AN EMPERICAL EVALUATION OF ADEQUACY CRITERIA FOR TESTING CONCURRENT PROGRAMS

by

GAURAV SAINI

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2005

ACKNOWLEDGEMENTS

I would like to express my profound gratitude to my supervising advisor Dr. Yu Lei and co-supervising advisor Dr. R.H. Carver for their advice, support, and continual encouragement throughout the thesis process. Sincere appreciation is expressed to Dr. Yu Lei, Dr. Leonidas Fegaras, and Mr. David Levine for their guidance as members of the author's graduate committee.

Finally, my earnest gratefulness is bestowed on all of my family members. Their love and encouragement contribute greatly to my master degree studies.

July 12, 2005

ABSTRACT

AN EMPERICAL EVALUATION OF ADEQUACY CRITERIA FOR TESTING CONCURRENT PROGRAMS

Publication No. ______

Gaurav Saini, M.S.

The University of Texas at Arlington, 2005

Supervising Professor: Jeff (Yu) Lei

A concurrent program contains two or more threads that execute concurrently and work together to perform some task. Concurrency increases the efficiency of a program. Testing of concurrent programs has been a challenging task because of the inherent non-determinism. Most approaches proposed for concurrent program testing employ, explicitly or implicitly, a coverage criterion to measure test adequacy. In order to apply those approaches, we must first choose a criterion that suits best for our programs.

There is a need for quantitative results of evaluation regarding the effectiveness of the various coverage criteria used for the testing of concurrent programs. Such an evaluation can be useful to select a particular criterion for a specific set of concurrent programs. This thesis tries to evaluate the effectiveness of some of these criteria and puts forth the quantitative results for such an evaluation.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

Concurrent programs are gaining more popularity in the ever growing world of software development. A concurrent program can be defined as one that contains multiple processes and/or threads which execute concurrently and work together to perform a given task. Concurrency can be used to improve the computational efficiency of a program. While one process (thread) is awaiting user input, another process (thread) can perform computational tasks in the background. In addition, concurrency is inherent in many problem domains that can be solved easily by creating multiple threads. For example, a web server typically creates separate threads to service incoming client requests. Other applications where concurrency arises naturally include operating systems, user interfaces, etc.

Although concurrent programs have a lot of advantages, testing of a concurrent program has always been a challenging task because of the inherent non-determinism displayed by concurrent programs [1]. Multiple executions of a concurrent program with the same input may exercise different sequences of synchronization events (or SYN-sequences) and may produce different results, making testing a difficult task. A few approaches have been proposed for concurrent program testing in the past. Most approaches employ, explicitly or implicitly, a coverage criterion to measure test adequacy [2, 3, 5, 6, 10]. In order to apply those approaches, we must first choose a criterion that suits best for our programs.

There has been no formal evaluation performed and no quantitative results of evaluation put forth regarding the effectiveness of various coverage criteria used for testing concurrent programs. This thesis tries to evaluate the effectiveness of these criteria and puts forth the quantitative results for such an evaluation to aid a fellow concurrent program tester in making a decision to choose one of the numerous coverage criteria for testing of concurrent programs.

1.2 Evaluation Methodology

In this thesis we present an evaluation of adequacy criteria for testing concurrent programs. Most of the approaches for the testing of concurrent programs employ a coverage criterion to measure test adequacy, but we still don't have any experimental results to guide us towards any one specific coverage criterion appropriate for the needs of a particular program testing.

At times a robust testing coverage may be essential during testing a particular concurrent program i.e., there can not be a compromise on the adequacy of the test, even at the expense of increase in cost. On other occasions, robustness of the test could be of moderate importance but the testing approach should be cost effective. As of now it is difficult to suggest a particular coverage criterion that is supported by empirical evaluation results. This work is an effort to address this problem.

The results of the proposed evaluation of this thesis can be used to guide a fellow concurrent program tester for selecting a specific coverage criterion. The following points dictate the methodology adopted by us during our evaluation:

1. As our first step we were required to select the coverage criteria which we seek to evaluate. After spending a lot of time and effort going through several different coverage criteria [2,3,5,6,10,21,22,36,38], we short listed the following seven criteria based on their importance in concurrent program testing and the feasibility of their empirical evaluation:
    - All synchronizable statement-pairs coverage
    - All branches coverage
    - All decision/condition coverage
    - All du-pair coverage
    - All synchronizable statement pairs & all branches coverage
    - All synchronizable statement pairs & all conditions coverage
    - All synchronizable statement pairs & all du-pair coverage
2. Our next step was to choose one or more concurrent programs which are suitable for the proposed evaluation and also has an added mechanism for *deterministic execution* [5]. Deterministic execution is to force a specific sequence to be exercised by the execution of a concurrent program. We are using a sliding window protocol implementation in Ada as our example of concurrent program; this program also has a support for deterministic execution [11].

3. Next we construct a sequence of synchronization events *(SYN-sequences)* [5] of the example program following the respective coverage criteria. These sequences will be *deterministically* executed on our example program during testing. The effort spent in construction of these sequences for every approach is measured in terms of the number of synchronizations each sequence has. The number of total synchronizations per criterion represents the *cost* of testing using that coverage criterion.
4. In the last step of our evaluation, mutation-based testing is used to assess the robustness of the SYN-sequences and consequently the *robustness* of the criterion which is covered by those SYN-sequences.
5. Finally, comprehensive empirical results of the evaluation of all the coverage criteria are analyzed and compared to aid the selection of a specific criterion on the analysis of its *robustness* and *cost*, in testing of concurrent programs.

1.3 Organization

Chapter 2 gives a basic idea about testing of concurrent programs and discusses the background of research in the relevant field of testing concurrent programs. Chapter 3 presents preliminary information about testing concurrent programs and the challenges faced while testing concurrent programs. Chapter 4 is primarily dedicated to the elaboration and explanation of the seven coverage criteria which we are evaluating in this Thesis. Chapter 5 gives an in-depth view of the evaluation method and the analysis of results of evaluation, separately for each coverage criteria followed by a comprehensive comparison. Finally the conclusion and suggestions for future work are presented in chapter 6.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 Background

Software testing can be defined as exercising the program using data similar to real data that the program is designed to execute, observing the outputs, and inferring the existence of errors or inadequacies from anomalies in that output [37]. Software testing is one of the most significant procedures in software engineering. It is a process used to identify the correctness, completeness and quality of developed software. In the conventional approach this is done by executing the program with a selected set of test cases. This is a non-trivial procedure and has long been studied and researched since the dawn of computer era, to the extent that it can be called as the heart of software development circle. Figure 2.1 shows the classic software life circle. A significant amount of research in software testing, validation, and verification has been proposed over the past two decade.

The conventional method to test a sequential program involves executing it with specific input and comparing the output with the expected result. If testing is carried out without analyzing the inner structure of the program, the test method is called "*black box*" approach. It considers the software-under-test to be an opaque box. This approach doesn't work with any kind of concurrent programs, which includes parallel, distributed, or centralized message-passing programs [17, 19]. In later chapters more general details in testing concurrent software can be found.

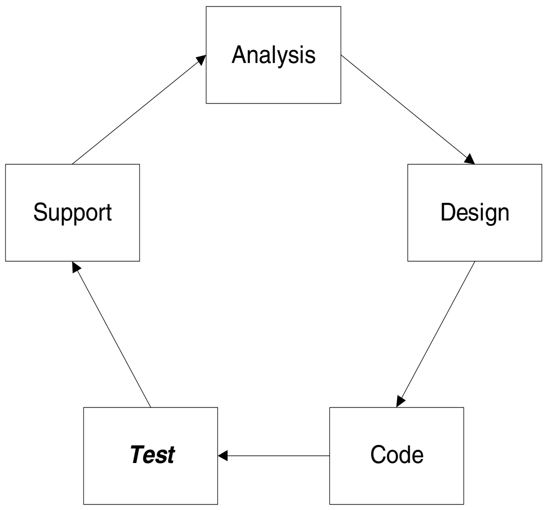


Figure 2.1 The linear sequential model or waterfall model

Another test strategy called the *"white box"*, or *"structural-based"* testing requires analysis of the control and/or data flow of a program. A tester requires to generate a test case by selecting specific sequence of statements or branches based on different testing criteria. If such a test case exists, we call this path a *"feasible path"*; otherwise, it is called an *"infeasible path"*.

A goal to select testing criteria is to decide when the testing efforts are sufficient enough to terminate the testing process with a level of confidence. This is called *"Test Adequacy Criteria"*. Formally, a test data adequacy criterion can be termed as a *stopping rule*. If a test data adequacy criterion focuses on the structural properties of a program or code, it is said to be a program-based adequacy criterion, including executing every statement or every branch in a program at least once [20]. These criteria are called *"all-statement coverage"* and *"all-branch coverage"*, respectively. To satisfy the all-statement coverage, for example, test cases must cause the program being tested to execute every statement within the program at least once.

In additional to the two basic coverage criterions above, *"all-du-path coverage"* is another major test data adequacy criterion. The *all-du-path* (definition-use) criterion requires that every definition-clear sub-path from every definition to all the successor nodes of each use reached by that definition to be tested. Uses may occur in predicates or in computations. In [21], Yang et al., have proposed the idea to extend this criterion, originally developed to test sequential program, to be applicable to parallel program testing.

Yang and Chung [35] proposed a model to represent the execution behavior of a concurrent program, and described a test execution strategy, testing process and a formal analysis of the effectiveness of applying path analysis to detect various faults in a concurrent program. An execution is viewed as involving a concurrent path (C-path), which contains the flow graph paths of all concurrent tasks. The synchronizations of the tasks are modeled as a concurrent route (C-route) to traverse the concurrent path in the execution, by building a rendezvous graph to represent the possible rendezvous conditions. The testing criterion is to examine the correctness of each concurrent route along all concurrent paths of concurrent programs. Their paper acknowledges the difficulty of C-path generation; however, the actual methodologies for the selection of C-paths and C-routes are not presented in the paper.

In [36], Taylor et al., proposes the notion of structural testing criteria to concurrent programs. Coverage criteria described include concurrency state coverage, state transition coverage, and synchronization coverage, which are analyzed after building a concurrency graph. No claim is made that a particular criterion suggested is ideal; rather the paper acknowledges that it is impractical to analyze a large program by building a concurrency graph.

Katayama et al., in [3] propose a test-case generation method with the event interaction graph (EIAG) and the interaction sequence testing criteria (ISTC). The proposed ISTC are based on sequences of interaction. A major problem with this approach is that path generation has exponential complexity.

All the aforementioned approaches propose a certain coverage criterion, which is accepted as the stopping condition for that approach. In the literature on testing concurrent programs, we can find no research work done to present empirical evaluation results of these criteria. In this Thesis we will provide the evaluation of some these commonly used coverage criteria, and present the quantitative comparison between them.

In the context of sequential programs, several researches have examined and evaluated test coverage criteria; we will be discussing them in the next section.

2.2 Related Work

In this section we discuss the research done in the field of evaluating adequacy criteria for sequential programs. There are a number of papers on the evaluation of test adequacy criteria in context to sequential programs [47]. In this section we provide a brief description of the related work done on evaluation and comparison of test adequacy criteria for sequential programs.

Three approaches to evaluate software testing methods have been proposed and developed, they are:

2.2.1 Formal analysis

Formal analysis of fault detecting ability is usually based on an abstract relation defined on test adequacy criteria. For instance, the studies of *subsume relation*, which defines a strictness order on adequacy criteria. This method has dominated the research in evaluation of test adequacy criteria [39, 40, 41]. It defines abstract relations on adequacy criteria, formally proves that particular pairs of adequacy criteria satisfy or dissatisfy the defined relation, and studies their relationships with fault detecting ability in various testing scenarios.

2.2.2 Simulation

Simulation is usually based on some simplified model of software testing [42, 43]. It can produce results in an ideal testing scenario and avoid some complicated factors, such as human factors [42]. However, it is difficult to apply this method to compare software test adequacy criteria of subtle differences.

2.2.3 Statistical experiment

Statistical experiment has the advantage of the universal applicability to compare all types of testing methods [44, 45, 46]. However, to prevent statistical experiment from certain potential invalidating factors, the experiments should be carefully designed and conducted. This will become clear in the next paragraph.

The basic form of statistical experiment with test adequacy criteria consists of three elements:

- A set of sample software systems.
- For each sample software system, there is a collection of adequate test sets with respect to the adequacy criteria under comparison.
- For each sample software system, there is a collection of known faults of the software.

As mentioned earlier, there are some potential invalidating factors in the application of this method. In [46], Hamlet pointed out the following two issues:

1. Sample programs: The particular collection of programs used in the experiment may be too small or too peculiar for the results to be trusted.
2. Test data: The particular test data created for each testing method may have good or bad properties that are not related to the testing method.

In addition to the above two problems, there can be another potential invalidating factor. That is:

3. Sample faults: The particular collection of known faults in each program used in the experiment may not be representative to the faults in real software development. They could be too easy or too difficult to be detected. They may have a peculiar distribution over the types of faults.

2.2.4 Test adequacy

Frankl and Weiss[40] in their experiments studied branch coverage and the use coverage data flow adequacy criterion using nine small programs of several different application areas. Their experiment attempted to avoid the human factors in the comparison of test adequacy criteria. Instead of using a small sample of adequate test sets generated by human testers for each criterion, they randomly generated a fairly large number of adequate test sets and calculated the probabilities of fault detecting.

They found that the probability of fault detecting is clearly dependent on the extent of test adequacy in such a way that only when the test adequacy is in the range of 97% to 100% coverage, can the probability of fault detecting be close to 1. The probability of fault detection drops rapidly when test adequacy is lower than 97%. This has been observed for both of the test adequacy criteria used in their experiment. Figure 2.2 provides an illustration of the dependence of the fault detecting probability on test adequacy [47].

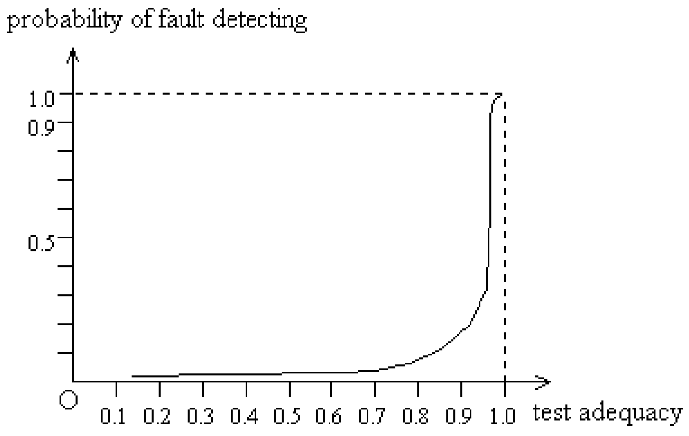


Figure 2.2 Illustration of the dependence of fault detecting probability on test adequacy

The perception that stricter adequacy criteria should have better fault detecting ability is confirmed by a number of experiments, such as Frankl and Weiss's experiment

[40], and Ntafos' [42] work on comparison of data flow and control flow test adequacy criteria.

CHAPTER 3

PRELIMINARIES

3.1 Overview

This chapter gives a brief overview about the testing of concurrent programs. Some terminologies and definitions relevant to our evaluation are also provided; this will be helpful in understanding the methodology of our experiment in the forthcoming chapters. In section 3.2 we discuss the common testing approaches for concurrent programs. Section 3.3 presents groundwork towards understanding evaluation and comparison of coverage criteria developed for various approaches of testing sequential and concurrent programs.

3.2 Approaches to testing concurrent programs

A concurrent program P or a sequential program is typically subjected to two types of testing:

- *black-box testing*: In this category, access to P's implementation is not allowed. Hence, only the specification of P can be used for test generation, and only the result (including the output and termination condition) of each execution of P can be collected.
- *white-box testing*: In this category, access to P's implementation is allowed. Here, both the specification and implementation of P

can be used for test generation. Also, any desired information about each execution of P and the structure of P can be collected.

Due to its complexity and size of the code or the inability to access code, white-box testing may not be practical during system or acceptance testing. There is a third type of testing, which is some where between the first two approaches:

- *limited white-box testing*: During an execution of P, only the result and *SYN-sequence* can be collected. An execution of a concurrent program P exercises a sequence of synchronization events, called a synchronization-sequence (SYN-sequence) [9]. Thus, only the specification and the SYN-sequences of P can be used for test generation. Also, an input and a SYN-sequence can be used to deterministically control the execution of P. Deterministic testing is described in the forthcoming subsection.

3.2.1 Non-deterministic testing

Non-deterministic testing of a concurrent program P involves the following steps:

1. Selection of a set of inputs for P
2. For each selected input X, P is executed with X many times and the result of each execution is examined.

If we execute P multiple times, then these non-deterministic executions of P with input X may exercise different SYN-sequences of P and thus may detect more failures than a single execution of P with input X. This approach can be used during (limited) white-box and black-box testing. The purpose of non-deterministic testing is to exercise as many distinct SYN-sequences as possible. However, experiments have shown that repeated executions of a concurrent program are not likely to execute different SYN-sequences [9, 19,23, 30, 31].

In general, non-deterministic executions are easy to perform and cost effective for many applications; however, each execution creates additional work since the result of the execution must be checked.

3.2.2 Deterministic testing

Deterministic testing of a concurrent program P involves the following steps:

1. Selection of a set of tests, each of the form (X, S), where X and S are an input and a complete SYN-sequence of P respectively.
2. For each selected test (X, S), a deterministic execution of P with input X according to S is forced. This execution determines whether S is feasible for P with input X. (Since S is a complete SYN-sequence of P, the result of such an execution is deterministic.)
3. Comparison of the expected and actual results (including the output, feasibility of S, and termination condition) of the forced execution. If the expected and actual results are different, a fault is detected in the program (or the test sequence). Fault can be located by using a replay tool. P is corrected after the fault has been located, P can be executed with each test (X, S) to verify that the fault has been removed and that in doing so, no new faults were introduced.

For deterministic testing, a test for P is not just an input of P. A test consists of an input and a SYN-sequence, and is referred to as an IN-SYN test. Deterministic testing provides several advantages over non-deterministic testing:

- Non-deterministic testing may leave certain portions or paths of P uncovered. Deterministic testing allows carefully selected SYN-sequences to be used to test specific portions or paths of P.
- Non-deterministic testing exercises feasible SYN-sequences only; thus, it can detect the existence of invalid, feasible SYN-sequences of P, but not the existence of valid, infeasible SYN-sequences of P. Deterministic testing can detect both types of faults.
- After P has been modified to correct an error or enhance its functionality, deterministic regression testing with the inputs and SYN-sequences of previous executions of P provides more confidence about the correctness of P than non-deterministic testing of P with the inputs of previous executions.

3.2.3 Combination of Deterministic and Non-deterministic testing

Although deterministic testing has advantages over non-deterministic testing, it requires considerable effort for selecting SYN-sequences and determining their feasibility. This effort can be reduced by combining deterministic and non-deterministic testing.

Prefix-Based Testing and Mutation-Based Testing are examples of a combination of Deterministic and Non-Deterministic Testing.

3.2.4 Reachability testing

Reachability testing is an approach that combines non-deterministic and deterministic testing. It is based on prefix-based testing; Prefix-based testing controls a test run up to a certain point, and then lets the run continue non-deterministically. The controlled portion of the test run is used to force the execution of a "prefix SYN-sequence", which is the beginning part of one or more feasible SYN-sequences of the program. The non-deterministic portion of the execution randomly exercises one of these feasible sequences

3.3 Evaluation of adequacy criteria

Most of the approaches we have mentioned in the previous section for testing concurrent programs make use of a certain coverage criteria to measure the test adequacy. A coverage criteria is used to determine when testing can stop and to guide generation of input values for test cases. In-spite of the fact that there has been a lot of research on evaluation of adequacy criteria for sequential programs, comparison of fault detecting ability is still remains notoriously difficult even in the case of sequential programs and especially in the case of concurrent programs.

The approach which we will follow in our evaluation of adequacy criteria falls under the category of statistical experiment [47] it has been discussed earlier in section 2.2. The following steps are involved in this approach of evaluation:

Assume that C_1, C_2,..., C_n are the test adequacy criteria under comparison.

- The experiment starts with the selection of a set of sample programs, say $P_1, P_2,..., P_m$.
- Each program has a collection of faults, which are known due to previous experience in the software development process or planted artificially, say, by applying mutation operations or manually.
- For each program P_i, i=1,2,..., m, and adequacy criterion C_j, j=1,2,...,n,k; test sets $T_{1ij}, T_{2ij},..., T_{kij}$, are generated in some fashion so that T_{uij} is adequate to test P_i according to criterion C_j.
- The proportion r_{uij} of faults detected by the test T_{uij} over the total number of known faults in the program P_i is calculated.
- Finally, statistical inferences are made based on the data r_{uij}, i=1, 2,..., n, j=1, 2,..., m, and u=1,2,..., k.

As any experimental method, there are some potential invalidating factors in the application of the method which have been previously discussed in section 2.2.

This thesis is essentially following the above stated approach of statistical experiment for the evaluation of adequacy criteria for the testing of concurrent.

CHAPTER 4

COVERAGE CRITERIA

4.1 Overview

In this chapter we will be presenting the description of all the coverage criteria that we are evaluating in this thesis. Coverage criteria are used to determine when testing can stop and to guide the generation of test cases. A number of coverage criteria have been developed for sequential programs. For instance, the *all-paths* criterion requires every path to be executed at least once; it is one of the strongest coverage criterions. Since the number of paths in a program may be very large or infinite, it may be impractical to cover them all. Thus, a number of weaker criteria have been defined by G.J. Meyers in [16].

The minimum structural coverage criterion is statement coverage, which requires every statement in the program to be executed at-least once.

Some stronger coverage criteria focus on the predicates in the program. The predicates in *if-else* and *loop* statements divide the input domain into partitions in the program. Simple predicates contain single condition which is either a single Boolean variable (e.g., if (B)) or a relational expression (e.g., if (e1 < e2)), possibly with one or more negation operators (!). Compound predicates contain two or more conditions connected by the logical operators AND ($\wedge$) and OR ($\vee$), (e.g., if ((e1 < e2) $\wedge$ (e2 < e3))). Predicate coverage criteria require certain types of tests for each predicate:

Decision coverage: requires that every (simple or compound) predicate evaluate to true and false at least once. Branch coverage is also knows as branch coverage.

Condition coverage: requires that each condition in each predicate evaluate to true and false at least once. Note that decision coverage can be satisfied without testing each condition in the predicate. For example, for the predicate (A $\wedge$ B), decision coverage is satisfied by the two tests (A=true, B=true) and (A=true, B=false), neither of which causes A to be false. Condition coverage requires A to be false at least once.

Decision/condition coverage: requires that both decision coverage and condition coverage be satisfied. Note that condition coverage can be satisfied without satisfying decision coverage. For example, for the predicate (A $\vee$ B), condition coverage is satisfied by the two tests (A=true, B=false) and (A=false, B=true), neither of which causes the predicate to be false. Decision/condition coverage requires the predicate to be false at least once.

Multiple-condition coverage: requires all possible combinations of condition outcomes in each predicate to occur at least once. Note that for a predicate with N conditions, there are 2N possible combinations of values for the conditions.

The criteria can be compared based on *subsumes* relation. A coverage criterion C1 is said to subsume another criterion C2 if and only if any set of paths that satisfies criterion C1 also satisfies criterion C2. For example, decision coverage subsumes statement coverage since covering all decisions necessarily covers all statements. This does not mean, however, that a coverage criterion that subsumes another is also more effective at detecting errors. Fig. 4.3 shows a hierarchy of criteria based on subsumes relation.

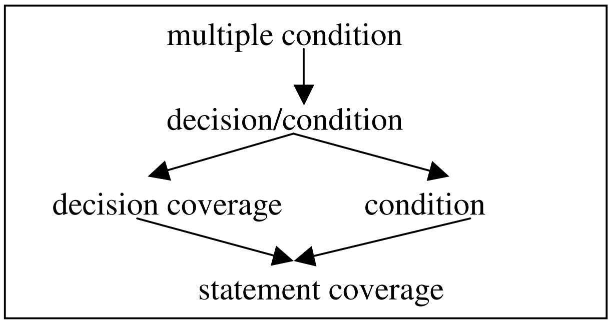


Fig. 4.1 Hierarchy of sequential structural coverage criteria based on subsumes relation.

4.2 Evaluated Coverage Criteria

This section discusses all the coverage criteria that our work evaluates, in terms of their respective adequacy. The selection of the specific coverage criteria listed in this section is a result of rigorous effort and time spent in the study of several different approaches which are proposed for testing concurrent programs, including those mentioned in section 2.1. We studied all the approaches and the respective coverage criterion employed by that approach. These coverage criteria were scrutinized for their relative importance in testing concurrent programs and their ease of empirical evaluation. We have already mentioned some of the possible problems that might arise in the evaluation of certain coverage criteria in section 2.1.

Following is the final list of seven coverage criteria that have been selected for the purpose of our evaluation:

All synchronization pair coverage criterion: The all synchronization-pair coverage criterion requires that every possible pair of statements that can be synchronized is actually synchronized at least once during testing.

All branches coverage criterion: This criterion requires that all the branches in the program are covered at least once during testing. This criterion is same as the aforementioned decision coverage criterion.

All decision/condition coverage criterion: This criterion requires that the truth and falseness of every simple condition in a branching statement is covered at least once by a test case. All conditions coverage criterion subsumes the all branches coverage criterion and is the same as all decision/condition coverage criterion to the.

All du-pair coverage criterion: This criterion requires that every distinct definition and use pair of all the variables in the concurrent program are covered at least once in the test execution.

All synchronization-pair and all branches coverage: This criterion requires the coverage of both the previously mentioned all-synchronizable-statement-pairs coverage AND all-branches coverage.

All synchronization-pair and decision/condition coverage: This criterion requires the coverage of both the previously mentioned all-synchronizable-statement-pairs coverage AND all decision/condition coverage.

All synchronization-pair and all du-pair coverage: This criterion requires the coverage of both the previously mentioned all-synchronizable-statement-pairs coverage AND all-du-pair coverage.

CHAPTER 5

EVALUATION AND RESULTS

5.1 Objective

Once we finalized the seven coverage criteria to be evaluated, we start with the formal evaluation procedure. The purpose of this thesis is to provide a useful and quantitative evaluation of the seven coverage criteria that can be useful in the selecting one or many of these coverage criteria for the testing of specific concurrent programs.

Keeping this in mind all the way through the evaluation procedure we have opted for a generic example of a concurrent program. The example program we are using for our evaluation is the implementation of the sliding window protocol and also includes the simulation of an unreliable medium and the application programs, in Ada. We received this program from the research work of R.H.Carver [11].

In our evaluation we are trying to come up with results on the testing of our example program following the seven approaches mentioned in chapter 4. We are expecting to measure the robustness of each coverage criterion in terms of the number of mutants which are killed by that criterion [6]. At the same time we are also going to keep a track of the cost, specifically the effort and time spent in following a coverage criterion to generate the synchronization sequences for our example. This can be done by keeping a track of the number of synchronizations generated in a particular synchronization sequence.

Evaluation objective is to determine:

1. Number of mutants killed.
2. Total number of synchronizations in the synchronization sequence.

Number of Mutants Killed $\rightarrow_{\text{signifies}}$ Robustness of a Coverage Criteria
Number of Synchronization in a SYN-sequence $\rightarrow_{\text{signifies}}$ Cost of the Coverage Criteria

Figure 5.1 Objective.

5.2 Methodology

For every coverage criteria the basic design of our evaluation involves the four major steps.

1. Construction of synchronization sequences
2. Deterministic execution of SYN-sequence
3. Mutation testing
4. Analysis of output file

These four steps will be explained in detail in the forthcoming subsections.

5.2.1 Construction of Synchronization Sequences

An execution of a concurrent program P exercises a sequence of synchronization events, called a Synchronization-sequence (or SYN-sequence) [9]. A feasible SYN-sequence is a sequence of synchronization events that can be exercised during the execution of a concurrent program. Construction of a synchronization sequence which covers a particular coverage criterion is the costliest part in the deterministic testing of concurrent programs.

Before we explain the construction of the SYN-sequence we will briefly give an explanation about the example program that we have used in our evaluation; this follows in the forthcoming subsection.

5.2.1.1 The Example Program SWP

For the purpose of our evaluation we needed such a concurrent program that is non-trivial and exemplifies the complexity and inter-process (inter-task) communication of concurrent programs that are used in real world scenarios.

We have used as our example an implementation of the Sliding Window Protocol in Ada language; this program has been adopted from “Selecting and Mapping Test Sequences from Formal Specifications of Concurrent Programs” [11]. Our version of the Sliding Window Protocol (SWP) consists of one Sender and one Receiver communicating across an unreliable medium. Messages are sent tagged with sequence numbers in the range 1..6, and these sequence numbers also constitute the acknowledgments. The size of the window is two. This Ada implementation of SWP has eight tasks (processes) and 530 statements.

In the forthcoming section on mutation testing we will see that this SWP program is later used in generating 1007 SWP mutants that will be tested with the SYN-sequence for each coverage criterion.

Figure 5.2 depicts the basic structure of the SWP implementation. Every ellipse depicts a task of the SWP program. The direction of the solid arrows shows the entry call, for instance, Sender task gives an entry call that is accepted by Medium_SR task. Synchronization in Ada is achieved by blocking entry and accept calls.

Client_S and Client_R simulate the application programs; Medium_SR and Medium_RS simulate the non-reliable medium from the sender to the receiver. The window size is 2 so there are two timers required at any instance of time; Timer1, Timer0.

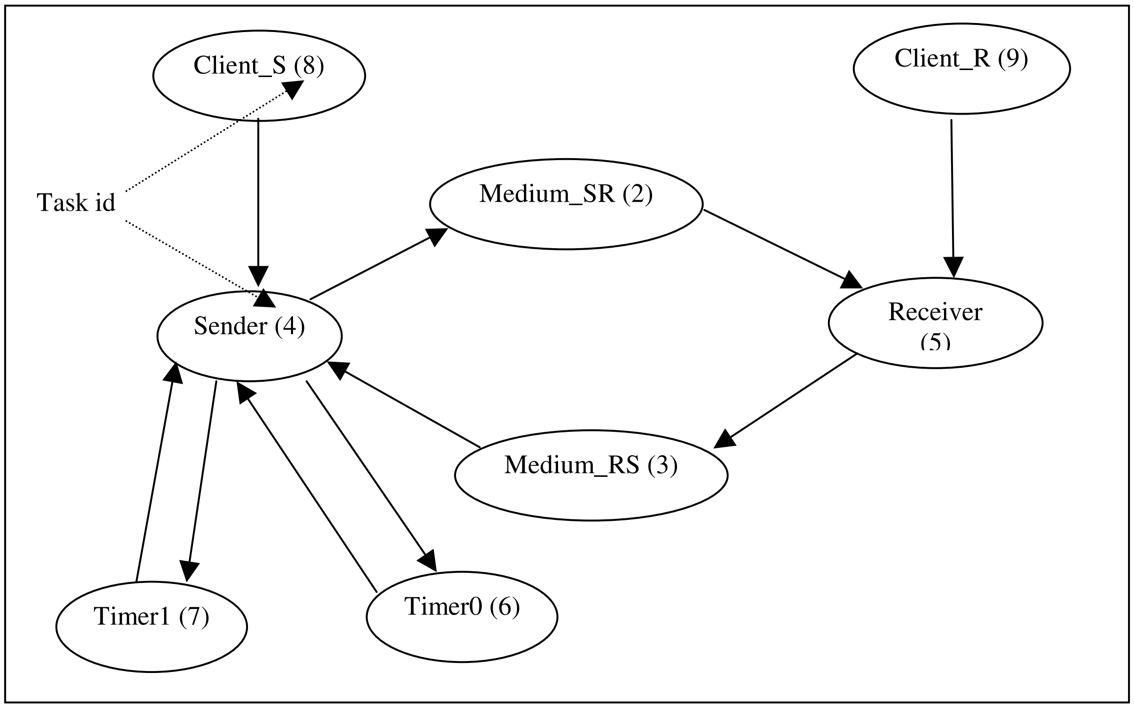


Figure 5.2 Structure of the SWP implementation

The following behaviors can be identified:

- Sender: local behavior of the sender.
- Receiver: local behavior of the receiver.
- Sender-to-Receiver: end-to-end behavior involving the flow of messages from the sender to the receiver.

- Receiver-to-Sender: end-to-end behavior involving the flow of acknowledgements from the receiver to the sender.

5.2.1.2 The construction of the SYN-sequence

The format of a test sequence for deterministic testing depends upon the concurrent programming constructs used in the program under test. Formal definitions of test sequences for various constructs and languages can be found in [12,13]. In Ada, processes are called tasks. Tasks communicate through ports, which are called entries. Communication occurs using synchronous message passing called rendezvous, in which messages are sent using blocking entry call statements, and received using blocking accept statements.

Every specific coverage criterion amongst the seven that we have chosen to evaluate in this thesis requires the fulfillment of certain conditions while testing, in order to claim that a specific test run covers the respective coverage criterion. We have to carefully select the sequence of synchronization events in order to fulfill this for every criterion. Selection of such synchronization sequences can be done by using reachability graph, but this becomes too intricate because of the state explosion problem. Moreover with our SWP example program having around 500 nodes in its control flow graph this problem becomes very pronounced. We followed a manual generation of synchronization sequences due to the absence of any available automation in this regard.

For P and each adequacy criterion Cj, k SYN-sequences S1j, S2j,...,Skj, are generated in some fashion so that k SYN-sequences are adequate to test SWP program according to criterion Cj.

We have kept no restrictions on the length of the generated sequences or the number of different sequences required to cover a specific coverage criterion, except for the restrictions imposed explicitly by the SWP program and Ada language. There is no maximum length of the feasible synchronization sequences for the SWP program; this is because of the presence of multiple while loops which can run through infinitely many iterations if the required conditions are met in the program flow. While generating the SYN-sequence our sole concern is the corresponding coverage criteria for which the SYN-sequence is being generated, the length of the sequence should be as small as possible. The length of a SYN-sequence symbolizes the cost of generating the SYN-sequence, therefore it should be kept in mind that the length of the sequence should be towards the minimum required to cover the coverage criteria. Absolute care is taken to generate as small sequences as possible, still their can be instances where the sequence which we are using for specific coverage might not be the smallest possible, such instances are minimized. Following this methodology will make our final result stronger and reliable in terms of calculation of cost and robustness of each coverage criteria.

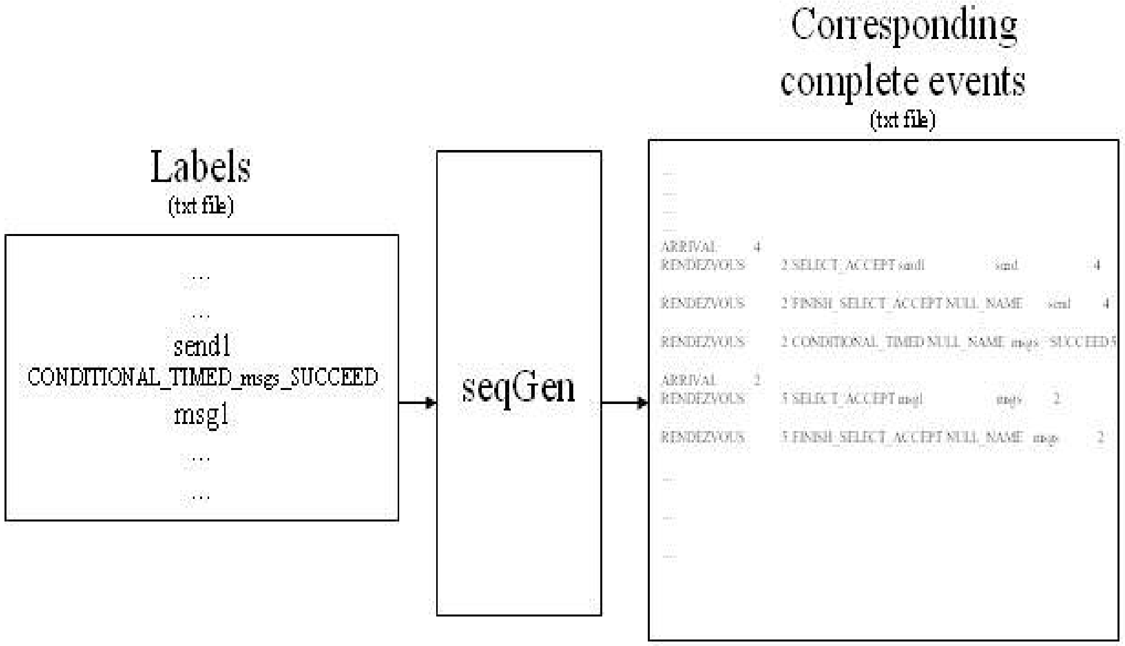
After the generation of feasible SYN-sequence we are required to transform this SYN-sequence into a format that is readable by another program that will deterministically force this sequence on the example SWP program [5].

A test sequence for a concurrent Ada program is a sequence of rendezvous events. A rendezvous event is denoted by (L, C, U, N, D), where L is a label (optional), C denotes the calling task, U the accepting task, N the entry name, and D other information for this rendezvous event. Thus a sequence of rendezvous events, referred to as a rendezvous sequence or R-sequence, can be represented by:

$((L_1,C_1,U_1,N_1,D_1), ((L_2,C_2,U_2,N_2,D_2),\ldots$

$(L_i,C_i,Ui,Ni,Di),\ldots)$

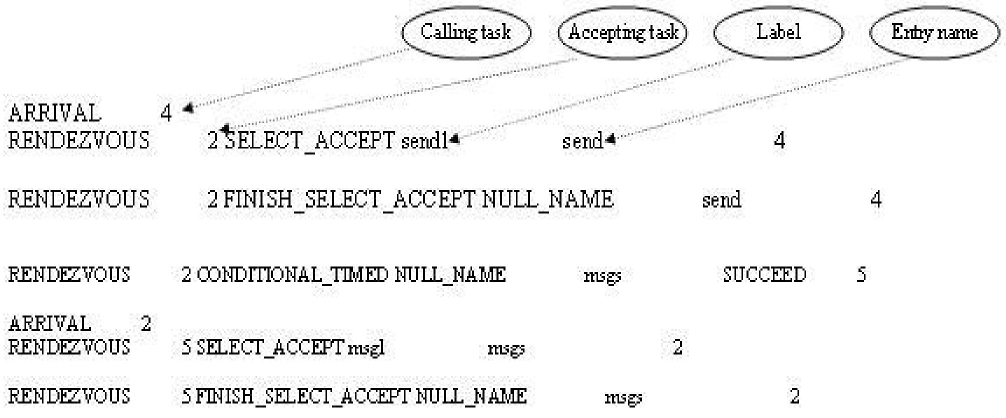
For the conversion of labels to complete formatted sequence we developed a simple java program named "seqGen" which could convert labels of specific rendezvous events in the SYN-sequence to formatted events having the complete rendezvous information about that event in a form readable by another program named Feasibility Control, a detailed explanation of such deterministic execution can be found in [5]. This program checks and deterministically runs the input SYN-sequence. Figure 5.3 provides a high level diagrammatic representation of the seqGen.



```
...
...
...
...
ARRIVAL       4
RENDEZVOUS        2 SELECT_ACCEPT send1             send              4

RENDEZVOUS        2 FINISH_SELECT_ACCEPT NULL_NAME     send      4

RENDEZVOUS        2 CONDITIONAL_TIMED NULL_NAME msgs  SUCCEED 5

ARRIVAL       2
RENDEZVOUS        5 SELECT_ACCEPT msg1              msgs     2

RENDEZVOUS        5 FINISH_SELECT_ACCEPT NULL_NAME  msgs        2
...
...
...
```

Figure 5.3 seqGen converts labels into complete rendezvous events.

Figure 5.4 depicts the complete SYN-sequence generated by seqGen when a sequence of labels is provided as an input.



```
ARRIVAL       4
RENDEZVOUS        2 SELECT_ACCEPT send1             send                  4

RENDEZVOUS        2 FINISH_SELECT_ACCEPT NULL_NAME          send                4

RENDEZVOUS      2 CONDITIONAL_TIMED NULL_NAME          msgs           SUCCEED       5

ARRIVAL       2
RENDEZVOUS      5 SELECT_ACCEPT msg1               msgs                2

RENDEZVOUS      5 FINISH_SELECT_ACCEPT NULL_NAME         msgs                   2
```

Figure 5.4 Complete SYN-sequence.

The next step to be followed after the creation of the feasible SYN-sequence is deterministic testing, which is explained in the forthcoming subsection.

5.2.2 Deterministic execution of the SYN-Sequence

In this section, we briefly describe the work done by Richard Carver and K. C. Tai in [5], which shows how to accomplish deterministic execution testing of a concurrent Ada program according to a given SYN-sequence.

As a first step the format of a SYN-sequence that provides sufficient information for deterministic execution is defined. Then it is shown how to transform a concurrent Ada program P into a slightly different program P* (also written in Ada) so that an execution of P* with (X, S) as input, where X and S are an input and SYN-sequence of P respectively, determines whether or not S is feasible for P with input X and produces the same result as P with input X and SYN-sequence S would, provided that S is feasible [5]. Tools for transforming concurrent Ada programs for deterministic execution testing are also described in [5].

Deterministic execution testing is needed not only for detecting errors in P but also for verifying corrections made to the concurrent Ada program. Assume that an execution of P with input X has exercised a SYN-sequence S, which is invalid according to P's specification. After an attempt has been made to correct the error(s) detected by this erroneous execution, we need to verify that S is infeasible for the corrected version of P with input X. Deterministic execution (regression) testing is also needed to verify that corrections made to P do not produce unexpected effects for previous successful executions of P.

There is also a description of the development of a language-based approach to solving the SYN-sequence definition, collection, and feasibility problems for language L. This approach is applied to concurrent Ada programs as follows.

1. The format of a SYN-sequence of an Ada program is defined in terms of the synchronization constructs available in Ada so that a SYN-sequence provides sufficient information for deterministic execution.
2. A SYN-sequence collection tool for Ada is developed which transforms a program P written in Ada into a slightly different program P’ (also written in Ada) so that P’ is equivalent to P except that during an execution of P’ the SYN-sequence of this execution is also collected.
3. A SYN-sequence feasibility tool for Ada is developed which transforms an Ada program P into a slightly different program P* (also written in Ada) so that an execution of P* with (X, S) as input, where X and S are an input and SYN-sequence of P respectively, determines whether or not S is feasible for P with input X and produces the same result as P with input X and SYN-sequence S would, provided that S is feasible.

A language-based approach to deterministic execution testing has the following advantages: (a) Since the transformation of a concurrent program for SYN-sequence collection or feasibility produces a program written in the same language, this approach does not create a portability problem; (b) the definition of a SYN-sequence and the development of a SYN-sequence collection or feasibility tool for a concurrent language are independent of the implementation of this language; and (c) the source transformation for SYN-sequence collection or feasibility for a concurrent language L can serve as a high-level design for L's implementation-based testing tools. (Implementation-based tools have the advantage of having less run-time overhead because the control of synchronization events is performed at the implementation level instead of the source level) Also, a language-based solution to SYN-sequence collection or feasibility may suggest an efficient implementation-based solution. More discussion of this language-based approach can be found in [10].

5.2.3 Mutation Testing

Mutation-based software testing is a powerful technique for testing software systems. It requires executing many slightly different versions of the same program to evaluate the quality of the test cases used to test the program. Mutation-based testing has been applied to sequential software; however, problems are encountered when it is applied to concurrent programs. These problems are a product of the non-determinism inherent in the executions of concurrent programs [6].

Mutation-based testing [7, 8] helps the tester in creating test cases and improves the quality of the tests. It involves constructing a set of mutants of the program under test. Each mutant differs from the program under test by one mutation. A mutation is a single syntactic change that is made to a program statement (generally inducing a typical programming error) [6].

Test cases are used to kill mutant programs by differentiating the output of the mutants from that of the program under test. If a test case causes a mutant program to produce incorrect output, then that test case is strong enough to detect the programming errors represented by that mutant, and the mutant is considered to be dead. The goal of mutation-based testing is to kill a large number of mutant programs. Each set of test cases is used to compute a mutation score; a score of 100% indicates that the test cases kill all mutants of the program under test. (Some mutants are functionally equivalent to the original test program and can never be killed. This is factored into the mutation score.)

In our evaluation to measure the adequacy of the generated test sequences, we have adopted the semi-automatically mutated Ada SWP implementation from [11] which was implemented using mutation operators of the Mothra mutation system [l4]. (Mothra was used to generate mutations of Fortran statements, and then these mutations were transformed into Ada.) Since Mothra was developed for Fortran 77, a few of the mutations were not applicable or they needed to be modified slightly for Ada.

In all we have 1007 different mutant programs (mutants) of our example SWP program. The generated SYN-sequence should be deterministically executed on all of these mutants. An SWP mutant is considered to be distinguished (killed) if a forced execution of the mutant according to a valid (invalid) test sequence is found to be infeasible (feasible), or the execution resulted in an incorrect result. The test execution is carried on a Windows platform, this execution is automated by writing a batch file which carries commands for the compilation, execution and finally writing the output generated for each execution in a text file.

Throughout the period of our mutation testing during the deterministic execution of the mutants we have to keep checking if an execution has ended up in a deadlock situation. If a mutant encounters a deadlock during the execution, then it is considered killed by the SYN-sequence. While this information will automatically be written in our output file we still have to break the deadlock of the execution manually and resume the execution of the batch file again from the next mutant onwards. On a UNIX system, script could have been written to avoid this manual termination in case of a deadlock, but I could not find a similar script for the Windows platform. Each mutant that encounters a deadlock is noted for the respective sequence of the respective coverage criteria. During the test execution of all the mutants the detailed output is written in the output file that is later analyzed for the number of mutants killed.

5.2.4 Analysis of output file

The final output file generated after the deterministic execution of the SYN-sequence on all the mutants, carries the final results and information about the number of mutants killed by that respective SYN-sequence. The output file has information about every mutant, which has been tested followed by its mutant number, which sequentially increases from 0 to 1146, for every SYN-sequence. In all there are 1007 different mutants within the range 1 to 1146.

Analysis of the final output file is done by analyzing the output produced by every mutant program. A mutant is assumed to be killed if the output produced by the mutant doesn’t match with the output produced on a simple unaltered program, the unaltered SWP program in our case. Whenever there is a mismatch of output then the mutant is assumed to be killed and the mutant number is noted. All the mutants which produce the same output as that of an unaltered SWP program are the ones which have not been killed by the SYN-sequence. These mutants are separately noted under the list of mutants not killed by that specific SYN-sequence. In the end of the execution of all the 1007 mutants, we have a comprehensive list containing:

- Mutants killed
- Mutants not killed
- Mutants which ended up in a deadlock

This list corresponds to a single SYN-sequence, which is one of the many used for a specific coverage criterion under test. A specific coverage criterion can require multiple SYN-sequences to cover the requirements of that coverage criterion. After the execution of all the SYN-sequences of a particular coverage criterion a final list of results is generated for that particular coverage criterion. This list also has the same three columns of the previous lists and includes all the mutants, which have been killed by all the SYN-sequences. From the complete set of 1007 mutants the mutants killed are marked and we are left with the total number of mutants which are not killed by this coverage criterion.

The set of mutants killed by a coverage criterion $\boldsymbol{m}$ = $\{\boldsymbol{MutKilled_m}\}$

Total mutants killed by $\boldsymbol{i}^{th}$ SYN-sequence covering $\boldsymbol{m}$ = $\{\boldsymbol{Killed_i}\}$

Thus we have the relation:

$$\{MutKilled_m\} = \bigcup \{ Killed_i \}$$

Total number of mutants killed by a specific coverage criterion will be the number of elements in the set *MutKilled*$_m$. Total number of mutants not killed will be 1007 less the number of elements in *MutKilled*$_m$.

Total number of mutants killed by coverage criterion **m** = ***num{MutKilled***$_m$***}***

Total number of mutants not killed by coverage criterion **m** = **1007 –** ***num{MutKilled***$_m$***}***

5.3 Results

This section presents the comprehensive results of the final evaluation of each of the seven coverage criteria against all the mutants.

In the results we have even included some very explicit details about the SYN-sequences used; these can be qualified as program specific to our example program of sliding window protocol. An SWP program essentially runs on messages passing from a receiver to the sender, in each SYN-sequences there essentially is going to be en exchange of at-least one or more messages and acknowledgements between the sender and the receiver within the program. In the results for every coverage criteria we have relevant data about the following:

1. Number of Mutants killed
2. Total number of Synchronizations
3. Total number of different sequences
4. Total number of messages sent

Of course, for the purpose of our evaluation we just require the first two statistics for each coverage criterion and hence the latter are not shown in the final evaluation results. The results are presented in the following subsections.

5.3.1 Evaluation of all synchronizable pair coverage criterion

This coverage criterion is specific for concurrent programs. We are definitely curious to know how well this criterion does in comparison to the other more traditional criteria, which are more frequently used in the testing of sequential programs.

After examining the results of this criterion (Table 5.1) we can infer that around 64% of the total mutants have been killed by this coverage, and additionally the required SYN-sequence used didn't have a lot of synchronizations. Even though the All-sync pair coverage criterion doesn't kill maximum number of mutants the cost incurred in testing through this coverage criteria is significantly less.

Table 5.1 Evaluation of all synchronization pair coverage criterion.

Total number of mutants tested	1007
Total number of mutants killed	644
Total number of Synchronizations (Rendezvous events)	54

5.3.2 Evaluation of all branches coverage criterion

The All Branches Coverage Criterion is typically used in the testing of sequential programs, our evaluation results with this coverage criterion show that it doesn't come out to be very expensive in terms of the cost (Total number of synchronizations) but still we are able to kill around 74% of all the mutants.

Table 5.2 Evaluation of all branches coverage criterion.

Total number of mutants tested	1007
Total number of mutants killed	748
Total number of Synchronizations (Rendezvous events)	81

5.3.3 Evaluation of all decision/condition coverage criterion

The Criterion of All decision/condition coverage is stronger than all branches coverage as explained in Chapter 4. The results also present a conforming view. The distinguishing feature in the evaluation results of this criterion is the number of synchronizations required for the coverage. There is a steep rise in the effort required for testing using this coverage criterion as compared to the afore mentioned criteria, which is measure in the terms of number of synchronization sequences.

Table 5.3 Evaluation of all decision/condition coverage criterion.

Total number of mutants tested	1007
Total number of mutants killed	876
Total number of Synchronizations (Rendezvous events)	485

5.3.4 Evaluation of all du-pair coverage criterion

There isn’t any strong subsumes relationship between the all Du-pair coverage criterion and the other control flow based criteria used in our evaluation. This criterion appears quite strong in its coverage, and this can also be inferred from the number of mutants killed by the criterion. On the down side, this criterion is very demanding in terms of effort and time spent in the generation of SYN-sequence for this criterion. We are able to kill 96% of all the mutants, but the cost increases manifold. It should also be noted that the generation of SYN-sequences for this criterion required effort and time which is manifolds the cost required for previously evaluated criteria.

Table 5.4 Evaluation of all du-pair coverage criterion

Total number of mutants tested	1007
Total number of mutants killed	967
Total number of Synchronizations (Rendezvous events)	1023

5.3.5 Evaluation of all Sync-pair and all branches coverage

This is our first approach in which we are combining two different coverage criteria, moreover, it should be noted that one of the criteria is specific for concurrent programs and the other one is specific for sequential programs. A substantial increase in the number of mutants killed can be eminent; at the same time there isn't a huge increase in the number of different synchronizations. It is important to note the difference between the cost required for this testing criterion from the other criterions.

Table 5.5 Evaluation of all Sync-pair and all branches coverage.

Total number of mutants tested	1007
Total number of mutants killed	801
Total number of Synchronizations (Rendezvous events)	142

5.3.6 Evaluation of all Sync-pair and all decision/condition coverage

The results of the evaluation of this combined coverage criterion are pretty consistent with the results of the two criteria presented separately, earlier in this section.

We can notice some increase in the number of mutants killed but the trade-off against cost cannot be ignored. This coverage is able to kill 89% of all the mutants.

Table 5.6 Evaluation of all Sync-pair and all decision/condition coverage.

Total number of mutants tested	1007
Total number of mutants killed	897
Total number of Synchronizations (Rendezvous events)	517

5.3.7 Evaluation of all Sync-pair and all du-pair coverage

The evaluations of this coverage criterion yields the best results in terms of the number of mutants killed, we are able to kill 97% of all the mutants which is the maximum amongst all the criteria which we have evaluated. However, criterion required the maximum number of synchronizations and hence the maximum cost. It can be inferred that this criteria is practically not very economical.

Table 5.7 Evaluation of all Sync-pair and all du-pair coverage

Total number of mutants tested	1007
Total number of mutants killed	974
Total number of Synchronizations (Rendezvous events)	1055

5.4 Comprehensive Evaluation Results

In the final table we are presenting the overall picture of the evaluations of all the coverage criteria. We can get a precise representation of the relative comparisons of robustness and the cost of each criterion. Amongst our seven criteria it can be discerned that the all sync-pair & all du-pair coverage criteria is the strongest amongst the seven.

At the down side this criterion happens to be one of the costliest criteria; this can be determined by comparing the huge number of synchronizations required for this specific coverage.

The most economic coverage criterion in terms of cost is the all-synchronizable pair coverage, but at the same time this criterion is the one which is the least strong in terms of number of killed mutants.

If we strictly analyze the final comprehensive evaluation results then we can infer that all sync-pair & all branches coverage criterion is the best suited if we consider the trade-off between the robustness and the cost of coverage of all the coverage criteria.

Table 5.8 Comprehensive Evaluation Results.

Coverage criteria	**Num. of mutants killed (Total=1007)**	**Total number of Sync.**	**Num of mutants killed per Sync.**
All sync pair	644	54	11.93
All branches	748	81	9.23
All decision/condition	876	485	1.81
All du-pair	967	1023	0.95
All sync pair & all branches	801	142	5.64
All sync pair & all decision/condition	897	517	1.74
All sync pair & all du-pair	974	1055	0.92

Figure 5.5 presents comprehensive evaluation result of all our seven coverage criteria in terms of their robustness to kill the mutants. The coverage criteria in this table are arranged in the ascending order of the respective number of mutants killed by each criterion. The bars indicate the robustness of the approach.

Inference drawn from this part of evaluation:

"All Synchronization pair & all du-pair coverage" is the most robust amongst the coverage criteria evaluated in our experiment.

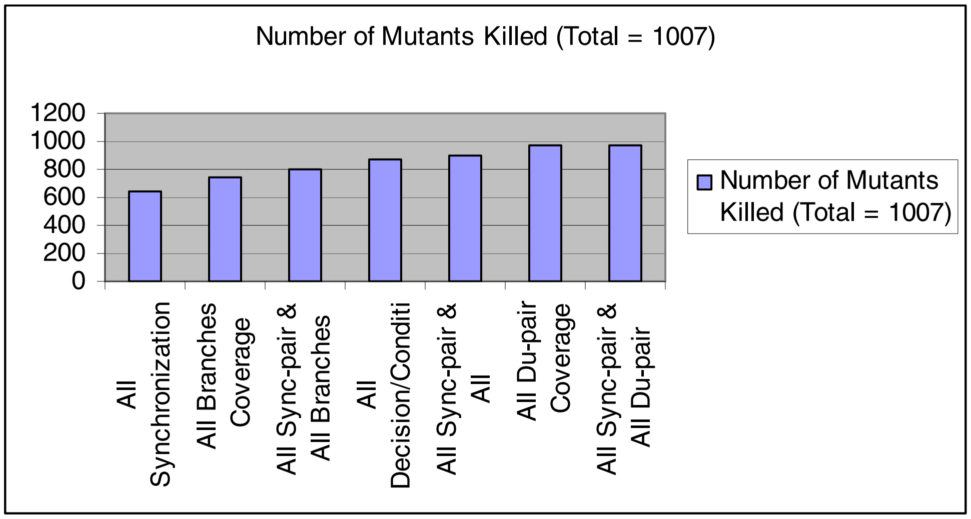


Figure 5.5 Number of Mutants killed.

Figure 5.6 presents comprehensive evaluation result of all our seven coverage criteria in terms of the number of synchronizations required in their SYN-sequence.

The coverage criteria in this table are arranged in the ascending order of the respective number of synchronizations required in their SYN-sequence. The bars indicate the cost of each approach.

Inference drawn from this part of evaluation:

"All Synchronization pair & all du-pair coverage" is the most costly amongst the coverage criteria evaluated in our experiment.

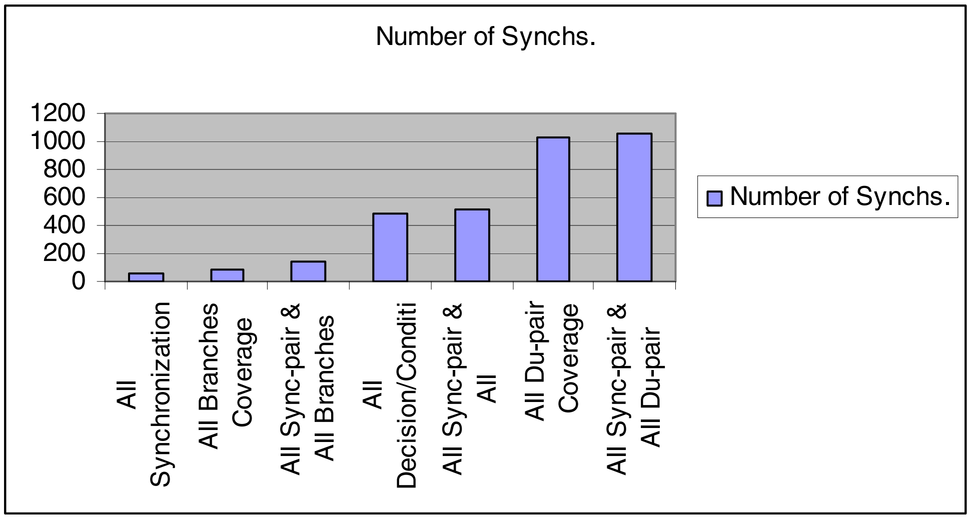


Figure 5.6 Number of Synchronizations.

Figure 5.7 presents comprehensive evaluation result of all our seven coverage criteria in terms of the number of mutants killed per synchronization.

The coverage criteria in this table are arranged in the ascending order of the respective number of mutants killed per synchronization. The height of the bars indicate the number of mutants killed per synchronization.

Inference drawn from this part of evaluation:

"All synchronization pair coverage" is the least costly amongst the coverage criteria evaluated in our experiment.

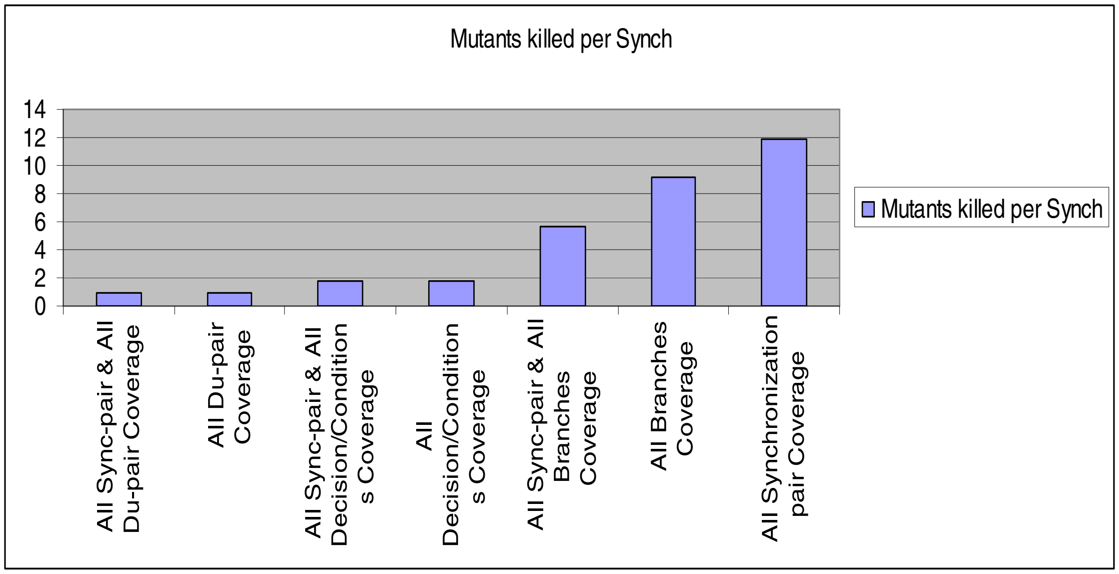


Figure 5.7 Mutants killed per Synchronization.

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

The primary objective of this thesis was to provide an empirical evaluation of adequacy criteria for testing concurrent programs. We provided evaluation results of seven different coverage criteria used in testing of concurrent programs based on our experiments.

We have used mutation based testing to set out benchmark for robustness.

Following are the contributions of this thesis based on the evaluation of the seven coverage criteria studied in the thesis:

- All synchronization pair coverage is the least costly amongst the coverage criteria evaluated in our experiment but it is not as robust in its adequacy as compared to the other six criteria.
- All branches coverage criterion is more robust than the all synchronization pair coverage but also happens to be a little costlier.
- All decision/condition coverage criteria lies somewhere in between all the coverage criteria in terms of its cost and robustness.
- All synchronization pair & all du-pair coverage is the most costly amongst the coverage criteria evaluated in our experiment.
- All synchronization pair & all du-pair coverage is the most robust amongst the coverage criteria evaluated in our experiment at the same

time it also happens to be the most costly of all the coverage criteria we evaluated.

- If we strictly analyze the final comprehensive evaluation results then we can infer that all sync-pair & all branches coverage criterion is the best suited if we consider the trade-off between the robustness and the cost of coverage of all the coverage criteria.

Sequential program testing criteria are more effective than expected when they are applied for testing concurrent programs.

6.2 Future work

Following are the suggestions for future work:

- The empirical results of evaluation of this thesis are based on the experiments conducted on just one example, more examples can be used to further consolidate the evaluation
- We studied several different coverage criteria and short-listed the seven criteria based on their importance in concurrent program testing and the feasibility of their empirical evaluation. More coverage criteria can be evaluated to make the list very comprehensive.
- Future work can be directed towards evaluation of multithreaded/distributed Java programs.
- Propose new coverage criteria that are more effective, efficient and scalable.

APPENDIX A

EXAMPLE PROGRAM SLIDING WINDOW PROTOCOL

EXAMPLE PROGRAM SLIDING WINDOW PROTOCOL

Here we are presenting the actual code of the SWP program that we have used for our evaluation. The program is in Ada and has sufficient comments to make it self-explanatory.

```
with fea_control; use fea_control;
Package SWP is
-- Sliding Window Protocol Implementation 2: two timers
        max_msg: constant integer := 100;
      -- messages are integers in the range 1..max_msg;
      -- sent in ascending order.
      window_size: constant integer := 2; -- max window size;
      -- window_size is adjustable, but depends on seq_no range.
      max_seq_no: constant integer := 6;  -- max sequence number
      min_seq_no: constant integer := 1;  -- min sequence number
      -- min_seq_no must be 1; max_seq_no is adjustable;
      subtype msg_type is integer range 1..max_msg;
      subtype seq_no_type is integer range min_seq_no..max_seq_no;
      procedure get_next_msg(m:in msg_type; myid: integer);
      -- Called by Client_S to send a message to Client_R
      procedure deliver_msg(m: in out msg_type; s: in out seq_no_type;
myid: integer);
      -- Called by Client_R to receive message from Client_S
end SWP;

with fea_control; use fea_control;
with text_io; use text_io;
Package body SWP is
      subtype window_type is integer range 0..window_size-1;
      subtype acked_seq_no_type is integer range 0..max_seq_no;
      trap_flag:boolean:=false;

      task Medium_SR is
      entry ttgetid(x: in TASK_ID);
            entry send(m:msg_type; s:seq_no_type; caller_id: integer);
      end Medium_SR;

      task Medium_RS is
      entry ttgetid(x: in TASK_ID);
            entry reply(a:acked_seq_no_type; caller_id: integer);
      end Medium_RS;
      task Sender is
      entry ttgetid(x: in TASK_ID);
            entry next_msg(m:msg_type; caller_id: integer);
            entry ack(a_no:acked_seq_no_type; caller_id: integer);
            entry timeout(t_no:window_type; caller_id: integer);
      end Sender;

      task Receiver is
      entry ttgetid(x: in TASK_ID);
            entry msgs(m:msg_type;s:seq_no_type; caller_id: integer);
            entry deliver(m:in out msg_type; s:in out seq_no_type;
caller_id: integer);
      end Receiver;

      task type timer_task_type is
      entry ttgetid(x: in TASK_ID);
            entry time(caller_id: integer);
            entry cancel(caller_id: integer);
            entry get_timer_no(t_no:window_type; caller_id: integer);
      end timer_task_type;

      Timer: array(0..1) of timer_task_type;

      procedure get_next_msg(m:in msg_type; myid: integer) is
      begin
      control.req_call_permit(myid)(4,"next_msg                 ");
            Sender.next_msg(m,myid);
      end get_next_msg;

      procedure deliver_msg(m:in out msg_type; s: in out seq_no_type;
myid: integer) is
      begin
      control.req_call_permit(myid)(5,"deliver                  ");
            Receiver.deliver(m,s,myid);
      end deliver_msg;

      function succ(x: in seq_no_type) return seq_no_type is -- value
in 1..6
      -- 1->2, 2->3, 3->4, 4->5, 5->6, 6->1
            t:integer range 0..max_seq_no;
      begin -- succ
            t:=(x+1) mod max_seq_no;
            if t=0 then
              return(max_seq_no);
            else return(t);
            end if;
      end succ;

      function pred(x: in seq_no_type) return seq_no_type is -- value
in 1..6
      -- 6->5, 5->4, 4->3, 3->2, 2->1, 1->6
            t:integer range 0..max_seq_no;
      begin --pred
```

```
            t:=(x-1) mod max_seq_no;
            if t=0 then return(max_seq_no); else return(t);
            end if;
      end pred;

      task body timer_task_type is
      myid: TASK_ID;
      arrivals_needed: INTEGER;
      saved_count: INTEGER;
      current_arrivals: INTEGER;
      call_successful: BOOLEAN;
      one_arrival: BOOLEAN;
      event_label: entry_name_type;

        timer_no:window_type;
        D: duration := 0.001;
      begin
      accept ttgetid(x: in TASK_ID) do
        myid := x;
      end;
      control.req_accept_permit(myid)("get_timer_no             ");
        accept get_timer_no(t_no:window_type; caller_id: integer) do
      if t_no=0 then
        event_label:= "get_timer_no0            ";
      else
        event_label:= "get_timer_no1            ";
      end if;
      control.accepted(myid)(caller_id,"get_timer_no             ",
                        event_label);
          timer_no := t_no;
      control.req_finish_accept(myid)(caller_id,"get_timer_no
");
        end get_timer_no;
        loop
      control.req_select_permit(myid)(one_arrival);
      if one_arrival = TRUE then
      current_arrivals := time'count + cancel'count +
get_timer_no'count;
      arrivals_needed := current_arrivals + 1;
      control.start_a_sequence;
      while current_arrivals < arrivals_needed loop
        delay 0.1;
      current_arrivals := time'count + cancel'count +
get_timer_no'count;
      end loop;
end if;
          select
            accept time(caller_id: integer) do
      if timer_no=0 then
        event_label:= "time0                    ";
      else
        event_label:= "time1                    ";
      end if;
      control.accept_selected(myid)(caller_id,"time
",
                              event_label);
      control.req_finish_select_accept(myid)(caller_id,"time
");
      end;
      control.req_select_permit(myid)(one_arrival);
      if one_arrival = TRUE then
      current_arrivals := time'count + cancel'count +
get_timer_no'count;
      arrivals_needed := current_arrivals + 1;
      control.start_a_sequence;
      while current_arrivals < arrivals_needed loop
        delay 0.1;
      current_arrivals := time'count + cancel'count +
get_timer_no'count;
      end loop;
end if;
            select
               accept cancel(caller_id: integer) do
      if timer_no=0 then
        event_label:= "cancel0                 ";
      else
        event_label:= "cancel1                 ";
      end if;
      control.accept_selected(myid)(caller_id,"cancel
",
                              event_label);
      control.req_finish_select_accept(myid)(caller_id,"cancel
");
      end;
            or delay D;
        event_label:= "tau                     ";
      control.delay_selected(myid)(event_label);
               loop
        event_label:= "tau                     ";
      control.req_cond_timed_permit(myid)(call_successful,"timeout
",4,event_label);
      if call_successful then
      control.req_call_permit(myid)(4,"timeout                 ");
                 Sender.timeout(timer_no,myid);
                exit;
               else
      control.req_select_permit(myid)(one_arrival);
      if one_arrival = TRUE then
      current_arrivals := time'count + cancel'count +
get_timer_no'count;
      arrivals_needed := current_arrivals + 1;
      control.start_a_sequence;
      while current_arrivals < arrivals_needed loop
        delay 0.1;
      current_arrivals := time'count + cancel'count +
get_timer_no'count;
```

```
      end loop;
end if;
                   select
                   accept cancel(caller_id: integer) do
      if timer_no=0 then
        event_label:= "cancel0                  ";
      else
        event_label:= "cancel1                  ";
      end if;
      control.accept_selected(myid)(caller_id,"cancel
",
                              event_label);
      control.req_finish_select_accept(myid)(caller_id,"cancel
");
      end;
                   exit;
                   else
        event_label:= "tau                      ";
      control.else_selected(myid)(event_label);
                   null;
      control.req_finish_else(myid);
                   end select;
end if;
              end loop;
      control.req_finish_delay(myid);
            end select;
           or
              terminate;
           end select;
         end loop;
      end timer_task_type;

      task body Medium_SR is
      myid: TASK_ID;
      arrivals_needed: INTEGER;
      saved_count: INTEGER;
      current_arrivals: INTEGER;
      call_successful: BOOLEAN;
      one_arrival: BOOLEAN;
      event_label: entry_name_type;

            msg: msg_type;
            seq_no: seq_no_type;
            D: duration := 0.0001;
      begin
      accept ttgetid(x: in TASK_ID) do
        myid := x;
      end;
        loop
      control.req_select_permit(myid)(one_arrival);
      if one_arrival = TRUE then
      current_arrivals := send'count;
      arrivals_needed := current_arrivals + 1;
      control.start_a_sequence;
      while current_arrivals < arrivals_needed loop
        delay 0.1;
      current_arrivals := send'count;
      end loop;
end if;
         select
          accept send(m:msg_type; s:seq_no_type; caller_id: integer)
do
      if s=1 then
        event_label:= "send1                    ";
      elsif s=2 then
        event_label:= "send2                    ";
      elsif s=3 then
        event_label:= "send3                    ";
      elsif s=4 then
        event_label:= "send4                    ";
      elsif s=5 then
        event_label:= "send5                    ";
      elsif s=6 then
        event_label:= "send6                    ";
      end if;
      control.accept_selected(myid)(caller_id,"send
",
                        event_label);
            msg := m; seq_no := s;
      control.req_finish_select_accept(myid)(caller_id,"send
");
          end send;
        event_label:= "tau                      ";
      control.req_cond_timed_permit(myid)(call_successful,"msgs
",5,event_label);
      if call_successful then
      control.req_call_permit(myid)(5,"msgs                     ");
            Receiver.msgs(msg,seq_no,myid);
      else
          --or
            --delay D;
            --put_line("lm");
            null;  -- loose it
      end if;
         or
          terminate;
         end select;
        end loop;
      end Medium_SR;

      task body Medium_RS is
      myid: TASK_ID;
      arrivals_needed: INTEGER;
      saved_count: INTEGER;
      current_arrivals: INTEGER;
      call_successful: BOOLEAN;
      one_arrival: BOOLEAN;
      event_label: entry_name_type;

            acked_seq_no: acked_seq_no_type;
            D: duration := 0.001;
      begin
      accept ttgetid(x: in TASK_ID) do
        myid := x;
      end;
        loop
      control.req_select_permit(myid)(one_arrival);
      if one_arrival = TRUE then
      current_arrivals := reply'count;
      arrivals_needed := current_arrivals + 1;
      control.start_a_sequence;
      while current_arrivals < arrivals_needed loop
        delay 0.1;
      current_arrivals := reply'count;
      end loop;
end if;
         select
          accept reply(a:acked_seq_no_type; caller_id: integer) do
      if a=0 then
        event_label:= "rep0                     ";
      elsif a=1 then
        event_label:= "rep1                     ";
      elsif a=2 then
        event_label:= "rep2                     ";
      elsif a=3 then
        event_label:= "rep3                     ";
      elsif a=4 then
        event_label:= "rep4                     ";
      elsif a=5 then
        event_label:= "rep5                     ";
      elsif a=6 then
        event_label:= "rep6                     ";
      end if;
      control.accept_selected(myid)(caller_id,"reply
",
                                event_label);
            acked_seq_no := a;
      control.req_finish_select_accept(myid)(caller_id,"reply
");
          end reply;
        event_label:= "tau                      ";
      control.req_cond_timed_permit(myid)(call_successful,"ack
",4,event_label);
      if call_successful then
      control.req_call_permit(myid)(4,"ack                      ");
            Sender.ack(acked_seq_no,myid);
      else
            -- put_line("la");
            null;  -- loose it
      end if;
        or
          terminate;
        end select;
        end loop;
      end Medium_RS;

task body Sender is
      myid: TASK_ID;
      arrivals_needed: INTEGER;
      saved_count: INTEGER;
      current_arrivals: INTEGER;
      call_successful: BOOLEAN;
      one_arrival: BOOLEAN;
      event_label: entry_name_type;
      h_s:seq_no_type; -- added for validty calculations

      subtype num_unacked_msgs_type is integer range 0..window_size;
      num_unacked_msgs:num_unacked_msgs_type:=0; -- unacknowledged
messages;
      lowest_unacked:seq_no_type:= min_seq_no; -- lowest msg sent and
unacked
      highest_sent:seq_no_type:= max_seq_no;    -- highest msg sent
and unacked
      msg:msg_type;                        -- message to be sent
      timer_no:window_type;         -- timed out timer:
0..window_size-1
      acked_seq_no:acked_seq_no_type;
      -- acknowledged sequence number 0..max_seq_no
      seq_no:seq_no_type;
      upper:seq_no_type;
      type msg_record_type is record
         msg:msg_type;seq_no:seq_no_type;
      end record;
      type store_type is array(window_type) of msg_record_type;
      msg_store:store_type;

      function compute_num_unacked_msgs(highest_sent:seq_no_type;
         lowest_unacked:seq_no_type)
       return  num_unacked_msgs_type is
      begin -- compute_num_unacked_msgs
            if highest_sent = pred(lowest_unacked) then
                  return(0);
            elsif highest_sent < lowest_unacked then  -- wrapped
                  return ((highest_sent+max_seq_no)-lowest_unacked+1);
            else return(highest_sent-lowest_unacked+1);     -- not
wrapped
            end if;
      end compute_num_unacked_msgs;

      function is_unacked_msg(acked_seq_no:acked_seq_no_type;
            highest_sent:seq_no_type;lowest_unacked:seq_no_type)
            return Boolean is
      begin -- is_unacked_msg
            if highest_sent = pred(lowest_unacked) then -- no unacked
msgs
                  return(false);
            elsif highest_sent < lowest_unacked then    -- wrapped
                  return(((lowest_unacked<=acked_seq_no) and
                        (acked_seq_no<=max_seq_no))
                        or ((min_seq_no<=acked_seq_no) and
                            (acked_seq_no<=highest_sent)));
            else
                  return((lowest_unacked<=acked_seq_no) and
                        (acked_seq_no<=highest_sent)); -- ~wrapped
            end if;
      end is_unacked_msg;

      Package Store_Messages is
            procedure store(msg_store:in out store_type;msg:msg_type;
                        seq_no:seq_no_type);
            -- store msg and seq_no for possible resending in
            -- position seq_no mod window_size.
            function
retrieve_msg(msg_store:store_type;position:window_type)
                  return msg_type;
            -- retrieve message stored in position position.
            function retrieve_seq_no(msg_store:store_type;
                  position:window_type) return seq_no_type;
            -- retrieve seq_no stored in position position.
      end Store_Messages;

      Package body Store_Messages is
            procedure store(msg_store:in out store_type;msg:msg_type;
                        seq_no:seq_no_type) is
            -- store msg and seq_no for possible resending in
            -- position seq_no mod window_size.
            begin
              msg_store(seq_no mod window_size).msg := msg;
              msg_store(seq_no mod window_size).seq_no := seq_no;
            end store;

            function
retrieve_msg(msg_store:store_type;position:window_type)
                  return msg_type is
            -- retrieve message stored in position position.
            begin
              return(msg_store(position).msg);
            end retrieve_msg;

            function retrieve_seq_no(msg_store:store_type;
                  position:window_type) return seq_no_type is
            -- retrieve seq_no stored in position position.
            begin
              return(msg_store(position).seq_no);
            end retrieve_seq_no;
      end Store_Messages;
      use Store_Messages;

begin
      accept ttgetid(x: in TASK_ID) do
        myid := x;
      end;
   for i in window_type loop
      control.req_call_permit(myid)(6+i,"get_timer_no             ");
      timer(i).get_timer_no(i,myid);
   end loop;
   loop
     num_unacked_msgs :=
compute_num_unacked_msgs(highest_sent,lowest_unacked);
      control.req_select_permit(myid)(one_arrival);
      if one_arrival = TRUE then
      current_arrivals := next_msg'count + ack'count + timeout'count;
      arrivals_needed := current_arrivals + 1;
      control.start_a_sequence;
      while current_arrivals < arrivals_needed loop
        delay 0.1;
      current_arrivals := next_msg'count + ack'count + timeout'count;
      end loop;
end if;
     select
      when num_unacked_msgs < window_size
        => accept next_msg(m:msg_type; caller_id: integer) do
      h_s := succ(highest_sent);
      if h_s =1 then
        event_label:= "next_msg1                ";
      elsif h_s =2 then
        event_label:= "next_msg2                ";
      elsif h_s =3 then
        event_label:= "next_msg3                ";
      elsif h_s =4 then
        event_label:= "next_msg4                ";
      elsif h_s =5 then
        event_label:= "next_msg5                ";
      elsif h_s =6 then
        event_label:= "next_msg6                ";
      end if;
      control.accept_selected(myid)(caller_id,"next_msg
",
                              event_label);
            msg := m;
      control.req_finish_select_accept(myid)(caller_id,"next_msg
");
           end next_msg;
      highest_sent := succ(highest_sent);       -- value in
1..max_seq_no
      store(msg_store,msg,highest_sent);  -- save msg and its seq_no
      control.req_call_permit(myid)(2,"send                     ");
      Medium_SR.send(msg,highest_sent,myid);    -- send msg and seq_no
```

```
        timer_no := highest_sent mod window_size;
      control.req_call_permit(myid)(6+timer_no,"time
");
      Timer(timer_no).Time(myid);   -- start timer
     or
      accept timeout(t_no:window_type; caller_id: integer) do
      if t_no =0 then
        event_label:= "timeout0                ";
      else
        event_label:= "timeout1                ";
      end if;
      control.accept_selected(myid)(caller_id,"timeout
",
                              event_label);
            timer_no := t_no;  -- value in 0 .. window_size-1
      control.req_finish_select_accept(myid)(caller_id,"timeout
");
      end timeout;
      msg := retrieve_msg(msg_store,timer_no);
      seq_no := retrieve_seq_no(msg_store,timer_no);
      control.req_call_permit(myid)(2,"send                   ");
      Medium_SR.send(msg,seq_no,myid); -- resend msg and seq_no
      control.req_call_permit(myid)(6+timer_no,"time
");
      Timer(timer_no).time(myid); -- restart timer
      -- (no need to cancel timer first, as timer timed out)
      -- resend all unacked messages sent after timed out message, if
any.
      seq_no := succ(seq_no);       -- seq_no of next msg to resend,
if any
      upper := succ(highest_sent); -- seq_no of succe of last msg to
be resent
      while seq_no /= upper loop
        msg := retrieve_msg(msg_store,seq_no mod window_size);
      control.req_call_permit(myid)(2,"send                   ");
        Medium_SR.send(msg,seq_no,myid); -- resend msg and seq_no
        timer_no := seq_no mod window_size;
      control.req_call_permit(myid)(6+timer_no,"cancel
");
        Timer(timer_no).cancel(myid);     -- cancel timer
      control.req_call_permit(myid)(6+timer_no,"time
");
        Timer(timer_no).time(myid);             -- restart timer
        seq_no := succ(seq_no);
      end loop;
     or
      accept ack(a_no:acked_seq_no_type; caller_id: integer) do --
accept acknowledgement
      if a_no=0 then
        event_label:= "ack0                    ";
      elsif a_no=1 then
        event_label:= "ack1                    ";
      elsif a_no=2 then
```

```
        event_label:= "ack2                    ";
      elsif a_no=3 then
        event_label:= "ack3                    ";
      elsif a_no=4 then
        event_label:= "ack4                    ";
      elsif a_no=5 then
        event_label:= "ack5                    ";
      elsif a_no=6 then
        event_label:= "ack6                    ";
      end if;
      control.accept_selected(myid)(caller_id,"ack
",
                              event_label);
            acked_seq_no:=a_no;
      control.req_finish_select_accept(myid)(caller_id,"ack
");
      end ack;
      if is_unacked_msg(acked_seq_no,highest_sent,lowest_unacked) then
        -- cancel timers and move window
        timer_no := lowest_unacked mod window_size;
      control.req_call_permit(myid)(6+timer_no,"cancel
");
        Timer(timer_no).cancel(myid);
        lowest_unacked := succ(lowest_unacked); -- value in
1..max_seq_no
        while lowest_unacked /= succ(acked_seq_no) loop
           timer_no := lowest_unacked mod window_size;
      control.req_call_permit(myid)(6+timer_no,"cancel
");
           Timer(timer_no).cancel(myid);
           lowest_unacked := succ(lowest_unacked); -- value in
1..max_seq_no
        end loop;
      end if; -- else ignore ack
    or
        terminate;
     end select;
  end loop;
end Sender;

task body Receiver is
      myid: TASK_ID;
      arrivals_needed: INTEGER;
      saved_count: INTEGER;
      current_arrivals: INTEGER;
      call_successful: BOOLEAN;
      one_arrival: BOOLEAN;

      next_required:seq_no_type:= min_seq_no;
      highest_received:seq_no_type:= max_seq_no;
      already_received : array(window_type) of boolean := (others =>
false);
      seq_no:seq_no_type;
      event_label: entry_name_type;
      msg:msg_type;
        received_first_one:boolean:=false;
      type msg_record_type is record
         msg:msg_type;seq_no:seq_no_type;
      end record;
      type store_type is array(window_type) of msg_record_type;
      msg_store:store_type;

      function in_window(seq_no:seq_no_type;next_required:seq_no_type)
            return boolean is
            upper:integer range 0..max_seq_no;
      begin
            upper := (next_required + window_size - 1) mod max_seq_no;
            if upper = 0 then upper := max_seq_no; end if;
            -- window = next_required .. upper
            if upper < next_required then -- wrapped
              return(((next_required<=seq_no) and
(seq_no<=max_seq_no))
                  or ((min_seq_no<=seq_no) and (seq_no<=upper)));
            else -- not wrapped
              return((next_required<=seq_no) and (seq_no<=upper));
            end if;
      end in_window;

function max(seq_no:seq_no_type;next_required:seq_no_type;
      highest_received:seq_no_type) return seq_no_type is
-- returns max of two seq_no's in the window
      upper:integer range 0..max_seq_no;
begin
      upper := (next_required + window_size - 1) mod max_seq_no;
      if upper = 0 then upper := max_seq_no; end if;
      -- window = next_required .. upper
      if highest_received = pred(next_required) then -- empty window
            return(seq_no); -- max by default
      elsif upper > next_required then
      -- not wrapped: next_required..upper..max_seq_no
        if seq_no > highest_received then
          return(seq_no);
        else
          return(highest_received);
        end if;
--                     | ---------- A -------|    |------- B -----|
      else -- wrapped:  next_required....max_seq_no min_seq_no...upper
            if ((min_seq_no<=seq_no) and (seq_no<=upper)) and
            ((next_required<=highest_received) and
            (highest_received<=max_seq_no)) then
            -- 1: seq_no in B, highest_received in A
              return(seq_no);
            elsif ((min_seq_no<=seq_no) and (seq_no<=upper)) and
            ((min_seq_no<=highest_received) and
            (highest_received<=upper)) then
             -- 2: seq_no in B, highest_received in B
             if seq_no > highest_received then
              return(seq_no);
             else
              return(highest_received);
             end if;
            elsif ((next_required<=seq_no) and (seq_no<=max_seq_no))
            and ((next_required<=highest_received) and
                   (highest_received<=max_seq_no)) then
             -- 3: seq_no in A, highest_received in A
             if seq_no > highest_received then
               return(seq_no);
             else
               return(highest_received);
             end if;
            elsif ((min_seq_no<=highest_received) and
                  (highest_received<=upper)) and
            ((next_required<=seq_no) and (seq_no<=max_seq_no)) then
             -- 4: seq_no in A, highest_received in B
             return(highest_received);
--          else
--          bail out!!
            end if;
      end if;
end max;

      Package Store_Messages is
            procedure store(msg_store:in out store_type;msg:msg_type;
                        seq_no:seq_no_type);
            -- store msg and seq_no for possible resending in
            -- position seq_no mod window_size.
            function
retrieve_msg(msg_store:store_type;position:window_type)
                  return msg_type;
            -- retrieve message stored in position position.
            function retrieve_seq_no(msg_store:store_type;
                  position:window_type) return seq_no_type;
            -- retrieve seq_no stored in position position.
      end Store_Messages;

      Package body Store_Messages is
            procedure store(msg_store:in out store_type;msg:msg_type;
                        seq_no:seq_no_type) is
            -- store msg and seq_no for possible resending in
            -- position seq_no mod window_size.
            begin
              msg_store(seq_no mod window_size).msg := msg;
              msg_store(seq_no mod window_size).seq_no := seq_no;
            end store;

            function
retrieve_msg(msg_store:store_type;position:window_type)
                  return msg_type is
            -- retrieve message stored in position position.
            begin
              return(msg_store(position).msg);
            end retrieve_msg;

            function retrieve_seq_no(msg_store:store_type;
                  position:window_type) return seq_no_type is
            -- retrieve seq_no stored in position position.
            begin
              return(msg_store(position).seq_no);
            end retrieve_seq_no;
      end Store_Messages;
      use Store_Messages;

begin
      accept ttgetid(x: in TASK_ID) do
        myid := x;
      end;
  loop
      control.req_select_permit(myid)(one_arrival);
      if one_arrival = TRUE then
      current_arrivals := msgs'count + deliver'count;
      arrivals_needed := current_arrivals + 1;
      control.start_a_sequence;
      while current_arrivals < arrivals_needed loop
        delay 0.1;
      current_arrivals := msgs'count + deliver'count;
      end loop;
end if;
    select
      accept msgs(m:msg_type;s:seq_no_type; caller_id: integer) do
      if s=1 then
        event_label:= "msg1                     ";
      elsif s=2 then
        event_label:= "msg2                     ";
      elsif s=3 then
        event_label:= "msg3                     ";
      elsif s=4 then
        event_label:= "msg4                     ";
      elsif s=5 then
        event_label:= "msg5                     ";
      elsif s=6 then
        event_label:= "msg6                     ";
      end if;
      control.accept_selected(myid)(caller_id,"msgs
",
                              event_label);
        msg := m; seq_no := s;
      control.req_finish_select_accept(myid)(caller_id,"msgs
");
      end msgs;
      if in_window(seq_no,next_required) and
      not(already_received(seq_no mod window_size)) then
          if not(received_first_one) and seq_no=1 then
          received_first_one := true;
          end if;
        store(msg_store,msg,seq_no); -- save msg for later delivery to
user
        already_received(seq_no mod window_size) := true; -- note
reception
        highest_received :=
max(seq_no,next_required,highest_received);
        while already_received(next_required mod window_size) and
            next_required /= succ(highest_received) loop
      --    deliver(retrieve(msg_store,next_required mod
window_size));
            -- deliver msg
      control.req_accept_permit(myid)("deliver                  ");
            accept deliver(m:in out msg_type; s: in out seq_no_type;
caller_id: integer) do
      if next_required=1 then
        event_label:= "deliver1                 ";
      elsif next_required=2 then
        event_label:= "deliver2                 ";
      elsif next_required=3 then
        event_label:= "deliver3                 ";
      elsif next_required=4 then
        event_label:= "deliver4                 ";
      elsif next_required=5 then
        event_label:= "deliver5                 ";
      elsif next_required=6 then
        event_label:= "deliver6                 ";
      end if;
      control.accepted(myid)(caller_id,"deliver                 ",
                            event_label);
              m:=retrieve_msg(msg_store,next_required mod
window_size);
              s:=next_required;
      control.req_finish_accept(myid)(caller_id,"deliver
");
            end deliver;
            already_received(next_required mod window_size) := false;
            next_required := succ(next_required);
        end loop;
      end if;
        delay 0.001;
        if received_first_one then
      control.req_call_permit(myid)(3,"reply                    ");
        Medium_RS.reply(pred(next_required),myid);
        else
      control.req_call_permit(myid)(3,"reply                    ");
          Medium_RS.reply(0,myid);
        end if;
      -- return ack0 until get first msg1, 1..6 thereafter
     or
      terminate;
     end select;
```

```
  end loop;
end Receiver;

-- Notes:
-- return Ack0 until receive msg1, then 1..6 thereafter.
-- may move compute to end of select so can derive a[# ;-> ?in]
-- make explicit the assumption about max_seq_no and window_size
(Holzmann)

      begin
medium_sr.ttgetid(2);
medium_rs.ttgetid(3);
sender.ttgetid(4);
receiver.ttgetid(5);
timer(0).ttgetid(6);
timer(1).ttgetid(7);
end SWP;

with fea_control; use fea_control;
with SWP; use SWP;
with text_io; use text_io;
procedure SWP_Test is
 max_msgs: constant integer := 30;

 task Client_S is
      entry ttgetid(x: in TASK_ID);
 end;
 task Client_R is
      entry ttgetid(x: in TASK_ID);
 end;

 task body Client_S is
      myid: TASK_ID;
      arrivals_needed: INTEGER;
      saved_count: INTEGER;
      current_arrivals: INTEGER;
      call_successful: BOOLEAN;
      one_arrival: BOOLEAN;

 begin
      accept ttgetid(x: in TASK_ID) do
        myid := x;
      end;
   for i in 1..max_msgs loop
      get_next_msg(i,myid);
   end loop;
 end Client_S;

 task body Client_R is
      myid: TASK_ID;
      arrivals_needed: INTEGER;
      saved_count: INTEGER;
      current_arrivals: INTEGER;
      call_successful: BOOLEAN;
      one_arrival: BOOLEAN;

  m:msg_type;
  seq_no:seq_no_type;
  seq_check:integer range 0..max_seq_no;
  package my_integer_io is new integer_io(integer);
  use my_integer_io;
 begin
      accept ttgetid(x: in TASK_ID) do
        myid := x;
      end;
  for i in 1..max_msgs loop
       deliver_msg(m,seq_no,myid);
       put(m);put("  ");put(seq_no);put_line("  ");
       seq_check := i mod max_seq_no;
       if seq_check = 0 then
        seq_check := max_seq_no;
       end if;
       if m /= i or seq_no /= seq_check then
        put_line("infeasible: invalid output");
       end if;
  end loop;
 end Client_R;
begin
  pragma main;
client_s.ttgetid(8);
client_r.ttgetid(9);
  null;
      end SWP_Test
```

REFERENCES

[1] K. C. Tai, *“Testing of Concurrent Software”,* Computer Software and Applications Conference, 1989, COMPSAC 89, Proc. of the 13th Annual International, 20-22 Sep 1989, pp. 62-64.

[2] Yu Lei and K. C. Tai, *“Efficient Reachability Testing of Asynchronous Message-Passing Programs”,* Proc. of the 8th IEEE International Conference on Engineering for Complex Computer Systems, Dec. 2002, pp. 35-44.

[3] Tetsuro Katayaam, Eisuke Itoh, Zengo Furukawa, Kazuo Ushijima, “*Test-cse Ganeration for Concurret Programs with the Testing Criteria Using Interaction Sequence”* Proceedings of Asia Pacific Software Engineering Conference'99 (APSEC'99), pp.590-597, 1999.

[4] Pichate Pluempatanakij, “*Improving non-deterministic testing of message-passing programs*”; MS Thesis, UTA, 2005.

[5] R. Carver and K.C. Tai, "*Deterministic Execution Testing of Concurrent Ada Programs*," Proceedings of TRI-Ada '89 -- Ada Technology In Context: Application, Development, and Deployment, October 23-26, 1989, Association for Computing Machinery, New York, New York, pp. 528 - 544.

[6] Richard H. Carver, “*Mutation-Based Testing of Concurrent Programs.*” ITC 1993: 845-853.

[7] DeMillo, R. A, Lipton, R. J., and Sayward, F.G., "*Hints on test data selection: help for the practicing programmer,*" IEEE Computer, 11(4), 1978, pp. 34-41.

[8] King, K. N., and Offutt, A. J., "*A Fortran Language System for Mutation-Based Software Testing,*" Software-Practice and Experience, Vol. 21(7), July 1991, pp. 685-718.

[9] Tai, K. C., "*On Testing Concurrent Programs*," Proc. of COMPSAC 85.

[10] Tai, K C., and Carver, R. H., "*Testing and debugging of concurrent software by deterministic execution*", TR-87- 19, Dept. of Computer Science, North Carolina State University, 1987.

[11] Jian Chen, Richard H. Carver, "*Selecting and mapping test sequences from formal specifications of concurrent programs*" HASE 1996.

[12] K. C. Tai, R. H. Carver, and E. E. Obaid. "*Debugging concurrent Ada programs by deterministic execution.*", IEEE Trans. Sofi. Eng., 17(1):45-63, Jan. 1991.

[13] K. C. Tai and R. H. Carver, "*Testing of Distributed Programs, Parallel and Distributed Computing Handbook*", 1996.

[14]A. J. Offutt, C. Z. Rothermel, and C. Zapf. "*An experimental evaluation of selective mutation.",* In Proc. International Conference on Software Engineering, 1993.

[15] Course-pack, Concurrent Programming , CSE6323, UTA, 2004.

[16] Myers, G. J., "*The Art of Software Testing*". John Wiley and Sons, IBM Systems Research Institute, Myers 1979.

[17] J. Gait, "*A probe effect in concurrent programs*", Software-Practice and Experience, Vol. 16, Issue 3, 1986, pp. 255-233.

[18] Ledoux, C.H., and D. Stott Parker. 1985, "*Saving traces for Ada debugging*". 1985 International Ada conference, Cambridge University Press, 97-108.

[19] K. C. Tai, "*Testing of Concurrent Software*", Computer Software and Applications Conference, 1989, COMPSAC 89, Proc. of the 13th Annual International, 20-22 Sep 1989, pp. 62-64.

[20] B. Beizer, "*Software Testing Techniques*", second edition, Van Nostrand Reinhold, New York, 1990.

[21] Cheer-Sun D. Yang, Amie L. Souter and Lori L. Pollock, "*All-du-path Coverage for Parallel Programs*", International Symposium on Software Testing and Analysis, Proc. of the 1998 ACM SIGSOFT Software Testing and Analysis, 1998, pp. 153-162.

[22] M. Pezze, R. N. Taylor and M. Young, "*Graph Models for Reachability of Concurrent Programs*", ACM Trans. On Software Engineering and Methodology, Vol. 4, Issue 2, April 1995, pp. 171-213.

[23] R. Carver and K. C. Tai, "*Replay and Testing for Concurrent Programs*", IEEE Software, Vol 8., No. 2, March 1991, pp. 66-74.

[24] Jong-Deok Choi and Harini Srinivasan, "*Deterministic Replay of Java Multithreaded Applications*", Proc. of the SIGMETRICS symposium on Parallel and distributed tools, August 03-04, 1998, pp. 48-59.

[25] R. H. B. Netzer, B. P. Miler; "*Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs*", Proc. of the 1992 ACM/IEEE conference on Supercomputing, December 1992, pp. 502-511.

[26] Michiel Ronsse and Koen De Bosschere, "*RecPlay: A Fully Integrated Practical Record/Replay System*", ACM Transactions on Computer Systems, Vol. 17, No2, 1999, pp. 133-152.

[27] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski, "*jRapture: A Capture/Replay Tool for Observation-based Testing*", Proc. of the International Symposium on Software Testing and Analysis, 2000, pp. 158-167.

[28] P. V. Koppol and K. C. Tai, "*An Incremental Approach to Structural Testing of Concurrent Software*", Proc of ACM International Symposium on Software Testing and Analysis, 1996, pp. 14-23.

[29] M. Pezze, R. N. Taylor and M. Young, "*Graph Models for Reachability of Concurrent Programs*", ACM Trans. On Software Engineering and Methodology, Vol. 4, Issue 2, April 1995, pp. 171-213.

[30] S. D. Stoller, "Testing *Concurrent Java Programs using Randomized Scheduling*", Proc. of the Second Workshop on Runtime Verification (RV), Vol. 70, Issue 4 of Electronic Notes in Theoretical Computer Science. Elsevier, 2002.

[31] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur., "*Multithread Java Program Test Generation*", IBM Systems Journal, Vol. 41, Issue 1, 2002, pp. 111-125.

[32] K. C. Tai, "*Race Analysis of Traces of Asynchronous Message-Passing Programs*", Proc. of 17th International Conference on Distributed Computing Systems, May 1997, pp. 261-268.

[33] C. Yang and L. L. Pollock, "*Identifying Redundant Test Cases for Testing Parallel Language Constructs*", ARL-ATIRP First Annual Technical Conference, 1997.

[34] M. Weiser, "*Program Slicing*", IEEE Transaction on Software Engineering, Vol. 10, Issue 4, 1984, pp. 352-357.

[35] Netzer, Robert H.B., and Barton P. Miller. 1992. "*What are race conditions? Some issues and formalizations.*", ACM Letters on Programming Languages and Systems(LOPLAS) Vol.1, Issue 1, 74-88.

[36] Taylor, R.N.; Levine, D.L.; Kelly, C.D., "*Structural testing of concurrent programs Software Engineering"*, IEEE Transactions, Volume: 18 , Issue: 3 , March 1992 Pages:206 – 215.

[37] Sommerville, "Concepts *of Software Engineering*," 1985.

[38] Yang, R.-D. , Chung, C.-G., "*A path analysis approach to concurrent program testing",* Computers and Communications, 1990. Conference Proceedings., Ninth Annual International Phoenix Conference on , Pages:425 – 432, 21-23 March 1990.

[39] Clarke, L.A., Podgurski, A., Richardson, D.J., & Zeil, S.J., "*A formal Evaluation of data flow path selection criteria*," IEEE Transactions on Software Engineering, Vol.15, No.11, pp1318-1332, November 1989.

[40] Frankl, P.G. & Weyuker, J.E., "*A formal analysis of the fault-detecting ability of testing methods*", IEEE Transactions on Software Engineering, Vol. 19, No. 3, March 1993, pp202- 213.

[41] Zhu, H., "*A formal analysis of the subsumes relation between software test adequacy criteria*", Technical Report No. 94/18, Department of Computing, The Open University, U.K., August, 1994.

[42] Duran, J.W. & Ntafos, S., "*An evaluation of random testing, IEEE Transaction on Software Engineering",* Vol. SE_10, No. 4, pp438-444, July 1984.

[43] Hamlet, D. and Taylor, R., "*Partition testing does not inspire confidence*", IEEE Transactions on Software Engineering, Vol. 16, pp206~215, Dec. 1990.

[44] Basili, V.R., and Selby, R.W., "*Comparing the effectiveness of software testing*", IEEE Transactions on Software Engineering, Vol. SE-13, No.12, pp1278-1296, December 1987.

[45] Box, G.E.P., Hunter, W.G. and Hunter, J.S., "*Statistics for Experimenters*", New York, Wiley, 1978.

[46] Hamlet, R., "*Theoretical comparison of testing methods*", Proceedings of the Third Symposium on Software Testing, Analysis and Verification, Florida, December 13-15, 1989, pp28-37.

[47] Hong Zhu, "*Adequate Testing of Computer Software*", Textbook Oxford Brookes University, August 1995, Chapter 8-9.

BIOGRAPHICAL INFORMATION

Gaurav Saini was born December 5th, 1979 in Jaipur, India. He received his Bachelor of Engineering degree in Computer Science and Engineering from Rajasthan University, Rajasthan, India in July 2003. In fall 2003 he started his graduate studies in Computer Science at the University of Texas at Arlington. He received his Master of Science from University of Texas at Arlington in July 2005. His research interests include Software Engineering and Concurrent Programming.