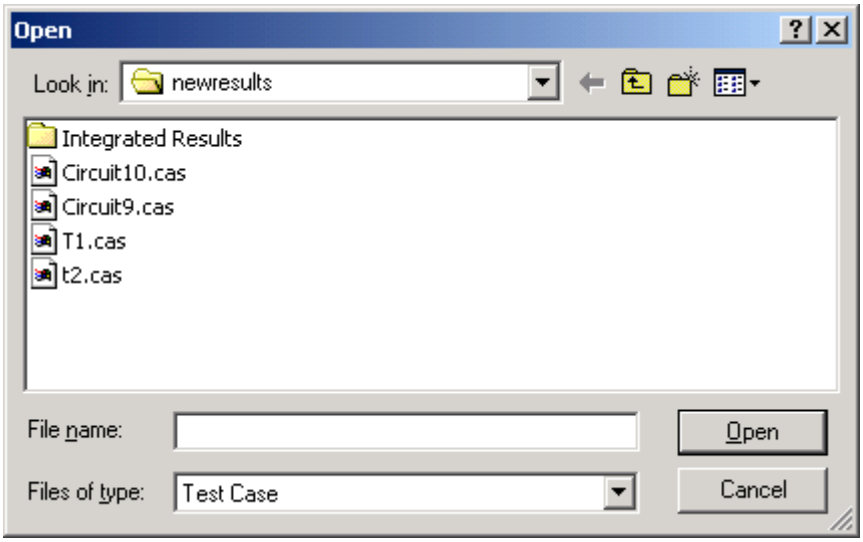


Figure 5.5: Open Dialog Box.

The maximum number of files that can be added to the system is 10 and can be changed by modifying the MaxTestCases entry in the system registry under 'HKEY_CURRENT_USER'.

Save: The save option displays a similar dialog box as in Figure 5.5 and allows the user to save the results generated using RATT as a comma separated value (CSV) file. This file can later be loaded into excel type applications for further analysis. RATT allows three types of files to be saved: file only with reliability metrics and domain coverage (.rlr), files only with test suggestions (.sug), and files with both reliability results and test

suggestions (.rtt). The save option is not available for user selection till the corresponding results are generated.

Print, Print Preview, and Print Setup: The print options allow the user to set up a printer, preview the document before printing, and finally print the document.

Exit: The exit option allows the user to close RATT gracefully.

Edit Menu

The Edit menu offers the following standard selections: Copy and Clear Screen. These selections are mainly aimed at providing the user with a reasonable view of the results on the screen.

Copy: The 'Copy' option allow the user to copy the text in the client area and paste this into a different application like a notepad for editing the results. The user will not be able to modify the data in the client area, and this decision is taken to preserve the authenticity of the results produced by RATT.

Clear Screen: Clear Screen option clears the complete client area and allows the user to concentrate one item at a time.

Run Menu

The Run menu has the following selections: Get Reliability Metrics, Get Test Suggestions, and Get Test Suite Suggestions. Selecting an option in this menu triggers methods in the workhorse classes and the results are computed.

Get Reliability Metrics: This option will trigger RATT to analyze all the files loaded into the system and then compute reliability and domain coverage metrics. At least one test file should be loaded before this option can be enabled. The compute methods of bucket coverage class, probability metrics class and MTTF class are invoked.

Get Test Suggestions: The user can select this option to get test suggestions aimed at improving further tests. As the test suggestions are based on the domain coverage metrics, this option is not available till the reliability metrics are computed using the 'Get Reliability Metrics' option. Also test suggestions differ in case of multiple files, and, hence, this option is disabled in case multiple files are loaded.

Get Test Suite Suggestions: Same as 'Get Test Suggestions' except that this option is used only when multiple files are loaded. Please refer Radhika's thesis [4] for more information.

Output Menu

The Output menu has selections that allow the user to view results on the client area. However, these results are available only after loading files and generating RATT results. The following selections are available here: Percentiles Coverage, Exception Coverage, MTTF, Probability Results, Test Suggestions, and Summary. Each of these selections may in turn pop up another sub-menu. The results can also be saved to a file using one of the File->Save options. All these options will use the inspector functions of the workhorse classes via manager class to display the results. Figure 5.6 gives a sample output generated by loading a single file 'Circuit9.cas'.

Percentile Coverage: This option allows the user to print the top 10 buckets with LEAST coverage of test values for every input and output variable.

Exception Coverage: When this option is selected, the top 10 buckets with highest number of exceptions for every input variable are printed on the screen. Combinations of input buckets producing highest number of exceptions are also printed.

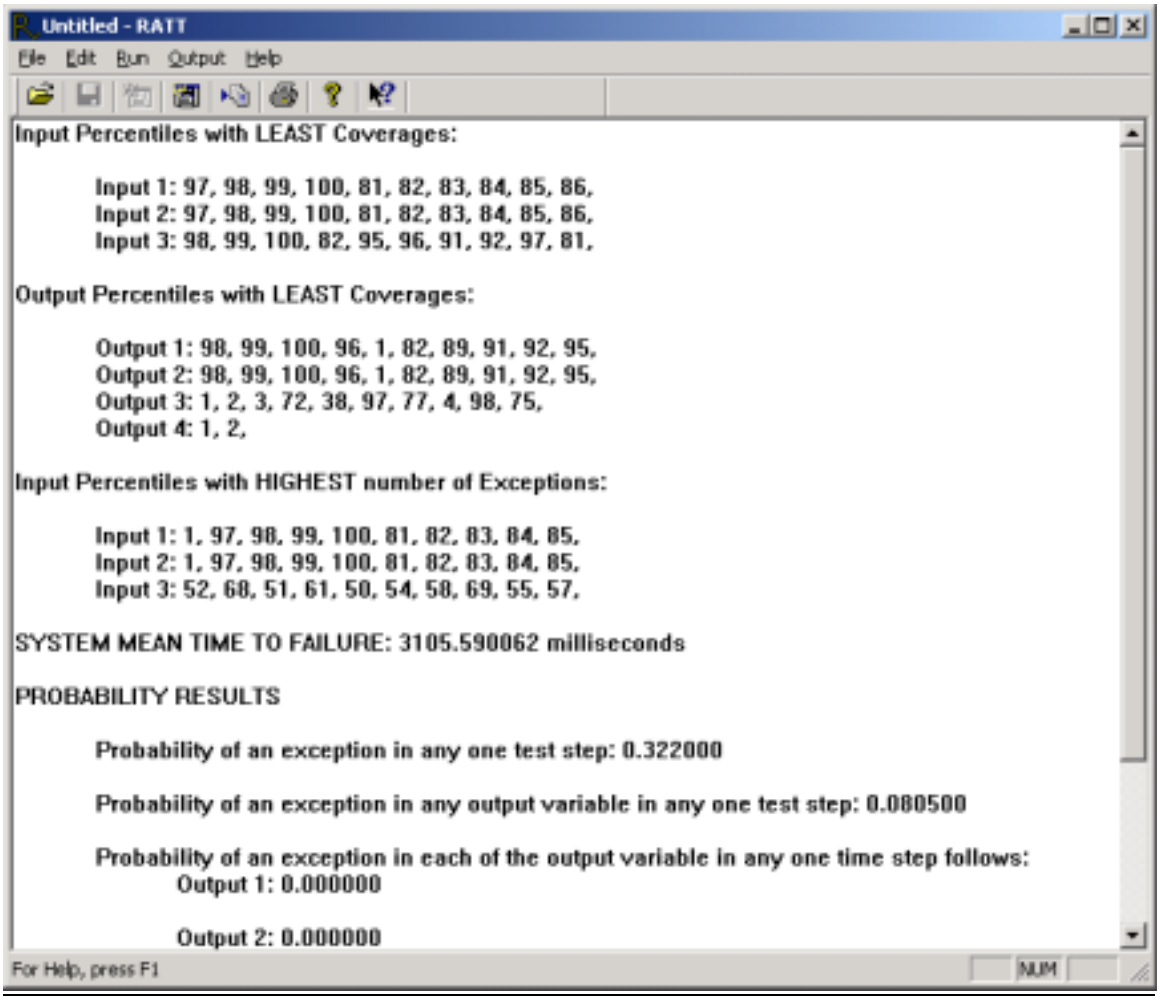


Figure 5.6: RATT Window with Results.

MTTF: The system MTTF is printed in milliseconds on the screen. In case of a single file, input buckets with highest MTTF are also printed. The later is not available when multiple test case files are loaded, as each file may have a different test time interval and all the values in the bucket may not come from the same test case file.

Probability Results: The probability metrics proposed in Chapter 4 are printed when this option is selected.

Summary: This option is provided to allow the user to print all the metrics using one mouse click instead of selecting every item in the Options menu.

CHAPTER 6

RESULTS

Software metrics should quantify the software characteristics and modifications to the software should be reflected in the metrics. In this chapter, the results of the test analysis produced by RATT are presented and analyzed. Analysis determines the effectiveness of the metrics proposed in chapter 4 in evaluating the effectiveness of testing real-time software.

The results presented in this chapter are produced by testing the ASCS_SCALING model used in controlling NASA's wind tunnel. This is a simple model with 8 inputs and 4 outputs. The model has been tested recursively before and after updates using MATT. The test case files produced by MATT for each test run are then fed to RATT to obtain reliability metrics. The sequence of the steps involved and the test parameters are as follows.

- Three independent test case files are generated without changing the model. Different input values are generated for each test run without changing the attributes of the outputs.
- Fourth test case file is generated after a minor modification to the software model. The modifications here include changing limits of an output variable, changing the exception criteria for at least one output variable.
- Fifth test case file is generated after a major update to the model resulting in a drastic reduction of the number of exceptions.
- The number of time steps in each test-run is set to 1000.

- The time interval between each time step for all the test-runs is set to 1000 milliseconds.

The generated test case files are analyzed using RATT, results saved, and changes in the metrics after loading each test case are represented graphically. For each test run, all the previously generated test case files are added to RATT for tracking the test progress and modeling reliability growth.

Domain Percentile Bucketing

Percentile Coverage

Figure 6.1 shows the graph for the coverage values achieved for the first ten buckets of the fourth input variable. The four columns for each bucket represent the number of values falling in that bucket after each of the four test-runs.

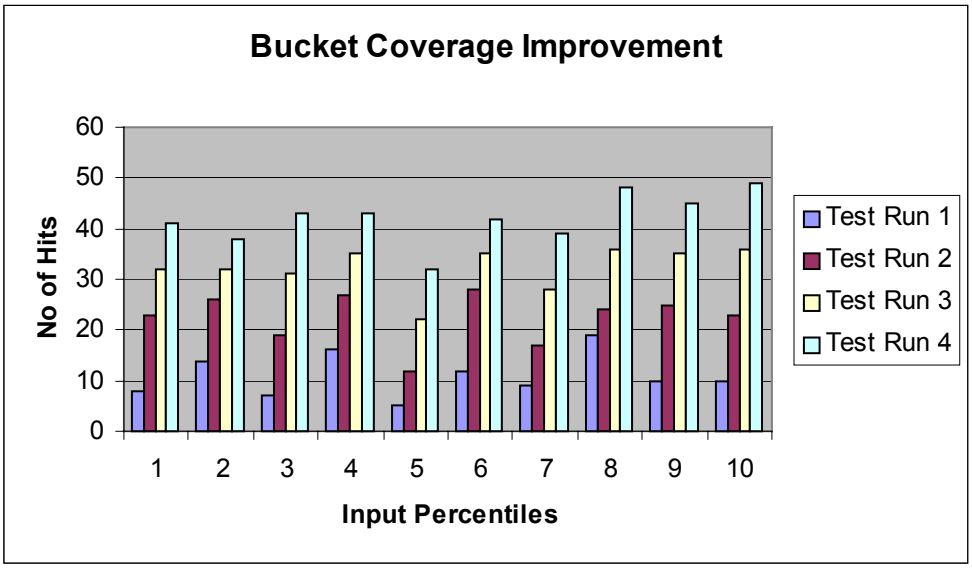


Figure 6.1: Bucket Coverage

The coverage improvement for every bucket from successive test runs is additively modeled and the user can easily keep track of the least covered buckets over

multiple test runs. Further tests can be concentrated in any uncovered buckets and additional tests run to increase domain coverage. If it can be assumed that the state of each bucket correctly represents the state of the system for all possible values in that bucket, then percentile bucketing reduces the number of possible states to K * 100, where K is a constant. The constant 'K' represents the desired level of coverage for every bucket set by software managers at the beginning of the test phase. For ASCS_SCALING model, the total number of values in the domain for fourth input variable with a minimum of 0, maximum of 100, and accuracy of 10 is $(100 - 0) * 10^{10}$. If the required coverage for every bucket is 100 values, percentile bucketing reduces the total number of values to be tested from 10^{12} to 10^{4}. Although the basic assumption is not valid for all real-time software, percentile bucketing still allows the user to keep track of the number of values in each sub domain and is much better than the brute force method.

Exception Coverage

Figure 6.2 shows the graph for the exceptions uncovered for the first ten buckets of the fourth input variable during four successive test runs. Exceptions in each bucket over different test runs are represented and the user can easily keep track of the current number of exceptions in each bucket, which in turn is an indication of the confidence of the user in the probability of success when a value is selected in that bucket. The graph shows that exceptions have increased during the first three test runs and more exceptions are found in buckets 4,6, 8, and 10. Based on the exceptions uncovered, the model is modified after the third test run and tested again in the fourth test run. The exceptions are reduced over all the buckets and the decrease is more significant in buckets 4 and 6.

Percentile bucketing helps identify the input sub domains producing a higher number of exceptions and allows the user to concentrate further tests within these faulty buckets to pinpoint the exact point of error. Instead of keeping track of distinct input values producing exceptions, a range of values producing exceptions is identified. As it is assumed that exceptions are linearly distributed over a given bucket, each bucket is assigned a failure probability. The failure probabilities for each bucket are listed in the reliability file saved by RATT. This file can be loaded into a spreadsheet application, and then buckets with higher failure probabilities can be identified by sorting. The software is updated accordingly, and the old test cases are run again (regression testing). If the number of exceptions uncovered after the update is less than the number of exceptions before the update, it can be inferred that the updates are improving the software reliability.

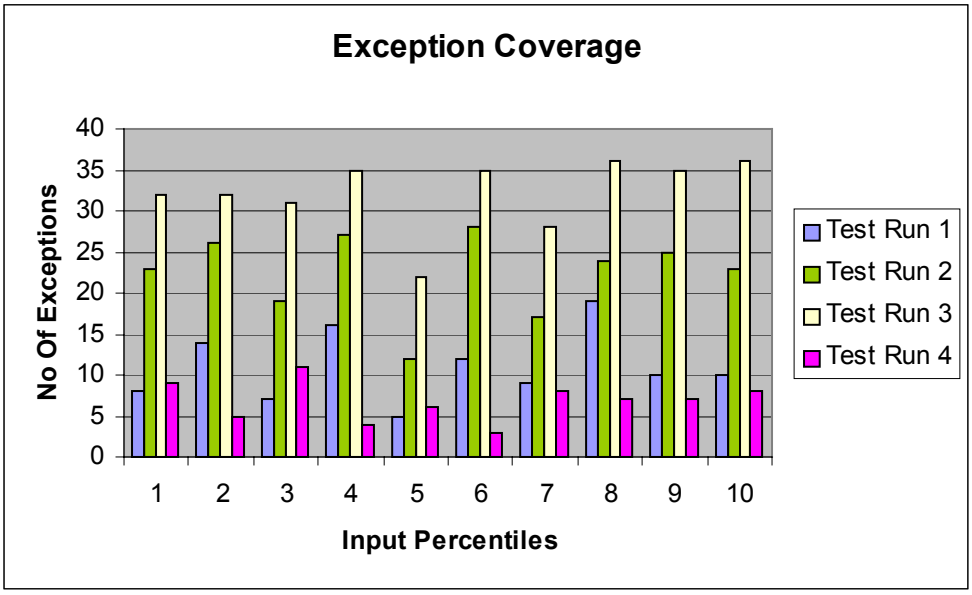


Figure 6.2: Exception Coverage

Hence, the graphs not only allow the user to identify defect prone sub domains but also allow the user to keep track of the test progress after software updates over multiple test runs.

Input Bucket Combinations

The input bucket combinations producing highest number of exceptions for the first test run of ASCS_SCALING model are shown in figure 6.3. Each row indicates the bucket number for every input variable producing exceptions. Only the top 10-bucket combinations producing exceptions are stored and displayed. The bucket combinations provide important information and can improve testing.

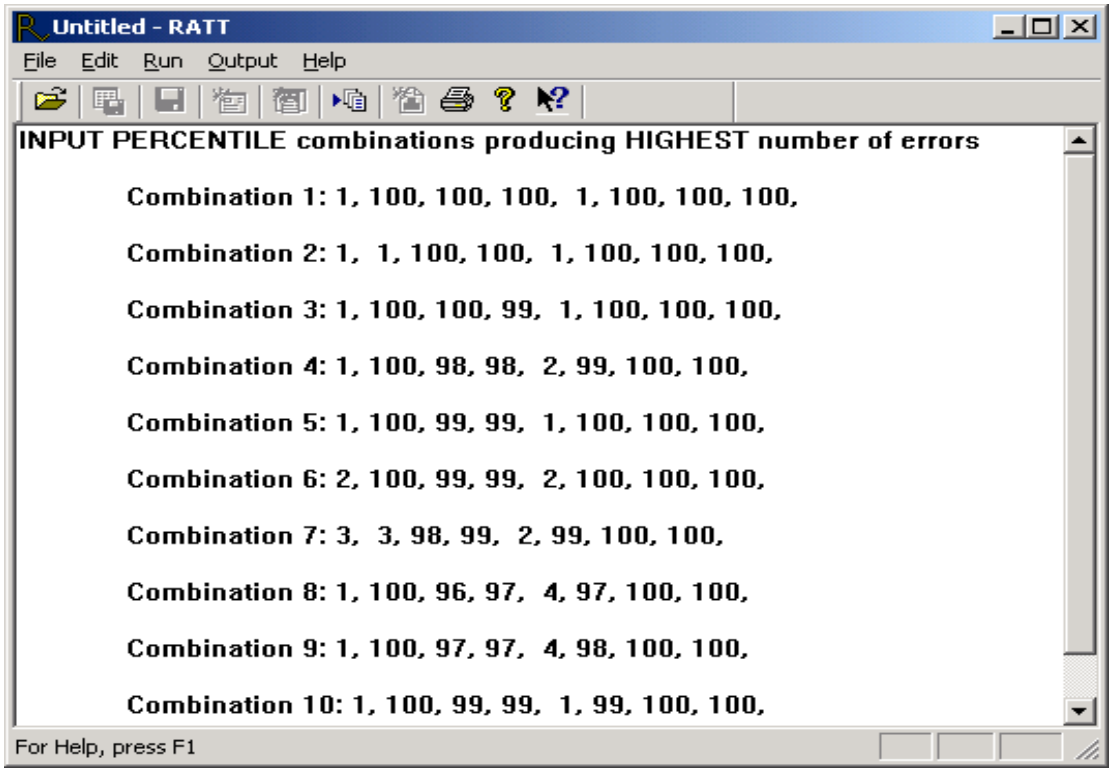


Figure 6.3: Input Bucket Combinations

The combinatorial problem discussed in Chapter 3 is reduced drastically. The total number of possible input combinations for ASCS_SCALING model with 8 inputs each taking values from 0 to 100 with a precision of 10 decimal places is $(10^{12})^{8}$, i.e.

10^{96}. The number of combinations with percentile bucketing is reduced to 100^{8}, i.e. 10^{16}.Although the combinatorial number is reduced by a large factor, this number can still be very high for large real-time systems. However, only combinations producing defects are reported by RATT allowing the user to tackle the problem by divide and conquer rule.

Quantitative Metrics

Reliability Results Based on Output Variable Correctness

The graph shown in figure 6.4 models, the changes to the probability of an exception in any one time step during successive test runs.

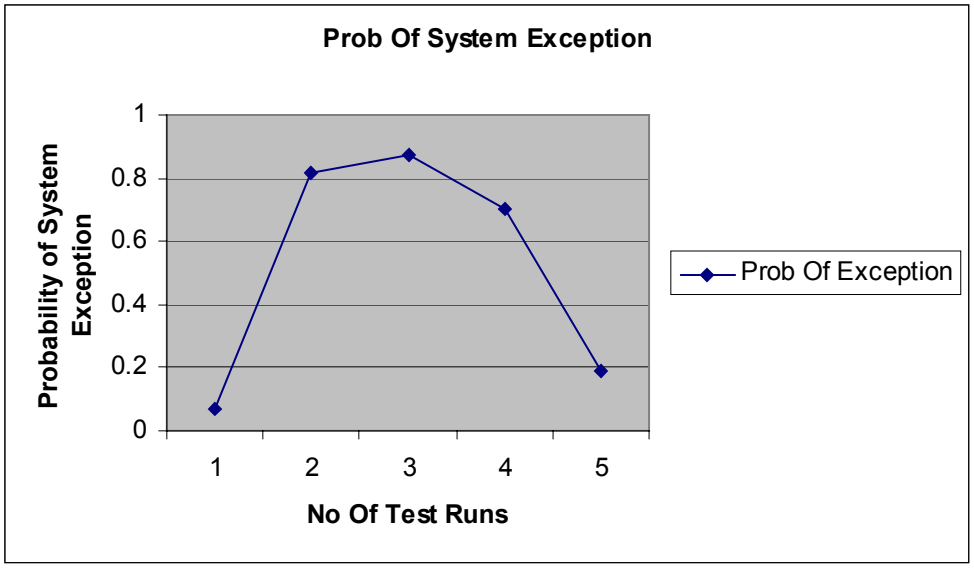


Figure 6.4: Probability of an Exception in any One Time-Step

The probability of an exception is increased during the first three test runs and this can be attributed to the fact that more errors are uncovered with rigorous testing. The update is reflected in the fourth test run, where the number of exceptions has reduced. The amount of decrease in the curve is directly proportional to the quality of the update to

the model. A larger decrease here means that the update was a major one in the desired direction, which is improving reliability. As such, the graph also clearly and correctly indicates that the first update after the third test run is a minor one compared to the update after the fourth test run. As the metric changes rapidly both by uncovering more errors and also by model updates, the model has to be tested until the metric is stabilized, i.e. the curve becomes linear over successive test runs.

The graph shown in figure 6.4 models, the changes to the probability of an exception for every output variable in any one time-step during successive test runs.

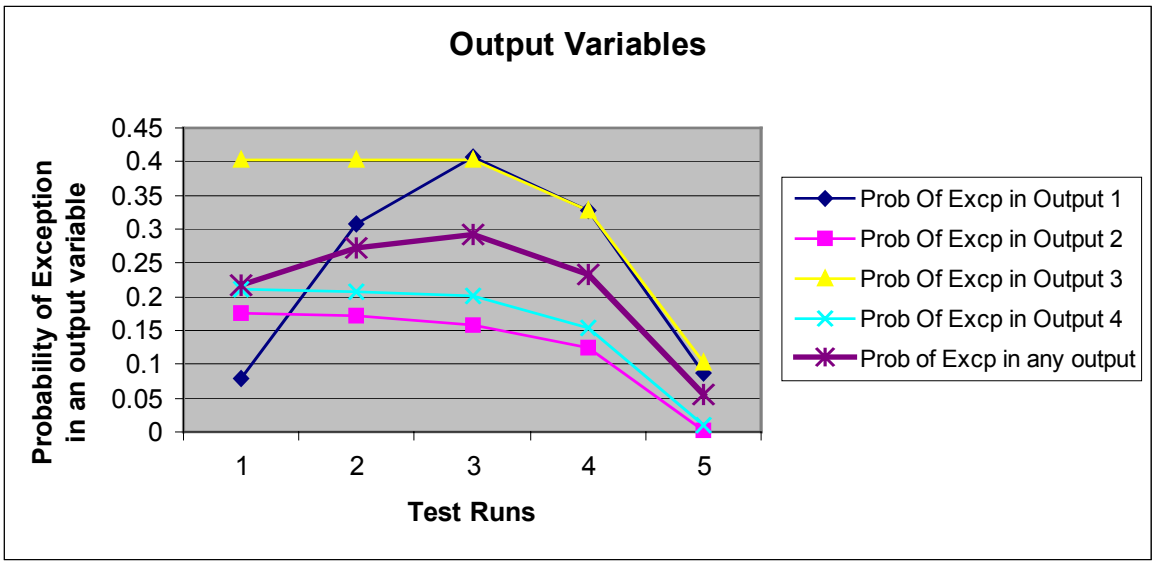


Figure 6.5: Probability of an Exception in an Output Variable in any Time-Step

The four curves in the graph show ways in which probabilities of an exception changes for each of the four output variables. The bold curve in red shows the average of all the four curves and gives the probability of an exception in exactly one output variable in any one time step. As the graph indicates, the probability of an exception in output variable 1 changes rapidly and indicates instability in this output. The graph is useful in predicting the stability of the system based on each individual output variable and points

out potential problem output states. This is important in systems where the outputs are prioritized. Lower probabilities should be achieved for outputs with higher priorities.

Reliability Results Based on Input Variable Correctness

Figure 6.6 shows the graph reflecting changes to the reliability of the system over all time steps based on valid input states. Only results from the first four test runs are shown in figure 6.6, as the value from the fifth run is much higher and cannot be interpolated in this graph. The value from the fifth test run is shown in figure 6.7.

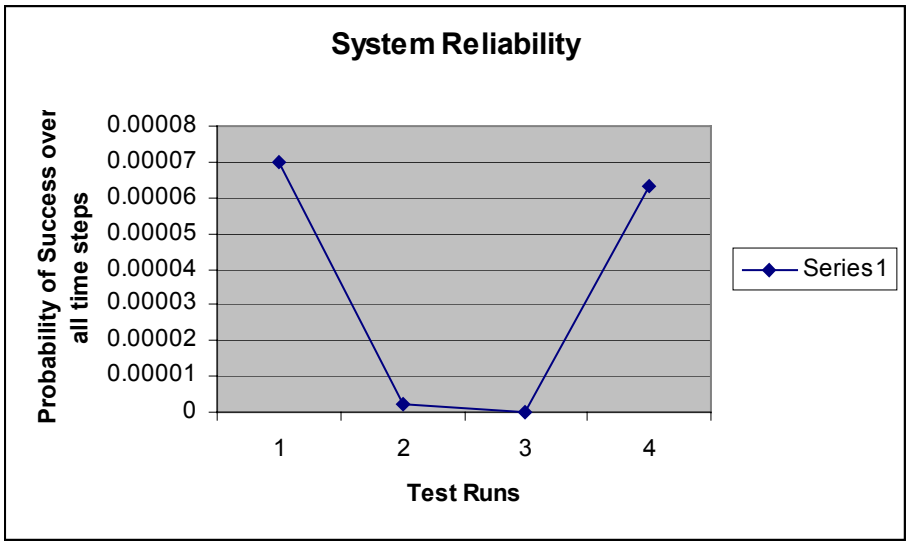


Figure 6.6: Probability of System Success Over all Time Steps (4 Test Runs)

The results obtained closely model the theoretical bathtub reliability graph shown in figure 3.1. The graph shows that reliability falls as more exceptions are detected and increases as the software is updated correctly. The results are based on the validity of all possible input combinations and greatly reduce the effort of the user in keeping track of the valid and invalid input states. This metric is also closely related to Probability of Failure On Demand (POFOD) defined in Chapter 2. A lower reliability value is an indication of two related concerns:

- Not Enough values have been tested in all buckets of all input variables. More testing without changing the model can improve this figure provided further test runs results in more successes than failures.
- The model is very poor and needs modifications.

However, in either case the curve should become linear over successive test runs before making a decision. The curve clearly shows the reliability growth over multiple test runs.

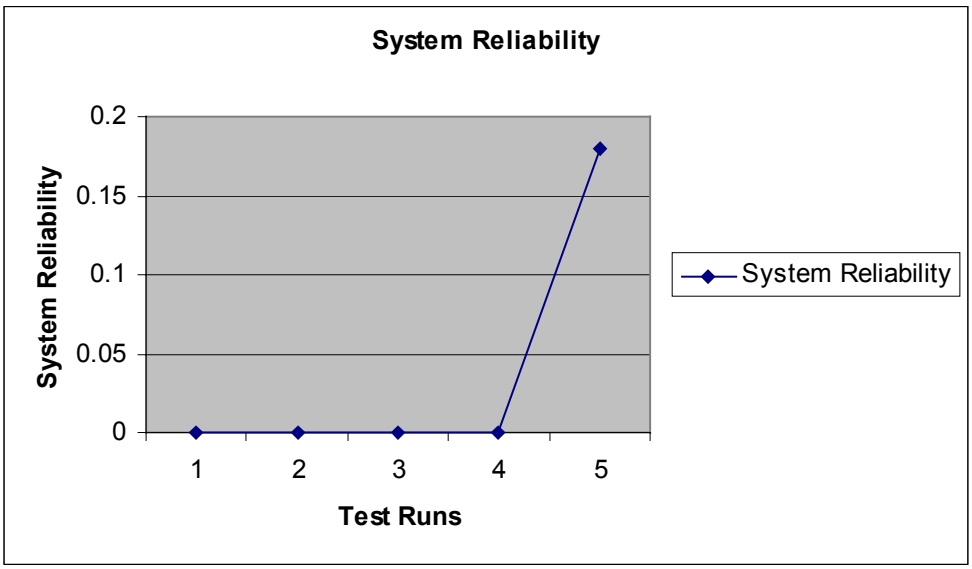


Figure 6.7: Probability of System Success over all Time Steps (5 Test Runs)

MTTF Results

Figure 6.7 shows the graph reflecting changes to the System Mean Time To Failure (MTTF). As the graph shows, more exceptions are uncovered during the first three test runs, and this reduces the MTTF value. However, upgrades to the model increase MTTF, and the amount of increase depends on the quality of the update. MTTF for individual buckets for every input variable are also listed in the RATT files and

provide useful information about the next failure interval when values from that bucket are selected. From the graphs, it can be concluded that MTTF growth is correctly modeled by the proposed metrics, and MTTF changes as the software is changed. MTTF gives useful information about the occurrence of next failure. For transaction based real-time systems the sum of response time and recovery time should be less than MTTF for uninterrupted operation.

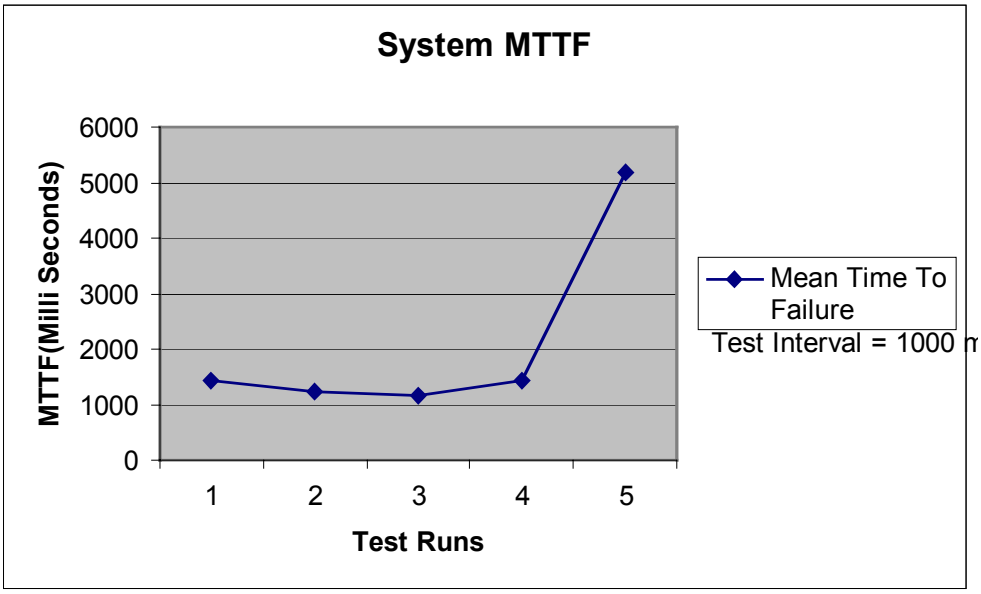


Figure 6.8: System Mean Time To Failure (MTTF)

The results shown in this chapter indicate that the domain coverage metrics provide valuable information to the user for uncovering defects, and the reliability metrics correctly describe the software under test and change as more defects are uncovered or software is modified. While the former is useful from a developer point of view, the latter is useful from the managerial perspective. Managers can pre-assign values for all the reliability metrics and the development process continues until these metrics are achieved. We have not only proposed reliability metrics in this project but also verified the feasibility of computing these metrics and the practical applicability of the

metrics by testing the results on models used in controlling NASA's wind tunnel. While $MATRIX_x^{®}$ and MATT solves the first three steps in real time software development, RATT solves the final step of analyzing the test results and producing quantitative metrics describing the software.

Finally, RATT is proposed to be used within Boeing for improving the test process and evaluating the models to be developed for NASA. At the same time, RATT is also proposed to be used by NASA's Independent Verification and Validation Center for evaluating the product delivered by Boeing. RATT is also proposed to be used within other organizations such as Allied signal and Sperry for improving the test process in the development of real-time control models. In essence, RATT will be used as a standard for assessing the quality of real time software produced via simulation.

CHAPTER 7

LIMITATIONS AND FUTURE WORK

This chapter provides an overview of the current limitations of the reliability metrics and also provides an insight to improvements that can be achieved. Features that can be implemented in RATT to make it more efficient and robust are also discussed.

One of the major limitations of the input based reliability metric is that the basic assumption that exceptions are linearly distributed over a given input bucket has to be valid. This limitation is particularly not valid when the input range is very large and also for control systems with discrete input points. One technique of overcoming this limitation includes testing incrementally by taking small ranges at a time. Another limitation of system reliability metric is that though the system reliability reaches one when there are no exceptions, the metric values fall rapidly when even a very small number of exceptions are uncovered. It can be attributed to the fact that multiplication of small floating-point numbers makes the product decrease rapidly. This might prevent managers from gaining confidence in the system based on a smaller system reliability value. Hence, the managers should be cautious in assigning a target value prior to testing or else the specified value may never be reached.

Currently MTTF metrics for individual input buckets are not provided when multiple test case files are loaded into RATT. The reason for this is the fact that the time interval between time-steps may be different for different test runs. MTTF for individual buckets are useful in predicting the failure interval when an input value in a bucket is selected. One way of obtaining MTTF for individual buckets is to average the time

intervals from all the test case files and apply the same calculations as in a single test case file. However the effectiveness of computing the average has yet to be analyzed with the help of a domain expert. Another improvement to the domain coverage metrics is the ability to predict the combinations of input buckets producing the highest number of exceptions. This can be achieved by applying pattern matching to the currently reported input bucket combinations. Reliability growth models based on random variable sampling can be studied in more depth as well, and applied to the current metrics by incorporating the operational profile into testing.

Further improvements to RATT could improve efficiency in computing the current metrics and also port RATT to other operating systems (i.e. Sun Solaris). The current design allows the development and integration of a new user interface component easily and hence RATT can be ported to other platforms as required. One feature that can be included in RATT is the ability to generate and display graphs from the user interface. Although graphs can be generated currently by loading RATT files into spreadsheet applications, incorporation into RATT would be a nice feature for the user.

In conclusion, the two goals of this project have been achieved: by providing metrics to quantify real-time software and by verifying the feasibility, and practical applicability, of the results by testing on real-time control models. More importantly, the limitations of the metrics are also identified to avoid any inappropriate applicability of the same to all real-time control systems.

BIBLIOGRAPHY

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

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

[4] Turlapati, R., "Leveraging Test Measurements into Proposing Additional Domain Tests", ETSU Master's thesis, 2001

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

[6] Jeff Tian, "Integrating time domain and input domain analyses of software reliability using tree-based models", *IEEE Transactions on Software Engineering* Dec 1995 v21 n12 p945 (14)

[7] Ricky W. Butler George B. Finelli, " The infeasibility of quantifying the reliability of life critical real time software", Available from: http://shemesh.larc.nasa.gov/fm/paper-nonq/nonq-tse.html

[8] Reliability: Embedded Systems, Available From: http://www.cs.cmu.edu/~koopman/des_s99/sw_reliability/#definition

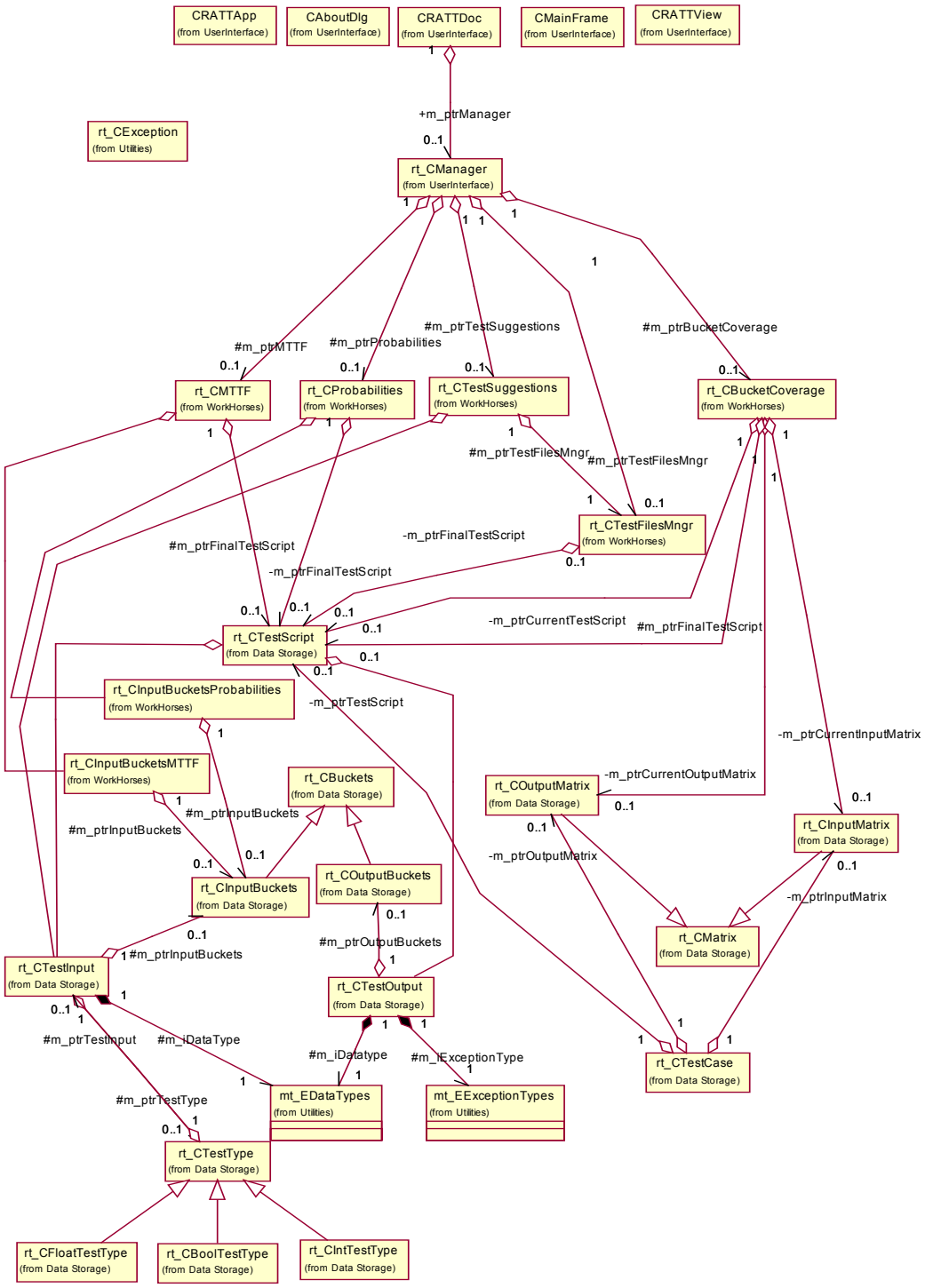
[9] Software Reliability, Available From: http://www.comp.lancs.ac.uk/computing/users/tam/CS231/slides-18/index.htm

[10] Phyllis G. Frankl, Richard G. Hamlet, Bev Littlewood, Lorenzo Strigini, "Evaluating testing methods by delivered reliability", *IEEE Transactions on Software Engineering* August 1998 v24 n8 p586 (16)

[11] Gutjahr, Walter-J, "Partition testing vs. random testing: the influence of uncertainty", *IEEE Transactions on Software Engineering* September1999 v25 n5 p661 (15)

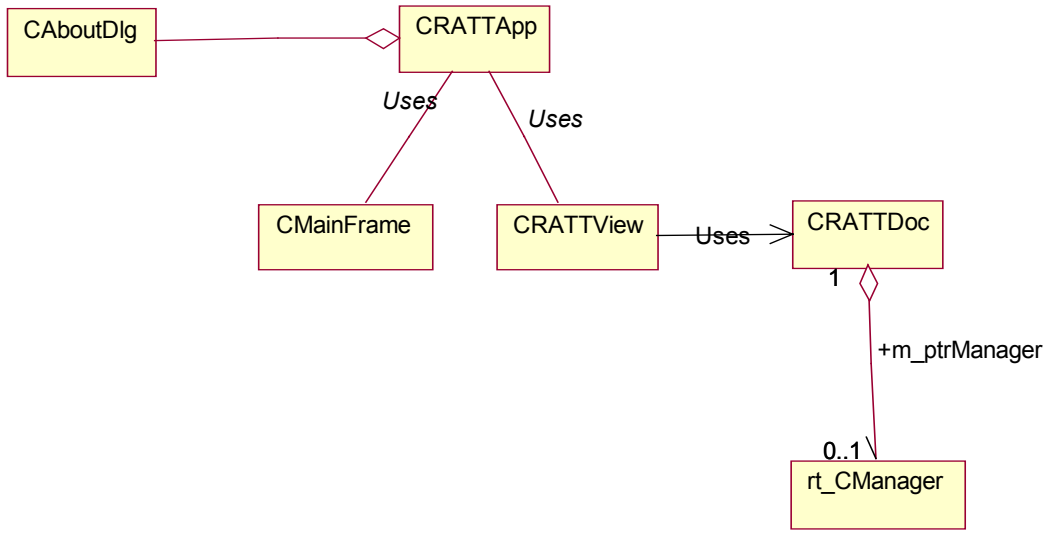
[12] Norman F. Schneidewind, "Reliability Modeling for Safety-Critical Software", *IEEE Transactions on Reliability,* August 1997 v46 n1

APPENDIX: RATT High-Level Class Diagram



APPENDIX (cont'd)

RATT User Interface classes



VITA

NARENDRA KONERU

Personal Data:	Date of Birth: June 8, 1978 Place of Birth: Hyderabad, INDIA
Education:	Bala Bharathi High School, Hyderabad, India St. Anthony's Junior College, Hyderabad, India University College of Engineering, Osmania University, Hyderabad, India; Computer Science, B.E., 1999 East Tennessee State University, Johnson City, Tennessee; Computer Science, M.S., 2001
Professional Experience:	Software Intern, ANURAG, DRDO, Hyderabad, India, 1998 Software Engineering Intern, 3CX, San Jose, 2000 Graduate Assistant, East Tennessee State University, Johnson City, Tennessee, 1999-2001
Publications:	Henry J., Koneru N., Turlapati R., "Quantitative Evaluation of Domain Testing", Testing Computer Software, June 18-22, 2001, Accepted for publication
Honors and Awards	Nominated for Best Graduate Student Award, 1999-2000. Outstanding Scholastic Achievement Award, East Tenessee State University, 1999 - 2000 and 2000-2001.