To the Graduate Council·

I am submitting herewith a thesis written by David Edward Pearson entitled "Generalized Testing Chains " I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Dr Jesse H Poore, Major Professor

We have read this thesis
and recommend its acceptance

Accepted for the Council

Interim Vice Provost and
Dean of the Graduate School

GENERALIZED TESTING CHAINS

A Thesis

Presented for the

Masters of Science Degree

The University of Tennessee, Knoxville

David Edward Pearson

December 2000

Dedication

To Rena, Emma, and the one on the way

Acknowledgments

I would like to thank my advisor, Dr Jesse Poore for all of the guidance, support, learning opportunities, and advice he has provided in both an academic and professional environment.

I want to thank Dr Stacy Prowell and Dr Robert Ward for serving on my committee and reviewing this thesis

I would also like to thank all of the former employees of Software Engineering Technology who served as a sounding board as I worked through several of the items discussed in this thesis, with special gratitude to Earl Billingsley Furthermore, I would like to express my gratitude to SET, Q-Labs, and Video Networks for supporting my education while continuing to work for the respective companies on a full time basis.

Finally, I would like to thank my family Thanks to my parents for all of their support Most of all, thanks to Rena for her continual support, patience and love

Abstract

This thesis discusses techniques for generalizing the usage of the testing chain in the Cleanroom Certification process Methods for using generalized testing chains to maintain process control, capitalize on previous testing results, incorporating results of nonrandom tests, and obtaining the discriminant calculation earlier in the testing process will be presented.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Statistical testing of software based on modeling expected usage is maturing as a practice. Several advances have been made in improving the modeling process, increasing the efficiency of testing, and optimizing the underlying calculations

Current tool support only measures individual testing cycles comprised of random tests. The need exists to support the ability to track reliability across multiple testing cycles and increments, as well as inclusion of nonrandom information in the reliability calculations. The following chapters will explore these underlying needs and propose methods for satisfying them.

Chapter 2

Background

This chapter presents a background of usage based statistical testing in an incremental development process for developing software.

2.1 Statistical Testing

One method for usage based statistical testing is Cleanroom Certification. [1] The key components to the process are the usage model, the test cases, and the testing chain

2.1.1 Usage Model

An outline for modeling the expected use of a software system is provided in [4] A Markov chain, referred to as the usage chain, is used to represent the anticipated user behavior The usage chain consists of a start state S_0, representing invocation of the soft-

ware, a final state S_F, representing termination of the software, and a set of intermediate states S_i, representing states of use for the software. The transitions between states represent the possible user actions with probability p_{ij} This usage chain can also be represented in matrix form, denoted P, where the entry for cell (i,j) is the probability of transitioning to state j from state i.

From this chain, several analytical results can be calculated One such result is the steady state distribution for each state i in the usage chain, denoted π_i This is calculated by solving the system of equations $\pi = \pi P$ [4]

2.1.2 Test cases

Another advantage of using a Markov chain to model the expected use of the software is that test cases are simple to generate. From the usage chain, a test case can be generated by taking a random walk through the usage chain from S_0 to S_F according to the transition probabilities for each state encountered. [4]

2.1.3 Testing chain

In order to record the results of executed tests, a new Markov chain is formed, called the testing chain [5] However, instead of probabilities, the testing chain stores frequencies

for the transitions For each successful transition in a test case, the frequency count for that transition in the testing chain is increased by 1.

If a failure is encountered between states S_i and S_j, a new state f_i is added to the testing chain, and the frequency count from S_i to f_i is incremented by 1 (If the failure is a recurrence of a previously encountered failure, the existing state for that failure is used)[5]

2.1.3.A Reliability Calculation

The expected reliability of the software system can then be defined as the probability of a realization from the testing chain reaching the final state S_F from the initial state S_0 without encountering a failure state By normalizing the testing chain, and modifying it to make the final state and all failure states absorbing, the reliability can be calculated with the following equation.

$$R_{S_0 S_F} = \hat{p}_{S_0, S_F} + \sum_{J \in \tau} \hat{p}_{S_0, S_J} R_{S_J, S_F},$$

where R_{S_i, S_j} is the probability of reaching state S_j from state S_i without encountering a failure, $\hat{p}_{S_i, S_j}$ is the normalized frequency count from state S_i to state S_j in the testing chain, and τ is the set of nonabsorbing states [5]

2.1.3.B Alternate Reliability Calculation

An alternate method for calculating the expected reliability from the testing chain stems from viewing it as a graph, and eliminating all states except for the invoke, terminate, and failure states. A state S_e can be eliminated as follows First, remove all transitions from state S_e to itself and normalize the remaining transitions Let S_I be the set of states with transitions to S_e, and let S_J be the set of states that have incoming transitions from S_e Then, for each state $S_i \in S_I$, remove the transition from S_i to S_e, and add a new transition from S_i to each state $S_j \in S_J$ with probability $p_{ij} = p_{ie}\ p_{ej}$. (If a transition from S_i to S_j already exists, it is incremented by this amount.) By repeating this process for all states in S_I, stateS_e can be eliminated An example of this is shown in Fig 2 1,

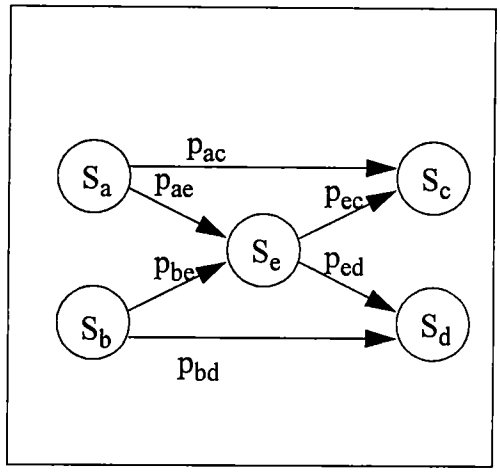

Figure 2.1: Graph Containing State S_e

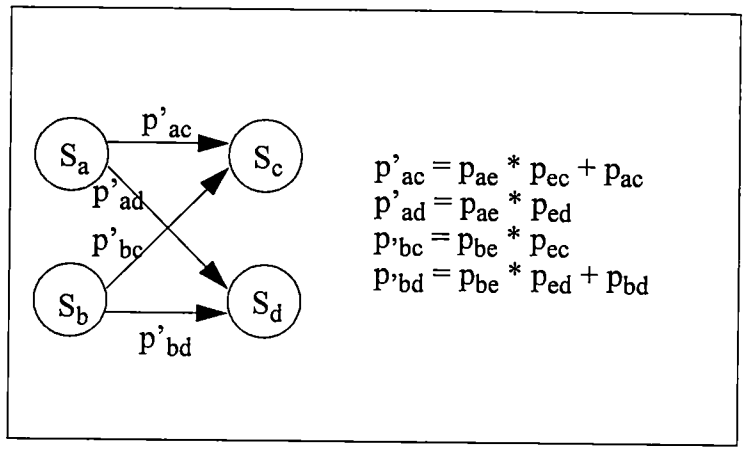

Figure 2.2: Graph After State S_e is Eliminated

before state S_e is removed, and in Fig. 2 2, after it has been eliminated. If this process is repeated for all states in the testing chain that do not represent the start state, the final state, or a failure state, the estimated reliability for the system will be represented by a single transition from the start state to the final state This is true because the expected reliability of the system is defined as the probability of a realization starting in the start state reaching the final state without encountering a failure

2.1.3.C Discriminant Calculation

It is also possible to calculate a quantitative stopping criterion for testing The discriminant, defined as the expected value of the log likelihood ration between the usage chain and testing chain, can be computed with the following formula

$$D(U, T) = \sum_{ij} \pi_i p_{S_i, S_j} \log \frac{p_{S_i, S_j}}{p_{S_i, S_j}},$$

where U is the usage chain and T is the testing chain [5] The value of $\hat{p}_{S_i, S_j}$ must be greater than 0 in order for the function to be defined, which means that all transitions in U must occur in T with a value greater than 0. This has the effect of requiring that all possible transitions must be taken during testing before the discriminant can be computed.

2.2 Testing in an Incremental Development Process

In an incremental testing process, the overall development of a software system is divided into several phases, called increments. Each of these increments should result in a fully functional executable system, whose functionality represents a subset of the final system behavior. Testing should be performed upon completion of the development phase of each increment. [1] Thus, for anything but a small system, there will be several phases of testing due to multiple increments

Within an increment, testing must demonstrate that the software meets predetermined reliability criteria in order for that increment to be successfully completed If testing reveals that the quality is unacceptable, the software should be returned to the development phase so the underlying faults can be fixed before the system returns to testing [1] This has the effect of producing several cycles of testing within each increment

Chapter 3

Multiple Testing Records

An important need for generalized testing chains is to support multiple testing records. This will allow for reliability to be measured for multiple testing cycles within an increment, as well as over multiple increments. Such measurements can provide useful information for process control and improvement Multiple testing records will also allow the effect of combining information within a testing chain to be measured

3.1 Multiple Records Within Increment

Software development usually consists of several iterations of testing followed by fixing the faults underlying the revealed failures and retesting Process control is critical during this phase of the process Modifying the code to fix one fault can easily result in the introduction of a new fault Thus, it is important to ensure that the process does not degen-

erate into a thrashing mode, where faults are introduced at the same rate they are removed and the overall reliability of the system is not improved.

Measuring the reliability of the system for each testing cycle during these iterations can demonstrate proof that the reliability is steadily increasing This can be done by forming a new testing record for each testing cycle, then comparing the reliability of the current cycle against the reliability of the previous cycles If the reliability is increasing, then the process is still in control. However, if the reliability is decreasing or remaining equal, it may signify a problem

Measuring the reliability across testing cycles in this manner will also allow for research on how this type of testing correlates with various reliability growth models

3.2 Multiple Records Across Increments

An incremental development process calls for successive rounds of development and testing, each of which results in an increasing subset of the final system, until the final increment completes the system [1] The usage model used for testing can also be successively increasing subsets of the final usage model that matches the functionality available in the system at the end of each increment. [3] When this is the case, the testing records for each of the increments can be combined into a single testing chain so that the reliability estimate can capitalize on all of the testing experience

If the individual increments consisted of multiple test and fix iterations, it will probably only be beneficial to combine the final testing cycle from each increment, since that represents the reliability of the software that was released from that increment

Tracking the testing results across increments can also help to optimize the testing of each increment. In [3], it is suggested that the usage model for each increment be modified so that the new functionality is more likely to occur in testing. It is discussed how this can improve the efficiency of testing in each increment since the new functionality will be emphasized. However, it also warns that the models for each increment will not accurately reflect the true expected use of the system. Combining the testing records of the previous increments can help to counteract this problem, thus improving the efficiency of testing without sacrificing the accuracy of the reliability estimate

3.3 Combining Information

Multiple testing records will also allow the information from testing performed in different environments to be combined into a single reliability estimate. If a system needs to run on several different platforms, the same test suite may be run on each platform Multiple testing chains can be used to track the testing information from each platform, as well as a chain containing the combined testing information from all platforms That will allow the tester to predict the reliability for each environment, as well as the overall system reliability Similarly, if different sets of probability distributions for a model are used to test

different categories of expected use, multiple testing chains can be used for the results of testing based on each distribution, as well as the combined overall testing effort.

3.4 Experimentation

Multiple testing records can also be used to measure the impact of combining information within a testing chain In the following chapters, multiple testing records will be used to measure the effect of introducing nonrandom information. This will be achieved by using one testing chain that contains purely random information, and a second chain that contains the random and nonrandom information. By comparing the calculations that can be made on each chain, the effect of the additional information can be measured

Chapter 4

Nonrandom Test Cases

In order to support the overall testing process, it is usually necessary for some nonrandom test cases to be run in addition to random testing However, only random test cases are used in determining the reliability calculation It would be beneficial to include the results of nonrandom tests in the testing chain, so that the entire testing experience will be reflected in the reliability calculations. This chapter will examine specific needs for nonrandom tests and how they can be included in the testing chain

4.1 Need for Nonrandom Tests

4.1.1 Minimal Transition Coverage Tests

The minimal transition coverage tests are the set of test scripts that will cover all transitions in the usage model in the fewest possible number of steps Although they are gen-

erated from the usage model, and represent valid realizations of the usage chain, they are not randomly generated.

Due to the fact that the minimal transition coverage tests cover all of the states and transitions in the model in as few steps as possible, they provide the testing process with a method of testing all of the available functionality in the system as quickly and cheaply as possible early in the testing process, which is a significant advantage Thus, they have become a common element in the Certification process.

4.1.2 Mission or Safety Critical Scenarios

Another instance where nonrandom testing may be required is to test functionality that is mission or safety critical A finite set of randomly generated tests cannot be guaranteed to cover specific functionality or create specific scenarios. Thus, if a particular feature or scenario of use has significant bearing on human life or health, or is particularly important from a business or pubic relations standpoint, it may be appropriate to test them with manually crafted test cases

This is especially true when the scenarios represent low-probability events A classic example is the emergency shutdown functionality of a nuclear reactor control program that prevents a meltdown In terms of expected usage, the probability of this event occurring is very low. However, in terms of public safety, it is absolutely essential to ensure that it works correctly. Thus, testing this functionality with manually created tests may be appropriate

Furthermore, federal regulations may require that the software must pass certain test suites, or demonstrate a successful response in specific scenarios Likewise, a customer may require similar demonstrations These circumstances may also warrant or require testing with nonrandom tests

4.1.3 Out of Testing Budget or Schedule

Another instance where nonrandom testing may be beneficial is when the testing process is running out of available budget or schedule time, but important features have not been tested. In this situation, it is only possible to execute a limited number of additional tests

If the remaining test resources are spent executing additional random tests, there is a significant risk that these tests will not cover the features that have been missed thus far Therefore, manually crafting a set of test cases to exercise the missing features may be the only way to ensure they get tested In these instances, executing nonrandom tests is the best utilization of remaining testing resources

4.2 Using Results of Nonrandom Tests

In this section, a method for including nonrandom tests in the testing chain will be presented The two aspects that need to be considered are adding the nonrandom testing expe-

rience to the testing chain and understanding the statistical ramifications of including nonrandom information in a reliability estimate

4.2.1 Inclusion in Testing Chain

As long as the nonrandom tests can be mapped onto the usage model, the testing results can be added into the testing chain in exactly the same manner as the results of random testing This includes any failure information that is revealed. Since the intent of the usage model is to characterize and represent all possible uses of the system, the nonrandom tests should correspond to the usage model in most instances

One possible scenario when the nonrandom test may not correspond to the usage model closely is if the model was constructed at a higher level of abstraction In this instance, the tester will need to translate the lower level nonrandom tests to the abstract level of the usage model before they can be stored in the testing chain

4.2.2 Statistical Ramifications

Since the reliability estimate is based on a random experiment, the effect of including nonrandom information must be examined

Any nonrandom test that can be mapped to the usage model represents a valid realization of that model, That is, that test case could be randomly generated from the model Thus, the nonrandom tests can be considered statistically valid

In terms of introducing nonrandom information into a random experiment, the goal would be to perform enough random testing to outweigh the nonrandom tests. Once this is accomplished, the benefits of including the nonrandom tests outweigh the disadvantages

The amount of random testing required to outweigh the nonrandom information can be measured with the discriminant calculation Nonrandom tests that differ significantly from expected usage will result in a higher discriminant value, signifying that a large amount of random testing will need to be performed. If the nonrandom tests do not differ significantly from expected usage, the discriminant value will be small In this case, the argument can be made that including the nonrandom tests in the reliability calculations does not hurt or will be quickly outweighed, since the low discriminant measure implies the testing chain is stochastically similar to the usage chain

4.3 Experimental Results

The discriminant can be used to measure the effects of the nonrandom information by calculating the discriminant value for all of the tests run for a given model, then calculating the discriminant based on just the random tests The difference between these values signifies the impact of the nonrandom information

As an example, the model of the simple GUI window specified in [4] will be used Suppose that the nonrandom test case shown in Fig 4 1 was used to exercise all of the functionality in the system at least once.

Script 1
- Invocation
- Window
- Maximize
- Window
- Minimize
- Icon
- Restore
- Window
- Move
- Drag Mouse
- Window
- Size
- Drag Mouse
- Up
- Drag Mouse
- Down
- Drag Mouse
- Left
- Drag Mouse
- Right
- Drag Mouse
- Window
- Close
- Termination

Figure 4.1: Nonrandom Script

Fig 4.2 plots the effect of this script as random tests are added The difference decreases very rapidly early in testing, implying that it is possible for random testing to minimize the effect of nonrandom information to an acceptable difference. However, it is important to note that the difference does not steadily decrease. Likewise, neither chain produces a discriminant value consistently greater than the other, since the difference briefly falls into negative territory

This implies that it is not a single threshold in terms of a number of random test cases that will outweigh nonrandom information. Instead, the effect of the nonrandom tests should be constantly measured and used as an additional stopping criteria

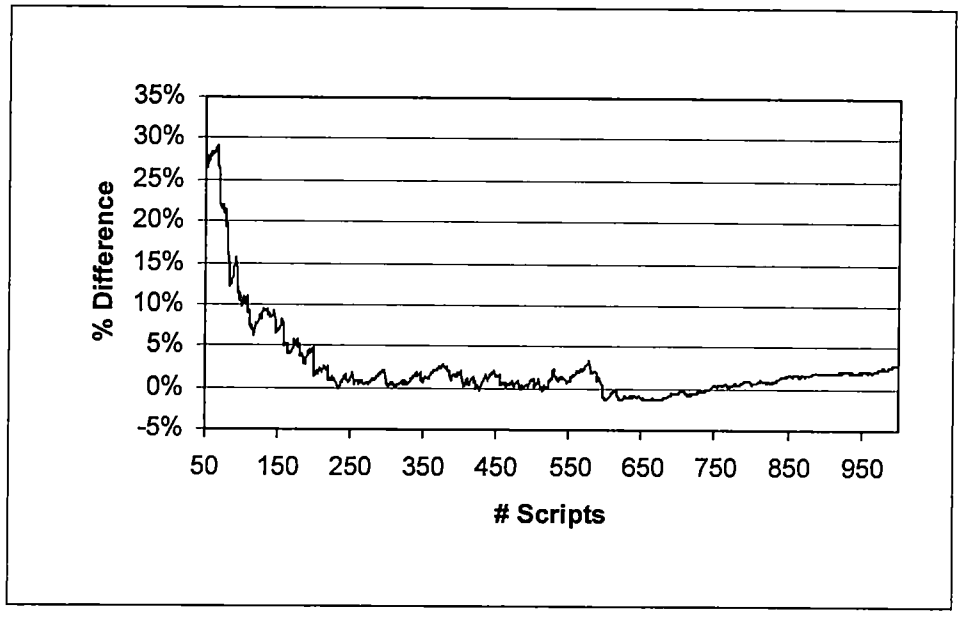

Figure 4.2: Discriminant Difference as Percentage

Chapter 5

Initialization of Testing Chain

One significant weakness in the current discriminant measurement is that it is not defined until all of the transitions in the usage chain have been covered in testing This has the effect of preventing the discriminant from being available to the tester early in the testing process Furthermore, in cases where a failure prevents a transition from being taken, it will never be possible to calculate the discriminant This poses a significant problem to the tester, since the discriminant is the strongest measure of how closely testing reflects expected usage.

One solution suggested in [2] is to assign initial values to all transitions in the testing chain. This has the advantage of allowing the discriminant to be calculated immediately, and is based on the assumption that enough random testing will be done to outweigh the introduction of nonrandom information

In the remaining sections of this chapter, we will examine different options for assigning initial values to the testing chain

5.1 Prior Testing Record

One option for initializing the testing chain is to use prior testing information If non-random tests are used, the advantages and disadvantages are identical to those of using the minimal arc coverage tests, discussed in §5.2. The nonrandom tests would merely be generated based on different criteria

The best option is to use information from prior random testing, if it is available This has the advantage of initializing the testing chain with random information, so there is no question as to the validity of the reliability calculations However, all of the transitions must be covered in the prior information in order to enable calculation of the discriminant

If prior random testing information is available, this is the best option However, due to the fact that a significant number of random tests usually need to be run in order to achieve full transition coverage, and there can be no failures preventing execution of transitions, this option will probably not be viable in most cases

5.2 Minimal Arc Coverage Tests

Another option for initializing the testing chain is to record the minimal arc coverage tests in the testing chain This is different from initializing the testing chain to a constant value of 1.0, since some states will likely be visited more than once in the minimal arc

coverage tests Since testing with the minimal arc coverage tests is a common element in the usage model based testing process, this information will usually be available.

This option has the advantage of reflecting actual testing that was performed. Furthermore, the minimal arc coverage tests represent statistically valid tests that could be generated from the usage chain, despite the fact that they are not randomly generated

This option has the disadvantage that any failures that occur could result in a transition not being visited, which would prevent calculation of the discriminant Another disadvantage is that since some of the values used to initialize the testing chain will be greater than 1 0, it will take longer for random testing to outweigh the nonrandom information.

5.3 Uniform Initial Value

A third option is to assign the same initial value to all transitions in the testing chain In [2] an initial value of 1 0 is used This is conceptually equivalent to populating the testing chain as if each transition had already been visited exactly once during testing It has the advantage that it will always enable calculation of the discriminant as soon as testing is begun, regardless of any failures that may occur

Another option is to assign a very small number to each transition This has the same advantage of always enabling calculation of the discriminant, and has the additional advantage that it will take less random testing to outweigh the very small number

A disadvantage of both of these methods is that initializing the testing chain with a constant value will not represent valid test cases for almost all models Even if an initial value of 1.0 is used, it is very unusual to be able to cover all arcs exactly once in a set of test scripts, so the resulting testing chain could not occur based on realizations of the usage model.

5.4 Experimental Results

Since the minimal arc coverage scripts and prior nonrandom tests are types of nonrandom information, the effect of using them to initialize a testing chain can be measured by the method outlined in Chapter 4

The effect of uniform initial values can be measured in a fashion similar to the method for measuring the effect of nonrandom information A series of random test cases can be generated and applied to testing chains initialized with different values, and a testing chain with no initial value The discriminants can then be compared against an uninitialized chain containing only the random tests to determine the effect of the initial value This was done for initial values of 1.0, 0 1, and 0 01

In order to account for statistical abnormalities within a set of scripts, ten sets of 1000 scripts were generated from the model of the simple GUI window specified in [4], then applied to a testing chain for each initial value. The value of the discriminant after each script was then averaged for each initial value.

The effect of the initial value on the testing chains needs to be measured in two primary areas the effect in the short term before enough random testing has been done to fully cover the testing chain, and the long term effect of the initial values.

5.4.1 Short Term Effect

The primary reason for adding an initial value to a testing chain is to allow for the discriminant to be calculated before all of the states have been covered by random tests Fig. 5 1 plots the average discriminant values from the testing chains for each initial value over the first 20 scripts

This demonstrates the discriminant values before enough random testing has been done to cover all of the states. Within the 10 sets of random scripts, the average full state coverage occurred in 10 1 scripts, with the fastest coverage in 4 scripts and the slowest in 17 scripts

As shown in Fig. 5 1, a lower initial value leads to a higher discriminant value This would intuitively imply that a higher initial value should be used However, a value closer to 0 implies that the testing is closer to expected usage Since the discriminant value from an initialized testing chain should only be used as an estimate, a larger initial value leads to a more aggressive estimate Conversely, a smaller value leads to a more conservative estimate for the discriminant when it would not otherwise be defined

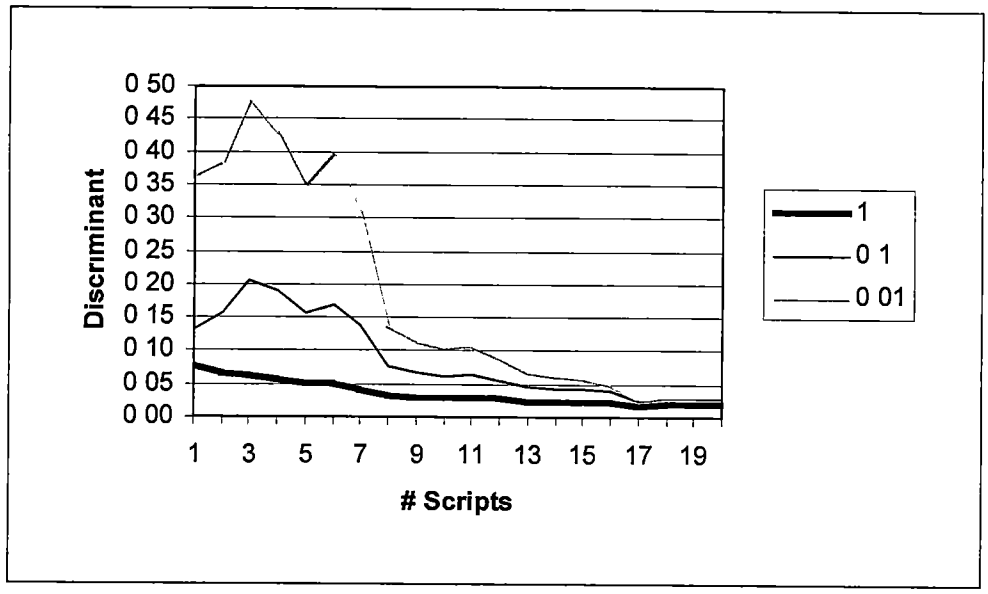

Figure 5.1: Short Term Average Discriminant

5.4.2 Long Term Effect

Table 5 1 shows the average discriminant values as more tests are added to the testing chains Fig. 5 2 plots the difference between the discriminant of the initialized testing chain and the uninitialized chain as a percentage for all three initial values

These results show that as the initial value increases by an order of magnitude, the effect of this initialization also roughly increases by an order of magnitude in early testing As expected, this effect decreases as more tests are run However, as can be seen in Fig 5 2, the chain with an initial value of 1.0 still shows a noticeable effect after 1000 scripts, even though the model only consists of 15 states By contrast, the effect of the initial value is negligible in the chain initialized with 0.01 after just 50 scripts and remains negligible as more tests are added.

Table 5.1: Long Term Average Discriminant

Number of Scripts	Discriminant (based on Initial Value)			
	0 0	0.01	0.1	1.0
100	0 00610794	0 00610386	0 00606807	0.00579651
200	0 00295012	0 00294954	0 00294453	0.00291146
300	0.00208595	0.00208546	0 00208117	0.00204598
400	0.00166516	0.00166492	0.00166282	0 00164590
500	0.00126304	0.00126292	0 00126189	0.00125405
600	0 00095934	0 00095921	0 00095603	0 00094807
700	0 00085530	0.00085519	0.00085425	0 00084607
800	0.00064543	0.00064538	0.00064140	0 00064093
900	0.00059268	0.00059264	0.00059230	0 0058968
1000	0 00057685	0 00057681	0 00057645	0 00057369

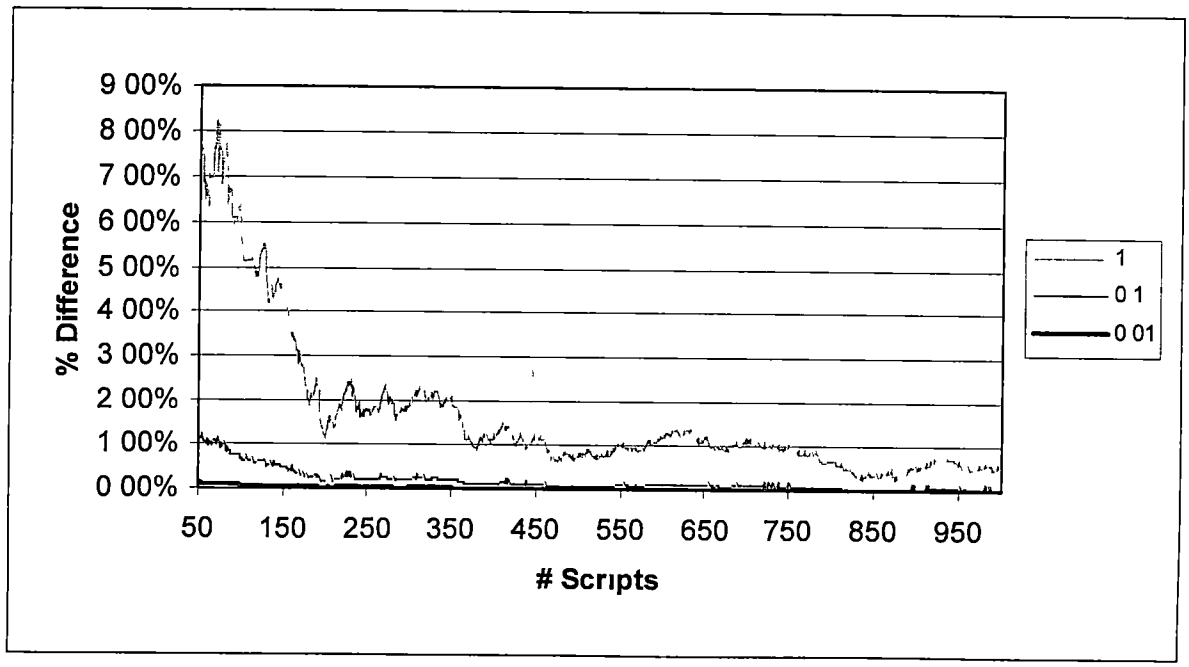

Figure 5.2: Long Term Discriminant Difference as Percentage

An important observation from the data in Table 5.1 is that the value of the discriminant calculated from testing chains with an initial value was always smaller than that of the uninitialized chain. This is significant since the discriminant is a measure of how well the tests match the expected use of the system, with a value closer to 0 meaning a closer match. Thus, an initialized chain will produce an overly optimistic discriminant value that implies the tests are closer to expected usage than they actually are

5.4.3 Conclusions

Based on these results, a smaller initial value should be used Since it provides a more conservative estimate in the early phases of testing, and has a negligible effect on the discriminant as more tests are added to the testing chain, a smaller initial value is the best approach.

Chapter 6

Conclusion

Several methods using generalized testing chains to support and improve the testing process were outlined Using multiple chains and capitalizing on the ability to add arbitrary information to them can help to address several issues encountered in the Certification process

6.1 Multiple Testing Records

A method for using reliability calculations from multiple testing chains to measure quality and process control within developmental increments was described By maintaining a separate testing chain for each cycle of testing within an increment, it can be determined if the quality of the software under test is improving with each iteration Similarly, a declining reliability will signify that the process is out of control

A method for using multiple testing chains to combine testing results across increments was provided By using a testing chain for each new increment that contains all new testing, in addition to all testing done in previous increments, it is possible to capture all prior testing knowledge.

Similarly, it was demonstrated how multiple testing chains can be used to combine testing results from different environments or user profiles By maintaining a separate chain for each environment and a chain combining all results, it will allow the tester to have specific reliability estimates for each environment as well as an overall quality measurement This same method can be used for experimentation purposes, with one chain representing a control group and all other chains representing experimental values

6.2 Nonrandom Test Cases

A method for adding nonrandom tests to a testing chain was described. By mapping the nonrandom tests onto the usage model, it is straightforward to add them to the testing chain It was demonstrated that the discriminant can be used to measure the statistical impact of the nonrandom information By maintaining one chain for the random tests, and a second chain for both the random and nonrandom tests, then measuring the difference in the discriminant values, the effect of the nonrandom information can be measured It was revealed that this difference needs to be constantly monitored, since it will arbitrarily increase or decrease as additional testing is performed.

6.3 Initialization of Testing Chain

Several options for initializing the testing chain were described to allow for computation of the discriminant value early in the testing process before all of the usage model has been tested It was concluded that prior test results should be used when available Otherwise, the minimal arc coverage tests can be used

However, it was also demonstrated that these methods may not be adequate in most cases Thus, initializing the testing chain with uniform initial values was shown to be a viable alternative. It was demonstrated that a smaller initial value should be used since it provides a more conservative estimate in the short term and has a smaller effect in the long run

REFERENCES

References

[1] R. C. Linger and C J Trammell, *Cleanroom Software Engineering Reference Model, Version 1 0*, Software Engineering Institute Technical Report CMU/SEI-96-TR-022, November 1996.

[2] J. M Morales, *Test Planning for Software Reuse*, Masters Thesis, Department of Computer Science, University of Tennessee, Knoxville, May 1997

[3] G H Walton, J H Poore, and C J Trammell, "Statistical Testing of Software Based on a Usage Model", *Software Practice and Experience*, Vol 25, No 1, January 1995, pp 97-108.

[4] J A. Whittaker and J. H Poore, "Markov Analysis of Software Specifications", *ACM Transactions on Software Engineering and Methodology*, Vol 2, No 1, January 1993, pp 93-106

[5] J A. Wittaker and M G. Thomason, "A Markov Chain Model for Statistical Software Testing", *IEEE Transactions on Software Engineering*, Vol 30, No 10, October 1994, pp 812-824

APPENDICES

Appendix A

GenChain Prototype Design Notes

This chapter discusses the design of the GenChain prototype, which implements the generalized testing chain features discussed in the proceeding chapters

A.1 Behavioral Specification

The behavior of the prototype towards the user is simplistic in nature The main complexity is in implementing the algorithms for the calculations The user will only be able to take a few actions, which limits the number of external stimuli the system can receive

A.1.1 Stimuli

The first step in specifying the behavior of the system is to identify the possible stimuli the system can receive These stimuli are listed in Table A 1

Table A.1: Stimuli

Stimulus Name	Corresponding User Action
New	Creating a new testing record for a specific usage chain
Open	Open an existing testing record
Save	Save the current testing record
Save As	Save the current testing record under a new name
Exit	Exit the system
New Uninitialized	Create a new test stand for the current testing record that is not set to an initial value
New Initialized	Create a new test stand for the current testing record that is set to an initial value
Random	Add random tests to the currently selected test stand
Nonrandom	Add nonrandom tests to the currently selected test stand
Name TS	Give the test stand the specified name

A.1.2 State Box

Since the behavior of the system is not extremely complex, the possible user actions are conditioned on the current state of the system, and most of the system responses involve updating specific values displayed to the user, a state-based specification of the system is appropriate However, before a state box specification can be completed, the state variables need to be defined

A.1.2.A State Variables

A convenient way of approaching the definition of state variables is to divide them into control state variables and data state variables

The control state variables are used to determine which response the system should provide to the user Table A.2 lists the control state variables for the prototype The initial value of the variable is listed in bold.

Table A.2: Control State Variables

Variable	Values	Description
TR_Open	true, **false**	true if a testing record is open, false otherwise
TS_Exists	true, **false**	true if at least one test stand exists for the current testing record, false otherwise

The data state variables are used to store the information necessary to provide the user with the correct value in a response Table A 3 lists the data state variables for the prototype.

A testing record will need to be able to store many test stands. Each test stand will have two testing chains, one for random tests, and one for all tests (The overall testing chain will store nonrandom tests as well as random tests, thereby being a superset of the random testing chain) For each of these testing chains, the prototype will need to report the number of tests in the testing chain, and the reliability and discriminant values calculated for the testing chain

Table A.3: Data State Variables

Variable	Description
TR_Name	Name of the current testing record (Also the name of the file in which it is stored.)
UsageChain	Usage chain for the current testing record
TS_List	List of test stands for the current testing record
TS_Name_i	Name of the i^{th} test stand
TS_Init_i	Initial value for the i^{th} test stand
TS_Testchain	Testing chain for all tests for the i^{th} test stand
TS_Tests_i	Number of overall tests for the i^{th} test stand
TS_D_i	Discriminant value from all tests for the i^{th} test stand
TS_R_i	Reliability value from all tests for the i^{th} test stand
TS_Randchain	Testing chain for random tests for the i^{th} test stand
$TS\_Rand\_Tests_i$	Number of random tests for the i^{th} test stand
$TS\_Rand\_D_i$	Discriminant value from random tests for the i^{th} test stand
$TS\_Rand\_R_i$	Reliability value from random tests for the i^{th} test stand

A.1.2.B State Box

Now that the state variables have been identified, the state box table can be defined. However, instead of listing a single large table, we will break the specification into logical groupings.

There are several stimuli that can be received at any time. These stimuli will also generate the same response, regardless of the current state of the system Table A 4 specifies these responses

There is also a set of stimuli that can only be received when a testing record is open (TS_Open = true) Their behavior is identical regardless of the state of the system, as long as TS_Open is true This behavior is specified in Table A.5.

The remaining stimuli are legal only when a testing record is open and at least one test stand exists. (TR_Open = true and TS_Exists = true) Again, when these conditions are true, their behavior will be identical regardless of any other system state, as specified in Table A 6

Table A.4: Stimuli not Affected by Current State

Stimulus	Response	State Change
New	A new testing record is created and displayed	TR_Open = true Usage_Chain = usage chain read from specified file
Open	Open specified testing record	TR_Open = true All other values set from file
Exit	System terminated	

Table A.5: Stimuli Valid when TS_Open = true

Stimulus	Response	State Change
Save	Current testing record saved to file	No change
Save As	Current testing record saved to specified file	TR_Name = specified name
New Uninitialized	New test stand without initial value created	TS_Init = 0 TS_Tests = 0 TS_Testchain = empty chain TS_Randchain = empty chain TS_D = -1 TS_R = -1 TS_RandTests = 0 TS_Rand_D = -1 TS_Rand_R = -1
New Initialized	New test stand with initial value created	TS_Init = specified initial value TS_Tests = 0 TS_Testchain = usage chain with frequencies set to initial value TS_Randchain = usage chain with frequencies set to initial value TS_D = -1 TS_R = -1 TS_RandTests = 0 TS_Rand_D = -1 TS_Rand_R = -1

Table A.6: Stimuli Valid when TR_Open = true and TS_Exists = true

Stimulus	Response	State Change
Random	Display new values	new tests added to TS_Testchain and TS_Randchain TS_Tests += number of tests read TS_D = new discriminant value for TS_Testchain TS_R = new reliability value for TS_Testchain TS_Rand_Tests += number of tests read TS_Rand_D = new discriminant for TS_Randchain TS_Rand_R = new reliability value for TS_Randchain
Nonrandom	Display new values	new tests added to TS_Testchain TS_Tests += number of tests read TS_D = new discriminant value for TS_Testchain TS_R = new reliability value for TS_Testchain
Name TS	Display new name	TS_Name = specified name

A.2 Object Oriented Design

Since Java is a pure object-oriented language, the necessary objects need to be defined before development can begin. Since all of the necessary state information has been specified in the state box, defining the objects is relatively straightforward

The first step was to define the main object In this case, since the user interface will only consist of a single screen with a small number of components, the main object can also contain the entire graphical interface with the user. This main class will be titled GenChain.

Now that the main class and user interface have been considered, the remaining functionality must be dealt with. In order to allow for a clean division between the primary functionality and the user interface, the best solution is to define a new class that provides the remaining functionality to the GUI. This class can represent an individual testing record, with an API that will allow access to all of its information This will allow the GenChain class itself to contain a single reference to this class, instead of tracking all of the test stands directly. Since this class will focus on a Markov usage model for the software being tested, it will be called the Model class

The Model class will need to store a usage chain Therefore, we will define a Chain class In order to conserve memory, the Chain class will store a list of transitions, rather than storing the chain as a matrix, since many of the cells would be empty Therefore, a Transition class will also be needed

The Model class will also need to store several testing chains However, since the user needs the ability to differentiate between random and nonrandom tests, and several values will correspond to each testing chain, it is cleaner to create an intermediate class This class was called the TestStand class.

Each TestStand class will then need to consist of two testing chains, one that stores all of the tests (random and nonrandom) imported into that test stand, and one that stores only the random tests Likewise, the TestStand will need to record the number of tests in each testing chain, as well as the reliability and discriminant values for each testing chain

Since the testing chain will store frequencies, but the reliability and discriminant values must be calculated on normalized testing chains, a NormalizedChain class extending the Chain class is needed. Since this is a case of an "is-a" rather than a "has-a" relationship, inheritance via subclassing is appropriate The NormalizedChain class extends the Chain class by defining a function to calculate the reliability and discriminant for that testing chain

Fig. A 1 shows the resulting class diagram

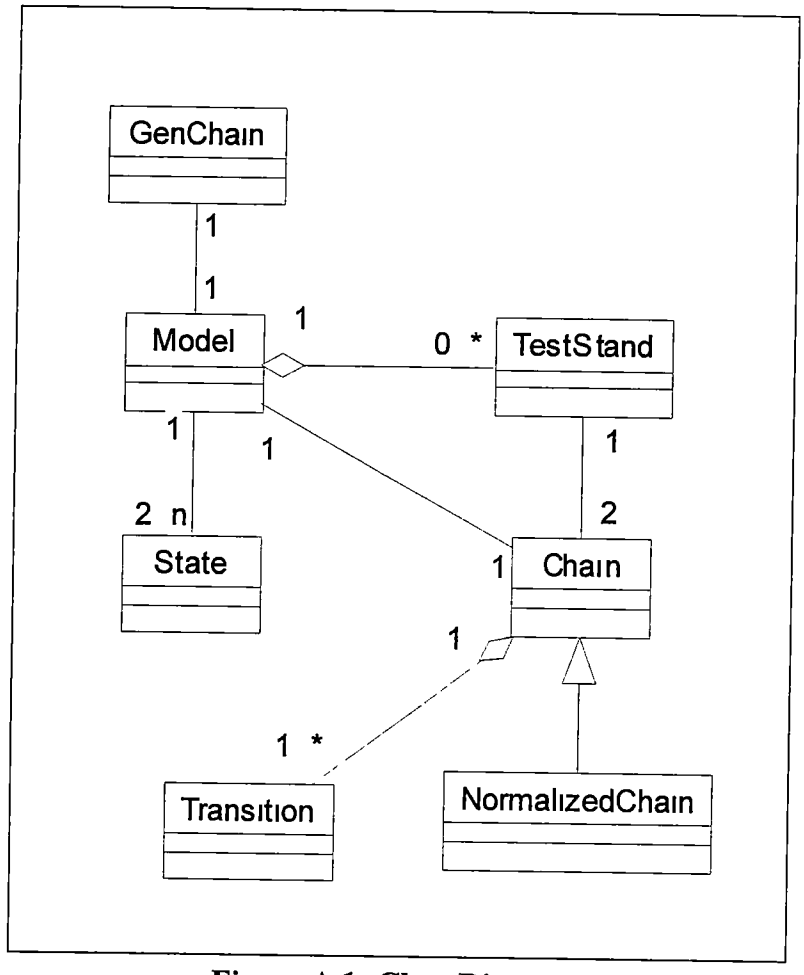

Figure A.1: Class Diagram

A.3 Implementation

Based on the state box and object oriented design discussed above, the implementation is very straightforward. Thus, once test scripts and failures are imported, the prototype has all of the necessary information for calculation of the reliability and discriminant

Since the steady state distribution π can be calculated by existing prototypes and commercially available tools, the decision was made to import these values along with the usage model, rather than calculating them internally

The graph approach to calculating the reliability of the system was utilized, as opposed to the matrix approach, since it is faster and requires significantly less memory for most models The algorithm to implement this approach is straightforward, but could not utilize existing numerical libraries as the matrix approach could

Fig A 2 and Fig A.3 show the sequence diagrams representing the method calls necessary to create a new initialized or uninitialized testing chain, respectively A different constructor is used for the TestStand object If an initial value is desired, the constructor with an initial value parameter should be used

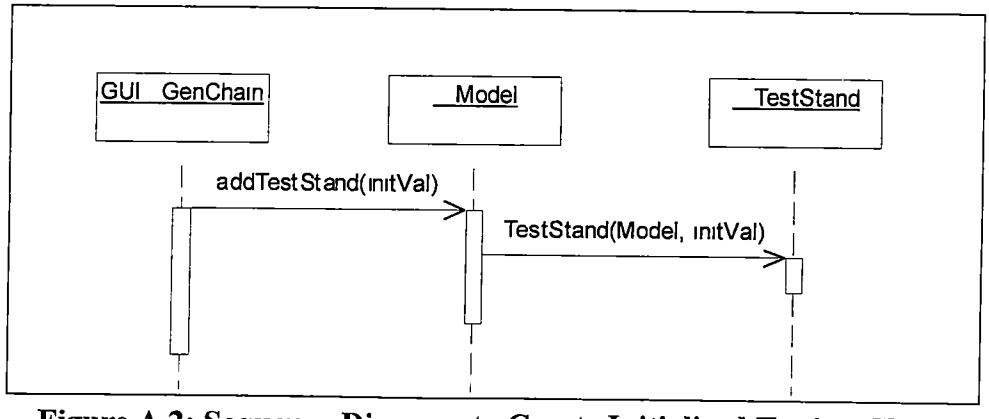

Figure A.2: Sequence Diagram to Create Initialized Testing Chain

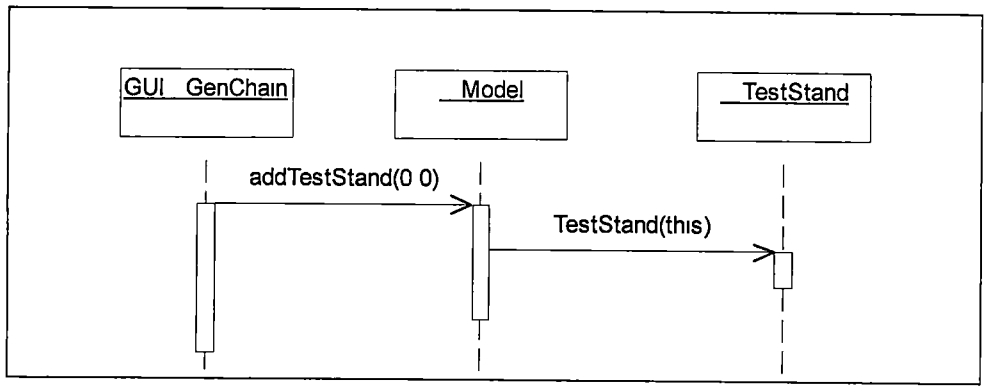

Figure A.3: Sequence Diagram to Create Uninitialized Testing Chain

The sequence diagram for adding nonrandom tests is shown in Fig. A 4 It should be noted that the tests are only added to the overall testing chain, and that the random testing chain is unaffected Adding new tests will result in the reliability and discriminant values for that testing chain being recomputed, taking the new tests and failures into consideration Note that the testing chain normalizes itself into a NormalizedChain, which then performs the reliability and discriminant calculations

The sequence diagram for adding random tests to a test stand is shown in Fig A 5 Note that it is identical to adding nonrandom tests, except that the tests are also added to the random testing chain, whose reliability and discriminant are also recalculated

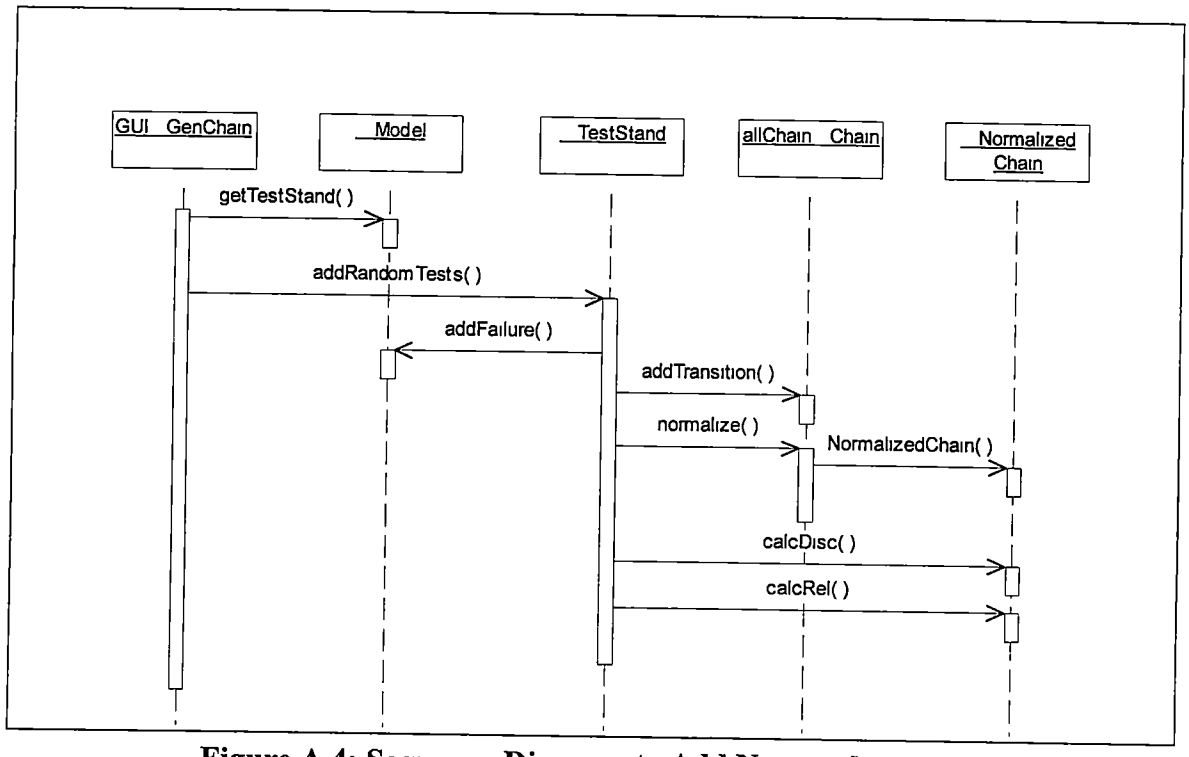

Figure A.4: Sequence Diagram to Add Nonrandom Tests

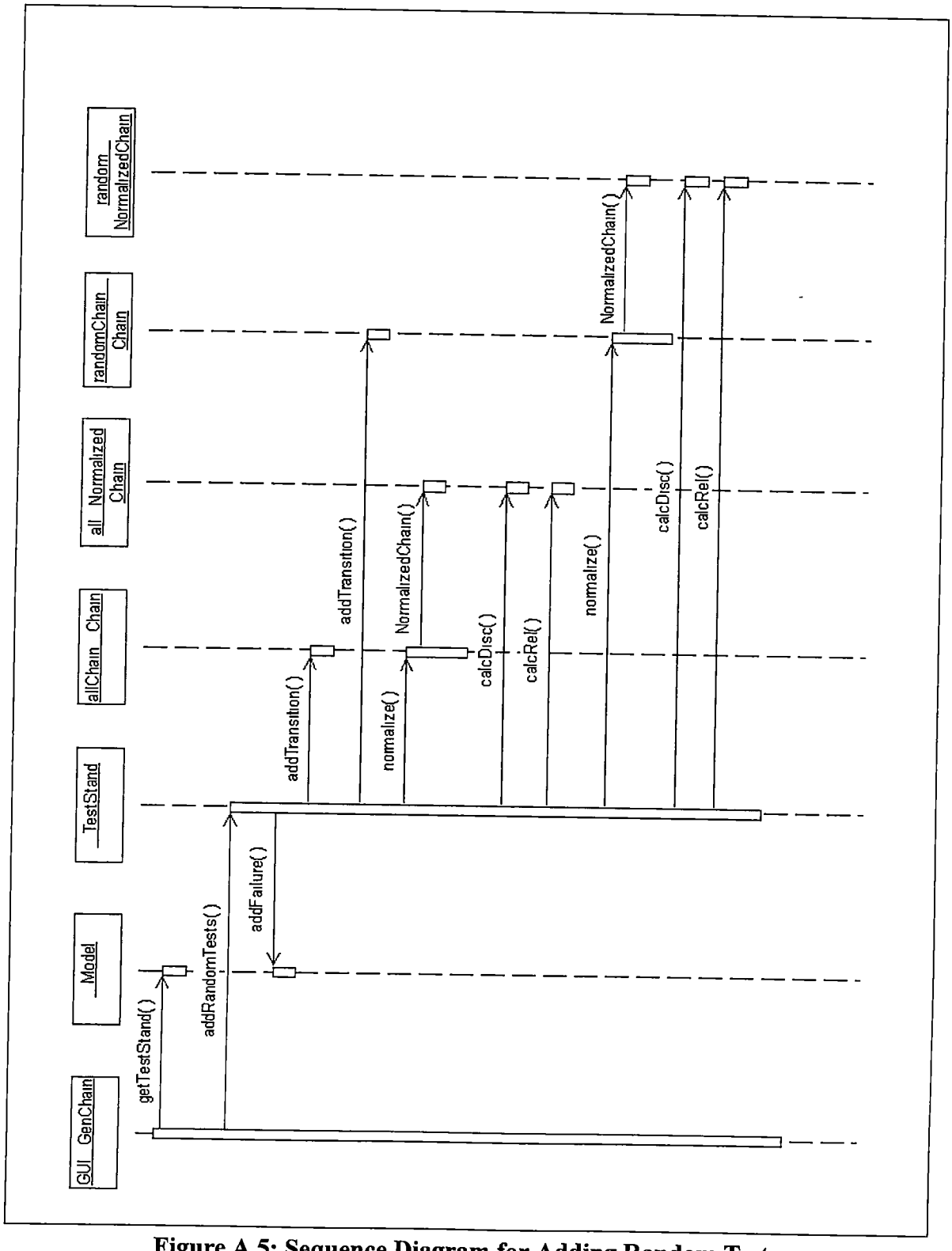

Figure A.5: Sequence Diagram for Adding Random Tests

Appendix B

GenChain Prototype Usage Notes

This appendix provides basic instructions for using the GenChain prototype to support generalized testing chains

B.1 General

B.1.1 Running GenChain

The GenChain prototype is written in Java 1.2. Thus, it can execute on any system that has a Java 1 2 runtime environment which has been installed correctly. To run GenChain, type `java GenChain` from the command prompt

B.1.2 Exiting the program

To exit the GenChain prototype, select the File - Exit menu option

B.1.3 Definitions

Testing record. The testing record is the central object for the GenChain prototype The testing record stores all of the testing chains associated with an individual model

Test stand. Individual testing chains are stored within a test stand The user can create as many test stands as desired for each testing record, each of which can be named Each test stand consists of a random testing chain and an overall testing chain. The test stand also reports the number of tests, reliability, and discriminant for both of these testing chains

Random testing chain. A random testing chain stores all of the random tests that have been imported to that test stand.

Overall testing chain. An overall testing chain stores both the random and nonrandom tests that have imported to that test stand

B.2 Working with testing records

The central object of the GenChain prototype is the testing record A testing record consists of a usage chain and all associated testing chains. This section discusses how to work with testing records in the prototype

B.2.1 Creating a new testing record

In order to create a new testing record, select the File - New menu option, or the New button on the toolbar. This will result in the Open Model file browser being displayed Select the file that represents the usage chain for the new testing record, and press the Open button A new, empty testing record will then be created.

B.2.2 Opening an existing testing record

To open an existing testing record, select the File - Open menu option, or the Open button on the toolbar This will result in the Open Testing Record file browser being displayed. Select the file that represents the testing record to be opened, and press the Open button The testing record will then be loaded, and all test stands will then be displayed on the main screen

B.2.3 Saving a testing record

In order to save a testing record, select the File - Save menu option, or press the Save button on the toolbar This will result in the current testing record, including all test stands, being saved to a file

B.2.4 Saving a testing record under a different name

In order to save a testing record to a different file, or a file with a different name, select the File - Save As menu option This will result in the Save As file browser being displayed. Enter the new name for the file and press the Save key The testing record will then be saved under the new name.

B.3 Working with Test Stands

Within each testing record, several test stands can be used Each test stand serves as a separate testing chain Furthermore, each test stand will track the reliability and discriminant for the overall tests, as well as just the random tests

B.3.1 Creating a new uninitialized test stand

In order to add a new test stand to the current testing record, select the **Test Stand - New Uninitialized** menu option. This will create a new test stand that is not initialized with a default value.

B.3.2 Creating a new initialized test stand

It is also possible to create a test stand with testing chains that are initialized to default value for all transitions This can be done by selecting the **Test Stand - New Initialized** menu option.

B.3.3 Naming a test stand

Each individual test stand can be assigned a name. This is done by clicking the name column in the row for the appropriate test stand and entering the desired name

B.3.4 Adding tests to a test stand

Testing scripts are added to individual test stands The test scripts can be designated as random or nonrandom In order to add random tests, select the **Test Stand - Add Ran-**

dom Tests menu option Nonrandom test tests can be added by selecting the Test Stand - Add Nonrandom Test menu option.

After either menu option is selected the file browser will appear. Select the file containing the scripts to be added and press the Open button At this time, the new scripts will be added, and the reliability and discriminate values will be updated for the test stand If nonrandom tests were added, only the overall reliability and discriminant values will be updated However, if random tests were added, both the random and overall values will be updated. This is because nonrandom tests are only added to the overall testing chain for the current test stand, while random tests are added to both the random and overall testing chain. This way the effect of nonrandom tests are displayed

B.4 Usage Notes

B.4.1 Tracking Reliability Within an Increment

To track reliability across multiple testing cycles within an increment, create a new test stand for each testing cycle. Record all scripts and failure information for each testing cycle in the appropriate test stand This will allow for the user to ensure that the reliability is increasing for each new testing cycle

B.4.2 Using Test Results from Previous Increments

The prototype will allow the user to capitalize on prior testing information, as long as the tests came from a usage model that is structurally equivalent to the model for the current testing record

Thus, to take advantage of testing knowledge from prior increments, create a testing record for the final usage model. Within the testing record, create a test stand for each increment in which testing information is available. Then, for each of these test stands, import the scripts and failure information for the corresponding increment and all previous increments. This will allow the user to take advantage of testing performed on previous increments of the system. However, it should be noted that this is based on the assumption that development of the current increment did not affect the reliability of the portions of the system tested previously.

B.5 File Formats

This section describes the file format required by the GenChain prototype.

B.5.1 Model Import File

The model import file consists of two sections; the first section lists all of the state names in the model, while the second lists all of the transitions exiting each state. There

must be exactly one blank line between each section. The state names may contain spaces, as well as any other non-control character The following specifies the file format·

```
state_name_1
state_name_2
...
state_name_n
<blank line>
from_state_1
<tab>        probability to_state_1
<tab>        probability to_state_2
...
<tab>        probability to_state_i
from_state_2
<tab>        probability to_state_1
...
<tab>        probability to_state_j

from_state_n
<tab>        probability to_state_k
```

In the first section, the entire line will be treated as the state name. Likewise, in the second section of the file, the entire line specified as from_state will be treated as the state name For the to_state field, the entire remainder of the line after the probability value will be treated as the name of the to state for the transition The probability value must be a floating point value between 0 and 1, inclusive, and the probabilities for all of the transitions leaving each state must sum to exactly 1 (The prototype will not normalize them) Each line specifying a transition must start with a tab, and the probability and the to_state fields must be separated by exactly one space

Following is an example of a valid model import file

```
a
b
c
d

a
        0.5 b
        0.25 c
        0.25 d
b
        0.75 c
        0 25 d
c
        0 5 a
        0 4 b
        0 1 d
```

B.5.2 Script Import File

The script import file also has two distinct sections, the first section lists all of the failures that were encountered while executing the scripts contained in the file, and the second lists the scripts themselves

Following is the specification for the script import file format.

Failure: failure_name_1 script_number transition_number halting_flag
Failure failure_name_2 script_number transition_number halting_flag

Failure: failure_name_n script_number transition_number halting_flag
<blank line>
Script 1.
start state
transition name 1
state name 1
transition name 2
state name 2
.
transition name n
terminate state
<blank line>
Script 2.
start state
transition name 1
state name 1
transition name 2
state name 2

transition name n
terminate state
<blank line>

Script n:
start state
transition name 1
state name 1
transition name 2
state name 2
. .
transition name n
terminate state

where script_number and transition_number for a failure corresponds to the script and transition within that script where the failure was observed The halting_flag field should be true for the failure if it prevented the remainder of the script from being executed

Following is an example script import file

Failure fail_1 1 10 true
Failure fail_2 3 1 false
Failure fail_1 3 4 false

Script 1
state a
transition
state b
transition
state c
transition
state a
transition
state b
transition
state c
transition
state a
transition
state d

Script 2
state a
transition
state d

Script 3
state a
transition
state c
transition
state b
transition
state c
transition
state a
transition
state b
transition
state d

B.6 Additional Utilities

In addition to the GenChain GUI, the underlying API can be accessed directly as a library, which allows for the creation of additional utilities to provide different functionality Two such utilities were created to produce the results of the calculations in a format that could be easily imported into a graphing application. These utilities are discussed below

B.6.1 RandomTestDriver

The RandomTestDriver utility was used to measure the effect nonrandom information had on a testing chain as random tests are added It allows the user to add an arbitrary number of randomly generated scripts to the random testing chain of an existing test stand within a model The discriminant for the random and nonrandom testing chains is output after each new script is added. It is executed as follows.

java RandomTestDriver model_name test_stand_index num_scripts

where *model_name* is the name of the model, (in CenChain format) *test_stand_index* is the index of the test stand within the model to which the random tests are to be added, and *num_scripts* is the number of new random scripts to be added

It will then output all of the results to stdout, which may be redirected to a file to store the results The first column of the output is the script number, followed by a column for the discriminant for the overall testing chain, followed by a column for the discriminant

for the random testing chain The columns will be delimeted by spaces, allowing the data to be easily imported into a spreadsheet or plotting utility as either space or whitespace delimeted information

The simulated tests will be added to the random testing chain of the specified test stand Once all scripts have been simulated, the model will be saved

B.6.2 DiscTestDriver

The DiscTestDriver utility was used to test the effect of initial values on a testing chain. It allows the user to specify an arbitrary number of initial values to test, and how many scripts should be applied to the resulting initialized chain The discriminant for each chain is output after each script is added It is executed as follows

java DiscTestDriver model_name num_scripts init_value [init_value.]

where *model_name* is the name of the model, (in GenChain format) *num_scripts* is the number of script to be simulated, *init_value* is the initial value for the first testing chain, and [*init_value*..] is zero or more additional initial values to be tested

It will then output all of the results to stdout, which may be redirected to a file to store the results The first column of the output is the script number, followed by a column for each initialized chain in the same order as they were provided. The columns will be delimeted by spaces, allowing the data to be easily imported into a spreadsheet or plotting utility as either space or whitespace delimeted information.

A new test stand will be created within the model for each initial value specified. Once all scripts have been simulated, the model will be saved. It should be noted that, even though the simulated scripts are randomly generated, they are added to the nonrandom chain for each test stand. This is purely for the sake of efficiency. Adding a script to the random chain would cause the discriminant to be calculated for both the random and overall chains, but adding them to the overall chain allows for a single calculation per script.

B.6.3 ScriptGenerator

ScriptGenerator is a utility class that will randomly generate scripts for a given model. It is not a standalone utility. Rather, it can be included in other utilities whenever there is a need to generate random scripts. The following method will cause a script to be generated and returned in a Vector object·

```
public Vector generateScript()
```

The script will be contained in the returned Vector, with each sequential Vector element containing the next state in the script. Thus, the calling program can walk through the script by creating an Enumeration over the Vector, or by sequentially accessing each element.

The seed being used to generate the scripts can be obtained by using the following method.

public long getSeed()

A set of scripts can be regenerated by using the same seed they were originally generated with

Vita

David Pearson was born in Cocoa Beach, Florida, on April 19, 1971 After being raised primarily in State College and Washington, Pennsylvania, he moved to Charlotte, North Carolina, where he graduated from East Mecklenburg High School in 1989 He then attended Virginia Polytechnic Institute and State University, earning a BSEE in Computer Engineering in 1994.

While attending Virginia Tech, David worked as a co-op for Digital Equipment in Hudson, Massachusetts for four semesters from 1991 to 1993. Upon graduating, he went to work as a consultant for Software Engineering Technology, Inc. in August of 1994 in Lanham, Maryland, until being transferred to Knoxville, Tennessee in October of that same year While working there, he took an increasing role in the design and development of a suite of CASE tools designed to support the Cleanroom software process.

In July of 1998, SET was acquired by Q-Labs, Inc., where David continued to work until August of 1999. At that time, he began working for Video Networks, Inc , in Atlanta, Georgia, leading the development of their family of products for delivering electronic media to the cable industry