

ABSTRACT

CHALKE, YATIN DATTARAM. Automatic Generation and Execution of Specification-Based Test Cases in a Network Based Environment. (Under the direction of Dr. Mladen Vouk)

Automatic testing of software has always been of interest. The concept has been receiving more and more attention in the software development climate of today (e.g., agile software processes) where managers and testers are being asked to turn around their products under shorter schedules and with smaller resources. Automation is being looked at as a pathway to meet the product testing and delivery deadlines. There are many research methods and tools, as well as a number of commercial tools that automate test design, generation and execution. Some are specification-based, some are structure based, and some are combinations of both. However the current state of automation of software testing has problems. These include design and maintainability of automation (e.g., current methods may not track product changes in a cost-effective way, and test-case suites do not always evolve easily and consistently), generation of excessive number of test-cases, end-user friendliness of the tool interfaces, measurability of the test-suite effectiveness, and so on.

The goal of the work presented here is to study automation of efficient specification-based test-case design, generation and execution. The investigation focuses on the principle known as pair-wise testing. Empirical evidence exists that shows that the technique can produce, depending on the algorithm used, test suites which are not only efficient in fault detection, but also have a manageable size. Objectives of the work were to a) enhance and further assess the open-source pair-wise test case design tool, developed previously by K.C. Tai and his students, called PairTest, b) develop a prototype of an adaptable easy-to-use system for automatic execution of the PairTest test cases, something that the original release of PairTest does not do, in a setting here one tests

network-based systems, and c) assess usability of pair-wise testing strategy and automation of testing in an industrial environment.

This work explicitly recognizes two stages of testing automation: i) automated test-case design and generation, and ii) automated construction of executable test-case from the generated suite. In assessing performance of PairTest, it was compared to the automatic test case generation practices in the industry with the help of different user scenarios. The prototype system for conversion of PairTest generated test-cases into executable test-suites is based on the open-source 'Expect' toolset. PairTest performs favorably where it comes to test-case design and generation, as well as fault-detection power. It exhibits an ability to contain exponential growth in the number of tests generated with the increase in the number of specification parameter values. A quantitative analysis of pair-wise testing and test automation in two industrial environment shows that, compared with manual methods, automatic test design and execution in general help a) increase the error detection capability (efficiency) of testing, b) reduce resource consumption, and c) increase evolvability (maintainability, adaptability) of the testing process. Results confirm that properly automated testing, as a 'complete' testing solution, not only improves the quality of the software and system, but also saves time and resources.

Automatic Generation and Execution of Specification-Based Test Cases in a Network Based Environment

By

Yatin Chalke

A THESIS SUBMITTED IN PARTIAL
FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF SCIENCE
COMPUTER NETWORKING

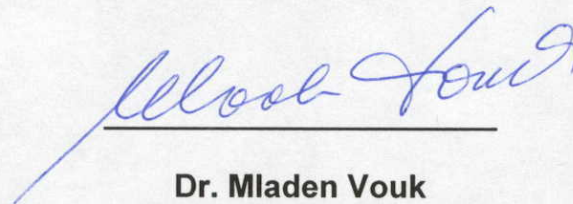
AT

NORTH CAROLINA STATE UNIVERSITY

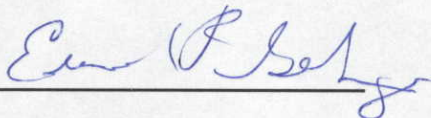
RALEIGH, NC

2003

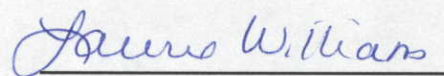
APPROVED BY:



Dr. Mladen Vouk
Chair of Advisory Committee



Dr. Edward Gehringer
Member of Advisory Committee



Dr. Laurie Williams
Member of Advisory Committee

DEDICATION

To my *Parents*
To my *brother* Ojas

Who have made my Higher Education dreams come true.

BIOGRAPHY

Yatin Chalke was born in Mumbai, Maharashtra, India. He did his undergraduate program at Sardar Patel College of Engineering, Mumbai, India and received a B.E. in Electronics Engineering from University of Mumbai in 2000. He joined North Carolina State University in fall 2000 to pursue M.S. in Computer Networking. He worked under Dr. Mladen Vouk since September 2001 on the topic Automatic Test Case Generation and Automation for High Performance systems. As a part of his MS Co - Operative Program, he has worked at Nortel Networks, Raleigh, North Carolina; Ericsson, Raleigh, North Carolina and IpInfusion Inc., Cary, North Carolina.

ACKNOWLEDGEMENT

I would like to thank Dr. Mladen Vouk, my advisor, for his guidance, and motivation and for providing the opportunity to work with him. Dr. Vouk has been instrumental in bringing me to the field of Networking and Testing and I am glad I got the chance to work with him.

I would like to thank Dr. Edward Gehringer for being on my thesis committee and providing invaluable advice and support in so many ways over past two years. I would like to thank Dr. Laurie Williams for accepting to be a committee member and providing useful comments on this thesis from time to time.

I would like to thank Mr. Christopher Lott from Telcordia for granting us the trial version of AETG for research use.

I would like to thank my friends for being there when I needed them.

Finally, I would like to thank my parents and family for their support and love. Thanks Mom for all the encouragement, you are the reason I am here. Thanks Dad for faith in me. Thanks Ojas for your support. Words alone cannot express the thanks I owe to my family, for their moral and emotional support.

This work was supported in part by NSF Award 9901004 and NC State Center for Advanced Computing and Communication.

Table of Contents

| | |
|--|------|
| LIST OF FIGURES..... | VIII |
| LIST of TABLES..... | X |
| CHAPTER 1..... | 1 |
| INTRODUCTION..... | 1 |
| 1.1 Background..... | 1 |
| 1.2 Nature of Testing..... | 2 |
| 1.3 Specification-Based Testing..... | 4 |
| 1.4 Automatic Test Case Generation..... | 5 |
| 1.5 Translation, Execution and Analysis..... | 6 |
| 1.6 Organization of Thesis..... | 7 |
| CHAPTER 2 | 8 |
| AUTOMATED TEST CASE DESIGN..... | 8 |
| 2.1 Test Generation Approaches..... | 8 |
| 2.2 Code-Based Test Generation..... | 9 |
| 2.3 Interface-Based Test Generation..... | 9 |
| 2.4 Specification-Based Test Generation..... | 10 |
| 2.5 Test Automation Framework Approach (TAF)..... | 13 |
| 2.5 Pairwise Strategy..... | 16 |
| 2.6 Illustration of Pairwise Strategy | 17 |
| 2.8 Summary..... | 18 |
| CHAPTER 3..... | 19 |
| IN-PARAMETER ORDER ALGORITHM..... | 19 |
| 3.1 The In-Parameter Order (IPO) Strategy..... | 19 |
| 3.2 An Algorithm for Horizontal Growth..... | 21 |
| 3.3 Algorithm IPO_H (T, P _i)..... | 22 |
| 3.4 An Algorithm for Vertical Growth..... | 22 |
| 3.5 Algorithm IPO_V (T, π)..... | 23 |
| 3.6 PairTest Test Generation Tool..... | 24 |
| 3.7 Test Generation Tools..... | 26 |
| 3.8 Automatic Efficient Test Case Generator..... | 27 |
| 3.8.1 AETG's Algorithm to Add New Configuration..... | 28 |

| | |
|--|-----|
| 5.3.8 Test Execution Evolvability..... | 105 |
| CHAPTER 6..... | 110 |
| CONCLUSIONS..... | 110 |
| 6.1 Summary..... | 110 |
| 6.2 Conclusion..... | 112 |
| 6.3 Future Work..... | 113 |
| REFERENCES..... | 114 |
| APPENDIX A..... | 122 |
| APPENDIX B..... | 154 |
| APPENDIX C..... | 172 |

List of Figures

| | |
|--|-----|
| Figure 1 Test Specification Elements..... | 15 |
| Figure 2 Edit Parameter window for PairTest (GUI Version)..... | 31 |
| Figure 3 Edit Relations window for PairTest (GUI Version)..... | 32 |
| Figure 4 Edit Constraints window in PairTest (GUI Version)..... | 33 |
| Figure 5 Parameter Values | 38 |
| Figure 6 Relations..... | 39 |
| Figure 7 Test Set Using PairTest..... | 40 |
| Figure 8 Test Set from PairTest (Network Processor Example)..... | 49 |
| Figure 9 Test Generation and Automation Process..... | 66 |
| Figure 10 Test Automation Process..... | 68 |
| Figure 11 Test-Bed..... | 80 |
| Figure 12 Failure Coverage Vs. Number of Test Cases (PairTest)..... | 86 |
| Figure 13 Fault coverage Vs. Number of Test Cases (PairTest)..... | 87 |
| Figure 14 Number of Test Cases Vs. Failure Coverage- INDUSTRIAL..... | 87 |
| Figure 15 Per Test Case Efficiency Vs. Number of Tests..... | 90 |
| Figure 16 Average Testing Efficiency Vs. Failure Coverage..... | 91 |
| Figure 17 Resource Distribution over Testing period- INDUSTRIAL..... | 93 |
| Figure 18 Resource Rate Vs. Time (Test Generation- PairTest) - LABORATORY..... | 94 |
| Figure 19 Resource Rate Vs. Time (Test Generation) - INDUSTRIAL..... | 96 |
| Figure 20 Resource Rate Vs. Time (Test Execution) data - LABORATORY..... | 99 |
| Figure 21 Resource Rate Vs. Time (Test Execution) - INDUSTRIAL..... | 100 |
| Figure 22 Evolvability Vs. Time (Test Generation – PairTest) - LABORATORY..... | 103 |
| Figure 23 Evolvability Vs. Time (Test Generation) -INDUSTRIAL..... | 104 |
| Figure 24 Evolvability Vs. Time (Automation) – LABORATORY..... | 106 |
| Figure 25 Evolvability Vs. Time (Automation) - INDUSTRIAL..... | 107 |

| | |
|--|------------|
| Figure 26 Edit Parameters from AETG..... | 154 |
| Figure 27 Edit Relations from AETG..... | 155 |
| Figure 28 Edit constraints from AETG..... | 156 |

List of Tables

| | | |
|------------------|---|------------|
| Table 1 | Fault/Failure Coverage Vs. Number of Test Cases Data..... | 85 |
| Table 2 | Resources Vs. Time (Test Generation) data..... | 94 |
| Table 3 | Resources Vs. Time (Test Execution) data..... | 98 |
| Table 4 | Evolvability Vs. Time (PairTest – Test Generation) data..... | 102 |
| Table 5 | Evolvability Vs. Time (Automation) data..... | 105 |
| Table 6 | Test Suite Analysis for Per Test Case Efficiency..... | 147 |
| Table 7 | Executable values of test cases - PairTest..... | 148 |
| Table 8 | Executable values of test cases – Manual Test | 149 |
| Table 9. | Manual Test Generation Incremental configuration..... | 150 |
| Table 10. | Manual Test Execution Incremental configuration..... | 150 |
| Table 11. | Failure Coverage data –Industrial..... | 151 |
| Table 12. | Test Generation- Industrial Resources..... | 151 |
| Table 13. | Test Execution- Industrial Resources..... | 152 |
| Table 14. | Evolvability –Industrial (Test Generation)..... | 152 |
| Table 15. | Evolvability – Industrial (Test Execution)..... | 153 |

Chapter 1

Introduction

1.1 Background

Testing is a process of establishing confidence that a program or system does what it is supposed to do [Few99]. Software has to go through many stages to reach an acceptable level of this confidence. The developed software has to be thoroughly checked to see that it serves its purpose. This checking is effected by verification, validation and testing [Adr82; IEEE97; Pres92]. Verification, validation and testing are ongoing processes which start with the software requirements and continue even after the development is completed. Software testing is one of the most visible aspects of this process. It is also a very challenging task.

Showing that software is 100% correct can be done in only two ways – through **program proving** and through **exhaustive testing**. Neither is a realistic option in general practice. The former is usually too expensive; the latter is usually defeated by the test-case (state) explosion problem. A partial answer, but a promising one, to this problem is a combination of

- a) A development and test-case generation approach that maximizes fault-detection properties of a test-suite and minimizes the resources needed to develop these test cases

- b) a method that allows automatic translation of above test cases from its abstract form into its executable form
- c) a method that allows a resource efficient execution of the test cases
- d) a method that allows a resource-efficient evaluation of the results of test-case execution, and
- e) a resource-efficient way of updating/maintaining the test-suite in the face of changes in the software it is intended to test.

Automation of some of the elements above is what this thesis investigates.

1.2 Nature of testing

Testing cannot guarantee the **absence** of faults¹ in complex computer-based systems [Het88]. *“If you think your task is to find problems, you will look harder for them than if you think your task is to verify that the program has none”* [Myers1979]. Hence, the purpose of the software testing is detection of failures, defects and faults rather than demonstration of their absence [Bei83]. Testing can be used only to show the **presence** of faults [DDH72]. The basic reason is the discrete nature of digital computer software and by implication testing. For example, if two test points in the input space of a piece of software (the space of all external inputs and stimuli a software program can receive directly or indirectly, including external interrupts) have been tested successfully, one can only conjecture about the behavior of the software for other inputs situated between or close to these test points. Hence, exhaustive testing (testing of all program input states) is desirable, however infeasible.

¹ This thesis adopts the following terminology [IEEE97; Musa97]: a human or automation mistake or ERROR results in a FAULT in the software at some stage of its development and process (e.g., concept, requirements, design, implementation, deployment, verification, validation, testing, management). The fault may propagate through the process stages that follow as a DEFECT. When software is deployed, execution of the software paths that intersect with the defect, may result in an internal ERROR-STATE. When, and if, that internal error-state becomes visible to an outside observer (or interfacing software), then a FAILURE is manifested. In general, observation of an ANOMALY in the observable behavior of software does not mean a true failure has been observed. This anomaly needs to be investigated to see if it is a genuine departure from the defined expected behavior, and then this needs to be followed up with defect location, root-cause analysis, and construction of an appropriate problem resolution response (e.g., defect patch, or fault removal).

Exhaustive testing requires that a program's responses be evaluated for all possible inputs [Bla98]. For example, one may have to consider [Bei95] the following.

- Testing the program for all valid inputs.
- Testing the program for all invalid inputs.
- Testing the program for all edited (changed/ mis-typed) inputs.
- Testing all variations on input timing.
- Testing the program for all combinations (Almost infinite)

Obviously, for anything but trivial specifications, the test cases will grow in numbers very fast. It is often unrealistic to expect that all possible test cases can be developed to start with, let alone run and analyzed. Yet, if any of the cases from the "exhaustive" suite are not run, it means the program was only tested in part.

Consequently, only what is considered to be a **representative** subset of inputs is usually manifested as real-life test suites. This means that, in practice, testing only shows that a system behaves correctly with respect to the requirements pertaining to the actual test cases run, the **transitivity** of that quality to intermediate states is an **assumption**. The issue is that of selection of "representative" test cases that assure that this assumption, and the resultant conclusions about the transitivity of the ascertained quality of the product, hold true all the time or at least most of the time.

Testing has to be considered with respect to the development **process** too. In the classical waterfall process model of software development, testing is presented as the last phase before software release [Roy70]. This may cause a tendency to cut the testing effort if a project exceeds their deadlines or resources. Consequently, unless the specifications are very static and the software problem is very well defined, the tests have to be designed and iteratively maintained as soon as, in the development process, the (testing) requirements are available. This problem was recognized and has been embodied in iterative software development processes [Ore93; Sot01], and software engineering standards [IEEE90]. Verification, validation and testing have to

start as soon as software development starts, they have to be flexible and adaptive, and they have to be pervasive. More recently, the issue has been highlighted by agile development processes [Aoy97; Aoy98; Gla01; New02; Boe02; Sch01]. In fact, selecting an appropriately timely and dynamic verification, validation and testing process and strategy is the key to combating the burden of “exhaustive” testing. .

1.3 Specification-Based Testing

In this thesis we focus on the test case generation and automation methods that derive from software specifications rather than structural characteristics. The aim is to develop test-suites that are both fault-effective (fault-sensitive) and resource-effective (e.g., small number of test cases).

In practice, almost 30% of testing effort is spent in writing test cases [Bla98]. Yet, these test cases may or may not be effective in actually detecting faults. Hence, the test-design and/or test-case generation algorithm(s) plays major role in generating efficient test cases. Unless fault-sensitivity, ability of test cases to detect faults, is the primary driver for test case development strategy, other factors, such as software life-cycle phase priorities or time constraints, may dictate test-suite composition. For example ad-hoc testing [Bei83] is the need to generate test cases which run only once and provide some immediate situation-specific assurances or feelings of adequacy. Alternately regression tests [Rot02; Mal02] serve as “benchmark” against which the product is tested repeatedly to assure that changes do not break functionalities that worked before the changes were made. This requires test cases which are not only very fault-efficient and resource-efficient, but also adapt to changes in the software in a cost-effective way.

As much as 60% of the defects uncovered in software are due to missing requirements or improper specifications [e.g., Per95]. Hence, it makes a lot of sense to make the specifications - so called “black-box” testing based on specifications - a principal driver for test case design [Sch02]. It is commonly accepted that formal

specifications provide the best basis for test case generation [Eka91; Boc88]. Unfortunately, a common drawback is that a large gap between a formal test-case generation strategy and the ability to automatically generate cost-effective executable test-cases. As mentioned earlier the issues are: test-design strategy (algorithms), translation of these test-cases into executable test-suites, actual execution of the test-suite, the development of an oracle for being able to tell when the results of test execution are correct, and adaptability and maintainability of test-suites.

There are a number of papers discussing the issue in automatic test case generation and automatic test execution [Dus99; Few99; Gri02; Hu01]. The former aspect, issue of automatic test generation, deals with item a) and e) of section 1.1 – a method to take formalized specifications and easily develop effective and efficient test suites. The later, automatic test execution, deals with items b), c) and d) – ways to automate generation, execution and analysis of the test suites and their results. This thesis is focused specifically on automation of the translation between abstract test-cases generated using an automated formal strategy, in this case PairTest [Coh94; Coh96; Tai01; Tai02], and on automation of test-execution of such test-cases.

1.4 Automatic Test-Case Generation

A major obstacle to automatic generation of test cases using specifications is the required support for going from the test-case design to the actual testing (design-to-test ability). Underlying development models often do not support automatic test vector generation for specification based testing [Bla98; Hie97]. There are many techniques available to convert such models into ones that do incorporate design-to-test for automatic test vector generation [Hie97; CHJ88]. However most of the approaches make the maintenance of generated test cases expensive with changes.

A general framework of interest is called Test Automation Framework (TAF) [BBNC01]. It is one of a number of such frameworks for automatic test generation [Bla98; Sto93; Sto96; Meu98]. What sets TAF apart is that it provides not only a

resource-efficient way for automatic test case generation using specifications, but also it has a reasonable maintenance overhead. Management of the generated test cases is simplified by TAF through the use of verification models which transform specifications into a pre/post condition pairs called test specification elements. In the present case, TAF was applied to the In-Parameter Order algorithm [Tai01], a member of the pair-wise testing family of solutions [Coh94; Coh97; Ding99; Wil96]. This model identifies the requirements in terms of logical entities representing the input parameters and the environment of the system under test. The verification part of model adds inter-parameter relations and constraints. The model is quite efficient in the number of test-cases it generates, and it appears to have very good fault-detection properties [Ding99; BBNC01; Tai01; DKLL98]. The model based on In-Parameter Order algorithm has been available as an open source abstract test-case generation tool called PairTest [Tai01; Tai02].

This work enhances the tool based on In-Parameter Order algorithm in two ways: (1) its handling of parameter options, and (2) the translation of abstract test-suites into executable test-suites.

1.5 Translation, Execution and Analysis

Efficient translation of formally developed test cases into executable test cases is very desirable. However, the actual execution of test cases and the analysis of the results for correctness is often more time consuming than test case development. The “oracle” issue is not within the scope of the present work, but translation and execution are.

Manual test translation from abstract to executable and manual execution is one option for test case execution. Unfortunately, this process is usually not resource efficient [Dus99]. Automation is desirable, but one has to be very careful that the chosen approach does not suffer from the one-way-street syndrome, i.e., it is easy to generate and execute a test-suite automatically, but it is not easy to maintain the

generated test-suite to account for incremental changes in the system specifications (requirements, design, environment, etc.) incrementally adaptable to changes in the system. *“The management and performance of test activities, to include the development of executable tests and execution of test scripts so as to verify test requirements, with adaptability to changes is needed ”* [Few99].

Development of application-specific test-drivers, software that effects translation of abstract test cases into executable test cases, can be very time consuming and error prone. This can lead to high maintenance costs. This is an issue that agile methodologies address explicitly, since early automated testing is often an integral part of such strategies [Poo01; Gla01; Boe02; New02, Sch01; Aoy98]. The resource efficient and adaptable test automation can be addressed using solutions such as the Automated Test Life-cycle methodology (ALTM) [Dus99]. This methodology can be used to provide a structured approach for automated execution of test cases leading to development of an adaptable test script. The ALTM concept was adopted in this work to develop a resource efficient and adaptable test script for translation, automated result analysis and execution of test cases generated by PairTest.

1.7 Organization of Thesis

Chapter 2 discusses test automation concept and tools. Chapter 3 focuses on PairTest and discusses the In-Parameter Order algorithm. It also describes another algorithm for pair-wise testing implemented in the commercial Automatic Efficient Test Case Generator (AETG) tool. It also compares the operational properties and performance of the implemented system, PairTest. Chapter 4 describes the “Expect”-based automation methodology implemented as part of present work. Chapter 5 provides a metric-based analysis of the value of test automation. Chapter 6 provides summary and conclusions.

Chapter 2

Automated Test Case Design

2.1 Test Generation Approaches

The process of testing involves many stages. The stages of identifying the requirements or test areas and design of test cases are considered intellectual activities [Few99]. Whereas, stages of execution of test cases and comparison of test outcomes are much more repetitive. The intellectual activities affect the quality of the test cases. In contrast repetitive activities are time consuming and boring.

Thus, activities of test execution and comparison are repeated many times, while the activities of identifying test conditions and designing test cases are performed less times. Due to limited time available for testing, testers have to select test cases which will be executed. But as there can be infinite conditions in an input domain, the choice of such cases becomes very important thus indicating need for effective test generation process. This emphasizes need of high fault detection capability for effective test generation.

In the following sections, we evaluate three types of test generation approaches: code, interface, and specification-based testing and their pros and cons to choose an approach for developing a cost effective and efficient automated test generation process.

2.2 Code-based test generation

Code-based test generation approach develops test inputs by examining the structure of the software code itself. A path through the code is composed of segments determined by branches at each decision point. The conditions required to produce such paths can be provided automatically to generate paths through the code.

The code-based test generation approach generates test inputs. But expected outcomes are required for each test for verification and analysis of the test results. Any code based test case design cannot tell whether the outcomes produced by the software are the correct values. So this approach is incomplete, since it cannot generate expected outcomes. *A Test-Oracle is the source of the correct expected outcomes but a code based test generation scheme has no test-oracle* [Het88].

Another type of code-based test design approach can generate tests to satisfy weak mutation testing criteria. Mutation testing is where the code or input is changed in some small way to see if the system can correctly deal with this mutated version. But still, such code-based test generation approach to satisfy weak mutation testing does not provide the appreciable level of fault detection coverage [Het88]. The main reason for this is attributed to newly generated execution paths through the changed code which reduce the fault detection ability.

2.3 Interface-based test generation

An interface-based test generation approach can generate tests based on some well-defined interface, such as GUI or web application. If a screen contains various menus, buttons and check boxes, this approach can generate test cases which can check each such menu. For example, test cases generated by this approach can verify if a check box is checked or if a field is selected.

A similar approach can be taken to develop test cases for Internet and intranet pages. The tools developed using this approach can activate each link on a World Wide Web

page and then do it for each of the pages linked to recursively. The interface-based approach can be useful for identifying some functionality-related defects. An expected outcome can be partially generated, but only in a very general sense and kind of in the binary form as it can just identify a correct working result and wrong result. This approach cannot tell if all the outcomes are logically right.

This approach can therefore generate tests by generating test inputs and thus identifying some functionality errors. It can be defined to be partial oracle [Het88]. It is very helpful in performing a *'roll-call' testing* [Het88] which means that it can test everything which should be there is there. Testing using test cases generated this way can be too tedious to do manually as it take lot of patience and time which defeated our goal of development of an efficient test case generation scheme.

2.4 Specification-based test generation

Specification-based test generation refers to creating test inputs from the software specifications. It allows tests to be created early in the system development and to be ready for execution, which simplifies incorporating changes in the test cases with changes in system. Additionally specifications can be used as a basis of output checking which can lead to reduction in one of the major costs of testing, the cost for analysis of the test results. The specification-based testing technique provides another advantage by keeping the essential part of the test data independent from the particular implementation of specifications. It can generate test inputs and also expected outputs, provided that the specification is in a form that can be analyzed by the tool. The specification may contain structured natural language constraints, such as business rules, or it may contain technical data as states or transitions. So, these tests do have an automatically-generated test oracle [Het88].

If the ranges for the input parameters are clearly defined, then the specification-based technique can generate boundary values as well as sample valid and invalid classes of values [Pos96]. Few specification-based tools can derive tests from cause-effect graphs [CL01] which can then identify some kind of ambiguities and omissions. A benefit of specification-based testing is that the tests generated test the aspect in

which tests are concentrated at what the software should do rather than concentrating at what software does [HJL96]. This helps in avoiding test generation for satisfying developed software criterion by sidelining the actual requirements. For such test generation expected outcomes can be generated if they are stored in the specification. Most of the defects in software are due to lack of consistency in implementation of requirements. Boehm and Basili state that finding and fixing a problem late in the development process can be more expensive than finding and fixing it during the requirement or design phase [BB2001]. Thus specification-based testing approach can help generate a cost-effective technique with high fault detection ability.

The limitations of the above mentioned automated test case generation approaches can be summarized as.

- Code-based methods do not generate expected outcomes.
- Interface-based methods can only generate partial expected outcomes.
- Code-based and interface-based methods find very less specification defects.

Considering these aspects of the test generation approaches, it can be seen that the approach using requirements of the software as a criterion to generate test cases, can help identify most defects thus increasing fault detection capability [Ding99; Hu01; BF98]. Thus specification-based test generation approach can be used to generate tests with high fault detection ability which can check software requirements effectively.

There are many specification-based testing techniques available which can be used for automated test generation abilities [Ding99; BF98]. This emphasizes choice of an effective specification-based test generation technique to provide high fault detection capability. We evaluate three specification-based techniques namely full-predicate testing, transition-pair testing and specification-mutation testing to choose a technique for implementation of test case generation tool.

State-based specifications describe software in terms of states and transitions. Typical state-based specifications define conditions on transitions, which are values that specific variables must have for the transition to be enabled. The *triggering events* [Ding99], which are changes in variable values that cause the transition, are evaluated in specification-based testing. A triggering event is a change in one or more variable that can cause a transition to enter a new state [Ding99].

Full-predicate testing and transition-pair testing are state-based specifications; specification-mutation testing, is a state-checking specification. Full-predicate coverage checks whether specifications are formulated correctly. The full-predicate coverage criterion demands that, at minimum, inputs derived from each specification predicate are provided. This criterion requires that each specification predicate is tested independently [Ding99]. Thus it attempts to address the question of whether each specification is necessary and is formulated correctly. Due to this, tests which do not account for the complex interactions among sequences of states in the specifications lead to ineffective test generation [Cha99]. Full-predicate coverage tests state transitions independently and tests sequences of state transitions. Invalid sequences of state transitions can lead to typical faults. But faults due to valid sequences are not allowed. To check for these kinds of faults, transition-pair coverage can be used. It requires that pairs of state transitions be evaluated to recognize faults. Mutation testing is another specification-based technique used for testing software units. It is defined in terms of the code statements in an implementation and is similar to the code-based testing. Mutation testing was adapted for deriving tests from functional specifications to avoid shortcoming of code-based test generation. A specification for mutation testing is divided in two parts. One part is a state machine defined in terms of variables and description of the conditions under which variables may change value. The other part is logic constraints on valid execution paths. As mutation testing is an adaptation of code-based testing it is not as efficient as full-predicate or transition-based testing criterions [Ding99; BF98; Wil96].

In comparison of full-predicate testing, transition-pair specification-based technique fares better due to its ability of fault detection for valid sequences [Ding99]. This ability gives it the edge in fault detection ability which indicates that a strategy based on transition-pair specification strategy can help in development of an efficient approach for automated test generation. Such a strategy using this principle is discussed in section 2.6.

2.5 Test Automation Framework (TAF) Approach

As from the above discussion it can be inferred that transition-pair specification-based test generation technique can provide high fault detection capability resolving the need for efficient test generation method. However, such a technique needs to be modeled for automated test case generation. Use of formal specifications to provide a basis for test case generation has been in use [Eka91]. Goodenough and Gerhart may have been the first to claim that the testing based only on a program implementation is fundamentally flawed [GG75]. Gourlay developed a mathematical framework for specification-based testing [Gou83]. A model-based specifications approach constructs an abstract model of the system states and characterizes how a state is changed by abstract or concrete operations [CHJ88; CGDDTK96]. The state changes or events that affect the model using existing mathematical constructs like sets and functions define operations in the system. State transitions define relationships between sequences of states based on conditions of system state. Event specifications define certain conditions related to change in the system state. A test specification element can be defined as an input-to-output relation. An associated constraint is defined by conjunction of Boolean value relations which define constraints on the inputs associated with the input to output relation.

A drawback of such model-based specifications approach is underlying development process model does not support automatic test generation. A key challenge is to translate such specification-based test generation models into a form which is suitable for automatic test case generation by incorporating the ability to handle changes in

requirements by automatic changes in tests generated. Model transformations are typically required to transform model-based specifications into a form which supports test generation. Hierons described writing rules for Z specifications [Hie97] to support test case generation, but did not address specifications composed of combinations of specification techniques; particularly specifications composed using event specification techniques. Model transformation to support test generation interoperability is an important aspect for development of cost-effective and efficient automated test generation scheme. Blackburn [Bla98] described an approach for transforming the model-based specification into a form which can support test vector generation but failed to identify composition of function and state specifications. Thus, the drawback of ineffective and expensive translation of specification-based test generation models was the issue in realizing the goal of developing an efficient automated test generation method.

To overcome the drawback of the ineffective translation of specification-based test generation models for automatic test generation leading to inefficient use of resources the Test Automation Framework (TAF) approach was evaluated. The core capabilities underlying the TAF approach were developed in the late 1980s and were shown to be effective through use in support of FAA certifications for flight critical avionics systems [BB96]. Statezni described how this approach supports requirements-based test coverage mandated by FAA with significant life cycle cost savings [Sta00]. Safford presented results stating the approach reduced cost, effort and cycle time by eliminating requirements defects and automating test generation process [Saf00]. The National Institute of Standards and Technology (NIST) assessed TAF approach as the basis for a methodology and supporting toolkit to automate major aspects of security functional testing [BBNC01].

The benefits of this approach can be summarized as

- Better quality requirements for design and implementation thus eliminating rework in those phases and testing phase.

- Use of models for verification reduces the time spent on verification test planning by up to 50 percent [Saf00]
- Test generation from a verification model helps to eliminate most of the manual test creation effort.
- Use of verification models for test generation helps in early defect identification.
- Ability to plan a known level of requirement coverage for analysis purposes during test execution helps in adaptation of test execution for changes.

Thus, TAF can be used to improve fault detection which can lead to reduction in resource consumption. The key change from other automated test generation approaches is the introduction of verification models for development of automated test vectors. These models support automated means of identifying model defects and generating tests that are highly effective in verifying a system consistent with the model. The TAF translator transforms and expands Software Cost Reduction (SCR) [Hei95] specifications into a form supporting automatic test generation [BBNC01]. For model analysis and test generation, the model is transformed into a set of precondition/ post condition pairs referred as Test Specification Elements as described in Figure 1. Test vector generation is then carried out to produce a test vector for every test specification element [BBF97].

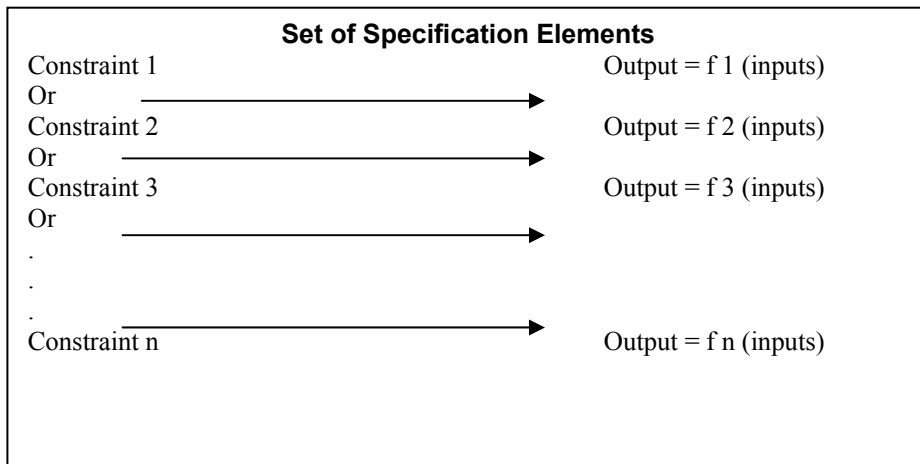


Figure 1. Test Specification Elements

Thus TAF approach can be used to generate automated test vectors in a cost effective way from test specifications. The use of this approach allowed us to provide a resource-efficient technique for automatic test case generation which can handle changes in requirements effectively.

2.6 Pairwise Strategy

Our goal is to provide a resource-efficient and effective approach for automated test generation. To be consistent with this goal, we have seen the benefits of specification based test generation to provide high fault detection capability and use of TAF approach to provide a cost-effective way for automated test generation. However another major issue affecting the automated test generation is generation of large number of test cases as specifications change. The motivation behind automated test generation, to provide a fast and effective way of test generation is almost nullified due to these large numbers of test cases. In this section we evaluate a transition-pair specification-based strategy called Pairwise in our effort to tackle issue of generation large number of test cases.

The notion of pairwise coverage of parameter values in software testing was first put forth by Mandl [Man85] for testing Ada compiler. Mandl used testing the types of operands as part of equations as an example in using pairwise coverage of parameters. Pairwise strategy considers pair of parameters for test generation. A possible trade off of pair wise between test coverage and the number of tests can be to determine a set of test configurations that test each pair wise combination of parameter values. This implies that for any two parameters of the system each combination of the possible values of the two parameters occurs in at least one test configuration. This is the basic principle of pairwise strategy.

This principle provides a well-defined level of test coverage, with a reduced number of system configurations. Thus issue of combinatorial explosion of generated test cases can be handled using pairwise strategy. Pair wise parameter coverage can detect any problem with pair wise compatibility of parameters, assuming the test suite for

any coverage has thorough coverage [Wil02]. In addition, it guarantees that all interfaces have been exercised as all combinations are covered at least once. These properties of pairwise strategy resolved the issue of generation of large number of test cases. However, pair wise parameter coverage cannot guarantee detection of problems involving a specific combination of three or more parameter values. The idea of using pairwise (or n-way) interactions among parameters as a coverage strategy for system testing has been useful in practice in a variety of situations [Wil96]. This strategy is illustrated with an example in the following section.

2.7 Illustration of Pairwise Strategy

Consider the following system of parameters to illustrate the concept of pairwise testing.

- Parameter A has values A_1 and A_2
- Parameter B has values B_1 and B_2
- Parameter C has values C_1 , C_2 and C_3

For parameters A and B, $\{(A_1, B_1), (A_1, B_2), (A_2, B_1), (A_2, B_2)\}$ is the only pairwise test set.

However, when we consider the three parameters A, B and C then there exist $A(2) * B(2) * C(3) = 12$ test cases with full dependence of parameter values on each other. Pairwise strategy states that each combination of valid values of a pair of parameters appears at least once in a test case. This provides partial dependence in the values of the parameters thus eliminating the redundancy.

When we consider parameters A, B and C a large number of pairwise tests exist depending on the heuristics applied. Example of such pairwise test suite is given.

- $\{(A_1, B_1, C_1), (A_1, B_2, C_2), (A_2, B_1, C_3), (A_2, B_2, C_1), (A_2, B_1, C_2), (A_1, B_2, C_3)\}$

Different test generation heuristics for pairwise testing exist. Two main heuristics are listed below.

- The strategy in [Coh97] starts with an empty test set and adds one test at a time. To generate a new test, the strategy produces a number of tests according to a greedy algorithm and then selects one that covers the most uncovered pairs.
- Another approach to generate pairwise test set is to use orthogonal arrays [Man85]. The original method of orthogonal arrays required that all parameters have the same number of values and that each pair of values be covered the same number of times [Man85].

2.8 Summary

We evaluated the code-based, interfaced-based and specification-based test generation approaches for choice of a test generation approach helpful in automatic test case generation. Specification-based test generation technique showed relatively better fault detection ability than code-based and interface-based test generation techniques. We also evaluated drawbacks of specification-based test generation for automated test generation. We discussed use of TAF approach to eliminate those drawbacks to provide a cost-effective automated test generation technique using specification-based test generation. We illustrated use of pairwise strategy to tackle the issue of generation of large number of test cases providing a good level of test coverage.

Our goal is to provide a resource-efficient automated test generation approach with high fault detection capability. To achieve this goal we enhance an automated test generation tool based on pairwise strategy which is described in Chapter 3. We enhance and assess the requirement model used by tool identifying the requirements in terms of logical entities representing the environment of the system under test. We also update the verification model of the tool which specifies requirements in terms of relations and constraints using TAF approach for test generation. The algorithm used for developing the tool is discussed in Chapter 3.

Chapter 3

In-Parameter Order Algorithm

Pairwise testing is a specification-based testing criterion that requires for each pair of input parameters of a system, every combination of valid values of these two parameters be covered by at least one test case. An algorithm based on pairwise strategy using greedy heuristics called In-Parameter Order algorithm was developed by Dr. K. C. Tai at North Carolina State University.

3.1 The In-Parameter (IPO) order strategy [Tai01]

The In-Parameter Order (IPO) method was developed by Dr. Tai [Tai01]. The method is based on the well-known greedy heuristics for set covers but is adapted to the interaction test coverage problem. The greedy heuristics for set covers is as follows: *add the subset that covers the most elements of the set to be covered* [Wil96]. Adaptations to this heuristics include checking subsets in decreasing order. For interaction among these subsets; to provide coverage of all such subsets, this involves checking each set configuration to see which one covers the most interaction elements. Unfortunately this increases the number of test configurations exponentially, so the direct application of the heuristics is not particularly useful.

Instead IPO applies this heuristics to choosing parameter values **individually**, instead of selecting an entire test configuration at once. It starts with a small problem of selecting the first two parameters, and creating $n_0 * n_1$ test configurations to cover all the combinations among those parameters. In this case, parameters are not ordered by the number of values which indicate that there is no assumption that $n_0 \geq n_1$. Then the solution is expanded in two dimensions. The first dimension is horizontal. Then another parameter is added and new values for the added parameter must be selected. These parameters are selected to cover the most number of pairs of parameter values as illustrated in the section 3.2. After this phase, there may still be uncovered pair wise combinations between the added parameter and the previous parameters. By using the greedy heuristic again, new test configurations are added to cover the missing combinations.

Thus for a system with two or more parameters, the IPO strategy generates a pairwise test set for the first two parameters, extends the test set to generate a pairwise test set for the first three parameters, and continue to do so for each additional parameter. The extension of the existing pairwise test set for an additional parameter contains following steps.

- **Horizontal growth**, which extends each existing test by adding one value of the new parameter [Tai01].
- **Vertical growth**, which adds new tests, if necessary, to the test set, produced by horizontal growth [Tai01].

The horizontal as well as vertical growth algorithms are explained in the following sections. Thus the result is the heuristic that is locally greedy in the horizontal and vertical dimensions. In this case vertical growth is locally optimal but fully optimal horizontal growth would be exponential in complexity [Tai01]. This is because entire test configurations are considered instead of just adding a single parameter value at a time.

3.2 An algorithm for Horizontal Growth [Tai01]

The horizontal growth algorithm in IPO can be described as follows:

Assume that T is a pairwise test set for parameters p_1, p_2, \dots, p_{i-1} . The horizontal growth of T for parameter p_i is to extend each test in T by adding a value of p_i [Tai01]. A high-level algorithm called IPO_H for growth of T for parameter p_i is shown in section 3.3.

An explanation of IPO_H involving the system described above can be given as follows.

$\{ (A_1, B_1), (A_1, B_2), (A_2, B_1), (A_2, B_2) \}$ is the only pairwise test set for A and B as described in section 2.6. Since C has three values C_1, C_2 and C_3 , these values can be added to the existing pairwise test set to extend it as $(A_1, B_1), (A_1, B_2)$ and (A_2, B_1) by adding C_1, C_2 and C_3 respectively.

The new extended tests are $(A_1, B_1, C_1), (A_1, B_2, C_2)$ and (A_2, B_1, C_3) . Thus the resulting set of missing pairs is $\{ (A_2, C_1), (B_2, C_1), (A_2, C_2), (B_1, C_2), (A_1, C_3), (B_2, C_3) \}$.

Now we need to choose one of C_1, C_2 or C_3 for the remaining pair (A_2, B_2) . If we add C_1 to it then the extended set will cover the two missing pairs namely (A_2, C_1) and (B_2, C_1) .

If we add C_2 to the (A_2, B_2) the extended set (A_2, B_2, C_2) will cover one missing pair namely (A_2, C_2) . If we add C_3 to the (A_2, B_2) the extended set (A_2, B_2, C_3) will cover only one missing pair namely (B_2, C_3) . So we choose (A_2, B_2, C_1) as our fourth choice of test as it covers more missing pairs thus reducing number of tests.

Still the following pairs are not covered: $(A_2, C_2), (A_1, C_3), (B_1, C_2)$ and (B_2, C_3)

Vertical growth algorithm as illustrated in section 3.4 can be used to cover these pairs.

3.3 Algorithm IPO_H(T, p₁) [Tai01]

// T is a test set. But T is also a list with elements in arbitrary order.

{Assume that the domain of p₁ contains values v₁, v₂,, and v_q;

Let $\Pi = \{ \text{pairs between values of } p_1 \text{ and values of } p_1, p_2, \dots \text{ and } , p_{i-1} \};$

If (|T| <= q)

{

for 1 <= j <= |T|, extend the jth test in T by adding value v_j and remove from Π pairs covered by the extended test;

}

else

{

for 1 <= j <= q, extend the jth test in T by adding value v_j and remove from Π pairs covered by the extended test;

for q < j <= |T|, extend the jth test in T by adding one value of p₁ such that the resulting test covers the most number of pairs in Π , and remove from Π pairs covered by the extended test;

}

}

3.4 An algorithm for Vertical Growth [Tai01]

To illustrate the vertical growth algorithm let us consider that the horizontal growth for parameter p_i has produced a test set T for p₁, p₂ and p_i. Let us consider that Π be the set of pairs not covered by T. Each pair in Π contains a value of p_i and a value of p₁, p₂, or p_{i-1}. Assume that | Π | > 0. The vertical growth T according to Π is to construct new tests for covering pairs in Π and add these new tests to T. Thus

resulting T is a pairwise test set for p_1, p_2, \dots, p_i . In the vertical growth algorithm “-“ represents the unspecified or any value of the parameter.

After the complete run of the IPO_V (Vertical Growth) algorithm the T (Test Set) may contain “-“ values. If p_i is the last parameter, each “-“ value for p_k , such that $1 \leq k \leq i$, is replaced by any value of p_k . Otherwise, these “-“ values are replaced by parameter values in the horizontal growth algorithm for p_{i+1} as follows.: *Assume the value y of p_{i+1} is chosen for the horizontal growth of a test that contains “-“ as the value for p_k , $1 < k < i$. If there are uncovered pairs involving y and some values of p_k , then “-“ for p_k is replaced by any value of p_k .* [Tai01]

Again considering the example system described above we can show that the horizontal growth of parameter C generates the following four tests. $(A_1, B_1, C_1), (A_1, B_2, C_2), (A_2, B_1, C_3)$ and (A_2, B_2, C_1) and that these four tests do not cover following four pairs : $(A_2, C_2), (A_1, C_3), (B_1, C_2)$ and (B_2, C_3) . Now we use IPO_V algorithm to construct a new test to cover these four pairs. To cover (A_2, C_2) we generate test $(A_2, -, C_2)$. To cover (A_1, C_3) we generate $(A_1, -, C_3)$. To cover (B_1, C_2) we change $(A_2, -, C_2)$ to (A_2, B_1, C_2) without adding a new test. To cover (B_2, C_3) we change $(A_1, -, C_3)$ to (A_1, B_2, C_3) similarly without adding a new test. Thus only two new tests are generated to cover the four pairs not covered by horizontal growth. So generated pairwise test will have six tests.

3.5 Algorithm IPO_V(T, Π) [Tai01]

{ Let T be an empty set ;

for each pair in Π

{ assume that the pair contains value w of p_k , $1 \leq k < i$, and value u of p_i ;

if (T contains a test with “-“ as the value of p_k and u as the value of p_i)

modify this test by replacing “-“ with w ;

else

```

    add a new test to  $T$  that has  $w$  as the value of  $p_k$ ,  $u$  as the value of  $p_l$ ,
    add “-” as the value of every other parameter;
};
 $T = TU T$ ;
};

```

As illustrated above the IPO strategy performs well according to the sizes of generated test sets by generating fewer test cases to cover all combinations. Also, IPO strategy can be easily extended to reuse the existing test sets when systems are modified due to changes in input parameters and/or their values as illustrated in earlier sections [Tai01]. This provides a very useful cost-effective way of automatic test generation, accounting for requirement changes in test system. Thus IPO strategy provides a way to handle issue of large number of test cases with high adaptability for changing requirements in a cost-effective way. This is in accordance with our goal to provide a resource efficient way of automatically generating manageable number of effective test cases. So we enhanced PairTest, an automated test generation tool based on IPO strategy, to provide a stable open-source automated test generation tool. The PairTest is described in section 3.6.

3.6 PairTest Test Generation Tool

A test case generation tool can prove a boon to the testers considering tough deadlines and deliverables in today’s industry. It can help testers to save some time by automatic generation of test cases to do decent amount of testing. Studies show that total testing time is just 30% of what it should be [BF98].

Test development has been known as a tedious and time-consuming process. All test generation tools rely on algorithms to generate the tests. Testing complex systems have a number of challenges: test cases take too long to create, too long to automate, too long to run and too long to verify. Even if test automation is introduced in the testing process, tools generate large number of test cases and make it very difficult to maintain such large number of test cases. Such test generation tools are expensive and

have low tolerance to incorporate requirement changes in test generation. However, these tools are relatively more thorough, accurate and faster than a human tester using the same algorithm.

In an effort to provide solution to these problems with test generation tools, PairTest, an automated test generation tool based on IPO strategy, was developed in Java [Sun] by Dr. Tai [Tai01]. PairTest version 1.0 was released in 1998. However, PairTest had crashing bugs. Also conversion of a large test system in terms of PairTest parameters was tedious and time consuming. To achieve our goal, we modified the PairTest tool to facilitate error free working of the tool. One major issue was the failure of the tool for incorporation of the ‘-‘ values or ‘don’t care’ values during test generation. We modified the tool such that the tool can be used to generate the test set and extend it for generation of new test set to accommodate the change in parameters. We also added a text-based version of PairTest in addition to the GUI version. PairTest was enhanced to add the feature of adding Parameters, Relations as well as Constraints from files which made adding large data very easy. Many feature such as parameter addition, test generation etc. were modified in the tool to correct the crashing bugs in the tool and make it more reliable. Flexibility of the tool was enhanced by addition of the text based version of the tool which helps to run in the tool in different operating environments.

The enhanced version 1.1 of PairTest is available as an open-source automated test generation tool.

The benefits of the PairTest tool can be summarized as follows.

- Automates tedious aspects of test case design.
- It can generate a complete set of test cases with respect to their source.
- It can identify certain kinds of defects, non-working items or software that does not conform to a stored specification.

3.7 Test Generation tools

From the industrial experience it is observed that test generation can become a bottleneck for product in the competitive nature of the industry by increasing the time required for the product to reach markets. In the past two decades, various test development automation tools have been introduced and gradually accepted by designers and test engineers to automate dozens of tasks that are essential for developing adequate tests. The test development automation tools can generally be classified into four categories as below [BB96].

- Design-for-testability (DFT) tools
- Test pattern generation tools
- Pattern grading tools
- Test program development and debugging tools.

DFT tools are used early in the design process and are often used by the design engineers instead of test engineers. Tools in this category include testability analysis tools, design rule checkers and test-structure insertion tools. Testability analysis tools report areas of a design that could cause testability problems (e.g. areas that have poor controllability or poor observability). These tools provide useful guidance in the redesign for improved testing ability [Zin97]. Different test structures (boundary scan, internal scan, memory Built-In-Self-Test (BIST) and logic BIST) impose slightly different design rules. Violations of the test case design rules have to be fixed before test structure can be inserted to avoid changes in the testing ability of tools for test generation. Test pattern generation tools attempt to automatically generate high-quality tests for specified fault models. Automatic test pattern generator (ATPG) is one of the most used pattern generation tool. There are a number of automatic test case generation tools available but these can suffer from combinatorial explosions in the number of possibilities to test [Wat96].

Another commercial tool, Automatic Efficient Test Case Generator (AETG) from Telcordia (formerly known as Bell Labs) uses a pairwise test case generation

technique to generate tables of test vectors, which a test driver can then execute immediately. Code coverage is then used to indicate missing functionality from AETG's model. AETG specifically promises to generate fewer test cases than other tools. This results in less time to test the product and verify results [Coh95]. The term "test vectors" is used as opposed to test cases as in most practical situations, AETG does not create actual test cases, but a table of input values needed to execute the desired test cases.

Test case generation in AETG is based on the greedy heuristics of pairwise coverage of parameter values in software testing process which was introduced by Mandl [Man85] for testing Ada compiler. This is the same technique used to develop IPO algorithm used to implement PairTest tool. Test generation ability of PairTest was effectively assessed in industrial as well as controlled laboratory environment. The comparison is discussed in section 3.9.

3.8 Automatic Efficient Test Case Generator (AETG)

The first report on AETG [Coh94] introduces the idea of a combinatorial approach to test generation based on statistical experimental design techniques. The use of the system is then described such that the user has to enter the set of fields (parameter values) and relations (dependencies among parameters). AETG can also generate n-way interactions and not just pair wise interactions. As for how AETG determines its designs, Telcordia has declared those algorithms as proprietary. However, there are clues that greedy heuristics may be involved as shown in section 3.8.1 [Coh96; Wil96]

While AETG is a tool, it is basically just a mathematical engine. Careful planning and design is extremely important to achieving success hence it is the methodology of test generation that is most important. In order to understand how AETG creates its test

vectors, a brief introduction to the basic constructs is necessary. The two main components of AETG are Relations and Fields [Coh96]. In terms of testing, a field is usually considered to be an input, and relations are simply how these fields or inputs interact with each other. In essence, a tester must determine the different possible values for each field in a relation. After these requirements have been captured, AETG will then proceed to find the minimal set of test vectors to cover a subset of all interactions. AETG is also able to generate test vectors covering 3-way or n-way interactions.

The test configurations in AETG are generated using greedy heuristic, where the objective is to find the maximum number of uncovered pairs at any stage [Coh97]. This helps to handle the issue of combinatorial explosion of test cases. PairTest also handles this issue in similar way. Also there is a random element to the heuristic and therefore the exact results are not predictable in advance. Code coverage is then used to indicate missing functionality from AETG's model.

3.8.1 AETG's algorithm to add new configuration [Coh94]

Algorithms for AETG are described as proprietary by Telcordia. The following algorithm is depicted in the first report [Coh94] of AETG. It gives a slight idea how test generation is implemented in AETG.

- Choose a parameter and a value for that parameter, such that the parameter value appears in the greatest number of uncovered pairs
- Choose a random order for remaining parameters.
- For each parameter, for each possible value for the new parameter, choose the value that covers the most pairs with the previously assigned parameters.

The above algorithm is almost similar as the IPO algorithm illustrated in section 3.1 [Wil96] as it tries to cover the most uncovered pairs with previously assigned

parameters. This helps in reduction of the generated number of test cases thus helping to contain combinatorial explosion problem in generation of test cases which is based on pairwise strategy of test generation. Actual algorithm of AETG is proprietary to Telcordia.

3.9 PairTest Evaluation

Our goal was to develop a resource-efficient approach with high fault detection capability for automated test generation. The implementation of such a method is seen in terms of PairTest, a tool for automated test generation. To verify success of this goal it is necessary to illustrate test generation ability of PairTest tool in industrial environment by analysis with different case studies.

The AETG Web Service offers facilities for creating and editing AETG system input, for using the AETG algorithms to create test sets, described in Chapter 3, and for viewing test sets created by the AETG System. We are thankful to Mr. Christopher Lott from Telcordia for providing us with a free trial account of AETG for research purposes. AETG trial versions can be obtained from <http://aetgweb.argreenhouse.com>. The AETG Web service generates efficient test sets by creating a small number of test cases such that all pairwise, triple, or n-way interactions of the field values are covered. AETG can generate n-way test sets but as PairTest tool based on IPO algorithm generates test sets by using 2-way interactions so we will be focusing on on pairwise test generations by PairTest.

3.10 Test Case Parameters and Specifications

The AETG system input consists of fields, field values, compounds and relations, which are stored together as a Test Specification (TS). A relation is composed of fields and compounds. The user specifies valid and invalid values for each field or compound in a relation. Relations also specify the constraints, which specify which values of the fields appear together [Coh96]. Edit parameter page from PairTest is

displayed in Figure 2. The screenshot of edit parameter page for AETG is displayed in Figure 26 in Appendix B.

AETG has one more parameter field for specifying the values for continuous parameter that is absent in PairTest. A compound is composed of fields and (optionally) other compounds. A compound value is named tuple [Coh96]. A tuple is an array that has at least one value for each field or compound in the compound.

PairTest test generation strategy can be described as

- PairTest has a GUI as well as a command line-based option for editing as well as creating a new test plan which runs on any machine with JDK 1.2[Sun] or higher version.
- PairTest allows the parameter values to be entered while entering the parameters. So one does not have to wait till relations are created containing respective parameters to enter respective parameter values.
- In PairTest selecting a common parameter in different relations is possible. This allows use of multiple parameters in relations. This allows to prevent situation where a parameter can't be included which is selected in other relation unless all the other parameters in that relation are included in the new relation.

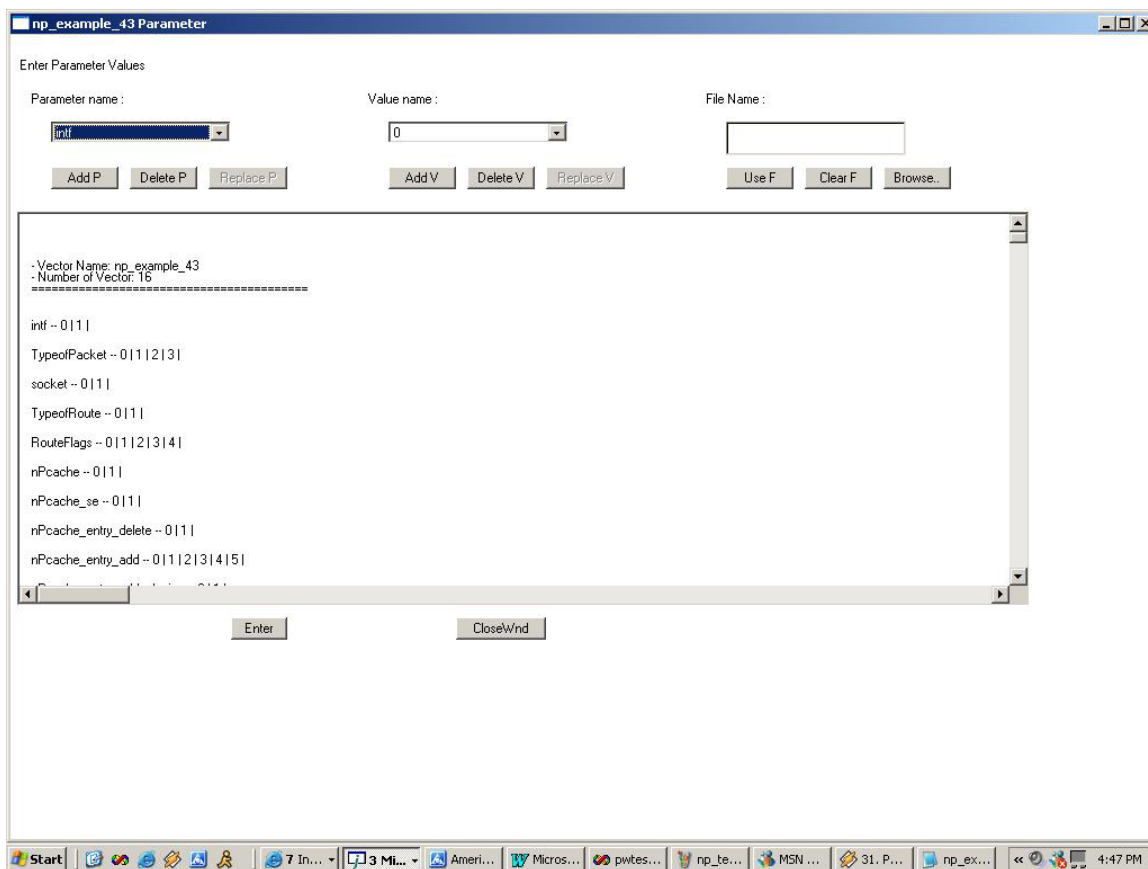


Figure 2. Edit Parameter Window for PairTest (GUI Version)

3.11 Relations

At least 2 steps are necessary to create a new relation.

- Selecting the fields and compounds required for creating the relation.
- Entering the values for the selected fields and choosing the named tuples.

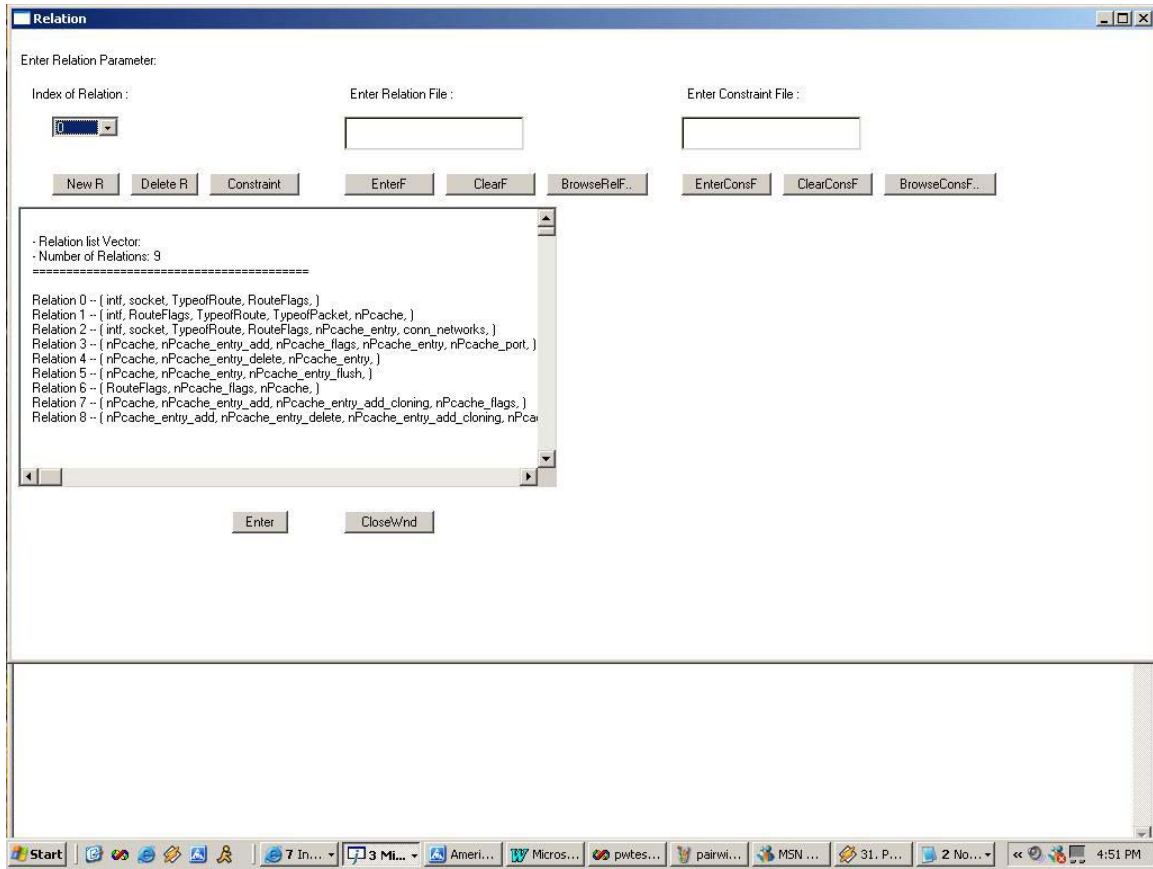


Figure 3. Edit Relations window for PairTest (GUI version)

In addition to select valid values of fields for the relations, AETG also lets set the invalid values of the fields in the relation [Coh96]. While creating relations, values for fields are entered. A number of constraints can be added for a given relation. Relation editing for PairTest is shown in Figure 3. The screenshot for edit relation page for AETG is displayed in Figure 27 in Appendix B.

In PairTest relations are selected by choosing the parameter names to add in a new relation. There is no restriction on duplicating parameters in multiple relations.

3.12 Constraints

Any number of constraints can be added to a single relation. Before a constraint is entered at least one valid value must be entered for each field or parameter. Adding constraint operation for PairTest is shown in Figure 4.

The overview of syntax for valid constraints can be given as follows.

Ex: *if field_i op value₁ then field_j op value₂*

In AETG the constraints can be entered as logical statements. The logical operators as =, !=, <, <=, > and >= can be used for defining a constraint statement. The screenshot for edit constraint page for AETG is displayed in Figure 28 in Appendix B.

In PairTest the constraints do not involve logical operators. Moreover they are defined as a static constraint by choosing one of the values of the parameters in relation with the other parameters. In order to cover the parameter value range PairTest allows a parameter value to be denoted as (*) which signifies parameter can take any defined value.

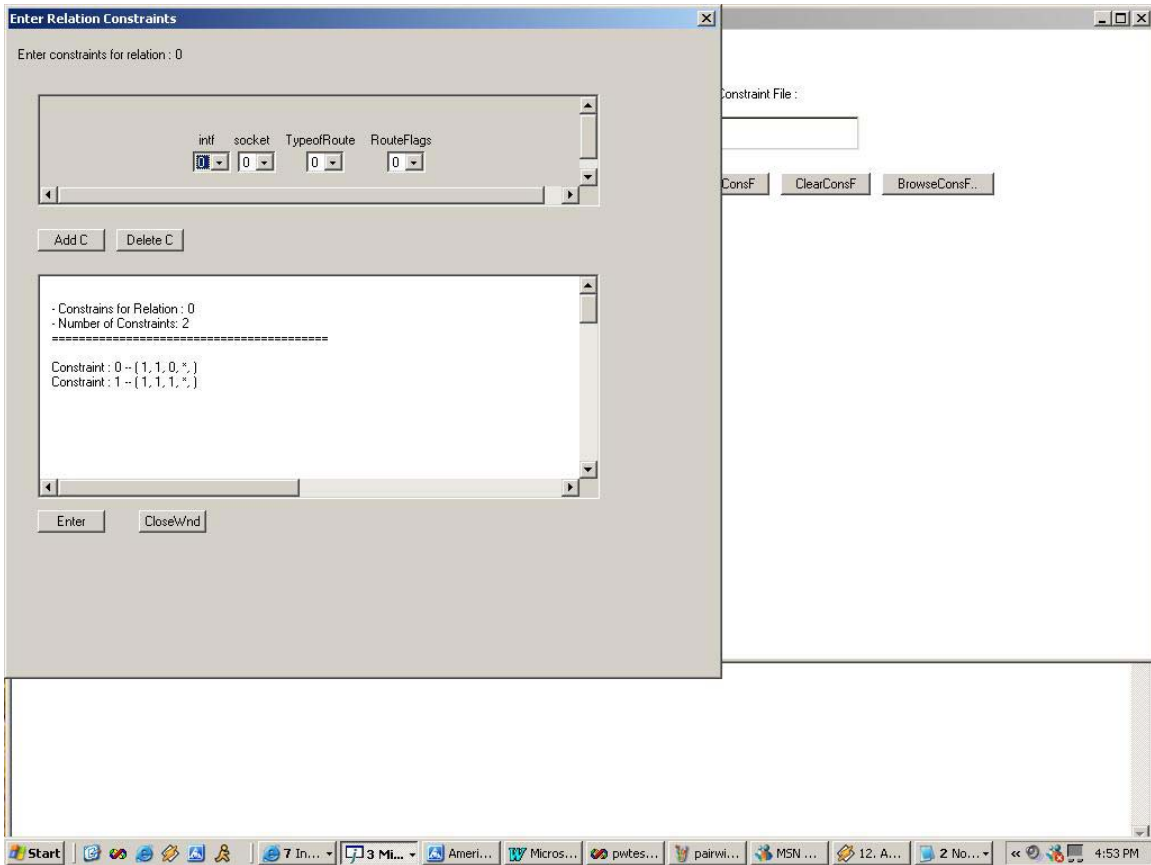


Figure 4. Edit Constraints window in PairTest (GUI Version)

3.13 Test Set

The AETG system generates test cases based on the fields, compounds, and relations in a test specification [Coh94]. When any test set is generated, the test case generation process for current specifications is displayed in PairTest.

In PairTest the test set can be generated by two options.

1. Replace: When the test set is generated with this option an existing file is modified with the new values of parameters, relations and constraints.
2. Extend: This option generates test set by combining the existing specifications with the new added specifications and generates the new test set.

This allows PairTest to generate new test sets as well as modify existing test sets without much effort.

3.14 Comparison of PairTest

3.14.1 Performance issues

Table below shows time information for test sets generated by PairTest. The execution time information was collected when PairTest was compiled and run on a PC with Intel 450 MHz Pentium II processor, Windows 98 and JDK 1.2.2.

Result of PairTest with N 4-value parameters

| | | | | | | | | | |
|---------------------|------|------|------|------|------|------|------|------|------|
| N(# of parameters) | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 |
| S (# of tests) | 31 | 34 | 41 | 42 | 48 | 54 | 59 | 61 | 66 |
| T (Time in seconds) | 0.25 | 0.37 | 0.42 | 0.64 | 0.77 | 0.98 | 1.37 | 1.81 | 2.23 |

Results of PairTest with 10 parameters each having d values

| | | | | | | | |
|---------------------|------|------|------|------|------|------|--|
| D (# of values) | 5 | 10 | 15 | 20 | 25 | 30 | |
| S (# of tests) | 47 | 169 | 361 | 618 | 956 | 1355 | |
| T (time in seconds) | 0.19 | 0.41 | 0.72 | 1.89 | 3.75 | 5.16 | |

Sizes of tests generated by AETG and PairTest [Tai02]

| | | | | | | | |
|----------|----|----|----|----|----|-----|--|
| System | S1 | S2 | S3 | S4 | S5 | S6 | |
| AETG | 10 | 17 | 33 | 26 | 13 | 191 | |
| PairTest | 8 | 18 | 36 | 25 | 15 | 216 | |

S1 : 4, 3-value parameters

S2 : 13, 3-value parameters

S3 : 61, parameters(15 4-value , 17 3-value , 29 2-value parameters)

S4 : 75, parameters (1 4-value , 39 3-value, 35 2-value parameters)

S5 : 100, 2-value parameters

S6 : 20, 10-value parameters

These values are results of the tests run 25 times to take average value of the GUI-based as well as command line version of PairTest. Also the results vary along with modifications made to relation and constraints vectors.

Comparison of other performance related issues are given as follows.

- Ease of use

PairTest with its runtime options for GUI-based or command line-based versions provides easy entering as well as editing existing parameters, relations or generating a

new test set. PairTest allows change in specifications after a test set generation to generate a new test set with the edited specifications.

Also the GUI as well as command line versions of PairTest provide user-friendly interfaces, which allow users to generate test sets and edit them as per their needs easily.

- While entering constraints, PairTest allows the parameter value to be specified as (*) which signifies all the values of the parameter. So there is no need to enter the constraint for each value of parameter which saves the tedious work of entering constraint for each value of parameter.
- PairTest also provides an option of adding parameters, relations and constraints from the files automatically. These files must be in specific format specified in the help section of the PairTest. This saves the tedious work of entering the parameters/relations/ constraints (saves much work when there are more than 100 parameters) and increases speed of PairTest by some extent.
- Parameter values need to be entered when the parameter is selected in a relation. So when a parameter appears in duplicate relations its values need to be entered again. In case of PairTest it allows the parameter values to be entered at the instant parameter is entered. So relations can choose parameter values without need for re-entering them.
- Test set generation in these tools is done by the participation of parameters in the relations. So PairTest does allow parameters to participate in multiple relations without the whole set of parameters constituting one relation is being repeated in the new relation.

3.14.2 Efficiency considerations

Evaluating and comparing the efficiency of the PairTest algorithm is a difficult task. It has specific ways of specifying the parameters, their values as well as relations and constraints though its underlying strategy is pairwise.

Following examples illustrates and compare the efficiency of PairTest.

Case Study – 1

The lists of parameters, relations and corresponding constraints are displayed in Figures 5 and 6. It is not possible to list all the test cases so a snapshot of the test cases is provided. For the given parameters and relations PairTest generated 336 test cases where as AETG generated 326 test cases.

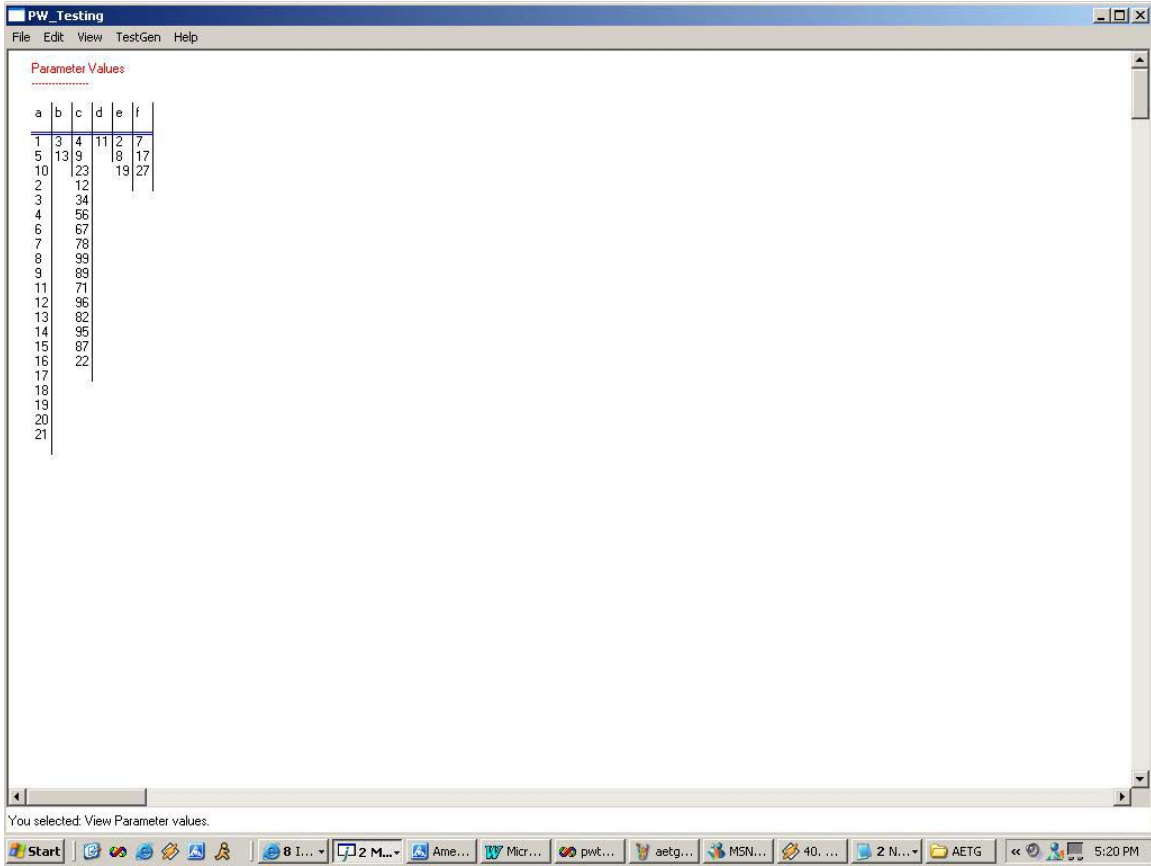


Figure 5. Parameter values

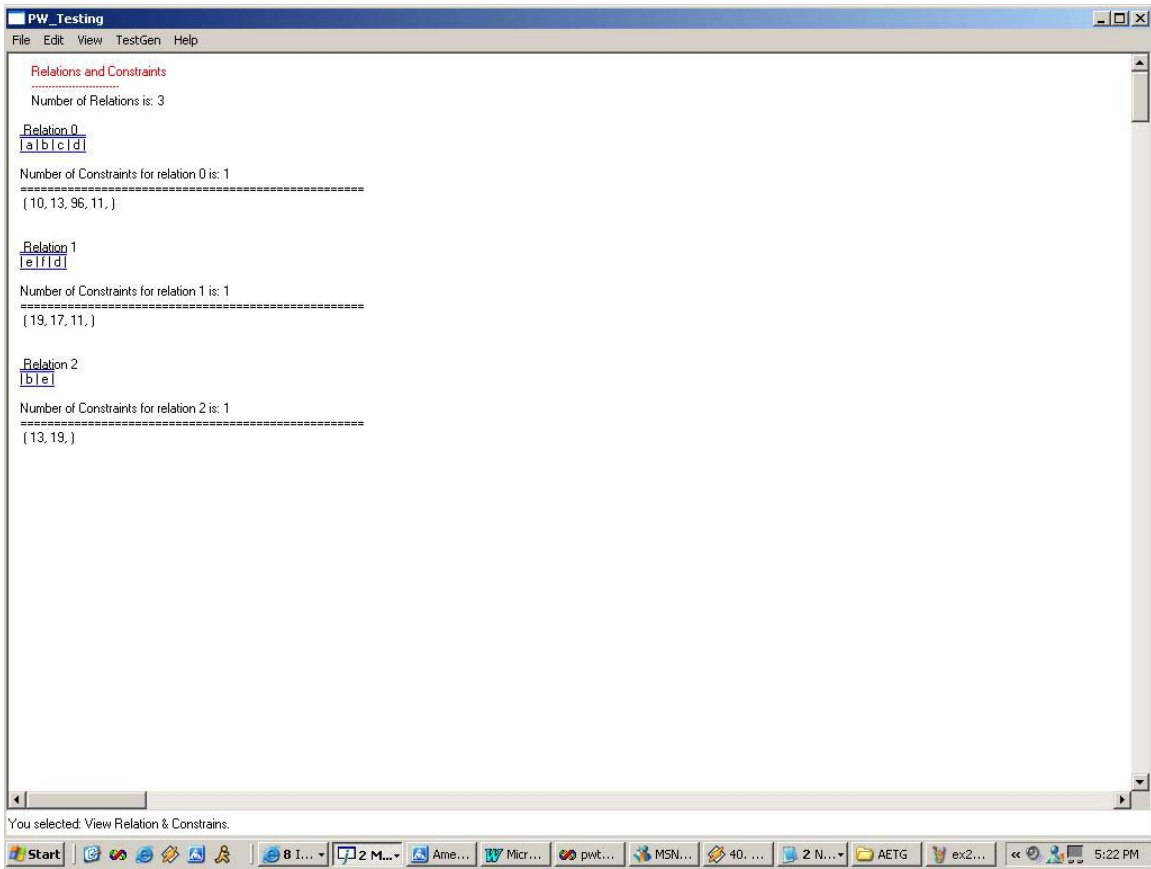


Figure 6. Relations

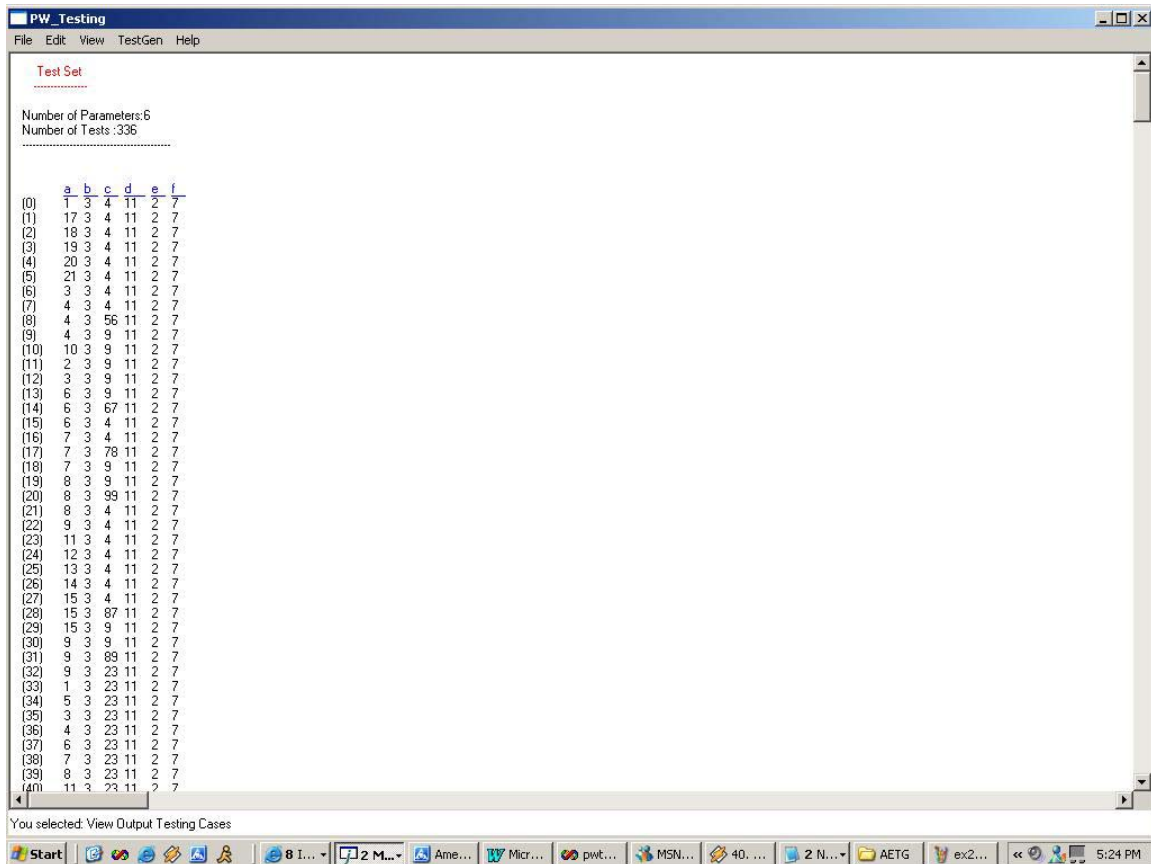


Figure 7. Test Set using PairTest

The PairTest file as well as AETG test set is displayed in Appendix B. Screenshot of PairTest test cases can be seen in Figure 7.

By comparing the two test sets we come to a conclusion that there is some difference in the number of test cases generated by AETG as well as PairTest. The basic difference lies in that fact that AETG algorithm can be used to add parameters n-way to generate test cases where as PairTest uses only pairwise addition of test parameters. In the above case study numbers of test cases generated by PairTest are slightly higher than AETG. This is due to the use of (*) Any values allowed by PairTest in defining constraints.

The analysis of the efficiency of the test generation abilities of PairTest is discussed in the section ahead.

3.14.3 Case Study -2: Network Processor Example (Practical Use)

Using the author's experience as a co-op in in the area of network processors we tried to test a system of network processors PairTest.

The system under test involved a network processor having 24 ports, each of which could be connected to a different network. Network processor was running vxWorks [vx] operating system. The software developed kept track of all the networks connected to the network processor. It also listened on a routing socket to keep track of any routing packet received by the kernel and process it. Network processor then dynamically added the necessary routes in its forwarding cache to keep it updated. Routes added by the kernel by cloning some other network route also needed to be maintained by the forwarding cache of the system.

Following types illustrate sample of route changes in the system.

- Routes can be network or host routes.
- Routes can be cloning/cloned/static routes.
- Routes can be added for the system in which network processor is on
 1. Local Networks
 2. Remote Networks
 3. CPU Hosted
- Route changes can involve addition, deletion or information transfer.

The system is converted to give inputs to AETG as well as PairTest. The conversion of the system to parameters is listed below.

Interface : intf
0 = Down
1 = UP

Type of Routing Packet : TypeofPacket
0 = RTM_GET
1 = RTM_ADD

2 = RTM_DELETE
 3 = Default

Socket Connection : socket
 0 = Unsuccessful
 1 = Successful

Route Type : TypeofRoute
 0 = Network
 1 = Host

Route Flags : RouteFlags
 0 = Static
 1 = host
 2 = Network
 3 = Cloning
 4 = Cloned

nP Forwarding Cache : nPcache
 0 = Invalid
 1 = Valid

nP Search Engine : nPcache_se
 0 = Absent
 1 = Present

nP Cache Delete Entry : nPcache_entry_delete
 0 = Unsuccessful
 1 = Successful

nPcache Add Entry : nPcache_entry_add
 0 = Remote Network
 1 = Remote Host
 2 = Local Network
 3 = Local Host
 4 = CPU Hosted Network
 5 = CPU Host

nPcache Cloned Route : nPcache_entry_add_cloning
 0 = Not cloned
 1 = Cloned Route

nPcache Flags : nPcache_flags
 0 = nP_Network
 1 = nP_Host

2 = nP_Valid

nPcache Entry Type : nPcache_entry
0 = Remote Network Entry
1 = Local Network Entry
2 = CPU Hosted Entry

nPcache Ports : nPcache_port
0 = Port not added
1 = Port added

Connected Networks : conn_networks
0 = No Network connected
1 = Networks Connected

nPcache Events : nPcache_event
0 = Add event
1 = Remove Event
2 = Update Event
3 = Flush Event
4 = Delete Event
5 = Destination Unknown Event
6 = Source Mismatch Event

nPcache Flush : nPcache_flush
0 = Unsuccessful
1 = Successful

3.14.3.1 Case Study -2: PairTest Testing

To generate the test set using PairTest these parameters and their respective values were entered. The dependencies among these parameters are converted in terms of relations and constraints, which are listed below.

PairTest File

PARAMETERS

Number of Parameters :16

param:intf-0|1|

param:TypeofPacket-0|1|2|3|

param:socket-0|1|
param:TypeofRoute-0|1|
param:RouteFlags-0|1|2|3|4|
param:nPcache-0|1|
param:nPcache_se-0|1|
param:nPcache_entry_delete-0|1|
param:nPcache_entry_add-0|1|2|3|4|5|
param:nPcache_entry_add_cloning-0|1|
param:nPcache_flags-0|1|2|
param:nPcache_entry-0|1|2|
param:nPcache_port-0|1|
param:conn_networks-0|1|
param:nPcache_event-0|1|2|3|4|5|6|
param:nPcache_entry_flush-0|1|

RELATIONS

Number of Relations :9

relation:0-(intf,socket,TypeofRoute,RouteFlags,)

CONSTRAINTS

Number of Constraints :2

constraint:0-<1,1,0,*,>

constraint:1-<1,1,1,*,>

relation:1-(intf,RouteFlags,TypeofRoute,TypeofPacket,nPcache,)

CONSTRAINTS

Number of Constraints :3

constraint:0-<1,1,1,* ,1,>

constraint:1-<1,0,0,* ,1,>

constraint:2-<1,4,1,* ,1,>

relation:2-

(intf,socket,TypeofRoute,RouteFlags,nPcache_entry,conn_networks,)

CONSTRAINTS

Number of Constraints :1

constraint:0-<0,*,*,*,*,*,>

relation:3-

(nPcache,nPcache_entry_add,nPcache_flags,nPcache_entry,nPcache_port,
)

CONSTRAINTS

Number of Constraints :6

constraint:0-<1,0,0,0,*,>

constraint:1-<1,1,0,0,*,>

constraint:2-<1,2,1,1,*,>

constraint:3-<1,3,1,1,*,>

constraint:4-<1,4,2,2,*,>

constraint:5-<1,5,2,2,*,>

relation:4-(nPcache,nPcache_entry_delete,nPcache_entry,)

CONSTRAINTS

Number of Constraints :0

relation:5-(nPcache,nPcache_entry,nPcache_entry_flush,)

CONSTRAINTS

Number of Constraints :1

constraint:0-<1,*,1,>

relation:6-(RouteFlags,nPcache_flags,nPcache,)

CONSTRAINTS

Number of Constraints :4

constraint:0-<0,0,1,>

constraint:1-<1,0,1,>

constraint:2-<2,1,1,>

constraint:3-<4,2,1,>

relation:7-

(nPcache,nPcache_entry_add,nPcache_entry_add_cloning,nPcache_flags,)

CONSTRAINTS

Number of Constraints :2

constraint:0-<1,2,1,1,>

constraint:1-<1,3,1,1,>
relation:8-(nPcache_entry,nPcache_event,)
CONSTRAINTS
Number of Constraints :3
constraint:0-<0,5,>
constraint:1-<1,6,>
constraint:2-<2,6,>

After entering these relations and constraints the test set was generated which is listed below:

Test Set by using PairTest :

[TESTSET]
NUM OF VARS:16
NUM OF TESTS:39
0:intf(0:0,1:1,)
1:TypeofPacket(0:0,1:1,2:2,3:3,)
2:socket(0:0,1:1,)
3:TypeofRoute(0:0,1:1,)
4:RouteFlags(0:0,1:1,2:2,3:3,4:4,)
5:nPcache(0:0,1:1,)
6:nPcache_se(0:0,1:1,)
7:nPcache_entry_delete(0:0,1:1,)
8:nPcache_entry_add(0:0,1:1,2:2,3:3,4:4,5:5,)
9:nPcache_entry_add_cloning(0:0,1:1,)
10:nPcache_flags(0:0,1:1,2:2,)
11:nPcache_entry(0:0,1:1,2:2,)
12:nPcache_port(0:0,1:1,)

```

13:conn_networks(0:0,1:1,)
14:nPcache_event(0:0,1:1,2:2,3:3,4:4,5:5,6:6,)
15:nPcache_entry_flush(0:0,1:1,)
0 0 x x x x x x x x x x x x
0 1 x x x x x x x x x x x x
0 2 x x x x x x x x x x x x
0 3 x x x x x x x x x x x x
0 x 0 x x x x x x x x x x x
0 x 1 x x x x x x x x x x x
0 x x 0 x x x x x x x x x x
0 x x 1 x x x x x x x x x x
0 x x x 1 x x x x x x x x x
0 2 x x 0 0 x x x x x x x x
0 3 x x 0 1 x x x x x x x x
0 1 1 1 0 x x x x x x x x x
0 1 1 0 2 x x x x x x x x x
0 2 1 0 4 x x x x x x x x x
0 0 1 1 3 x x x x x x x x x
1 1 1 0 1 x x x x x x x x x
0 2 1 0 1 0 x 0 2 1 1 1 x x x
0 3 1 1 1 0 x 0 3 0 0 2 x x x
0 0 0 1 1 1 x 1 2 0 2 0 x x x
0 0 1 x 2 0 x x 4 0 0 0 1 0 x x
0 3 1 x 2 0 x x 5 0 1 0 1 1 x x
1 3 0 0 3 1 0 1 1 0 1 2 0 1 0 0
1 1 0 0 4 1 0 0 4 1 1 2 0 0 2 0
1 0 0 0 4 1 0 0 3 1 0 0 0 1 2 0
1 0 0 1 0 1 0 0 5 1 2 1 0 1 2 0
1 2 0 1 2 1 0 0 0 1 2 1 1 1 0 0
1 3 0 1 4 0 0 1 1 1 0 1 1 0 1 1
1 2 0 0 3 0 0 0 0 0 0 0 0 0 1 1

```

1 1 0 0 3 0 0 0 0 0 2 2 0 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0
1 0 0 0 4 0 0 0 1 0 2 0 0 0 3 0
1 0 0 0 3 0 0 0 2 0 0 1 0 0 3 0
1 0 0 0 2 0 0 0 5 0 0 2 0 0 4 0
1 0 0 0 0 0 0 0 2 0 0 2 1 0 5 0
1 0 0 0 0 0 0 0 0 0 1 2 1 0 3 0
1 0 0 0 1 0 0 0 3 0 1 1 1 0 4 0
1 0 0 0 1 0 0 0 3 0 2 0 0 1 4 0
1 0 0 0 0 0 0 0 4 0 2 1 0 0 5 0

x = Any

The test set as viewed in the GUI version of PairTest is displayed as shown below in Figure 8.

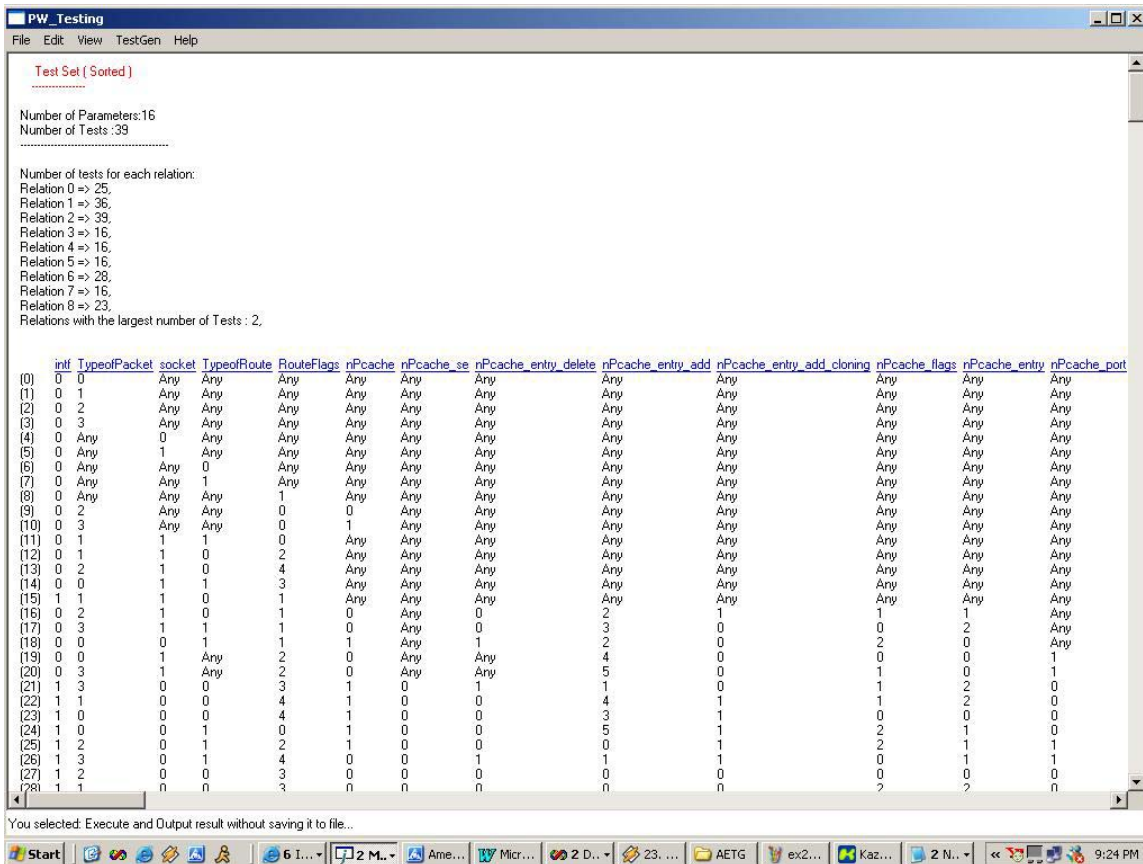


Figure 8. Test Set from PairTest (Network Processor Example)

3.14.3.2 Case Study-2 :AETG Testing

The test set displayed below is generated using AETG for research purposes of only this thesis. This test set is generated using AETG web and reprinted with specific permission.

* AETG Web, Copyright © 2003, Telcordia Technologies; reprinted with specific permission

The AETG specifications

View Test Specification 'NP_EX'

```

1 # Spec file generated by the AETG Web Service.
2
3 # Fields
4 field conn_networks;
5 field intf;
6 field nPcache;
7 field nPcache_entry;
8 field nPcache_entry_add;
9 field nPcache_entry_add_cloning;
10 field nPcache_entry_delete;
```

```

11 field nPcache_entry_flush;
12 field nPcache_event;
13 field nPcache_flags;
14 field nPcache_port;
15 field nPcache_se;
16 field RouteFlags;
17 field socket;
18 field TypeofPacket;
19 field TypeofRoute;
20
21 # Compounds
22
23 # Relations
24
25 R1 rel 2 {
26 RouteFlags : 0 1 2 3 4 ;
27 TypeofPacket : 0 1 2 3 ;
28 TypeofRoute : 0 1 ;
29 conn_networks : 0 1 ;
30 intf : 0 1 ;
31 nPcache : 0 1 ;
32 nPcache_entry : 0 1 2 ;
33 nPcache_entry_add : 0 1 2 3 4 5 ;
34 nPcache_entry_add_cloning : 0 1 ;
35 nPcache_entry_delete : 0 1 ;
36 nPcache_entry_flush : 0 1 ;
37 nPcache_event : 0 1 2 3 4 5 6 ;
38 nPcache_flags : 0 1 2 ;
39 nPcache_port : 0 1 ;
40 nPcache_se : 0 1 ;
41 socket : 0 1 ;
42 if intf =1 then socket =1;
43 if socket = 1 then TypeofRoute = 0 1;
44 if intf = 1 then nPcache = 1;
45 if TypeofRoute = 1 then RouteFlags = 0 4;
46 if nPcache =1 then nPcache_se = 1;
47 if nPcache = 1 then nPcache_entry_delete = 1;
48 if nPcache = 1 then nPcache_flags = 0 1 2;
49 if nPcache_entry_add = 0 1 2 then nPcache_event = 0 1 2 3 4 5 6;
50 if RouteFlags = 1 then nPcache_entry = 0 1;
51 if nPcache_entry_add = 2 3 then nPcache_entry_add_cloning = 1;
52 if socket = 1 then TypeofRoute = 1;
53 }
54
55 # Spec file ends.

```

The test set displayed below is generated using AETG for research purposes of only this thesis. This test set is generated using AETG web and reprinted with specific permission.

* AETG Web, Copyright © 2003, Telcordia Technologies; reprinted with specific permission

The AETG Test Set

Valid Test Cases:

The parameter order is in following sequence

(First number denotes the test case number.)

Parameter Order:

conn_networks, intf , nPcache, nPcache_entry, nPcache_entry_add ,
nPcache_entry_add_cloning , nPcache_entry_delete , nPcache_entry_flush,
nPcache_event , nPcache_flags, nPcache_port, nPcache_se, RouteFlags, socket,
TypeofPacket, TypeofRoute.

```
1 1 0 1 0 5 1 0 1 4 0 0 1 0 1 1 0
2 0 1 1 0 1 1 1 1 6 0 0 1 0 1 0 0
3 0 1 1 0 0 0 0 0 0 0 0 1 0 1 1 0
4 0 1 1 0 2 1 1 0 1 0 1 0 0 1 2 0
5 1 1 0 0 2 0 1 0 3 0 1 1 0 1 2 0
6 0 0 0 0 3 0 1 0 2 0 0 1 0 1 3 0
7 1 0 1 0 4 0 0 0 6 0 0 0 0 1 0 0
8 0 1 0 0 1 1 1 0 3 0 1 0 0 1 3 0
9 1 1 0 0 1 0 0 1 0 0 1 1 0 0 2 0
10 0 1 0 0 3 1 0 1 5 0 0 0 0 0 2 0
11 1 1 0 0 1 0 0 1 4 0 0 0 0 1 2 0
12 1 1 0 0 2 1 1 1 0 0 1 0 0 0 3 0
13 1 1 1 0 3 0 0 0 0 0 1 1 0 0 2 0
14 1 1 0 0 3 0 1 1 3 0 1 1 0 0 1 0
15 1 1 1 0 4 0 1 0 0 0 0 0 0 0 3 0
16 1 1 0 0 4 1 0 0 5 0 0 1 0 1 2 0
17 0 0 0 0 5 0 0 1 0 0 0 0 0 1 3 0
18 0 0 0 0 0 0 1 1 0 0 0 0 0 0 2 0
19 1 0 1 0 4 0 1 0 1 0 1 0 0 0 3 0
20 0 1 1 0 3 0 1 0 1 0 0 1 0 0 0 0
21 1 0 1 0 2 1 1 1 2 0 0 0 0 1 1 0
22 1 0 1 0 1 1 1 0 5 0 0 0 0 0 3 0
23 1 0 0 0 5 1 1 0 3 0 1 0 0 0 0 0
```

24 1 1 0 0 1 0 0 1 2 0 1 0 0 0 3 0
25 0 1 1 0 2 0 1 0 6 0 1 0 0 0 2 0
26 0 1 0 0 5 0 1 1 1 0 1 1 0 1 3 0
27 0 1 1 0 4 0 1 0 2 0 1 1 0 1 0 0
28 1 0 1 0 2 0 1 1 4 0 1 1 0 1 3 0
29 1 1 0 0 5 1 1 1 6 0 1 1 0 0 3 0
30 0 0 0 0 4 0 0 0 3 0 0 1 0 0 1 0
31 1 0 1 0 5 0 0 0 5 0 1 0 0 0 1 0
32 0 1 1 0 0 1 1 1 0 0 1 0 0 0 0 0
33 1 1 0 0 3 1 1 0 4 0 0 1 0 0 0 0
34 0 0 1 0 4 0 0 0 4 0 1 0 0 1 2 0
35 1 1 1 0 4 1 1 1 3 0 1 1 0 0 3 0
36 0 0 0 0 3 0 1 1 6 0 0 0 0 1 1 0
37 1 0 0 0 0 0 0 0 0 0 0 1 0 1 3 0
38 0 1 1 0 1 1 0 1 1 0 1 0 0 0 1 0
39 1 1 1 0 5 0 1 0 2 0 0 0 0 0 2 0
40 0 0 0 0 2 1 0 1 5 0 1 1 0 1 0 0

* All values and combinations are valid.

Invalid Constraint Test Cases

1 1 1 1 1 4 1 1 1 4 0 0 0 0 1 3 0
2 0 1 1 2 0 1 1 1 0 0 1 0 0 0 3 0
3 0 0 0 0 0 1 0 1 1 0 0 1 0 1 1 0
4 1 1 1 0 0 1 0 0 2 0 1 1 0 0 0 0
5 0 0 0 0 0 0 1 1 3 0 1 1 0 0 3 0
6 0 0 0 0 0 0 1 0 4 0 0 0 0 1 3 0
7 0 1 0 0 0 1 1 0 5 0 1 1 0 1 3 0
8 0 1 1 0 0 1 1 1 6 0 1 1 0 1 0 0
9 1 1 0 0 1 1 1 0 1 1 1 0 0 1 1 0
10 1 1 0 0 3 1 0 0 6 2 0 1 0 1 3 0
11 0 1 1 0 4 0 1 0 6 1 0 0 0 0 3 0
12 1 1 1 0 0 0 0 0 0 2 0 0 0 1 2 0
13 0 0 0 0 5 0 1 1 6 0 0 1 1 1 2 0
14 0 1 1 1 0 1 0 1 0 0 0 1 2 0 2 0

15 0 0 1 0 0 0 1 1 0 0 0 0 3 1 1 0
16 0 0 0 0 1 1 1 0 1 0 1 1 4 0 3 0
17 1 1 1 0 4 0 1 1 5 0 1 1 0 0 2 1
18 1 1 0 0 3 1 1 0 6 0 1 0 0 1 3 1

* All values are valid, but tuples violate constraints.

As we see from above results there is not much difference between test cases generated by PairTest as well as AETG. The analysis of the above two scenarios and their comparison with the manual tests is given below.

3.14.3.3 Case Study-2: Manual Tests

Sample of the rules used to develop manual tests to test the above mentioned system are shown below.

- Make sure all the interfaces in question are up. If they are not active then disregard all the route changes.
- Check if routing socket is established.
- Check if network processor forwarding cache is valid.
- Check if search engine is added to the network processor.
- Check the type of routing packet received when the route table is accessed.
 - Is RTM_ADD received if route added
 - Is RTM_DELETE receive if route deleted
 - Is RTM_GET received if route deleted
 - Is Default received for any other operation
- Check type of route modified
 - Network Route
 - Host Route
- Check route flags on the modified routes
 - Static

- Host
- Network
- Cloning
- Cloned
- Check if network processor ports are active.
- Check if network processor ports are connected to other networks
- Check if nP cache entry is recognized correctly from the route being modified.
 - Local
 - Remote
 - CPU Hosted
- Check if correct entry is being added to nP forwarding cache when a new route is added.
 - Local Network
 - Remote Network
 - CPU hosted Network
 - Local Host
 - Remote Host
 - CPU hosted host
- Check if the route being added is cloned.
- If the route is cloned appropriate entry is added to the forwarding cache.
- Route table is updated when the cloned route times out.
- Check entry in forwarding cache is added with appropriate flags for appropriate route additions.
 - nP_Network
 - nP_Host
 - nP_Valid
- Check if appropriate events are received by control plane for appropriate actions.
 - Add
 - Remove

- Update
- Flush
- Delete
- Destination Unknown
- Source Mismatch
- Check if forwarding cache entry is deleted when a route is deleted.
- Check if forwarding cache flushed correctly.

By considering all the above combinations the total number of manual test cases generated was 116. By analyzing Manual and PairTest test sets we came to a conclusion that all of the manual tests were covered by PairTest test set. Moreover, PairTest as well as AETG test sets provided a few more tests for testing addition of forwarding cache entries, checking addition of cloned routes and validity of interfaces.

When we ran the tests on the system under tests we observed that both PairTest and AETG gave a test for testing interfaces which were not included in manual test set. This test uncovered a possible failure situation in the system.

This condition was tested by the following test condition values of PairTest
 0 1 1 0 2 x x x x x x x x x x

By considering the respective values with the respective order of parameters.

for PairTest where

| | |
|--------------|------------------|
| intf | = 0 (not active) |
| TypeofPacket | = 1 (RTM_ADD) |
| Socket | = 1 (Active) |
| TypeofRoute | = 0 (Network) |

RouteFlags = 2 (Network)

Rest all values can take any values.

In this case even though interface was not active a route was added which signified interface might become active during the operation.

Similarly AETG also displayed this test condition by following test case.

1 0 1 0 2 1 1 1 2 0 0 0 0 1 1 0

By considering these values by respective order of parameters mentioned above we got

Conn_network = 1 (Active Networks connected)

Intf = 0 (Inactive)

nPcache = 1 (Active)

nPcache_entry_add = 1 (Network entry added)

TypeofPacket = 1 (RTM_ADD)

TypeofRoute = 0 (Network route)

RouteFlags = 0 (Static Network)

nPcache_event = 2 (Update Event)

socket = 1 (Active)

Rest all values were not significant.

This test tested the condition for the route addition for a different network interface connected to network processor. It tested that when any interface connected to a separate network, which was connected to network processor ports, became active during operation then the corresponding network route is added to the forwarding cache.

Thus PairTest displayed ability to generate less number of efficient test cases.

3.14.4 Discussion

Thus by analysing the test sets generated by the PairTest we saw a little variation in the efficiency of the test sets dependent on the system under test. PairTest and AETG didn't give much variation in the generated test sets for the less number of parameters. The variation increased with the increasing number of parameters and their values. Advantage of AETG was in the fact that AETG gave a list of invalid tests which could be used to avoid wrong results from such tests. The main source of this variation was due to PairTest's ability to accept the parameter values as 'Any' from the available parameter values.

Following points can be illustrated by analysis of the test sets generated by using PairTest for the above examples.

- While using AETG, tester specifies a degree to test for each relation. If the tester specifies pairwise testing, the AETG system generates tests that cover all valid pairwise combinations of values of the relation's fields [Coh96]. This implies that for any two parameter fields f_1 and f_2 and any values v_1 for f_1 and v_2 for f_2 , there is some test in which f_1 has the value v_1 and f_2 has value v_2 . [Coh94;Coh96] This is in accordance with PairTest tool strategy. If n-way testing is specified, the AETG system generates a test set that covers all n-way parameter combinations for each relation. PairTest generates test cases using only 2-way test generation.
- The AETG system generates tests for a set of relations by combining tests for individual relations [Coh97]. The algorithm for combining tests

ensures that for each combined test there is a set of relations such that the projection of the test onto the fields in each relation in the set is a test for that relation. If two relations have no common fields, the combined tests for the two relations are simply concatenations of the tests for each individual relation [Coh97]. The PairTest generates tests by extending each pairwise set such that every combination of valid values of the considered two parameters is covered by at least one test. Tests by PairTest are not concatenation of the tests for each relation.

- The AETG system generates an *invalid* test for each individual value specified for a field in relation. A value is an invalid value only within the context of the relation. A value which is invalid for a field in one relation may be a valid value for that field in another relation [Coh96]. To avoid having one invalid value mask, AETG system uses only one invalid value per test case [Coh96]. AETG creates the test for an invalid value by taking a valid test for the relation and substituting the invalid value in place of the field's valid value in that test [Coh96]. PairTest has no mechanism of identifying the invalid tests in the system. PairTest tries to cover every combination of valid values for the considered pair of parameters by at least one test by applying greedy algorithm to produce a number of candidate tests and then selecting one which covers the most uncovered pairs.
- PairTest uses greedy algorithm to generate a number of candidate tests. The requirements for pairwise test set generation that all parameters have same number of values is relaxed by assigning don't care or 'Any' values for the missing values. Also another requirement of having each pair of values for parameters to be covered same number of times is bypassed in PairTest as it is considered unnecessary for Software Testing [Tai02] and it gives rise to extra tests for pairwise testing.

3.15 Drive for Automation

We enhanced PairTest, an automated test generation tool based on IPO strategy, to achieve our goal of providing a cost-effective automated test generation approach. We also demonstrated test generation ability of PairTest in industrial environment. However, test cases generated by PairTest need to be converted into executable test cases which take lot of time and resources. Also execution of these test cases for result analysis is a time consuming task. To tackle these issues we research our second goal of providing a cost-effective and easy to use, adaptable method of translation of test cases generated by PairTest into executable test cases and their automated execution in Chapter 4. This method will be referenced as automation in the following chapters.

Chapter 4

Automation Using ‘Expect’

4.1 Objectives of Test Automation

Automating tests is a skill but a very different skill from testing. Many organizations are surprised when they find that it is more expensive to automate a test than to perform it once manually. *Whether a test is automated or run manually doesn't affect its effectiveness or how exemplary it is* [Bei83]. Automating test affects only how economic and evolvable it is. Once automated; an automated test is generally much more economic as the cost of running it is just a fraction of the effort to perform it manually. However automated tests generally cost more to create and maintain if automation is not developed efficiently. The better the approach to automating tests the cheaper it is to implement in long term. If no thought is given to maintenance when tests are automated, updating the entire test suite can cost as much, if not more, than the cost of performing all the tests manually.

To follow our goal of implementing a resource-efficient method for automated test execution and result analysis and to avoid the maintenance issues leading to high cost of automation we adopted objectives summarized below.

- Consistent repeatable testing
- Running tests unattended
- Finding bugs in regression testing
- Running tests more often.
- Increase confidence in testing process.
- Measure performance
- Reduce cost of testing.
- Running tests more quickly.

4.2 Metrics of Test automation

Before implementing any automation it is necessary to evaluate the attributes against which automation will be tested. To measure success of test automation it is very essential to measure the attributes of test automation to verify that it is consistent with the objectives [Pet99]. We study these attributes to incorporate them in the automation system developed for automated execution of test cases. We have evaluated these attributes in Chapter 5.

Few such measurable attributes are listed below with obtained directions for implementation of automation to be consistent with or goals.

- Maintainability

An automation regime is considered highly maintainable where it is easy to keep the tests in step with the software [Pet99]. It is fact that software changes but the cost of updating the automated tests should not be too great or the entire test automation effort will be abandoned in favor of cheaper manual testing. Thus maintainability is an important attribute which gave direction to the test automation developed by us to be consistent with the objectives. We evaluated maintainability of the developed automation in terms of its evolvability.

- **Efficiency**

Efficiency is generally one of the main reasons testing is automated so that testing can be performed with less effort and in shorter time. Usually early automation efforts are less efficient than manual testing. Measurement of test automation efficiency is done from cost of automation effort. The cost of automating includes the overhead and the time and effort spent in various activities related to test automation so usually efficiency and evolvability is measured over time. This gave us a direction to evaluate efficiency of automation in terms of evolvability and economic considerations.
- **Reliability**

Automation should be reliable so that it can increase user confidence. Reliability of an automated test suite is related to its ability to give accurate and repeatable results. A common direction in measuring reliability is the number of failing tests due to defects in test design or automation which was covered in the measurement of evolvability for the developed automation.
- **Flexibility**

Flexibility of an automation test suite can be defined as the extent with which different subsets of tests can be worked with the automation [Pet99]. A flexible automation test regime allows test cases to be combined in many different ways for different test objectives. This attribute gave us direction to evaluate flexibility of developed automation in terms of evolvability.
- **Portability**

The portability of automation is related to the ability to run in different environments. This is considered as a very important attribute, as it is very difficult to test the automation across different platforms if software supports a new platform. Automation was developed by keeping in mind the portability issue so it can be run in any environment with minor adjustments.

4.3 Automated Test Life- Cycle Methodology (ALTM) [Dus99]

The Automated Test Life-Cycle Methodology represents a structured approach for the implementation and performance of automated testing. The ALTM approach shows the benefits of modern application development efforts. To develop a successful automation ALTM demands that requirement analysis of the incremental stages of the software must be performed.

The ALTM, which is invoked to support test efforts involving automated test tools, incorporates a multistage process [Dus99]. This methodology supports the detailed and related activities which are required to decide whether to use an automated testing tool and which automated tool should be selected for automation. It considers the process needed to introduce and utilize an automated test tool with coverage of test generation technology and test execution process. The methodology also supports the development and management of test data and the test environment. The ALTM is geared toward ensuring successful implementation of automated testing [Dus99].

Due to these advantages of ALTM approach showing a pathway to successful implementation of automation, we chose this approach to implement the automation depicted in Appendix A.

The phases of the ALTM were followed to implement the automation. We list the phases below with the goal of each phase while explaining the steps taken for our implementation of automation. These phases can be summarized as follows.

- Decision to automate test

While implementing the idea of automation of testing there is always a need for automatic test plan generation. Throughout the industry automated test tools are viewed as enhancements to manual testing process. But ALTM approach suggests that automation be treated as a separate project. During this phase the automation test tool requirements are outlined. With introduction of automation it cannot be

expected that it will immediately reduce the test effort as well as minimize the test schedule. Careful analysis of the testing cycle as well as test generation process (PairTest) was done in this phase to make accommodation for learning curve before implementation of automation.

- Test Tool Acquisition

In industrial environment this phase depicts the approval from management for implementation of automation. This phase also involves evaluation and selection process for the tool which will be implemented using automation. As a tool should cover all areas of the testing requirements choice of tool is very important. We analyzed various tools which can be used to implement our implementation such as Tcl/Tk, Perl etc. We selected Expect as a tool to implement automation due to its interactive scripting abilities better than other tools available.

- Automated Testing Introduction Process

Test Process Analysis ensures that an overall test process and strategy are in place and are modified to allow successful introduction of automation [Dus99]. Pros and cons of introduction of automation in the testing environment were evaluated by consideration of available testing resources and test environment. It was learnt during this phase that introduction of an automation using expect in the available test environment would help reduce manual testing efforts by considerable amount thus increasing the efficiency of testing process.

- Test Planning, Design and Development

Test planning stage includes review of test planning activities. During this time test procedure standards and guidelines were identified. Goals of automation were outlined. As automation needed an existing efficient test suite as a requirement, test suite generated by PairTest was selected due to its high fault detection capability along with ability to generate reduced number of tests providing cost effective way of generating tests. In the development phase for the automation

quality of the test suite is of prime importance as to avoid later problems in maintenance [MAL94].

- Execution and Management of Tests

In this stage automation was developed and the tests generated by PairTest were executed using automation. Analysis of the test results obtained from test execution stage was performed in this stage. Also automation was tested for its ability to handle changes in the system under test. It was observed that automation showed higher evolvability than manual testing process to accommodate changes in the system.

- Test review and Assessment

In this phase metrics were evaluated and modifications were conducted in the automation to improve maintainability as well as reliability of the tests executed. This phase helped us improve the adaptability of the automation.

4.3.1 Practical Implementation

Our goal was to develop an automation which can be easily modified to apply to a new system with same effectiveness. This helped us in carrying out testing of any system with much more efficiency and relatively fast.

The Block diagram implementation of such a system is shown in Figure 9 which was implemented using PairTest as an automatic test case generator and 'Expect' as a tool for automating the existing test cases.

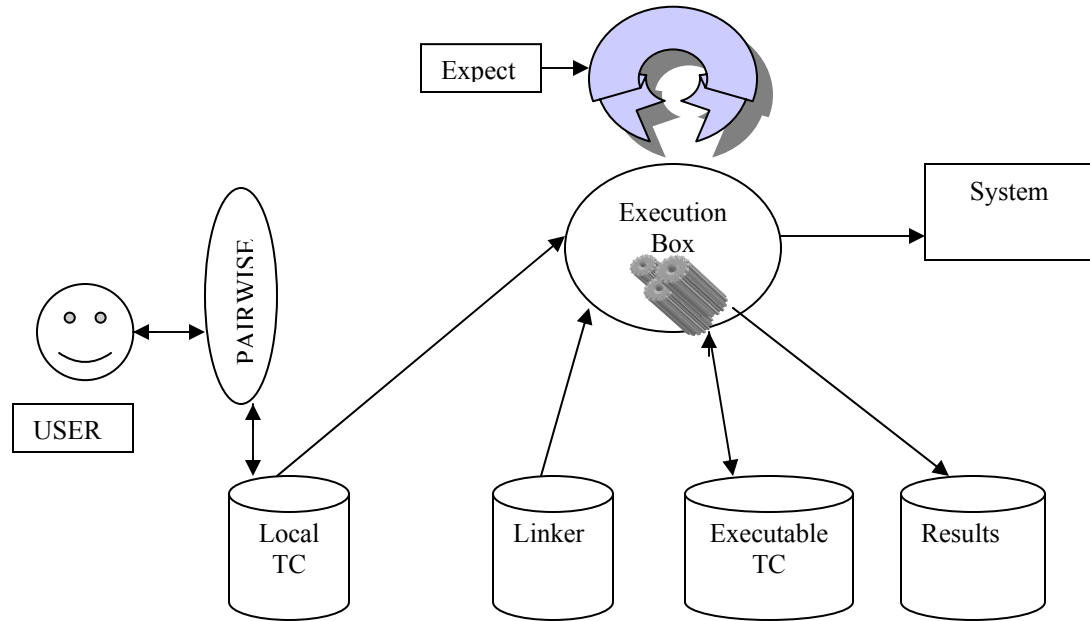


Figure 9. Test Generation and Automation Process

Expect, an extension of the simple yet powerful Tool Command Language (Tcl), is a latecomer to the UNIX [Unix] toolkit. It's an extensible interactive language whose scripting facilities surpass even those the best terminal programs offer which made it the prime choice for implementation of automation. Expect is a program that "talks" to other interactive programs according to a script. Following the script, expect knows what can be expected from a program and what the correct response should be. An interpreted language provides branching and high-level control structures to direct the dialogue. As compared to other tools available as Tcl/Tk [Tcl] or perl [Per], Expect, an extension of the simple yet powerful Tool Command Language (Tcl), is a latecomer to the UNIX [Unix] toolkit. It's an extensible interactive language whose scripting facilities surpass even those the best terminal programs offer. Expect has the unique capability of "chatting" to otherwise impossible programs and can smoothly juggle input and output to many places at once. The interactive facilities Expect offers open new dimensions of opportunity in automation, capability, and testing. Don Libes wrote Expect at

the National Institute of Standards and Technology (NIST) in 1987. A popular use of Expect is for testing other software. Expect can supply specific outputs, given certain inputs. Expect is a complicated language that takes time to master. Tcl, on the other hand, is a much simpler language and can be described in a more linear fashion. The free download as well as installation instructions can be found at expect.nist.gov.

Thus Use of Expect as automation tool helped us to meet our goal of development of a resource efficient and adaptable method for automated execution of test cases.

The basic flow diagram of automation is as displayed in Figure 10.

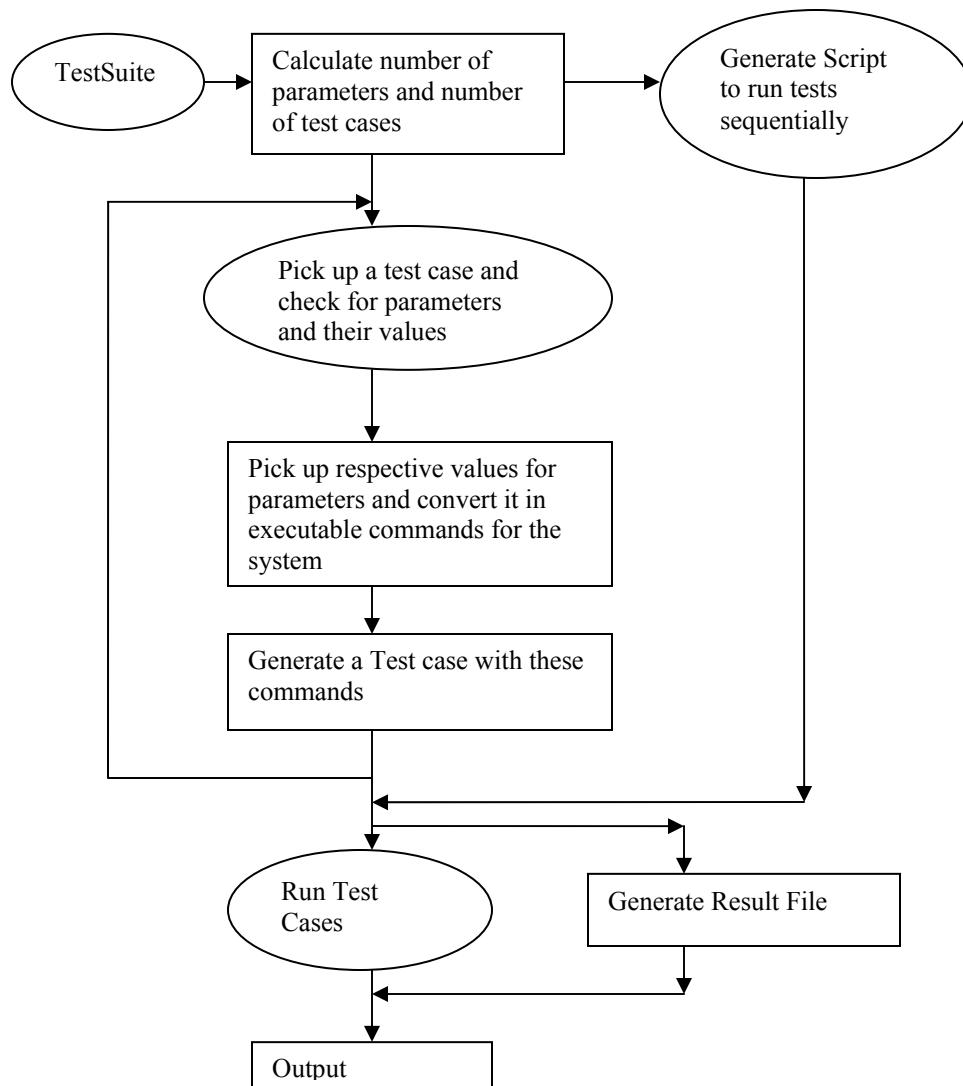


Figure 10. Test Automation Process

We developed our automation on RedHat Linux 7.3 [Linux73] machine with Pentium 3 [Intel] processor and JDK 1.2.1[Sun] for Linux. We also used Cisco Catalyst 5500 [Cisco] as a system under test for demonstration to test our automation. Minicom [Min] utility was used for talking to the switch using serial port.

Automation code is provided in Appendix A. Problems in development of automation and benefits of the developed automation are discussed in following sections. Evaluation of automation in industrial as well as laboratory environment to assess value of automation is discussed in Chapter 5.

4.4 Common Problems in Test Automation [Mar97]

There are a number of problems that may be encountered in trying to automate test execution. Problems, which come as a complete surprise, are more difficult to deal with. Some of the problems commonly encountered are listed below. We also discuss how we tackled these problems while developing automation described in Appendix A.

- **Poor Test Selection**

It is expected that automating tests will help tests find more defect which is not true. A test is most likely to find defects when it is run first time. If a test has already been run and passed then it is much less likely to find a new defect unless it is testing code, which has been changed. *Test execution tools are usually 'replay tools'* [Mar97]. Automated testing is not effective when the tests on which it is based on are not efficient in testing. This problem was successfully overcome by us with selection of test suite from PairTest as an automable test suite. Due to high fault detection capability of PairTest we were assured about the satisfactory fault detection capability of PairTest and did not depend on automation to help find new defects with more runs of the tests.

- **Maintenance of automated tests**

When software is changed it is necessary to update some or even all the tests so that they can be re run successfully. If automation is not efficient to adapt to changes in the software then maintenance efforts for those tests consume high amount of resources. This problem of adaptability of automation was handled by careful analysis of test generation ability of PairTest, the tool used to generate the automable test suites. PairTest generated new test suites to depict the change in

the test system very easily with those changes incorporated in the tests generated. This provided us with good level of adaptation properties for automation as automation was developed to adapt to changing test suites of PairTest.

- **Technical problems**

Interoperability of the automation with other software or tools cause serious problems in test automation. These problems were handled using Expect as a tool for development of automation. Expect's ability to provide easy way of adaptation of the environment change helped us build our automation which can handle changes in the test environment.

4.5 Current problems in Industrial Automation

As part of this thesis work, industrial experience was gained by working as a co-op. This co-op experience was used to develop automation in industrial environment. The common problems encountered while developing this automation are listed below. These experiences proved helpful in implementation of our automation to avoid problems.

4.5.1 Commitment of automation [Pet01]

Software testers, under pressure to do more testing in less time, often find themselves rushed and eager for anything that will help them in testing process. Their fear and desperation leads them to seek a "silver bullet" [Pet01] that will help them regain a rational pace to their work. Test automation is often considered as that silver bullet. It is assumed that it will make their job simpler and easier and help them contend with unrealistic schedules.

Understanding of testing requirements and the situations faced by testers is essential for a successful implementation. It is observed from the co-op experience that knowledge of software development as well as maintenance and reliability issues is essential for successful automation. From this experience we studied in detail the test system to make automation more

reliable before implementing it. This made the automation system easy to update with changes to the product under test and maintain which are major challenges in path of successful automation. These were dealt with careful analysis of system under test in the developed automation in Appendix A. We developed an automation which can serve as a primary choice for test execution rather than being secondary in nature.

Unless someone who is motivated and working closely with the rest of the development group does test automation, automation will not succeed [Pet01].

4.5.2 Choice of automation [Mar98]

It is not possible to automate 100% of testing [Dus99]. There are some test cases, which require manual attention. Automation can be built by analyzing where time is spent while testing manually [Mar98]. Focusing automation efforts on tasks that otherwise may go untested is a mistake. Automation is coded after a solid testing procedure is established as test automation always breaks down at some point. For successful automation choice of automable test cases is important to improve the productivity. Experience as a co-op helped us to get insight to develop automation by making sure the tests were important enough that the time to maintain them to keep automation running was less. We chose tests developed from PairTest tool described in the following sections. Standardized and effective testing procedures and practices are considered basis for success in automation. In our experience during work in industry as a co-op we learnt that decisions about what to automate can be critical to successful test automation which helped us in development of automation.

4.5.3 Approach for Late Start on Test Automation [Pet01]

As part of our work in industrial environment it was observed that most common problem in developing an efficient automation was the late of the

automation. The three keys to Test Automation are defined as; successful test automation requires team commitment, teamwork between testers and developers, and getting an early start [Pet00].

The major concern, which can be related to late starts in automation, can be defined as the test strategy changes in the automation which can be difficult to incorporate. These conditions inhibit the automation and may hinder development of automation by preventing from automating parts of test suite. *In general, the late start always asks for more investment before returns are seen* [Pet01]. Testing strategy includes the way test cases are designed and described, how testing progress is reported, how tests are grouped and assigned, and how decisions are made regarding what tests need to be run when. All these aspects change with automation to provide ease of use and maintainability to the automation.

This problem was tackled by use of PairTest as test generation tool. As PairTest was able to handle changes in requirements by incorporating them in generated test cases automatically, there was no need for test generation strategy change. This allowed us to develop an efficient automation by automatically following testing strategy changes in terms of PairTest test suite thus eliminating problems associated with late starts.

4.6 Intelligent Test Automation [Rob00]

In a process to create an efficient automation scheme there can be many approaches, which can be adopted by testers while testing the product. Few of such approaches practiced in industry are listed below with their relative positive and negative sides. In order to improve automation efficiency and in all testing efficiency using automation use of intelligence is very essential.

In order to evaluate and emphasize the need of intelligent automated testing from the analysis of earlier discussion we will consider four imaginary testers' progresses, which are testing the product software.

Approach 1:

Tester 1 starts hands on testing the product and finds some nice bugs. The development team happily fixes the bugs, and gives tester 1 a fresh version of the software to test, which results in more testing, more bugs and more fixes. Tester 1 feels productive and happy.

After several rounds of this find and fix cycle, he becomes bored and bleary eyed from running virtually the same tests over and over again by hand. When Tester 1 finally runs out of patience and enthusiasm the software is declared as ready to ship.

Customers find it too buggy and buy competitor's product.

Approach 2:

The second tester starts testing by hand, but soon decides that it makes more sense to create test scripts that would perform the keystrokes automatically. After carefully figuring out tests that would exercise useful parts of software, he records the actions in scripts. These scripts soon increase in the hundreds. At the push of a button, the scripts spring to life and run the software through its paces. Tester 2 feels clever and happy. The scripts require a lot of maintenance when the software changed. He spends weeks arguing with developers to stop changing the software because it breaks the automated tests. Eventually, the scripts require so much maintenance that there is little time left to do testing.

When the software is released the customers find lots of bugs that scripts didn't cover. They stop buying the product and wait for next version.

Approach 3:

Tester 3 doesn't want to maintain hundreds of automated test scripts. She writes a test program that goes around randomly clicking and pushing buttons in the application. This 'random' test program is hypnotic to watch and finds a lot of crashing bugs. This randomness of the program makes it somewhat more effective than earlier approaches to find the bugs and defects in the software. It is effective in uncovering bugs which crashed the program.

Tester 3 enjoys uncovering such dramatic defects and is happy. Since the random test program can only find bugs that crash the application, Tester 3 still has to do a lot of hands on testing, getting bored and bleary eyed in the process. Customers find so many functional bugs in the software when it is released that they lose trust and stop buying the product.

Approach 4:

Tester 4 begins with hands-on, exploratory testing to become familiar with the application and uses the knowledge gained during the hands-on testing to create a very simple behavioral model of the application. Tester 4 then uses a test program to test the application's behavior against what the model predicted. The behavioral model is much simpler than the application under test, so it is easy to create. Since the test program knows what the application is supposed to do, it can detect when the application is doing the wrong thing. As the product cycle progresses, developers write new features for the application. Tester 4 quickly updates the model, and the tests continue running. The program runs day and night, constantly generating new test sequences. Tester 4 is able to run the tests on a dozen machines at once and get several days of testing done in a single night.

After several rounds of testing and bug fixes, Tester 4's test generator begins to find fewer bugs. Tester 4 upgrades the model to test for additional behaviors and

continues testing. Tester 4 also does some hands-on testing and static automation for those parts of the application, which are not yet worth modeling. When Tester 4's software is released, there are very few bugs to be found. The customers are happy. And Tester 4 is happy.

As we go through above four plausible situations they show us some of the approaches available today in software testing.

Tester1 is a typical *hands-on* [Rob00] tester, manually running all tests from keyboard. Hands-on testing is common throughout the industry today. It does provide immediate benefits but in the long run it is tedious for the tester and expensive too.

Tester 2 practices static test automation. Static automation scripts use the same sequence of commands in same order every time. The scripts are costly to maintain when the application changes. The tests are repeatable but since they always run the same commands they hardly find any new bugs.

Tester 3 operates closer to the *cutting edge of automated testing* [Rob00]. The types of random test programs are called '*dumb monkeys*' [Rob00] because they essentially bang the keyboard aimlessly. They come up with an unusual test action sequences and find more crashing bugs. But it is very hard and time consuming to direct them to the specific parts of the application to be tested [Rob00].

Tester 4 combines the other testers' approaches with a type of intelligent test automation called '*model based automation*' [Rob00]. Model based approach doesn't record test sequences like static automation does, nor does it enter blind keystrokes. It uses specifications of the software's behavior to determine what outcome is expected. This automation generates new test sequences endlessly, adapts well to changes in the application and can be run on many machines at once.

From the analysis of the above approaches we got important direction about the approaches in implementing the automation while avoiding the mistakes to make automation more static or random making it difficult to adapt to changes.

4.7 Benefits of test automation

Automation helped to improve the overall testing process by execution of tests in a cost effective way which was consistent with our goals. Following points summarize some of the benefits of the developed automation emphasizing our goals to implement automation.

- A clear benefit of automation is that it led to greater confidence in system by execution of more test cases in less time making it possible to run them more often.
- Using automation, tests which were more difficult to execute manually could be performed easily. Large number of test cases could be executed using automation, which could take days to test manually.
- Automating menial or boring tasks, such as repeatedly entering the same test inputs, gave greater accuracy as well as improved morale. It also allowed putting more effort into designing better test cases to run by making better use of available resources.
- Automation provided consistency and repeatability of tests by repeating tests that were automated exactly (ex: same inputs). This provided a level of consistency which was very difficult to achieve with manual testing.
- The efforts put into deciding what to test, designing tests and building them got distributed over many executions of the test due to automation. Automation made reuse of tests easier.

- After a test suite had been automated it took a much shorter time to run than running it manually so effective testing time was shortened.

Chapter 5

Value of Automation

5.1 Introduction

Empirical evaluation of the value of automation is important. Some of the questions are: “How does automated test case generation and execution compare with manually designed and executed test cases?”, “Where are the trade-offs?”, and “Which metrics to use to make the assessment?”

The author of the work spent co-op time with a telecommunications company and have had first-hand experience automating and assessing the value of test automation. This chapter reports on some of the findings from that experience. However, in order to protect the confidentiality of the information, the actual data from the industrial setting are shown in a sanitized form. To ease the understanding of those results, experiments were devised which used same or similar metrics, but were conducted at NC State using both manually generated test cases (using a strategy similar to that used in during the co-op experience), and test cases using PairTest toolset and its automation extensions described earlier.

Two aspects of testing were selected as most important (from practical perspective) to assess automatic (algorithmic) vs. manual test-case generation. One is the fault-coverage provided by the generated test-suites, i.e., their ability to detect actual (industrial setting) and/or seeded faults (laboratory experiments). The other is

efficiency of the test suites, i.e., their fault-detection capability compared to their size (i.e., cost). Two other aspects were selected to judge the automation itself: resource consumption over the testing cycle, and evolvability of the test-suite, i.e., the cost in keeping the test-suites synchronized with changes that may be occurring to the specifications (requirements, design, detailed-design, environment) during software development.

5.2 Experiment

The experiment consisted of three parts. Definition of the experimental test-bed, development of manual test-cases, and development of automated test-cases using PairTest. It was decided to use a test-bed that resembled the industrial environment. In the industrial environment, testing was aimed at a series of software-based networking devices such as switches and routers. Therefore, the NC State illustration testbed consisted of a network routing device (not by the same manufacturer as was used in the industrial environment), and some network-based clients that acted as sources and destinations for network-based activities.

The test-bed is shown in Figure 11. It consists of a Cisco 5500 routing switch with a workstation-based source and a suite of destinations on the “internet”. In actuality, the whole test-bed operated in a private networking environment, 10.x.x.x class network, to avoid interfering with the operational campus environment. The experiment was concerned with reachability of destination internet protocol (IP) numbers from the workstation via the 5500 device. Reachability depended on different settings of the 5500 and the workstation routing tables. Some of the settings were considered to be erroneous (these were the seeded faults). The changes were made in the workstation routing tables and in the 5500 settings. The goal was for the test-suites, generated using certain rules explained below and in the Appendix A, to detect them. The issues were, how many of these faults did get detected, how many test-cases did it take to do that, and what was the “cost” of that.

Reachability from the terminal computer to certain destination IP addresses was tested using the Linux ‘Ping’ [Ping97]. When the system functioned properly, all the destination IP addresses should have been reachable. Failure to reach a destination IP address was considered as a failure of the system resulting from a fault (in reality this could be a set-up fault, or it could be an actual software problem).

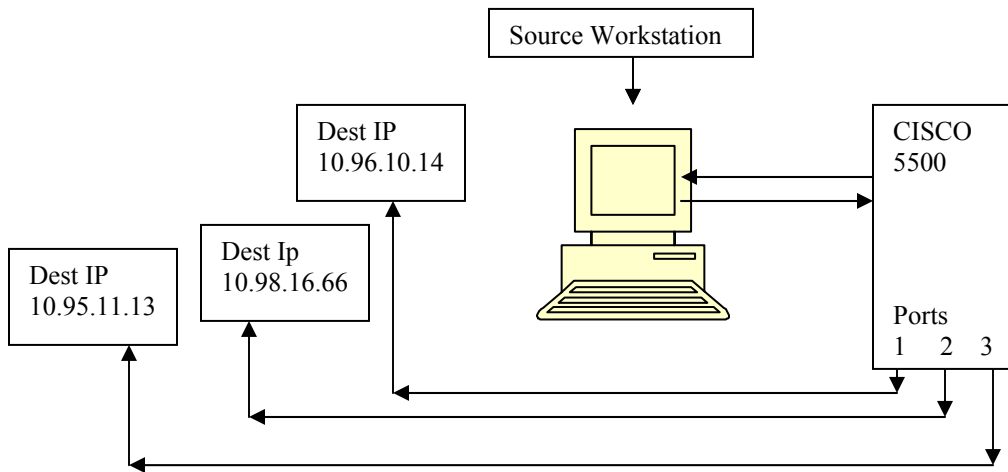


Figure 11. Test-bed

Specification Parameters

The specification parameters relevant to the the functioning of the test-bed system are discussed below. Actual values of the parameters are shown in Appendix A.

Destination Interface – It has value 1 when interface is up and in working condition. Value of this variable is 0 when the interface is down and cannot be used to pass ICMP data through Cisco 5500 which is used to test reachability.

destaddr – This variable contains various values for the destination IP addresses for which the reachability from the terminal computer was tested. Four values of different destination addresses were used which are listed in Appendix A. When

the whole system functions properly (according to the specifications), all the destination IP addresses should be reachable.

ipgw– This variable shows various values of the Internet Gateways used. These values were set as the next hop addresses on Cisco 5500. Three values of the gateways were used which are listed in Appendix A. The traffic to the destination IP addresses was passed through these gateways. If the traffic was not able to locate the destination IP address through a gateway it was considered as the fault of the system.

permitip – The input port of 5500 was setup to permit certain IP addresses. When the value of permitip was 0 it rejected traffic to those addresses thus causing failures of the system. When the value was 1 permitip allowed reachability to those IP addresses.

redirect - The port of 5500 was setup to redirect traffic to other ports. When the traffic for those IP addresses was received by the port, depending on the value of the variable, decision to redirect traffic to other port was taken. For Ex: The routing table of the 5500 is used to send traffic to the IP address 10.98.16.66 using next hop address 10.98.16.1 on port 2/1. If 5500 is configured to redirect traffic then data received on the port of 5500 was redirected to other port. If value of redirect was 0 then the traffic was not redirected where as 1 value redirected the traffic allowing the reachability.

Parameter Relations and Constraints:

Relationships among the parameters may complicate things. If there are no relationships (total independence), then so long as all parameter values are tested at least once the test-suite could be considered complete, if not exhaustive. If all parameters are known to depend on other parameters, then another extreme applies; all combinations of all parameter values need to be tested. If the parameters have certain relationships (e.g., mutual exclusion), then some of the

test-cases may be eliminated as “not possible” – of course they still may need to be generated and tested to account for surprises.

The test system described in Figure 11 had the following two relations defined. Parameters (destaddr, ipgw, redirect) depend on each other – This relation says that reachability to a particular destination IP address depends on the destination address itself, the value of the next hop address or gateway used on 5500 and on whether the 5500 re-directs the traffic to that address.

A separate relationship was set for (destaddr, ipgw, permitip) triplet. This relation accounts for the reachability to a particular destination IP address being dependent on whether the 5500 port allowed traffic to that destination IP address.

Of course, if the traffic to a destination IP address is not permitted then system will indicate failure to reach the corresponding destination address regardless of whether there is re-direction or not, and vice versa. Furthermore, settings of the destination interface (up or down) are independent of the other parameters. The actual values of the variables are listed in the Appendix A. The total number of seeded failures into the system was 12. These seeded failures are listed in Appendix A.

Manual Test Cases

Manual test cases were generated for the system using the combinatorial approach. These test cases are listed in Appendix A. Since there are 5 parameters, the total possible number of test cases generated using all combinations of all parameter values $INT. (2) \times DEST (4) \times G (3) \times R (2) \times P (2) = 96$. As all combinations of parameter values are covered with these combinations maximum number of test cases achieved with all relations were 96. This is the worst-case situation. All parameters are inter-dependent; hence all combinations have to be tested. However, by using the experience and knowledge of the human tester as additional constraints the number can be reduced. Note that the tester is aware of

the basic parameter relationships and constraints, but may process them differently than algorithms such as pairwise. For example, in the test-bed it is known that a certain destination IP address cannot be reached when its interface is down. Since that state is set by the tester (and is independent of other parameters), the manual test case for that situation will certainly fail, and the test case may not need to be run. In this case the number of manual test cases are $INT. (1) \times DEST (4) \times G (3) \times R (2) \times P (2) = 48$. Of course, in practice similar information may or may not be available. It may come from other verification and validation activities. If it does not (i.e., if the failure of the interface needs to be tested for), then the test case may need to be run. Thus testing for the failure of the interface might account for another test case thus giving rise to the 49 th manual test case. However, this test checking the failure of the interface can be considered independent from the other manual test cases and be accounted as an independent test case not counted towards the manual test cases generated for testing the system due to assurance of the interface going down in a limited setting environment. These manual test cases are a small subset of the test cases required to test the whole test system and are derived for testing partial aspect of the whole test system applying limited setting with the defined parameters. Hence, through the use of some in-dependence relationships the manual test cases were reduced to 48.

Automatic Test Case Generation

PairTest was used to generate test cases for the test system described above. Given the same parameters and constraints, PairTest generated 26 test cases as shown in new_m9 under Appendix A. These test cases generated by PairTest as well as manual test cases were used for analysis as described below.

Fault Injection

Faults in the system were injected by changing the configurations of the system using changes on 5500. The changes were done gradually so that all faults were seeded in the system. Faults were indicated as configuration changes in the

systemn causing the failure of the system to get a response from Linux ping [Ping97] utility. At each point numbers of failures in the system were dependent on the configuration of the system. The system was then tested with the test cases generated by PairTest as well manual test cases to test fault detection capabilities of the test cases. Each configuration changes caused certain failures in the system. However maximum number of failures in the system was 12 due to respective configuration changes in the system and response of the test system to those changes. These configuration changes were selected randomly. Faults seeded in the system are listed in Appendix A.

5.3 Metrics

As mentioned earlier, choice of appropriate metrics is the key element in the comparative analysis of the value of automated testing vs. manual testing. This study uses four metrics: failure coverage (in percent of total failures detected), testing efficiency (in units of faults per test case), and resource usage (in units of effort, e.g., person hours), and evolvability in effort (e.g., person hours) per unit of change.

5.3.1 Fault Coverage

Fault coverage is a measure of the fault detection capability of the test suite. It is defined as shown in equation (5.3.1.1)

$$\text{Fault coverage} = \frac{\text{No. Of faults uncovered by the test suite}}{\text{Total No.of faults}} * 100 \quad (5.3.1.1)$$

Similarly Failure coverage of the system can be defined as (5.3.1.2)

$$\text{Failure coverage} = \frac{\text{No. Of failures uncovered by the test suite}}{\text{Total No.of failures}} * 100 \quad (5.3.1.2)$$

The experiment was run buy a) generating manual and PairTest test cases, b) injecting faults into the system, and c) running the manual and the PairTest test cases to see which of the test cases detected which fault and failure. The fault coverage as well as failure coverage results are summarized in Table 1. Details of which test case detected which fault/failure and when are given in Appendix A.

| By Test Case Number | 5 | 10 | 15 | 18 | 20 | 24 | 25 | 26 | 30 | 35 | 40 | 48 |
|--|---|----|----|----|----|----|----|----|----|----|----|----|
| No. Of Failures Found by Automated Testing | 2 | 3 | 7 | 9 | 10 | 11 | 11 | 12 | 12 | 12 | 12 | 12 |
| No. Of Failures Found by Manual Testing | 0 | 1 | 3 | 5 | 6 | 7 | 7 | 7 | 9 | 9 | 11 | 12 |
| No. Of Faults Found by Automated Testing | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| No. Of Faults Found by Manual Testing | 0 | 1 | 2 | 3 | 4 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |

Table 1. Fault/Failure Coverage Vs. Number of Test Cases data

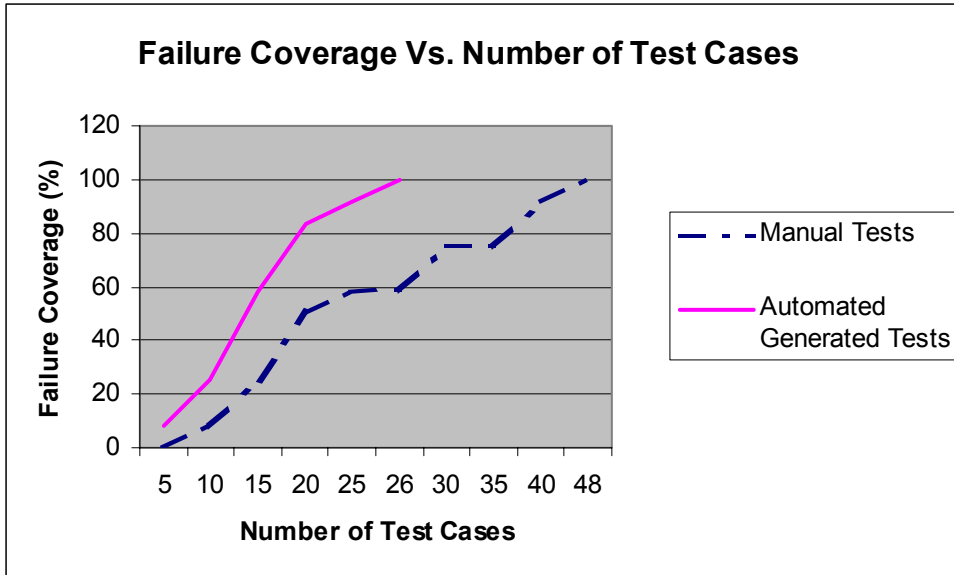


Figure 12. Failure coverage Vs. Number of Test Cases (PairTest)

* Failure coverage is the ratio of the failures detected

It can be seen from Figure 12 that PairTest generated 26 test cases which identified all the known failures in the system. Manual situation required all 48 test cases (in practice perhaps more, depends on how principal parameters are defined) to guarantee detection of all the seeded faults. According to this experiment, both PairTest and manual test-suites provided 100% coverage. However, PairTest testing appears to be about twice as efficient as the Manual one, i.e., the ratio is 48 to 26 (factor of 1.85) to as much as 96 to 26 (factor of 3.69) in favor of algorithmic test case generation. Similarly PairTest detected all the configuration faults in 18 test cases where as 24 manual test cases were required to detect all the configuration faults in the system. Using these results as summarized in Table 1, the graph of fault coverage vs. number of test cases was plotted as shown below in Figure 13.

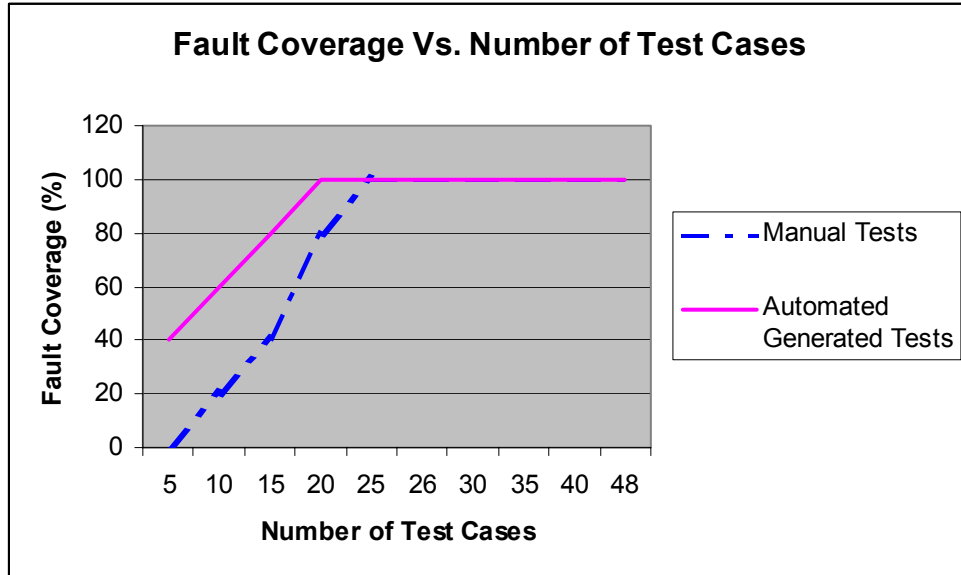


Figure 13. Fault coverage Vs. Number of Test Cases (PairTest)
 * Fault coverage is the ratio of the faults detected

The failure coverage obtained from the empirical data from industry is plotted as shown in Figure 14.

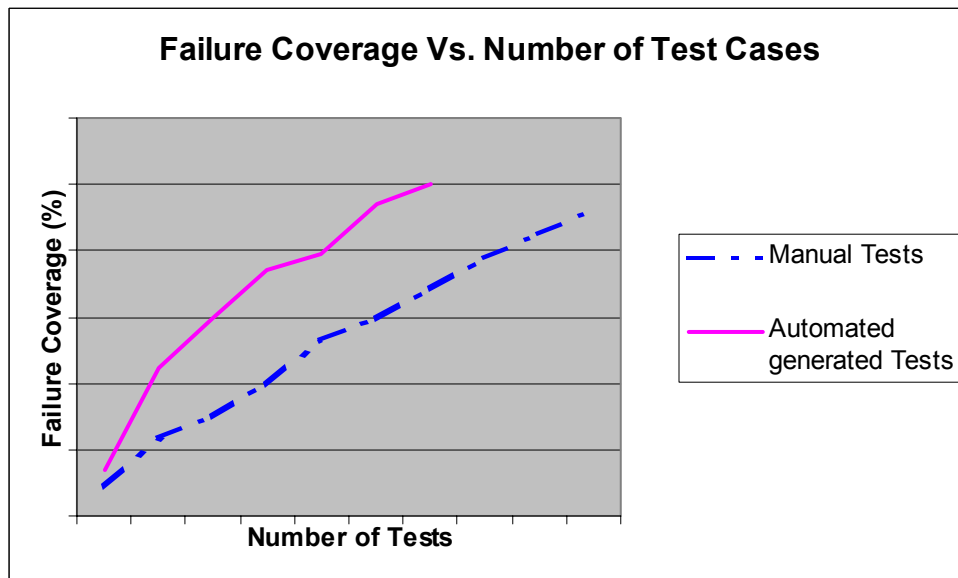


Figure 14. Number of test cases Vs. Failure Coverage -INDUSTRIAL
 * Failure coverage is the ratio of the failures detected

* Raw data values are presented in Appendix A in sanitized form to preserve the proprietary nature of data

Figure 14 illustrates the relative relationship between manual (dashed line) and algorithmically generated test suites observed in the industrial setting. The target was a network-based routing system, manual test cases were generated based on specifications, and automated test cases were generated using a variant of pairwise approach such as AETG. We see that, as in the laboratory experiment, the failure detection rises faster for algorithmic test cases than it does for manual test cases. However, both suites eventually detect a similar number of faults in the system. Of course, since the total actual number of faults is not known in the industrial case, this graph can only tell us algorithmic (in this case specification-based testing) appears to be more efficient in terms of resources (number of test cases) it uses to achieve a similar quality to the one achieved with manual test suites. The actual raw data is listed in Appendix A.

5.3.2 Efficiency

To further investigate the dynamics of the testing efficiency we turn to another metric: testing efficiency as defined in [RV95]. Assuming that in practice resources are constrained, the number of test cases could be considered as the “space” that needs to be covered. Hence, one can “cover” the test-suite (which does not change dynamically once generated). Equally, one could cover any other fixed (planned) resource, such as testing time.

Coverage based testing implies that “good” test cases are those which increase overall coverage. However, the main concerns are whether the test cases find faults, how efficient they are in finding these faults and how repeatable is the process of generating the test cases with same or better quality (efficiency) [RV95]. One family of such coverage based models is based on Hyper-Geometric Distribution [RV95]. Applying the Fault detection model to hyper geometric distribution empirical function for testing efficiency can be defined as shown in equation (5.3.2.1).

$$G_i(Q_i) = \frac{\Delta E_i (1 - Q_i)}{(N - E_i) \cdot \Delta Q_i} \quad [\text{RV95, RV98}] \quad (5.3.2.1)$$

Where $G_i(Q_i)$ is the testing efficiency at instant (step) i

ΔE_i is the difference between the observed number of cumulative failures detected at i and $i - 1$

Q_i is the cumulative coverage till i

ΔQ_i is the difference in coverage metric at i and $i - 1$

The above equation basically provides an estimate of the efficiency normalized with respect to the remaining uncovered “space,” which is a somewhat fairer metric than just instantaneous efficiency.

However, average testing efficiency (running average) may be a more stable an indicator of how the efficiencies of the two approaches compare. It is calculated as shown in equation (5.3.2.2).

$$\text{Average Testing Efficiency} = \frac{\sum_{i=1}^n G_i(Q_i)}{n} \quad [\text{RV98}] \quad (5.3.2.2)$$

Where $G_i(Q_i)$ is the testing efficiency at the instant i

n is the number of intervals

Figure 15 shows the plot of the instantaneous (normalized) and average testing efficiency of the experimental PairTest test set and the experimental Manual test suite against the number of test cases.

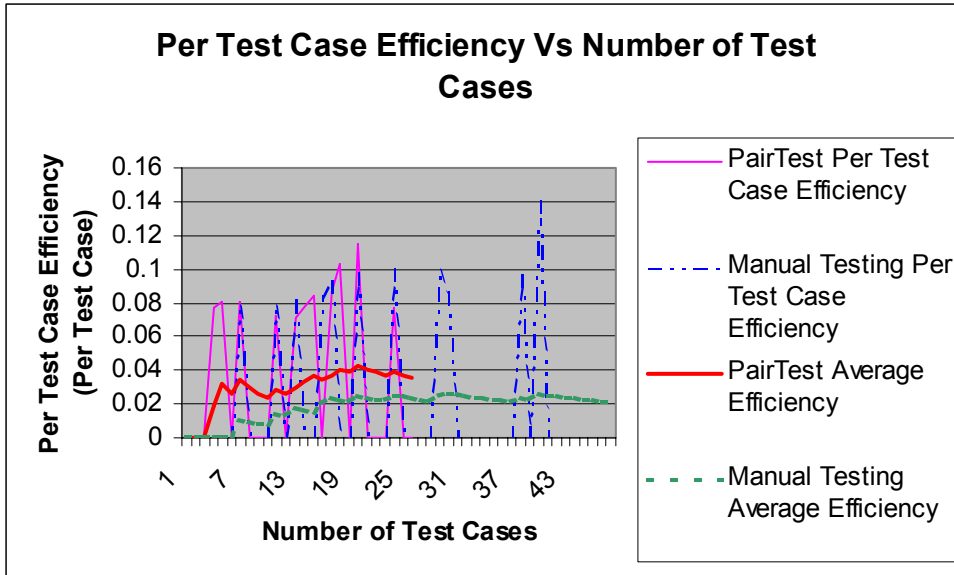


Figure 15. Per Test Case Efficiency Vs. Number of Tests

* Per Test Case Efficiency is the function of number of failures

As can be seen from Figure 15, average PairTest efficiency was higher than the average manual efficiency. In both cases, average efficiency appeared to be a slow varying function of the number of test cases.

A somewhat different view of the same information can be obtained by plotting the efficiency against the fault coverage. The reader should note that this is not possible in real life situations since we never know what the total number of faults/failures is, however this is a good tool in fault-seeding experiments where the number of seeded faults is known. Figure 16 shows such a plot. We see that PairTest appears to be consistently more efficient than manual testing. Of course, the finding may not hold for another algorithmic method.

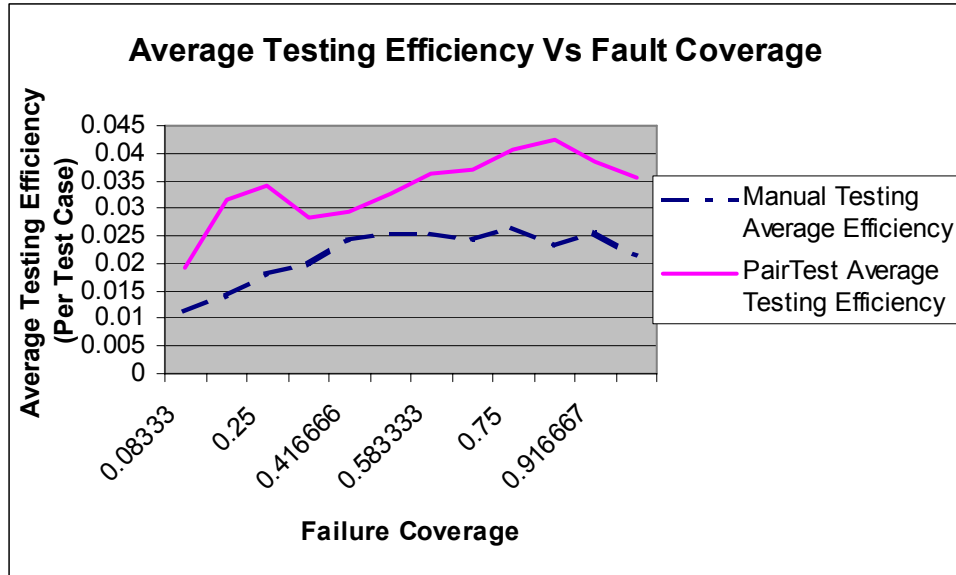


Figure 16. Average Testing Efficiency Vs. Failure Coverage

* Average Testing Efficiency is the function of number of failures

5.3.3 Resources

Resources used for test case generation and execution are another measure that can be used to assess the value of automation. Resources are calculated as a percentage of the total resources spent. In general, resources account for monetary, mechanical, and human related expenditures and effort required for generation of test cases and/or automation. For our experiment we have converted all resources in person-hours units. For example:

- Resources required for provision of the tools for test case development and automation, this includes the monetary value of the tool. (Units person-hours)
- Resources required for incorporation of the tool into the development process.
- Resources required for training in the use of the tool (Units person-hours).
- Resources required for mechanical modifications in the test system due to incorporation of the tool. (Units person-hours)

- Resources required in terms of human effort needed to use the tool, this includes salary and other wages for the people using the tool (Units person-hours)
- Resources required for actual test generation/ automation process using the tool. (Units person-hours)
- Etc.

In general, although the cost of testing assisted with automated tools will be less than that of manual testing, there is an initial cost associated with both manual test case generation and with the set-up and use of automated tools. This is depicted in Figure 17 which lists the distribution of efforts in the industrial environment for such test generation tools.

As shown in the Figure 17, in the initial part of the testing cycle a large amount of resources was consumed in setting up automated tools. On the other hand, the actual resources required for test generation are less than those for manual testing. Automated tools consume an almost constant amount of resources as testing progresses, whereas resource consumption for manual testing shows an increasing trend with incremental changes in the system. However, resource consumption for manual test generation process stabilizes and is constant at a much later time in testing cycle and this stable value is much higher than respective value for automated test generation. Explanation for these using incremental changes is given in following sections and in Appendix A.

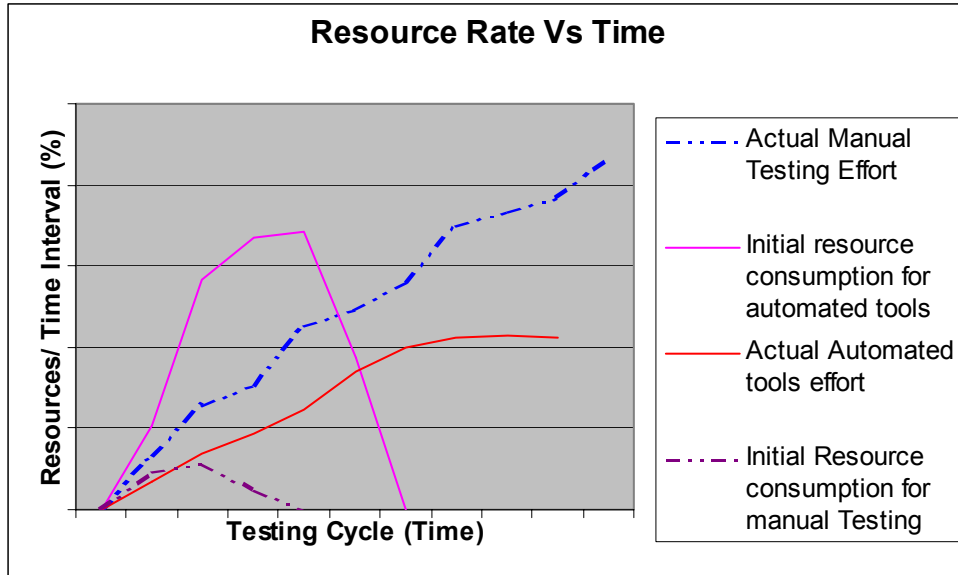


Figure 17. Resource Distribution over Testing Period – INDUSTRIAL

* Same amount of resources are available for automated as well as manual testing in each interval

5.3.4 Test Development Resources

This subsection explores the relative cost of manual and automated test case generation in our laboratory setting. PairTest generated 26 test cases to test the system described in Figure 11. Table 2 shows the resources spent on automated and manual test case generation over a period of about 20 days. For normalization purposes, it was assumed that the total effort or time that could have been spent for test generation during each time period of two days was about 15 hours per activity (manual or automated) - as automated test generation and manual test generation was not done simultaneously. Table 2 depicts the values of resource consumption in person-hours units from our laboratory experiment Figure 18 shows normalized resource expenditures (per two day period) for automated and manual testing vs. the testing wall-clock time in days.

| | | | | | | | | | | |
|---|-----|-----|------|----|-----|-----|------|----|------|----|
| No. Of Days | 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 | 20 |
| Total Resources used for Automatic Test Generation (person-hours) in 2 day period | 2.5 | 10 | 11.5 | 12 | 8 | 6.5 | 6.5 | 5 | | |
| No. of test cases generated in each interval | 0 | 1 | 1 | 2 | 3 | 6 | 7 | 6 | | |
| Total Resources used for Manual Test Generation (person-hours) in 2 day period | 2 | 3.5 | 4.5 | 7 | 7.5 | 8.5 | 10.5 | 11 | 11.5 | 13 |
| No. of test cases generated in each interval | 0 | 1 | 2 | 4 | 5 | 6 | 7 | 7 | 7 | 9 |

Table 2. Resources Vs. Time (Test Generation) data

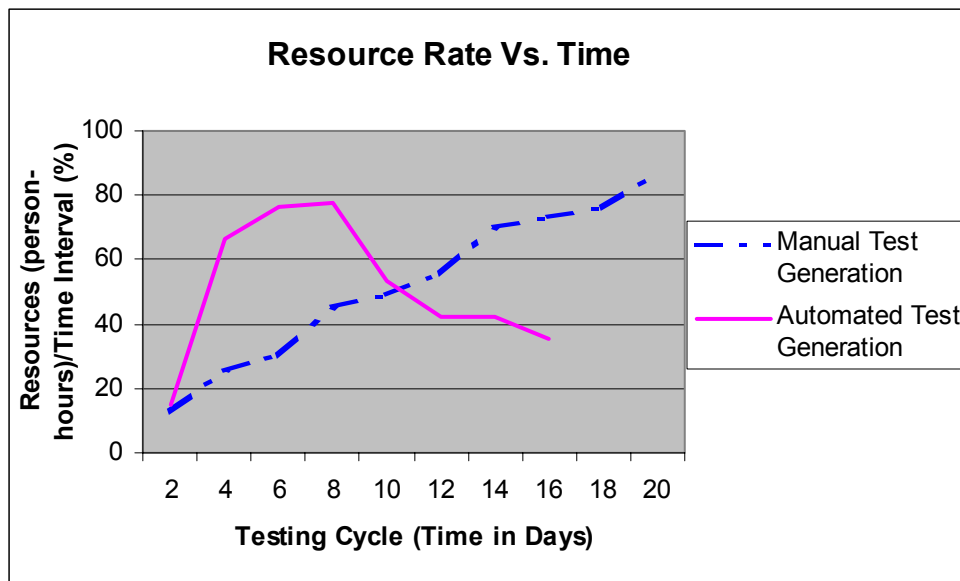


Figure 18. Resource Rate Vs. Time (Test Generation- PairTest) – LABORATORY

- * For each time period maximum amount of resources available were fixed in person-hours and same for manual as well as automated testing.
- * Resources / time period (%) = (Amount of resources consumed in person hours/ Total amount of resources available in each time interval)* 100

We see that PairTest consumed more resources in the initial phase of the testing cycle due to the setup costs, as well as due to the resources required for conversion of the test system parameters into PairTest parameters and relations. In the later part, PairTest generated tests by simple modifications of parameters which required almost same amount of resources. In contrast, resources required for manual test case generation in each time period increased steadily as analysis of the system progressed. The reason for this can be attributed to more number of test cases generated in each time period with incremental changes in the system during each time period. As manual test generation progressed more number of test cases was generated as the test system changed with incremental changes in each period thus accounting for increased resources. The effect caused by the incremental changes on the test system was same in each time interval. These incremental changes in the system are listed in Appendix A.

However, in complex systems with many numbers of test cases, there will be a point when manual test generation will reach a point at which maximum number of test cases will be generated in each time period. So the resources required for generation of such test cases will be constant making manual test generation effort almost constant for each time period. This point was not observed in our laboratory experiment. Even though resources required for manual test generation will stop increasing and is constant after that point, its value will still much higher than the corresponding automated test generation resources. Also after stabilization of automated test generation process number of test cases generated will be dependent on the configuration of the system inspite of the constant amount of resources required for test generation.

The graph in Figure 19 demonstrates the general trend obtained from empirical data in industrial environment for consumption of resources during testing cycle

with manual generation of test cases as well as use of automatic test generation tools. The actual values are presented in sanitized form to preserve the proprietary nature of the data.

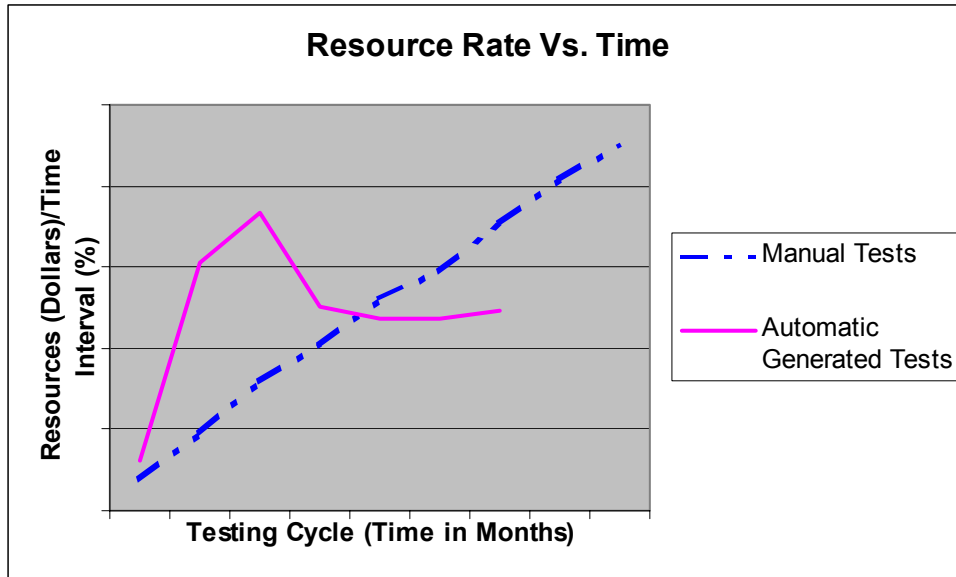


Figure 19. Resource Rate Vs. Time (Test Generation) – INDUSTRIAL

* Raw data values are presented in Appendix A in sanitized form to preserve the proprietary nature of data

As can be seen from the graph above; resources utilized for the generation of test cases increase with the progress of the testing cycle of the system for initial part. For manual test cases, consumption of resources is more or less linear considering the equal amount of incremental changes in product under test, in each time interval and progress of testing at a steady pace with increase in the number of test cases generated in each time period.

Automated tests consume large amount of resources for initial setup costs. Resources required to setup the tools as well as training make automated test case generation costly compared to manual tests in the initial part of the testing cycle. But after a period of adjustment for the tool, the curve for automated test generation shows decrease in the resources consumed before stabilization. For the rest of the testing cycle test generation tools display almost constant consumption of resources accounted due to almost same amount of effort

required for test generation thus reducing resource consumption for the overall testing cycle. However, number of test cases generated after stabilization is dependent on the configuration of the system. Manual test generation requires almost constant resources for test generation in each interval after reaching the maximum threshold for test generation in each interval. This point was not reached in the graph in Figure 19. However, even after that point, at same instance of testing cycle, automated test generation process requires much less amount of resources for test generation in each time interval than manual counterpart.

Intersection point of the two curves is primarily dependent on complexity of testing as well as type of tool utilized.

5.3.5 Test Execution Resources

Automation for execution of test cases developed by PairTest for test system in Figure 11 was developed which is displayed in Appendix A. Values in Table 3 indicate effort required for development of automation in number of person-hours for a five day period. Automation system adapted to evolution in the system of Figure 11 when parameters of the system were changed to develop new PairTest test set new_m10 as described in Appendix A. It was assumed that the total time that can be spent on manual testing as well as automation was 40 hours each five day period – as automation and manual test execution were not performed as simultaneous processes.

Automated test execution requires high amount of resources in the initial part of testing cycle as analysis needed for development of automation requires more effort. The large parts of this consumption are the resources required for tool setup for automation. For manual test execution resource consumption increases linearly as there are more number of test cases executed in each time period accommodating for equal amount of incremental changes in respective time period, in the system with progress of testing cycle. After stabilization of

automation, number of test cases executed per time period is dependent on complexity and configuration of the system.

However, in complex systems with many numbers of test cases, there will be a point when manual test execution will reach a threshold point at which maximum number of test cases will be executed in each time period. So the effort required for execution of such test cases will be constant making manual test execution effort almost constant for each time period. This point was not observed in our laboratory experiment. Even though manual test execution effort will stop increasing and is constant after that point, its value will still much higher than the corresponding automation. Also at any instant number of test cases executed by automation will be higher than the corresponding number of test cases executed by manual test execution.

| | | | | | | | | | | |
|--|-----|------|----|----|------|------|------|----|------|------|
| No. Of Days | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 | 50 |
| Resources for Automation (person-hours) each time period | 6.5 | 28.5 | 31 | 32 | 24 | 18 | 18.5 | 20 | 20.5 | 18.5 |
| No. of test cases executed in each time interval (Automated) | 2 | 2 | 3 | 6 | 6 | 7 | 8 | 8 | 9 | 11 |
| Resources for Manual Testing (person-hours) each time period | 5 | 9.5 | 13 | 18 | 20.5 | 22.5 | 27.5 | 29 | 30 | 34 |
| No. of test cases executed in each time interval (Manual) | 1 | 2 | 3 | 4 | 4 | 5 | 6 | 7 | 7 | 9 |

Table 3. Resources Vs. Time (Test Execution) data

The graph in Figure 20 was plotted from values in Table 2.

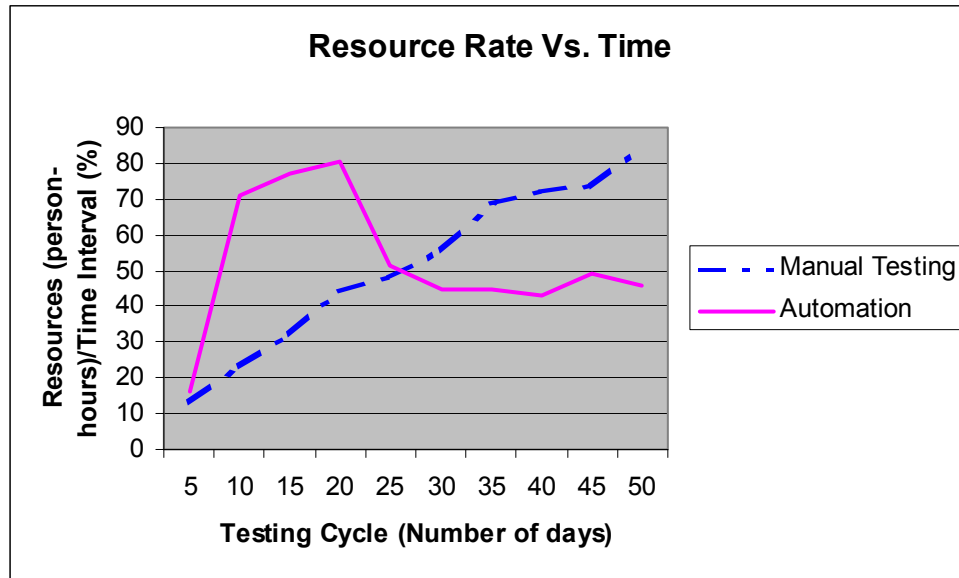


Figure 20. Resource Rate Vs. Time (Test execution) data - LABORATORY

- * For each time period maximum amount of resources available were fixed in person-hours and same for manual as well as automated testing.
- * Resources / time period (%) = (Amount of resources consumed in person hours/ Total amount of resources available in each time interval)* 100

Above graph displayed that during initial phase of automation large amount of resources were required for careful analysis and development of automation for system under test. Also mechanical setup for the automation tools, “Expect”, consumed large amount of setup time during the initial phase thus increasing the resource consumption. After initial increase, number of hours required to carry out test execution using automation reduced and was almost constant as automation adapted to system to execute test cases. When system under test changed slight rise was seen in the number of hours required to adapt to changes in system under test but it was observed that automation adjusted to the change effectively. For manual test execution number of hours required for test execution increased as more number of test cases was executed each time with progress in testing cycle accounted by incremental changes in the system under test. Effect of incremental changes to the system under test was same in each time interval. These incremental changes are

listed in Appendix A. It was seen that automation adjusted to the changes in the system efficiently thus maintaining resource consumption almost constant for execution of more number of test cases in contrast to increasing resource consumption for manual test execution.

Automation executed 26 test cases generated by PairTest in around 30 days where as to test same system using manual testing, manual test execution required more days to execute the test cases generated manually. The graph for automation was extended to match manual test execution process to show that automation adapted to changes in system effectively.

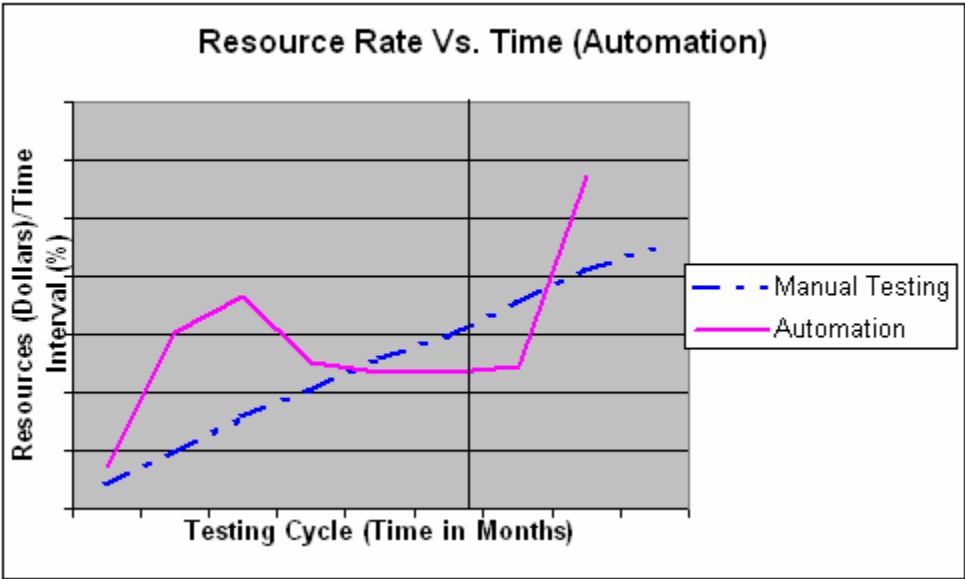


Figure 21. Resource Rate Vs. Time (Test Execution) –INDUSTRIAL

* Raw data values are presented in Appendix A in sanitized form to preserve the proprietary nature of data
 * The vertical line in the graph is indicator of the plot of evolvability in next section

Similarly from graph in Figure 21 obtained from empirical data in industrial environment, it can be observed that the curve of manual test execution for resource consumption continues rising linearly as depicted in the earlier graph due to increase in the number of test cases executed. This increase in the number of test cases executed is attributed to the equal amount of incremental changes in the system in each time period. Automation curve shows a spike in

the consumption of resources due to initial setup costs. It then reduces the resource consumption during the testing cycle due to reduction in manual as well as mechanical efforts required for the test execution. If automation is not efficient to adapt to changes in system then during later part of the testing cycle it can again show a spike in the consumption of resources as automation can consume more resources for maintenance as system under test evolves with new features. This was observed in industrial setting as can be seen from graph in Figure 21. It can be observed by comparison of the above graphs that if the automation is not efficient to adapt to changes in the test system then edge of automation over manual test execution is lost due to increasing maintenance costs. The raw data for the graph in Figure 21 is shown in Appendix A in sanitized form to preserve the proprietary nature of data. Figure 20 confirms that current automation depicted in the Appendix A adapts to changes in the system efficiently.

The vertical line in the Figure 21 indicates the approximate testing cycle time used for explanation of evolvability issue in industrial environment in the following section.

The intersection for both the curves varies according to complexity of system under test as well as changes in the system over a period of time. It is of prime importance that region of the automation curve approximately parallel to x-axis is maximum so as to utilize automation to improve efficiency. So automation should be developed so as to adapt to system changes efficiently

5.3.6 Evolvability

Evolution can be defined as a process which encompasses not only development but all aspects of adaptation, fixing and maintenance. Any software generally does not deteriorate through use but it must be evolved to maintain it satisfactory with respect to the changing operational domain or purposes. If

measure for evolvability is not provided, increasing complexity and other aging effects become a significant cost factor. Analysis of evolvability can be provided in various ways. We have chosen to provide analysis for evolvability using cost model incorporating modification complexity.

Evolvability can be defined as shown in equation (5.3.6.1).

$$\Delta \text{ Effort} \cdot \Delta \text{ No. of Test cases (t)} = A \quad (5.3.6.1)$$

Where Δ effort = change in resources in consecutive time intervals

Δ No. of Test cases (t) = coverage of number of changes in system.

A = Variable value (Measure of evolvability.) [CJH01; Pus02]

Lower value of Measure of evolvability indicates high evolvability as shown in equation (5.3.6.2). Unit change is considered to be minimum value for Δ effort and Δ No. of Test cases (t).

1

$$\text{Actual evolvability of system} = \frac{1}{\text{Measure of Evolvability}} \quad (5.3.6.2)$$

5.3.7 Test Generation evolvability

By considering the system in Figure 11 and considering the PairTest set new_m9 and manual tests shown in Appendix A, evolvability for the automated as well as manual test suites was calculated.

| Time Interval (days) | (2-4) | (4-6) | (6-8) | (8-10) | (10-12) |
|---|-------|-------|-------|--------|---------|
| Δ efforts coverage PairTest (Person-hours) | 35 | 38 | 26 | 14 | 9 |
| Δ efforts coverage Manual Tests (Person-hours) | 24 | 28 | 24 | 21 | 14 |
| Cumulative No. of Test Cases (PairTest) | 1 | 3 | 6 | 12 | 19 |
| Cumulative No. of Test Cases (Manual) | 2 | 4 | 8 | 15 | 27 |

Table 4. Evolvability Vs. Time (PairTest – Test Generation) data

Δ No. of test cases includes coverage of new generated test cases. By using values in Table 4 and formula for evolvability from equation (5.3.6.1) and (5.3.6.2), the graph in Figure 22 was plotted.

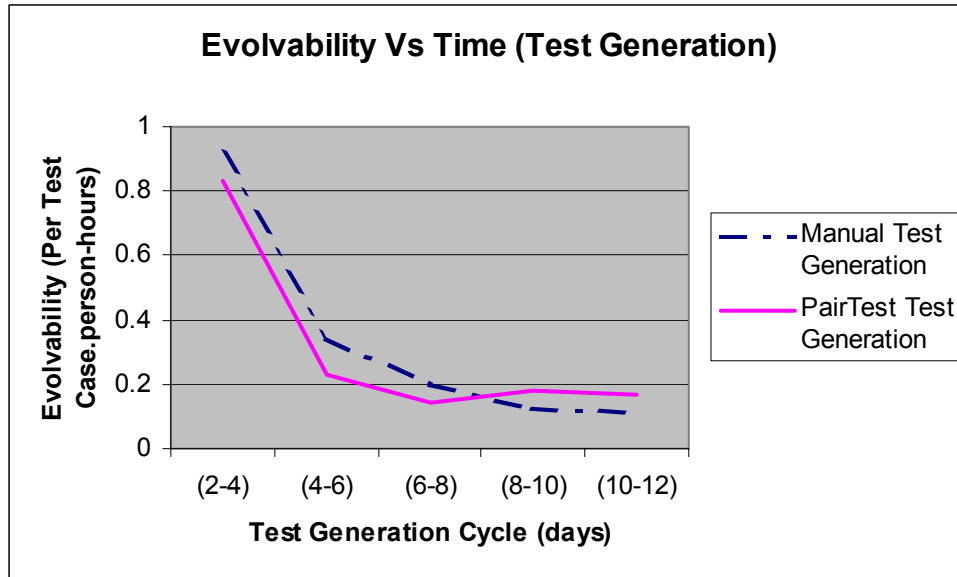


Figure 22. Evolvability Vs. Time (Test Generation - PairTest)-LABORATORY

* Evolvability is calculated using cost model taking into account modification complexity

Evolvability was considered to be of value one initially as system without any constraint was considered most evolvable. It can be seen from the above graph that PairTest and manual test suites displayed linear decrease in the evolvability with the progress in the testing cycle for initial part. This could be verified by rise in the amount of resources required to adapt to changes in the system under test. PairTest displayed little lower evolvability than manual test suite accounted by initial high setup cost than manual test generation as well as cost to acclimate to the system as new parameters or relations were added.

In the latter part of the testing cycle, as PairTest got acclimated to the system; evolvability of the test suite generated by PairTest stabilized as compared to manual test cases as resources required to generate test cases by PairTest

stabilized. However, manual test generation showed decreasing evolvability accounted by increasing effort required for generation of new test cases.

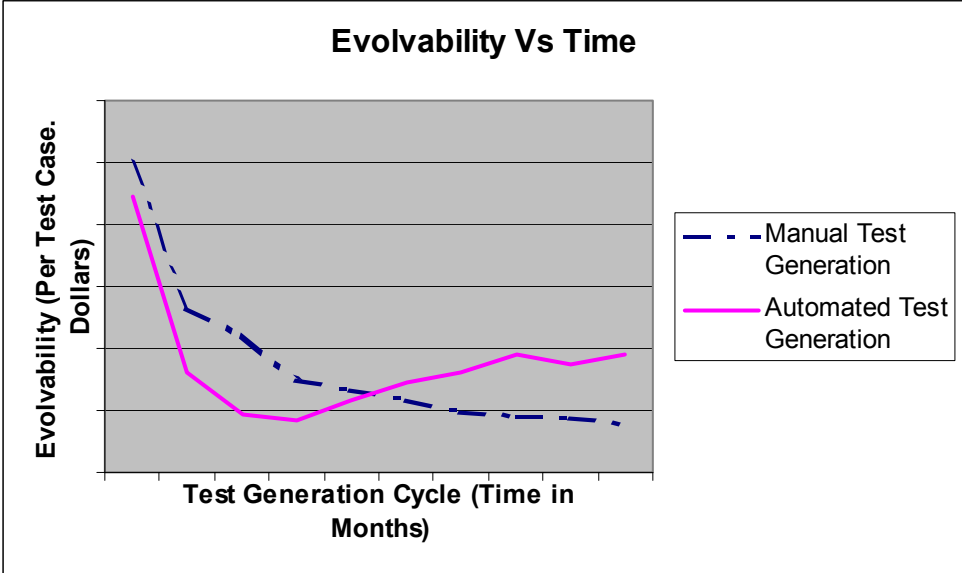


Figure 23. Evolvability Vs. Time (Test Generation) - INDUSTRIAL

* Evolvability is calculated using cost model taking into account modification complexity
* Raw data values are presented in Appendix A in sanitized form to preserve the proprietary nature of data

It can be seen from the graph in Figure 23, obtained from the empirical data in industrial environment, that evolvability of the tool-based and manual test cases decreases more or less with the progress in the testing cycle of the system which is attributed to increase in the amount of efforts required to adapt to system changes. For manual test cases evolvability of test cases is higher in the initial part considering the high consumption of resources for setup of test generation tools which reduces evolvability due to cost model. But as test generation tools get acclimated to the product under test, rate of decrease of evolvability of automatic test generation process reduces as compared to manual test generation.

Meanwhile evolvability of manual test suite continues to show decreasing trend as more effort is required to generate tests to adapt to changes in system. It appeared that the rate of decrease of evolvability for manual test cases is higher than that of test generation tools.

The intersection for both the curves varies according to complexity of system under test as well as changes in the system over a period of time.

5.3.8 Test Execution Evolvability

Following table indicates the respective values for Δ effort and No. of modifications for manual testing as well as automation and implementation described in Appendix A. At a particular instance of time in testing cycle number of test cases executed by automation and manual testing were different. Similarly number of modifications required to adapt to changes in the system in automation as well as manual tests were different at that instance of time.

| Time Interval (days) | (5-10) | (10-15) | (15-20) | (20-25) | (25-30) | (30-35) | (35-40) | (40-45) |
|---|--------|---------|---------|---------|---------|---------|---------|---------|
| Δ efforts coverage Manual Testing (Person-hours) | 11 | 11.5 | 12 | 12.5 | 13.5 | 13 | 12 | 11.5 |
| Δ Efforts coverage Automation (Person-hours) | 55 | 26 | 19 | 11 | 5 | 3 | 2.5 | 1.5 |
| No. of modifications (Manual Testing) | 4 | 9 | 12 | 20 | 28 | 32 | 39 | 56 |
| No. of Modifications (Automation) | 1 | 1 | 1.5 | 2 | 6 | 11 | 13 | 24 |

Table 5. Evolvability Vs. Time (Automation) data

The graph in Figure 24 was plotted by using the formula and values in table 5.

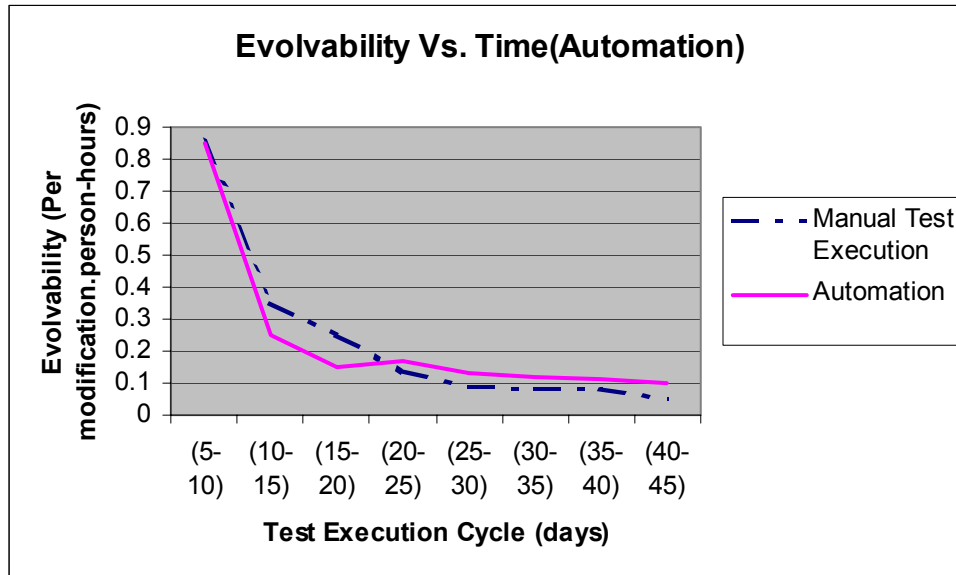


Figure 24. Evolvability Vs. Time (Automation) – LABORATORY

* Evolvability is calculated using cost model taking into account modification complexity

The graph showed decrease in the evolvability for automation as well as manual test execution in early stages of test execution cycle due to resource consumption to acclimate to the test execution parameters during development of automation. Automation showed higher rate of decrease in the rate of evolvability due to high resources needed for initial setup costs for automation.

Rate of decrease of the evolvability reduced as automation got accustomed to the system under test thus reducing resources required for automation to adapt to changes causing evolvability of the system to stabilize. Slight increase in the evolvability was attributed to the decrease in the consumption of resources due to updated system parameters in automation. Automation displayed very slow rate of decrease in evolvability as changes in the system were accommodated with minor modifications as compared to manual test execution process. Manual test execution process showed continuing trend of decreasing evolvability as time and resources required to execute more

number of test cases continued to increase. It appeared that current automation technique adapted quickly to the changes in the system.

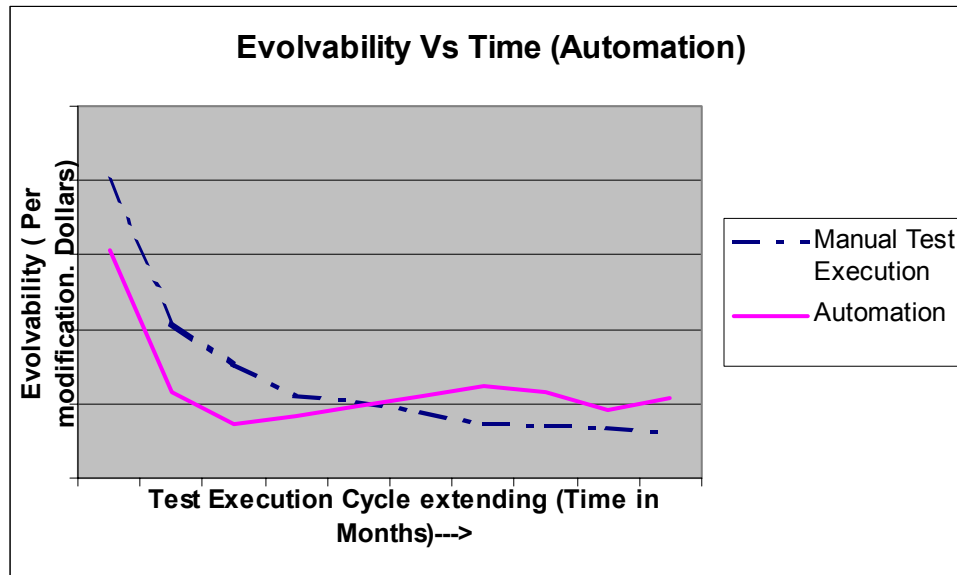


Figure 25. Evolvability Vs. Time (Automation)-INDUSTRIAL

* Evolvability is calculated using cost model taking into account modification complexity

* Raw data values are presented in Appendix A in sanitized form to preserve the proprietary nature of data

Similarly from the graph in Figure 25, obtained from the empirical data in industrial environment, it can be seen that automation and manual tests show decreasing trend in the evolvability of the test execution process in the initial part of the testing cycle. This is attributed to the increasing resource consumption for setup. Automation shows higher rate of decrease in evolvability as setup costs for automation are higher than the manual test execution process with reference to cost model.

But as time passes, with the addition of new features to the system under test, rate of decrease of evolvability stabilizes for automation as almost same amount of resources are needed for test execution in contrast to manual test execution process. Thus efficient automation process shows higher evolvability to adapt to changes in system under test than manual test execution process.

Therefore care should be taken before implementing automation that it is evolvable for considerable period of time of the total testing cycle. The intersection point of the two curves varies according to the complexity of the system under test, changes in the system over time as well as development of automation.

Following observations were done from the analysis of the above graphs.

- As number of test cases increased, manual test cases and test cases generated using test generation tools showed linear increase in the fault detection coverage with increasing number of test cases. Fault detection coverage for PairTest was higher than manual counterpart due to algorithmic approach.
- Test cases generated using tools displayed higher testing efficiency than manual test cases for the system under test. PairTest test suite showed an increasing trend in the efficiency of the test cases with increasing fault coverage where as almost constant trend was observed for manual test suite for similar conditions.
- Automatically generated test cases displayed high consumption of resources in the initial part of the testing cycle but the consumption reduced in the later part and was constant for most part of testing cycle. Consumption of resources for manually generated tests was more or less linear with increasing trend due to incremental changes in the system.
- More resources were used in the initial part of testing cycle for automation due to setup costs. As automation was developed to adapt to changes in the system it maintained consumption of resources for changes in system. Automation depicted in Appendix A showed such a constant resource consumption thus displaying higher adaptability. Manual test case execution showed linear increase in resource consumption with incremental changes in system for most part of testing cycle indicating high overall resource consumption.

- Test suites developed using test generation tools appeared to show higher evolvability as compared to the manual test suites which was accounted by the algorithmic test generation process in tools which reduced amount of effort required to adapt to changes in system.
- It appeared that efficiently developed automation showed higher evolvability than manual test execution process which was characterized by the adaptive nature of automation which reduced effort needed for execution of test cases as compared to manual test execution process.

Chapter 6

Conclusions

6.1 Summary

We enhanced PairTest and developed automation to achieve following goals: a) To develop an approach that maximizes fault-detection properties of the test-suite and minimizes the resources needed to develop these test cases b) To develop a method that allows a resource-efficient execution of these test cases c) To develop a resource-efficient way of updating/maintaining the test-suite and corresponding automation in the face of changes in the software it is intended to test. These goals were motivated by the need for efficient testing tools in the current industrial environment to facilitate and improve testing process. To achieve these goals we analyzed the concept of automating test case design and studied specification based test generation strategy called pairwise which maximized fault detection capability in a cost effective way.

Industry definition of *Testing can be defined as a process of establishing confidence that a program or system does what it is supposed to do* [Few99]. But *complete testing of any program or software is not possible* [Few99]. The basic premises, which automatic test generation helps are based on the management of the resources used for the testing. Automatic test generation techniques can save the time for test case generation for time consuming and/or complex test case generation. Using techniques such as specification based test case generation; effective algorithms for test case generation were developed to aid testing cycle. The need for translation to

convert specification based test generation models into automated test vector generation was accommodated using Test Automation Framework. Using pair wise strategy for test generation high fault detection capability along with ability to generate lesser number of test cases was achieved. High fault detection capability of pairwise along with its ability to generate less number of test cases was used in In-Parameter Order algorithm to develop and enhance an open source automated test generation tool called PairTest for generation of effective test suites thus aiding test generation process. Cost effectiveness of pair wise strategy with high fault detection ability was proved as an efficient approach for test generation in the competitive and time sensitive nature of the industry with case studies and their analysis. The advantage of tools lies in the fact that these tools are more thorough and accurate than a human tester using same algorithm and also they are much faster than most of testers. Test generation tools such as AETG, PairTest or DFT tools are revolutionizing test generation process and redefining testing cycle.

In an attempt to achieve more with fewer resources many organizations are researching ways to test software effectively. Similar to automatic test case generation, execution of those test cases automatically is skill very different from testing. The advantage of automation is it establishes greater confidence in the system under test in less time by simulating large input bases or by automating menial tasks thus making better use of available resources [Per95]. It also helps to provide consistency and repeatability of tests as the efforts put into design and generation of a test case gets divided over many executions of the test. Using the automated life cycle test methodology, a method for effectively translating PairTest test cases into executable test cases was developed. Using Expect as a tool, cost effective and adaptable test execution and result analysis approach was developed to aid testing process. Development of adaptable, intelligent and easy to use automation as showed in Appendix A is helping many organizations automate menial tasks of test execution to effectively manage available resources.

A quantitative analysis of pair-wise testing and test automation in industrial environment showed that, compared with manual methods, automatic test design and execution in general help a) increase the error detection capability (efficiency) of

testing, b) reduce resource consumption, and c) increase evolvability (maintainability, adaptability) of the testing process. We were successfully able to provide a cost effective open source test generation tool with high fault detection capability and easy to use, adaptable automation prototype to provide their combination as a complete testing solution to improve testing process and meet current demand of efficient testing tools in industry.

6.2 Conclusion

This work demonstrated use of specification-based test generation approach for development of a resource-efficient method for test generation with high fault detection capability. This work showed use pairwise strategy, a specification-based test generation strategy for automated test generation using Test Automation Framework approach. It covered use of In-Parameter Order algorithm, an algorithm based on pair wise strategy, to enhance an automated test generation tool called PairTest. This work assessed and enhanced feasibility, usability and effectiveness of PairTest. It tested, analyzed and assessed test generation ability of the same in industrial environment. This work emphasized use of PairTest as an open source test generation tool, which provides cost effective and less number of effective tests for testing. It also upgraded and enhanced PairTest to add stability to its GUI based version and added a text based version of PairTest to add flexibility and robustness to PairTest.

This work encompassed generation of a prototype method for translation of test cases into executable test cases. It also used Automated Life cycle Methodology to develop a prototype of an adaptable easy-to-use system, using Expect as a tool, for automatic execution of the PairTest test cases; in a setting to test network-based systems. It assessed the usability of the PairTest as well as automation in an industrial environment. This work also analyzed use of test generation tools as well as automation in industrial environment in comparison to manual testing efforts to provide general trends supported by research and empirical data emphasizing benefits of use of such tools. This work provided combination of PairTest and automation as

‘complete’ testing solution to improve quality and efficiency of the testing process in a cost effective way saving valuable testing time.

6.3 Future Work

Algorithms used by automatic test generations tools are developed by human intelligence. These tools using these algorithms are not able to identify additional requirements in testing cycle or they cannot generate additional tests yet as an expert human tester. Human experience in the area of testing is still irreplaceable in spite of the use of tools. Use of memory or artificial intelligence along with such tools to store the ‘knowledge’ gained by use of them can be very helpful in developing test cases. Such tools which can ‘learn’ from the previous experiences and can use that experience to enhance quality of the test cases can be area of future research.

Automation suffers from drawback of reduction in evolvability due to increasing maintenance costs as product evolves if not carefully planned. Need of accurate evolution plan of product hinders extensive use of automation in testing process. With use of databases and artificial intelligence, an automation tool which incorporates product plan to generate flexible and executable automation and executable test cases such that any future changes in the product evolution can be accommodated by reflecting them in the existing automation in a cost effective way with generation of executable test cases to develop updated automation system can be area of future research.

Automatic test generation tools are combining with automation tools to present a ‘complete’ test tool for testing any system. Modification of PairTest to provide n-way test generation along with incorporation of standardized system automations for various systems (Ex: Web, GUI, Network, Function etc.) with use of databases to provide ability of generating user specific executable automations can be area of future research.

References

- [Adr82] Adrion, Richards; Branstad, Martha; Cherniavsky, John. Validation, Verification and Testing of Computer Software. Pages: 159 - 192 Periodical-Issue-Article, Year of Publication: 1982, ISSN: 0360-0300, ACM Press New York, NY, USA.
- [Aoy97] Aoyama, M.; Agile Software Process model, Computer Software and Applications Conference, 1997. COMPSAC '97. Proceedings, The Twenty-First Annual International, Page(s): 454 -459, 11-15 Aug 1997.
- [Aoy98] Aoyama, M.; Agile Software Process and its experience, Software Engineering, 1998. Proceedings of the 1998 (20th) International Conference on software engineering, Page(s): 3 -12, 19-25 Apr 1998.
- [Bei83] Boris Beizer. Software Testing Techniques. Van Nostrand Reinhold Company Inc, New York, Book, 1983.
- [Bei95] Boris Beizer. Black – Box testing techniques for functional testing of Software and Systems. Book by John Wiley & Sons Inc., 1995.
- [Bla98] Blackburn, M. R., Using Models For Test Generation And Analysis, Digital Avionics System Conference, October, 1998.
- [BB96] Blackburn, M.R., R.D. Busser, T-VEC: A Tool for Developing Critical System. In Proceeding of the Eleventh International Conference on Computer Assurance, Gaithersburg, Maryland, June 1996.
- [BBF97] Blackburn, M. R., R.D. Busser, J.S. Fontaine, Automatic generation of Test Vectors for SCR Style Specifications, In Proceedings of the 12th Annual Conference on Computer Assurance, Gaithersburg, Maryland, pages 54-67, June 1997.
- [BF97] Blackburn, M. R., J.S. Fontaine, Test Automation Framework. Herndon, Virginia: SPC-97055-MC, version 02.10.00., Software Productivity Consortium, 1997.
- [BF98] Blackburn, M.R., J.S. Fontaine, Specification Transformation to support Automated Testing,TR SPC-97036-MC, Version 02.00.01, Software Productivity Consortium, March 1998
- [BBNC01] Blackburn, M.R., R.D. Busser, A. M. Nauman, R. Chandramouli, Model based Approach to security test Automation, Quality Week 2001, June 2001.
- [Boc88] Bochmann, G.; Cerny, E.; Gagne, M.; Jard, C.; Leveille, A.; Lacaille, C.; Maksud, M.; Raghunathan, K.; Sarikaya, B.; Experience with Formal Specifications Using an Extended State Transition Model, Communications, IEEE Transactions on [legacy, pre - 1988] , Volume: 30 Issue: 12 , Page(s): 2506 -2513, Dec 1982
- [Boe02] Boehm, B. Get Ready for Agile Methods, with Care. IEEE Computer, pp. 64-69. (Jan. 2002)

- [Burr97] Burr, K. , Young, W., Test Acceleration and Automatic Efficient Test case Generation, Nortel Technical Report, May 1997.
- [CGDDTK96] Cooke,D., A. Gates, E. Demirors,O, Demirors, M. Tankik, B. Kramer, Languages for the specification of Software, Journal of Systems Software,32:pp 269-308,1996.
- [Cha99] Chang, Juei; Richardson, Debra; Structural specification-based testing: automated support and experimental evaluation, SIGSOFT : ACM Special Interest Group on Software Engineering, Pages: 285 - 302 Series-Proceeding-Article, Year of Publication: 1999, ISBN:3-540-66538-2.
- [Chi99] Chillarege, Ram. Software Testing Best Practices, Center for Software Engineering, IBM Research, Technical Report, April 1999.
- [Cisco] Cisco Catalyst 5500 switch
- [CJH01] Cook, Stephen; Ji He; Harrison, Rachel. Dynamic and static views of software evolution. Proceedings of IEEE International conference on Software Maintenance (ICSM'01) pp. 592-601. IEEE Computer Society Press 2001.
- [CL01] T.Y Chen and M.F.Lau. Test case selection strategies based on boolean specifications. Software Testing, Verification and Reliability, Volume 11, Issue 3, 2001.
- [CLI] CLI commands for DiffServ, Ericsson Documentation.
- [COH94] Cohen, D.M., Dalal, S. R., Kajla A., Patton G.C. The Automatic Test Generator (AETG) System, Proceedings of the 5th International Symposium on Software Reliability Engineering (ISSRE '94), Monterey CA USA, 1994, pp. 303-309.
- [COH96] Cohen, D. M., Dalal S.R., Parelius J., Patton G. C., The Combinatorial Design Approach to Automatic Test Generation, International Symposium on Software Reliability Engineering, White Plains, N.Y., Vol.30, No.5, pp 83-88, October 30 – November 2 1996.
- [COH97] Cohen, David M. The AETG System : An Approach to Testing Based on Combinatorial Design, IEEE Transactions on Software Engineering, Vol. 23, No. 7, pp 437-444, July 1997.
- [Dav93] Davis, Alan. Software Requirements, Objects, Functions and States. Book by Englewood cliffs, NJ: Prentice Hall , 1993
- [DDH72] O.J.Dahl, E.W.Dijkstra & C.A.R. Hoarc. Structured Programming, chapter I Notes on Structured programming, A.P.I.C. Studies in Data Processing. Academic Press, London and new York, 1972.
- [Ding99] Ding, Wei; Ammann, Paul; Abdurazik, Aynur; Offutt, Jeff. Evaluation of Three Specification Based Testing criterion, Department of Information and Software Engineering, George Mason University, 1999.

- [DKLL98] Dalal, S.R., Jain A., Karunanithi N., Leaton J.M., Lott C.M., Model Based Testing of a Highly Programmable System, Proceedings of ISSRE'98, p. 174-178, 5-7 November 1998.
- [Dus99] Elfriede Dustin. Automated Software Testing. Book by Addison-Wesley Publications, 1999.
- [Eka91] Ekambareshwar, S.; A Suite of Software Tools for Executing Formal Specifications, TENCON '91.1991 IEEE Region 10 International Conference on EC3-Energy, Computer, Communication and Control Systems, Volume: 2, Page(s): 322 -329, 28-30 Aug 1991.
- [Eric] Ericsson Internal Documentation.
- [Exp] Expect Language (expect.nist.gov)
- [Few99] Mark Fewster. Software Test Automation. Book by Addison-Wesley, ACM Press, 1999.
- [FB98] Fontaine, Joseph and Blackburn M. Automatic Test Generation Support for Object Technologies, Herndon, Virginia: Report in Software Productivity Consortium, 1998.
- [FH88] P.A. Freeman and H.S. Hunt. Software quality improvement through automated testing. In Fifth International Conference on Testing Computer Software, Washington, DC, June 13-16, 1988.
- [Gla01] Glass, R.; Extreme Programming: The Good, the Bad, and the Bottom Line. IEEE Software, November/December 2001, pp. 112, 111.
- [Gou83] Gourlay, J.S., Introduction to the Formal Treatment of Testing, Software Validation. Proceeding of the Symposium on Software Validation, 1983.
- [Gri02] Griche, K.-C.; Automatic inter-procedural test case generation, 17th IEEE International Conference on Automated Software Engineering, 2002. Proceedings. ASE 2002., Page(s): 316, 2002
- [GW94] Ghiassi, M. , K.I.S. Woldman, Dual Programming approach to Software Testing, Software Quality Journal,3:45-58, 1994.
- [Hel95] Heller, E., Using Design of Experiment structures to generate Software Test Cases, 12 th International Conference Testing Computer Software, pp 33-41, June 1995
- [Het88] William Hetzel. The complete guide to software testing. Book by QED Information Sciences, 1988.

- [Hei95] Heitmeyer, C.; Bull, A.; Gasarch, C.; Labaw, B.; Proceedings of the Tenth Annual Conference on Computer Assurance, 1995. COMPASS '95. 'Systems Integrity, Software Safety and Process Security'. Page(s): 109 -122, 25-29 Jun 1995
- [Hie97] Hierons, R. M., Testing from a Z specification, Journal of Software Testing, Verification and Reliability, 7:19-33, 1997.
- [HJL96] Heitmeyer, C., R. Jeffords, Labaw. Automated Consistency Checking of Requirements Specifications. ACM TOSEM, 5(3):231-261, 1996.
- [Hu01] Hu, M.; An application of multiple-valued logic to test case generation for software system functional testing, 31st IEEE International Symposium on Multiple-Valued Logic, 2001. Proceedings. , Page(s): 35 -40, 2001.
- [IEEE90] IEEE Std 610.12-1990 IEEE standard glossary of software engineering terminology.
- [IEEE95] IEEE Std 1348-1995 IEEE recommended practice for the adoption of Computer-Aided Software Engineering (CASE) tools.
- [IEEE97] IEEE Std 1278.4-1997 IEEE trial-use recommended practice for distributed interactive simulation - verification, validation, and accreditation
- [Intel] Intel Pentium III Processor 700MHz
- [Kaner93] Kaner, C., Falk J., Nguyen H.Q., Testing Computer Software, Book by Van Nostrand Reinhold, 1993.
- [Kaner97] Kaner, Cem. Improving the Maintainability of Automated Test Suites, Proceedings of the Tenth International Quality week (Software Research), San Fransisco, Ca, 1997.
- [Kaner97a] Kaner, Cem.Pitfalls and strategies in automated testing; IEEE Journal, Computer , Volume: 30 Issue: 4 , Apr 1997 Page(s): 114 -116
- [Lal97] Jagseesan, Lalita; Porter, Adam; Puchol, Carlos; Votta, Lawrence. Specification Based Testing of Reactive Software: A Case Study in Technology Transfer, International Conference on Software Engineering, 1997.
- [Linux73] RedHat Linux 7.3 Operating system(Valhalla)
- [MAL94] Malaiya, Y. K., Karcich F., Li N., Skibbe R. The Relationship Between Test Coverage and Reliability, Proceedings of the Fifth International Symposium on Software Reliability Engineering, Monterey, CA, pp186-195, Nov 6-9 1994.

- [Mal02] Malishevsky, A.G.; Rothermel, G.; Elbaum, S.; Proceedings of International Conference on Software Maintenance, Page(s): 204 -213, 2002
- [Man85] Mandl, R., Orthogonal Latin Squares: An application of Experimental Design of Computer Testing, Comm. ACM, Vol. 28, No. 10, pp. 1054-1058, October 1985.
- [Mar97] Marick, Brian. Classic Testing Mistakes, Report by Testing Foundations, Consulting in Software Testing, 1997.
- [Mar98] Marick, Brian. When should a Test Be Automated?, Report by Testing Foundations, 1998.
- [Meu98] Meudec, C.; Automatic Generation of Software Test Cases From Formal Specifications, Thesis in Queen's University of Belfast, May 1998.
- [Min] Minicom – a UNIX Telecomm Program
- [Musa97] Musa, J.D.; Introduction to software reliability engineering and testing, Software Reliability Engineering - Case Studies, 1997. Proceedings., The Eighth International Symposium on , Page(s): 3 -12, 2-5 Nov1997.
- [New02] Newkirk, J.; Introduction to agile processes and extreme programming, Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002, Page(s): 695 -696, 2002
- [Ore93] Oresky, D.F.; Haapala, C.W.; Verification and validation in an iterative software development environment, Rapid System Prototyping, 1993. 'Shortening the Path from Specification to Prototype'. Proceedings. Fourth International Workshop on, Page(s): 57 -67, 28-30 Jun 1993.
- [Per] Perl Scripting Language
- [Per95] William Perry. Effective methods of software testing. Book by John Wiley & Sons, 1995.
- [Pet96] Pettichord, Bret. Success with Test Automation, Proceedings of the Ninth International Quality week , Software Research, San Francisco, Ca, 1997.
- [Pet99] Pettichord, Bret. Seven Steps to Test Automation Success, Report by STAR West, San Jose, November 1999.
- [Pet00] Pettichord, Bret. Three Keys to Test Automation, Report by Stickyminds.com, Bret Pettichord's Software Testing hot list. December 4, 2000.
- [Pet01] Pettichord, Bret. Getting a Late Start on Test Automation, Stickyminds.com, Report by Bret Pettichord's Software Testing hot list, January 22, 2001.

- [Ping97] Unix Ping utility, RFC 2151 : A Primer On Internet and TCP/IP Tools and Utilities
- [Poo01] Pole, C.; Huisman, J.; Using Extreme Programming in a Maintenance Environment. IEEE Software, November/December 2001, pp. 42-50.
- [Pos96] Poston, Robert. Automating Specification Based Testing, Los Alamitos, CA: IEEE Computer Society Press, 1996.
- [Pres92] Pressman, Roger. Software engineering (3rd ed.): a practitioner's approach, a book by McGraw-Hill, Inc. New York, NY, USA.
- [Pus02] Pussinen, Mika. A Survey of Software Product Line Evolution, Institute of software systems, Tampere University of Technology, December 19 2002.
- [PB89] Patil, S. and Banerjee, P. A parallel branch and bound approach to test generation in Proceedings of the Design Automation Conference, (Las Vegas, NV), pp 590-594, June 1989.
- [Rob00] Robinson, Harry. Intelligent Test Automation, Article in Software Testing and Quality Engineering Magazine, October 2000.
- [Rot02] Rothermel, G.; Elbaum, S.; Malishevsky, A.; Kallakuri, P.; Davia, B.; Proceedings of the 24rd International Conference on Software Engineering, 2002. ICSE 2002, Page(s): 130 -140, 2002
- [Roy70] W.W Royee. Managing the development of large software systems. In IEEE WESCON, August 1970.
- [RV95] Rivers, A; Vouk, M.A. An Empirical Evaluation of Testing Efficiency During Non-Operational Testing, in Proc. 1995 Software Engineering Research Forum, Boca Raton, FL, pp. 41-50, November 1995.
- [RV98] Rivers, A; Vouk, M.A. Resource Constrained Non-operational Testing of Software, in Proceedings of the 9th International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Paderborn, Germany, pp. 154-163, November 1998.
- [Saf2000] Safford, Ed. Test Automation Framework, State Based and signal flow examples, Twelfth Annual Software Technology Conference, May 2000.
- [Sch01] Schuh, P.; Recovery, Redemption, and Extreme Programming. IEEE Software, November/December 2001, pp. 34-41
- [Sch02] Schroeder, P.J.; Faherty, P.; Korel, B.; Generating expected results for automated black-box testing, 17th IEEE International Conference on Automated Software Engineering, 2002. Proceedings. ASE 2002, Page(s): 139 -148, 2002.

- [Sot01] Sotirovski, D.; Heuristics for iterative software development, IEEE Software, Volume: 18 Issue: 3 , Page(s): 66 -73, May 2001.
- [Sta99] Statezni, David, Industrial Application of Model Based Testing, 16 th International Conference and Exposition on Testing Computer Software, June 14-18, 1999
- [Sta2000] Statezni, David. Test Automation Framework, State Based and Signal Flow examples, Twelfth Annual Software Technology Conference, May 2000.
- [Sto93] Stocks, P.; Applying formal methods to software testing, Thesis in Department of Computer Science, University of Queensland, December 1993.
- [Sto96] Stocks, Phil; Carrington, David. A framework for Specification Based Testing, pp 777 -793, Transactions on Software Engineering, November 1996
- [Sun] Sun Microsystems version of Java Developer Kit 1.2
- [Tai01] Tai, K.C. and Lei, Yu. In-Parameter Order: A Test Generation Strategy for pairwise Testing, Technical Report TR-2001-03, Dept. of Computer Science, North Carolina State University, Raleigh, North Carolina, March 2001.
- [Tai02] Tai, Kuo-Chung., Lei, Yu. A Test Generation Strategy for Pairwise Testing, IEEE Transactions on Software Engineering, Vol. 28, No. 1, January 2002.
- [Tcl] Tool Command Language
- [Unix] UNIX operating system.
- [Voas91] Voas J., Morell L. and Miller K., Predicting Where Faults can hide from Testing. IEEE Software, March 1991.
- [vx] vxWorks real time operating system. Windriver systems.
- [Wat96] Watanabe, Aki; Sakamura, Ken. A specification-based adaptive test case generation strategy for open operating system standards, International Conference on Software Engineering, Proceedings of 18th International Conference on Software Engineering, pp 81-89, Berlin, Germany, 1996.
- [Wil96] Williams, A.W. and Probert, R. L., A Practical Strategy for Testing Pairwise Coverage of Network Interfaces, Proceedings of IEEE International Symposium of Software Reliability Engineering, pp. 246-254. 1996.

- [Wil02] Williams, A. W., Software component Interaction Testing, Doctorate Dissertation, Ottawa-Carlton Institute of Computer Science, University of Ottawa. 2002.
- [Zin97] Zin, A.M.; Al-Amayreh, A.; Foxley, E. An Approach to Specification Based Testing, Proceedings of International Conference on Software Engineering Quality SQE'97, Udine, Italy, 1997.

Appendix A

The test cases obtained from the PairTest are automated to reduce the manual work load. Automation scripts are written in 'Expect' and 'Tcl/Tk' to automate the test cases.

These scripts take the output file from the PairTest as an argument. They take each individual test case from PairTest, set the appropriate parameter values from the values in PairTest and the list of values provided for each parameter for the respective system under test. These scripts then generate executable test cases, each of which test a condition and have appropriate parameters set. They also execute the test cases sequentially and store the result of each test case in a result file. These automation scripts can be modified to test different systems with little modifications.

The below mentioned scripts should be kept in the same directory. The test cases are generated in /testdir directory. For execution of these script Expect version 5.3(lowest) should be installed. The automation scripts used are listed below. The main script files are commands and funcTest.inc. New_m10 is an example output file from PairTest which can be different for every run assuming new example file is generated for testing the same system with the list parameters which include the parameters of the system incorporated in the script.

Commands – File to create the executable test cases and create an executable file to run them.

Functest.inc – File with all the functions for executing the test cases.

New_m10 – Output file from PairTest. (This is an example file)

Commands

Copyright (c) North Carolina State University, 2002.

Commands – This file is the script to create all the executable test cases. This script takes as input the PairTest file and generates executable test cases. It also generates a file called 'alltest' which can be used to execute the test cases automatically. The results of automated execution of test cases are listed in file called 'results' in '/testdir' directory. This script can be run as 'commands *PairTest filename*'. However, Expect must be installed on the machine before running this file.

```
#!/usr/local/bin/expect --  
#
```

```
if {$argc != 1} {
```

```

puts "commands.tcl filename"
exit -1
}

#source /root/fdtest2.inc

puts "args : $argc"
puts "argv :$argv"

#gets stdin

set tmpfile [lindex $argv 0]

puts $tmpfile

#gets stdin

#set file1 [ open "d:/$tmpfile" "r"]

set file1 [ open "/root/code/$tmpfile" "r"]

set string1 [read $file1]
puts $string1
set temp "[TESTSET]"
set list1 [split $string1 "\n"]

set index [lsearch -exact $list1 $temp]

set list2 [split [lindex $list1 [expr $index +1] ] ":"]

set numpara [lindex $list2 1]

set list3 [split [lindex $list1 [expr $index +2] ] ":"]

set numtests [ lindex $list3 1]

lindex $list1 [expr $index+2+$numpara+1]

set list4 ""

for { set i 0} {$i < $numpara} {incr i} {
    lappend list4 [lindex $list1 [expr $i+2] ]
}

set flag_int 0
set flag_destaddr 0
set flag_permitip 0
set flag_ipgw 0
set flag_redirect 0
set flag_sendipaddr 0

set flag_portenable 0
set flag_portduplex 0

```

```

set flag_portspeed 0
set flag_portpriority 0
set flag_portmembership 0
set flag_porttrap 0

set mod_num 3
set port_num 1

set destaddr 10.98.16.66
set ipgw 10.98.16.1

set templist1 ""

set templist1 ""

set templist1 ""

for {set i 0} { $i < $numpara } { incr i } {

puts "i value :$i"
set para$i ""
set paraname$i ""
}

for { set i 0} {$i < $numpara} {incr i} {

set templist1 [split [lindex $list4 $i] "-"]

set templist2 [lindex $templist1 1]

set tempnamepara [lindex $templist1 0]
#puts "tempnamepara : $tempnamepara"

set tempnamepara1 [lindex [split $tempnamepara ":"] 1]
#puts "tempnamepara1 : $tempnamepara1"

#lappend paraname$i [lindex [split $tempnamepara ":"] 1]

set paraname$i [lindex [split $tempnamepara ":"] 1]
#puts "paranem$i : [subst ${tmp1}$i] "
set templist3 [ split $templist2 "|"]

set count [llength $templist3]

for { set j 0} { $j < [expr $count -1] } {incr j} {

lappend para$i [lindex $templist3 $j]

}

}

set tmp "para"
puts "numpara : $numpara\n"

```



```

#
# testname = test$i
#
#
# Sourcing common procedure library
#
source /root/fdtest2.inc

#-----
# Compulsory variables
#-----
set testname \"Testing Interface Parameters\"           ;# Name of
the test case
set temp_ip_addr \"10.98.16.10\"
set destaddr $destaddr
set ipgw $ipgw

#set resultfile \[open \"/root/testdir/results\" \"w\" \]
"

set chkval 0

for { set j 0 } { $j < $numpara } {incr j} {

set tmpIgnoreVal [lindex [subst ${tmp}$j] 0]

set tmpindex [lindex $testlist1 $j]
set tmpindexcnt 0
set ignoreValCnt 0

if { $tmpIgnoreVal == $ignoreVal } {
    set ignoreValCnt 1
    puts "The ignore set"
    # ignorevalcnt =1 means value has to be ignored
}

if { $tmpindex == $valX } {
    set tmpindex 0
    set tmpindexcnt 1
    # tmpindexcnt = 1 means value is any
}

set tmpval [lindex [subst ${tmp}$j] $tmpindex]

# chkval = 0 means interface is down

if { $i == 11 } {
puts "para$j : $tmpval"
}
# if { [subst ${tmp1}$j] == "int" }

#if { $j == 0 }

```

```

if { [subst ${tmp1}$j] == "int" } {

    set flag_int 1

    if { !$ignoreValCnt} {

        if { $tmpval == 0 } {

            puts "$i : int down"
            set int_status down
            if { $tmpindexcnt == 1 } {
                set chkval 1
            } elseif { $tmpindexcnt == 0} {
                set chkval 0
            }

        } elseif { $tmpval == 1} {

            puts "$i :int up"
            set int_status up
            set chkval 1
        }

    } else {

        set int_status ignore

    }
    puts "intstatus: $int_status"
}

#if { $j == 1 }
if { [subst ${tmp1}$j] == "destaddr" } {

    set flag_destaddr 1

    #puts "inside destaddr*****"
    if { !$ignoreValCnt} {

        if { $tmpval == 0} {
            set destaddr 10.98.16.67
        } elseif { $tmpval == 1} {

            set destaddr 10.98.16.67
        } elseif { $tmpval == 2} {
            set destaddr 12.98.16.0
            puts "destaddr : $destaddr"
        } elseif { $tmpval == 3 } {
            set destaddr 127.0.0.0
        }

    }

} else {

    set destaddr ignore
}

```

```

}

}

#if { $j == 2 }
if { [subst ${tmp1}$j] == "ipgw" } {

set flag_ipgw 1
if { !$ignoreValCnt} {

    if {$tmpval == 0} {
        set ipgw 10.98.16.1
    } elseif {$tmpval == 1} {
        set ipgw 10.98.16.2
    } elseif {$tmpval ==2 } {
        set ipgw 10.98.16.3
    }

} else {

set ipgw ignore

}

}

#if { $j == 3 }
if { [subst ${tmp1}$j] == "permitip" } {

set flag_permitip 1
if { !$ignoreValCnt} {
    if {$tmpval == 0} {
        set permitip disable
    } elseif {$tmpval == 1 } {
        set permitip enable
    }

} else {

set permitip ignore

}

}

#if { $j == 4 }
if { [subst ${tmp1}$j] == "redirect" } {

    set flag_redirect 1
    if { !$ignoreValCnt} {

        if {$tmpval == 0} {
            set redirect enable
        } elseif {$tmpval == 1 } {
            set redirect disable
        }

    }

}

```

```

} else {

set redirect ignore

}

}

#if { $j == 5 }

if { [subst ${tmp1}$j] == "sendipaddr" } {

    set flag_sendipaddr 1
    if { !$ignoreValCnt} {

        if { $tmpval == 0} {
            set sendipaddr 0.0.0.0
        } elseif { $tmpval == 1} {
            set sendipaddr 0.0.0.1
        } elseif { $tmpval == 2 } {
            set sendipaddr 1.1.1.1
        }

    }

} else {

set sendipaddr ignore

}

}

if { [subst ${tmp1}$j] == "portenable" } {

    set flag_portenable 1
    if { !$ignoreValCnt} {

        if { $tmpval == 0} {
            set portenable disable
        } elseif { $tmpval == 1} {
            set portenable enable
        }

    }

} else {

set portenable ignore

}

}

if { [subst ${tmp1}$j] == "portduplex" } {

    set flag_portduplex 1
    if { !$ignoreValCnt} {

```

```

    if {$tmpval == 0} {
        set portduplex half
    } elseif { $tmpval == 1} {
        set portduplex full
    }
} else {
set portduplex ignore
}
}

if { [subst ${tmp1}$j] == "portspeed" } {

    set flag_portspeed 1
    if { !$ignoreValCnt} {

        if {$tmpval == 0} {
            set portspeed 4
        } elseif { $tmpval == 1} {
            set portspeed 10
        } elseif { $tmpval == 2 } {
            set portspeed 16
        } elseif { $tmpval == 3 } {
            set portspeed 100
        } elseif { $tmpval == 4 } {
            set portspeed auto
        }

    } else {

set portspeed ignore

    }

}

if { [subst ${tmp1}$j] == "portpriority" } {

    set flag_portpriority 1
    if { !$ignoreValCnt} {

        if {$tmpval == 0} {
            set portpriority normal
        } elseif { $tmpval == 1} {
            set portpriority high
        }

    } else {

set portpriority ignore

    }

}

```

```

}

if { [subst ${tmp1}$j] == "portmembership" } {

    set flag_portmembership 1
    if { !$ignoreValCnt} {

        if {$tmpval == 0} {
            set portmembership dynamic
        } elseif { $tmpval == 1} {
            set portmembership static
        }

    } else {

set portmembership ignore

    }

}

if { [subst ${tmp1}$j] == "portttrap" } {

    set flag_porttrap 1
    if { !$ignoreValCnt} {

        if {$tmpval == 0} {
            set portttrap enable
        } elseif { $tmpval == 1} {
            set portttrap disable
        }

    } else {

set portttrap ignore

    }

}

}

if { $flag_int == 1 } {
puts $tmpfile "set int_status $int_status"
"
}

if { $flag_destaddr == 1 } {
puts $tmpfile "set destaddr $destaddr"
"
}

```

```

if { $flag_ipgw == 1 } {
    puts $tmpfile "set ipgw $ipgw"
}

if { $flag_permitip == 1 } {
    puts $tmpfile "set permitip $permitip"
}

if { $flag_redirect == 1 } {
    puts $tmpfile "set redirect $redirect"
}

if { $flag_sendipaddr == 1 } {
    puts $tmpfile "set sendipaddr $sendipaddr"
}

if { $flag_portenable == 1 } {
    puts $tmpfile "set portenable $portenable"
}

if { $flag_portduplex == 1 } {
    puts $tmpfile "set portduplex $portduplex"
}

if { $flag_portspeed == 1 } {
    puts $tmpfile "set portspeed $portspeed"
}

if { $flag_portpriority == 1 } {
    puts $tmpfile "set portpriority $portpriority"
}

if { $flag_portmembership == 1 } {
    puts $tmpfile "set portmembership $portmembership"
}

if { $flag_porttrap == 1 } {
    puts $tmpfile "set porttrap $porttrap"
}

#puts $tmpfile "set int_status $int_status"
#set destaddr $destaddr
#set ipgw $ipgw
#set permitip $permitip
#set redirect $redirect
#\#set sendipaddr \$sendipaddr
#set portenable $portenable

```

```

#set portduplex $portduplex
#set portspeed $portspeed
#set portpriority $portpriority
#set portmembership $portmembership
#set porttrap $porttrap
#
#"

puts $tmpfile "
    set tmp_commands \"

if { $flag_int == 1 } {
    puts $tmpfile "set interface sl0 \"$int_status
}

if { $flag_destaddr == 1 || $flag_ipgw == 1 } {
    puts $tmpfile "set ip route \"$destaddr \"$ipgw
}

if { $flag_permitip == 1 } {
    puts $tmpfile "set ip permit \"$permitip
}

if { $flag_redirect == 1 } {
    puts $tmpfile "set ip redirect \"$redirect
}

if { $flag_sendipaddr == 1 } {
    puts $tmpfile "set interface sl0 \"$sendipaddr \"$sendipaddr
}

if { $flag_portenable == 1 } {
    puts $tmpfile "set port enable $mod_num/$port_num
}

if { $flag_portduplex == 1 } {
    puts $tmpfile "set port duplex $mod_num/$port_num \"$portduplex
}

if { $flag_portspeed == 1 } {
    puts $tmpfile "set port speed $mod_num/$port_num \"$portspeed
}

if { $flag_portpriority == 1 } {
    puts $tmpfile "set port level $mod_num/$port_num \"$portpriority
}

```

```

if { $flag_portmembership == 1 } {
    puts $tmpfile "set port membership $mod_num/$port_num
    \ $portmembership
    "
}

if { $flag_porttrap == 1 } {
    puts $tmpfile "set port trap $mod_num/$port_num \ $porttrap
    "
}

puts $tmpfile "disable
    \"

"

#puts $tmpfile "
#     set tmp_commands \"
#         set interface s10 \ $int_status
#         set interface s10 0.0.0.0 0.0.0.0
#         set ip permit \ $permitip
#         set ip redirect \ $redirect
#         set ip route \ $destaddr \ $ipgw
#         disable
#         \"
#
#     "

puts $tmpfile "
    set tmp_commands1 \"
        ping \ $destaddr
    \"
"

puts $tmpfile "
    set tmp_commands2 \"
        ping 124.12.30
    \"
"

puts $tmpfile "set router2_cli_sid \[open_router_session \ $temp_ip_addr
\"Console\" \"enable\" \"\\n\"\"
#send_user \"telnetted\"
enter_router_priviledged_mode \ $router2_cli_sid
"

if { $chkval == 0 } {
puts "chkval1 : $chkval"
puts $tmpfile "submit_to_cli1 \ $router2_cli_sid \ $tmp_commands $i
\#submit_to_cli \ $router2_cli_sid \ $tmp_commands1
\#submit_to_cli1 \ $router2_cli_sid \ $tmp_commands2 $i

"

} elseif { $chkval != 0 } {
puts "chkval2 : $chkval"
puts $tmpfile "submit_to_cli1 \ $router2_cli_sid \ $tmp_commands $i
\#submit_to_cli \ $router2_cli_sid \ $tmp_commands1

```

```

\#submit_to_cli1 \$router2_cli_sid \$tmp_commands1 $i
"
}

puts $tmpfile "send \"exit\n\"

"
close $tmpfile

# gets stdin
}

close $alltestfile

```

FuncTest.inc

Copyright (c) North Carolina State University, 2002.

FuncTest.inc – This file lists the functions needed to connect to the respective networking equipments. It also accommodates the functions to submit the cli to the appropriate networking equipment. This file is included in the 'Commands' script.

```

#
#In submit to cli... check for error messages
# automatic interface/ip address configuration on router
# automatic test pc configuration

#~~~~~
# Verify expect version number
#~~~~~
exp_version -exit 5.38.0 ;# Verify version number

#~~~~~
# Common variables
#~~~~~
set prompt "(%|#|>|\$) $" ;# Default prompt
set debug 0 ;# Debug is off by default

#~~~~~
# For Interface scripts
#~~~~~

set set_int sc0
set set_status up

```

```

set flag 0

set ipaddr 10.98.16.10
set nmask 255.255.255.0

set dest 10.98.16.67
set gw 10.98.16.1

#~~~~~
# Compulsory variables
#~~~~~
set logfile "$argv0.log" ;# Name of log file

#~~~~~
# Test-specific variables
#~~~~~

# Create null variables to be filled in once command line parms are
read

set router2_cli_sid -1
set router1_cli_sid -1
set orange_cli_sid -1
set send_to_addr 0

set n_ping 10 ;# # of ping packets to

set send_from_addr "10.1.1.2" ;# Src interface on
pkt_gen

set tcpdump_file "/tmp/tcpdump.out" ;# Temporary TCPDUMP
output file

#~~~~~
# Functions for error and debug messages; option handling; etc
#~~~~~

#
# Print process usage

```

```

#
proc print_usage {} {
    global argv0 usage_str
    regsub PROCNAME $usage_str $argv0 tmp_str
    send_user "$tmp_str\n"
}

#~~~~~
# Session opening and closing functions
#~~~~~

proc ssh_error host {
    err_exit "Ssh to host $host failed\n"
}

#
# Open an ssh session at a remote host with given user-name and
password
#
proc open_ssh_session {host user passwd} {
    global logfile

    set prompt "$host*.~>"
    set ssh_pid [spawn ssh -l $user $host]
    set timeout 30
    expect {
        eof          {
            send_user "EOF\n"
            ssh_error $host ;# Never returns
        }
        timeout      {
            send_user "timeout\n"
            ssh_error $host ;# Never returns
        }
        "password: " {
            send "$passwd\r"
            exp_continue;
        }
    }
    -re $prompt
}

return $spawn_id
}

#
# Open an ssh session to a router
#
proc open_router_session {host_address host_name user passwd} {
    global logfile

    set prompt "$host_name.*(>|#)"
#
    set ssh_pid [spawn ssh -l $user $host_address]
    set ssh_pid [spawn telnet $host_address]
    set timeout 30
    expect {

```

```

        eof          {
                        send_user "EOF\n"
                        ssh_error $host_name ;# Never returns
                    }
        timeout      {
                        send_user "timeout\n"
                        ssh_error $host_name ;# Never returns
                    }
        "Enter password: " {
                        send "$passwd\r"
                        exp_continue;
                    }
        -re $prompt
    }

    return $spawn_id
}

#
# Close an ssh session
#
proc close_ssh_session sid {
    set spawn_id $sid
    send "logout\r"
    expect {
        timeout {
            err_exit "Couldn't close ssh session\n"
        }
        eof;
    }
}

#-----
# Various remote program execution utilities
#-----

#
# Remote cd (change directory on a remote session).
#
proc remote_cd {sid dir} {
    global prompt

    set spawn_id $sid
    send "cd $dir\r"
    expect {
        -re "No such file or directory." {
            err_exit "Couldn't cd to directory $dir"
        }
        -re $prompt ;# cd was successful
    }
}

#
# Create a file with a given content in a remote session
#
proc remote_create_file {sid filename content} {
    global prompt

```

```

set spawn_id $sid
#
# Start cat
#
send "cat > $filename\r"
expect {
    -re $prompt {
        err_exit "Failed to start cat\n"
    }
    -timeout 1 timeout
}
#
# Pump variable "content" into cat process
#
send "$content"
#
# Close file by sending Ctrl-D to process; should get prompt back
#
send "_ "
expect {
    timeout {
        err_exit "Failed to close cat process"
    }
    -re $prompt
}
}

#
# Execute ping in a remote session with specified number of packets
#
proc remote_ping {sid pingcnt dest} {
    global prompt debug

    set spawn_id $sid
    send "ping -c $pingcnt $dest\n"

    expect -re "ping -c"

    set timeout 5
    expect {
        "bytes from" {
            exp_continue ;# Prevent premature timeout
        }
        "unknown host" {
            err_exit "Error in ping--unknown host\n"
        }
        timeout {
            err_exit "Ping timed out\n"
        }
        -re $prompt ;# Complete!
    }
    if $debug {
        send_user "Ping Successful\n"
    }
}
}

```

```

#
# Send Ctrl-C to running (foreground) process
#
proc remote_send_ctrl_c sid {
    global pktsink          ;# XXX should not be here!!!

    set prompt "$pktsink*.~>"
    set spawn_id $sid
    send "_"
    expect {
        timeout {
            err_exit "Couldn't stop process with Ctrl-C\n"
        }
        -re $prompt ;# Yupp, that's what we want
    }
}

# one more submit

proc submit_to_cli1 {sid commands val} {
    global debug

#    set prompt "(%|#|>|\\$)"
    set prompt "Console"
    set spawn_id $sid
    set tmpval $val
    set out_str ""

    # Split command variable into individual commands
    set command_list [split $commands "\n"]
    set command_cnt [llength $command_list]
    if {$debug} {
        puts "$command_list"
        puts "$command_cnt"
    }
    #uts "command cnt : $command_cnt"
    # NOTE: The first and last commands are null... Skip them
    for {set i 1} {$i < [expr $command_cnt-1]} {incr i} {

        # Remove leading whitespace from command
        regsub "^( )" [lindex $command_list $i] "" cmd
        regsub "^(\\t)*" $cmd "" cmd

        # Send next command to CLI
        send "$cmd\r"

        # Wait for CLI prompt
        set timeout 15
        expect {
            timeout {send_user "CLI did not return..."}
            -re $prompt {
                append out_str $expect_out(buffer)
                #send_user "got prompt"
                #send_user "out str : $out_str"
            }
        }
    }
}

```

```

        }

    }

    #send_user "\n#####\n"
    #send_user "\n out_str :$out_str\n"
    #send_user "\n@@@@@@@@@ \n"
    #sleep 2

}

#send_user "\n#####\n"
#send_user "out_str: $out_str\n"
#send_user "\n $$$$$$\n"

regsub -all "\r" $out_str "" tmpstr
set tmplist [split $tmpstr "\n" ]
#send_user "\n***\n$tmplist\n***\n"
set tmpflen [llength $tmplist]
set resultval 2

for {set m 1} {$m < $tmpflen } {incr m} {
set tmplist1 [split [lindex $tmplist $m] " " ]
#send_user "\n^^^\n $tmplist1 \n^^^\n"
set idx [lsearch $tmplist1 "Usage:"]
set idx2 [lsearch $tmplist1 "Unknown"]

if {$idx != -1 || $idx2 != -1} {
set resultval 0
break
#set resultfile [open "/root/testdir/results" "a+"]
#puts $resultfile "Test $val Failed "
#close $resultfile
#exp_continue;
}

if {$idx == -1 && $idx2 == -1} {
set resultval 1
#set resultfile [open "/root/testdir/results" "a+"]
#puts $resultfile "Test $val Passed "
#close $resultfile
#exp_continue;
}

}

#send_user "\nresultval : $resultval "
if { $resultval == 0} {
set resultfile [open "/root/testdir/results" "a+"]
puts $resultfile "Test $val Failed "
puts $resultfile "temp 2"
close $resultfile
}
if { $resultval == 1} {
set resultfile [open "/root/testdir/results" "a+"]
puts $resultfile "Test $val Passed "
}

```

```

    close $resultfile
    }
    # Sleep for 5 seconds to allow changes to take effect
    sleep 3
}

#
# Enter privileged mode on router
#
proc enter_router_priviledged_mode {sid} {

    set prompt "Console"
    set spawn_id $sid
    send "enable\r"

    expect {
        eof          {
                        err_exit "cannot enter priv-mode"
                    }
        timeout      {
                        err_exit "cannot enter priv-mode"
                    }
        "Enter password:" {
                        #send "enable\r"
                        send "\n"
                        exp_continue;
                    }
        -re $prompt
    }
}
}

```

Copyright (c) North Carolina State University, 2002.

New_m9 – This file is a PairTest example output file.

new m9

```

PARAMETERS
Number of Parameters :5
param:int-0|1|
param:destaddr-0|1|2|3|

```

```

param:ipgw-0|1|2|
param:permitip-0|1|
param:redirect-0|1|
RELATIONS
Number of Relations :2
relation:0-(destaddr,ipgw,redirect,)
CONSTRAINTS
Number of Constraints :1
constraint:0-<*,*,0,>
relation:1-(destaddr,ipgw,permitip,)
CONSTRAINTS
Number of Constraints :1
constraint:0-<*,*,1,>
[TESTSET]
NUM OF VARS:5
NUM OF TESTS:26
0:int(0:0,1:1,)
1:destaddr(0:0,1:1,2:2,3:3,)
2:ipgw(0:0,1:1,2:2,)
3:permitip(0:0,1:1,)
4:redirect(0:0,1:1,)
0 0 0 0 1
0 2 0 0 1
0 3 0 0 1
1 3 1 1 0
1 0 0 x 1
0 1 1 0 1
1 2 1 0 x
0 2 2 0 1
0 0 2 0 1
0 1 2 0 1
1 1 0 0 x
0 1 0 0 x
1 0 2 x 1
1 1 1 0 x
1 2 2 0 x
x 3 x 1 x
1 3 0 1 1
1 0 1 0 x
x x 2 1 x
1 1 2 0 x
x x 0 x 0
x x 1 x 0
x 3 x x 0
1 2 0 0 x
x 2 x x 0
1 0 1 1 1

```

Copyright (c) North Carolina State University, 2002.

New_m10 – This file is a PairTest example output file.

new m10

PARAMETERS

Number of Parameters :4

param:int-0|1|

param:destaddr-0|1|2|3|

param:ipgw-0|1|2|

param:redirect-0|1|

RELATIONS

Number of Relations :1

relation:0-(destaddr,ipgw,redirect,)

CONSTRAINTS

Number of Constraints :1

constraint:0-<*,*,0,>

[TESTSET]

NUM OF VARS:4

NUM OF TESTS:19

0:int(0:0,1:1,)

1:destaddr(0:0,1:1,2:2,3:3,)

2:ipgw(0:0,1:1,2:2,)

3:redirect(0:0,1:1,)

0 0 0 1

0 2 0 1

0 3 0 1

0 3 1 1

0 0 1 1

0 1 1 1

0 2 1 1

0 2 2 1

0 0 2 1

0 1 2 1

0 3 2 1

1 1 0 1

x 1 x 0

x 0 x 0

x 2 x 0

x 3 x 0

x x 0 0

x x 1 0

x x 2 0

Manual Tests

The approach taken for generation of test cases was a combinatorial approach. By considering the values assigned to the parameters they can be arranged as follows for generation or manual test cases. All the combinations of the parameter values were evaluated to cover all possible combinations. The development of the manual test cases is shown below. The actual values of the parameters are listed below.

Parameter values

Interface – 0: Interface is down
1: Interface is up

Destination IP address : 0 : 10.98.18.10
1: 10.98.16.66
2: 10.96.10.14
3: 10.95.11.13

IP Gateway Values 0: 10.98.16.1
1: 10.96.10.1
2: 10.95.11.1

Permitip 0 : IP address is not permitted
1 : IP address is permitted

Redirect 0: Traffic to IP address is not redirected
1: Traffic to IP address is redirected

These values were used as parameter values to generate tests using PairTest.

Also by using these values manual test cases were generated using combinatorial approach. To cover each combination of the parameter values, single parameter values were varied at a time by keeping values of other parameters fixed. In the flow diagram below the pseudo parameter values are listed along with location in the test-bed of Figure 11 where parameter values were changed (for Ex.: Interface parameter values were modified on 5500).

| Values | Parameters with temporary values. |
|--------|--|
| 2 | Inter face (5500) - up /down |
| * | |
| 3 | Gateway (5500) – 0 1 2 |
| * | |
| 4 | DestinationIP (respective workstations) – 0 1 2 3 |
| * | |
| 2 | permitIP (5500) – enable /disable |
| * | |
| 2 | redirect (5500) - enable/disable |
| ----- | |
| = 96 | ← Total Manual Test Cases |

It can be observed from above flow diagram that by using combinatorial approach the total number of test cases generated covering each combination of parameter values are $INT (2) * Gate (3) * DestIP (4) * PermitIP (2) * Redirect (2) = 96$ test cases. These test cases were generated by changing values of a single parameter with respect to other parameters while keeping their values fixed at that instance. These manual test cases represented a small part of test cases considering whole test system. These test cases were part of a larger system and represented complete combinatorial manual test suite for the limited setting with the chosen parameters. The variation in the value of any parameter with respect to other parameters for generation of manual test cases was done by changing the value of the respective parameter by changing setting of 5500 or respective workstation as indicated above while keeping the setting for other parameters same keeping their values fixed.

So total number of manual test cases generated will be 96. It was deduced from the knowledge of the system that destination IP address can not be reached when the interface is down. Considering this aspect all the combinations in which interface is down become invalid due to specifications of the system. So accounting for those invalid test cases remaining combinations of the parameters were 48. Thus giving rise to 48 manual test cases. However, while using human tester experience to deduce that when interface is down destinations can't be reached, an additional test was required to check if interface is down. This can be accounted as the 49 th manual test case. For analysis purposes this test is considered not being part of actual manual test cases. This test is discussed in the discussion of manual test cases in section 5.1.

Faults in the System:

To assess test generation ability of PairTest, faults were seeded in the system and then tests were generated using PairTest.

The configuration faults or root faults causing failures in the system are listed as below.

- A. Destination IP address 10.98.18.10 doesn't exist.
- B. IP address 10.96.10.14 is blocked using permit IP list.
- C. IP address 10.98.16.66 is blocked using permit IP list.
- D. Redirection on Port 2/1 is disabled.
- E. Port 1/1 can redirect to port 2/1.

The above configuration faults gave rise to the failures in the system. By incorporating these configuration changes or seeded configuration faults in the system, the test cases generated by PairTest as well as manual test cases were run. Failure of the test case is defined as the ability of the test case to not get a response from ping [Ping] utility. By analysis of the system taking into consideration of seeded configuration faults, it was observed that there were 12 known failures of the system. The test cases generated to test the system were run to find the following 12 failures. The system observed 12 failures as described below.

1. The destination IP address 10.95.11.13 is not reachable with the use of gateway 10.96.10.1. (Port 2 redirection failure)
2. The destination IP address 10.98.18.10 is not reachable with the use of gateway 10.98.16.1. (IP address doesn't exist)
3. The destination IP address 10.96.10.14 is not reachable with the use of gateway 10.96.10.1. (IP address blocked)
4. The destination IP address 10.98.16.66 is not reachable with the use of gateway 10.98.16.1. (IP address blocked)

5. The destination IP address 10.98.18.10 is not reachable with the use of gateway 10.95.11.1. (IP address doesn't exist)
6. The destination IP address 10.98.16.66 is not reachable through use of gateway 10.96.10.1. (IP address blocked)
7. The destination IP address 10.96.10.14 is not reachable through the use of gateway 10.95.11.1. (IP address blocked)
8. The destination IP address 10.95.11.13 is not reachable through the use of gateway 10.98.16.1. (Port 3/1 redirects to only port 1/1)
9. The destination IP address 10.98.18.10 is not reachable through the use of gateway 10.96.10.1. (IP address doesn't exist)
10. The destination IP address 10.98.16.66 is not reachable through the use of gateway 10.95.11.1. (IP address blocked)
11. The destination IP address 10.96.10.14 is not reachable through the use of gateway 10.98.16.1. (IP address blocked)
12. The destination IP address 10.98.18.10 is not reachable through the use of gateway 10.96.10.1. (Port 2 redirection error)

Table 6. Test Suite Analysis for Per Test Case Efficiency

* indicates that the Test detected or uncovered a failure/fault in the system.

| Test Case Number | PairTest Test Suite | Manual Test Suite |
|------------------|---------------------|-------------------|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | *(1)/(D) | |
| 5 | *(2)/(A) | |
| 6 | | |
| 7 | *(3)/(B) | *(4)/(C) |
| 8 | | |
| 9 | | |
| 10 | | |
| 11 | *(4)/(C) | *(6)/(C) |
| 12 | | |
| 13 | *(5)/(A) | *(3)/(B) |
| 14 | *(6)/(C) | |
| 15 | *(7)/(B) | |
| 16 | | *(7)/(B) |
| 17 | *(8)/(E) | *(5)/(A) |
| 18 | *(9)/(A) | |
| 19 | | |
| 20 | *(10)/(C) | *(1)/(D) |
| 21 | | |
| 22 | | |
| 23 | | |
| 24 | *(11)/(B) | *(8)/(E) |
| 25 | | |
| 26 | *(12)/(A) | |
| 27 | | |
| 28 | | |
| 29 | | *(2)/(A) |
| 30 | | *(9)/(A) |

| | | |
|----|--|-----------|
| 31 | | |
| 32 | | |
| 33 | | |
| 34 | | |
| 35 | | |
| 36 | | |
| 37 | | |
| 38 | | *(12)/(A) |
| 39 | | |
| 40 | | *(10)/(C) |
| 41 | | |
| 42 | | |
| 43 | | |
| 44 | | |
| 45 | | |
| 46 | | |
| 47 | | |
| 48 | | *(11)/(B) |

It can be seen from the table 7 and 8 below that when the test cases generated by PairTest as well as manual test cases were executed, most of the test cases gave expected outcomes. However, few test cases failed. These test cases identified failures in the systems, which are listed above. The executable PairTest test cases are listed in Table 7 and executable manual test cases are listed in Table 8.

Table 7. Executable values of test cases PairTest

| Interfa. | Dest Addr | Mask | Gateway (port) | Permit IP | Redirect | Expected Outcome | Result (actual) |
|----------|-------------|-------------|----------------|-----------|----------|------------------|-----------------|
| Down | 10.98.18.10 | 255.255.0.0 | 10.98.16.1 | No | Yes | Pass | Passed |
| Down | 10.96.10.14 | 255.255.0.0 | 10.98.16.1 | No | Yes | Pass | Passed |
| Down | 10.95.11.13 | 255.255.0.0 | 10.98.16.1 | No | Yes | Pass | Passed |
| Up | 10.95.11.13 | 255.255.0.0 | 10.96.10.1 | Yes | Yes | Pass | Failed |
| Up | 10.98.18.10 | 255.255.0.0 | 10.98.16.1 | Yes | Yes | Pass | Failed |
| Down | 10.98.16.66 | 255.255.0.0 | 10.96.10.1 | No | Yes | Pass | Passed |
| Up | 10.96.10.14 | 255.255.0.0 | 10.96.10.1 | No | Yes | Pass | Failed |
| Down | 10.96.10.14 | 255.255.0.0 | 10.95.11.1 | No | Yes | Pass | Passed |
| Up | 10.98.18.10 | 255.255.0.0 | 10.95.11.1 | No | Yes | Pass | Passed |
| Up | 10.98.16.66 | 255.255.0.0 | 10.95.11.1 | No | Yes | Pass | Passed |
| Up | 10.98.16.66 | 255.255.0.0 | 10.98.16.1 | No | Yes | Pass | Failed |
| Down | 10.98.16.66 | 255.255.0.0 | 10.98.16.1 | No | Any | Pass | Passed |
| Up | 10.98.18.10 | 255.255.0.0 | 10.95.11.1 | Yes | Yes | Pass | Failed |
| Up | 10.98.16.66 | 255.255.0.0 | 10.96.10.1 | No | Any | Pass | Failed |
| Up | 10.96.10.14 | 255.255.0.0 | 10.98.16.1 | Yes | No | Pass | Failed |
| Any | 10.95.11.13 | 255.255.0.0 | Any | Yes | Any | Pass | Passed |
| Up | 10.95.11.13 | 255.255.0.0 | 10.98.16.1 | Yes | No | Pass | Failed |

| | | | | | | | |
|-----|-------------|-------------|------------|-----|-----|------|--------|
| Up | 10.98.16.66 | 255.255.0.0 | 10.96.10.1 | No | Any | Pass | Failed |
| Any | Any | 255.255.0.0 | 10.95.11.1 | Yes | Any | Pass | Passed |
| Up | 10.95.11.13 | 255.255.0.0 | 10.96.10.1 | No | Any | Pass | Failed |
| Any | Any | 255.255.0.0 | 10.98.16.1 | Any | No | Pass | Passed |
| Any | Any | 255.255.0.0 | 10.96.10.1 | Any | No | Pass | Passed |
| Any | 10.95.11.13 | 255.255.0.0 | Any | Any | No | Pass | Passed |
| Up | 10.96.10.14 | 255.255.0.0 | 10.95.11.1 | Yes | No | Pass | Failed |
| Any | 10.96.10.14 | 255.255.0.0 | Any | Any | No | Pass | Passed |
| UP | 10.95.11.13 | 255.255.0.0 | 10.98.16.1 | Any | No | Pass | Failed |

Table 8.Executable values of test cases – Manual Tests

| No. | Int. | Dest Addr | Gateway | Mask | Permit IP | Redirect | Expected Outcomes | Result |
|-----|------|-------------|------------|-------------|-----------|----------|-------------------|--------|
| 1 | Up | 10.98.18.10 | 10.98.16.1 | 255.255.0.0 | No | No | Pass | Passed |
| 2 | Up | 10.98.16.66 | 10.96.10.1 | 255.255.0.0 | No | Yes | Pass | Passed |
| 3 | Up | 10.96.10.14 | 10.95.11.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 4 | Up | 10.96.10.14 | 10.98.16.1 | 255.255.0.0 | Yes | Yes | Pass | Passed |
| 5 | Up | 10.96.10.14 | 10.95.11.1 | 255.255.0.0 | No | No | Pass | Passed |
| 6 | Up | 10.98.16.66 | 10.98.16.1 | 255.255.0.0 | Yes | Yes | Pass | Passed |
| 7 | Up | 10.98.16.66 | 10.98.16.1 | 255.255.0.0 | No | Yes | Pass | Failed |
| 8 | Up | 10.95.11.13 | 10.96.10.1 | 255.255.0.0 | Yes | Yes | Pass | Passed |
| 9 | Up | 10.98.16.66 | 10.98.16.1 | 255.255.0.0 | No | No | Pass | Passed |
| 10 | Up | 10.96.10.14 | 10.95.11.1 | 255.255.0.0 | No | Yes | Pass | Passed |
| 11 | Up | 10.98.16.66 | 10.96.10.1 | 255.255.0.0 | No | Yes | Pass | Failed |
| 12 | Up | 10.95.11.13 | 10.96.10.1 | 255.255.0.0 | Yes | Yes | Pass | Passed |
| 13 | Up | 10.96.10.14 | 10.96.10.1 | 255.255.0.0 | Yes | No | Pass | Failed |
| 14 | Up | 10.95.11.13 | 10.98.16.1 | 255.255.0.0 | No | Yes | Pass | Passed |
| 15 | Up | 10.96.10.14 | 10.96.10.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 16 | Up | 10.96.10.14 | 10.95.11.1 | 255.255.0.0 | No | Yes | Pass | Failed |
| 17 | Up | 10.98.18.10 | 10.95.11.1 | 255.255.0.0 | No | Yes | Pass | Failed |
| 18 | Up | 10.98.18.10 | 10.98.16.1 | 255.255.0.0 | No | No | Pass | Passed |
| 19 | Up | 10.96.10.14 | 10.98.16.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 20 | Up | 10.98.18.10 | 10.95.11.1 | 255.255.0.0 | Yes | No | Pass | Failed |
| 21 | Up | 10.98.18.10 | 10.95.11.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 22 | Up | 10.98.18.10 | 10.95.11.1 | 255.255.0.0 | Yes | Yes | Pass | Passed |
| 23 | Up | 10.95.11.13 | 10.98.16.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 24 | Up | 10.95.11.13 | 10.98.16.1 | 255.255.0.0 | Yes | Yes | Pass | Failed |
| 25 | Up | 10.95.11.13 | 10.98.16.1 | 255.255.0.0 | No | No | Pass | Passed |
| 26 | Up | 10.98.18.10 | 10.96.10.1 | 255.255.0.0 | No | Yes | Pass | Passed |
| 27 | Up | 10.96.10.14 | 10.95.11.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 28 | Up | 10.95.11.13 | 10.96.10.1 | 255.255.0.0 | Yes | Yes | Pass | Passed |
| 29 | Up | 10.98.16.66 | 10.98.16.1 | 255.255.0.0 | No | No | Pass | Failed |
| 30 | Up | 10.98.18.10 | 10.96.10.1 | 255.255.0.0 | No | Yes | Pass | Failed |
| 31 | Up | 10.98.16.66 | 10.95.11.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 32 | Up | 10.98.18.10 | 10.96.10.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 33 | Up | 10.98.16.66 | 10.98.16.1 | 255.255.0.0 | No | No | Pass | Passed |
| 34 | Up | 10.96.10.14 | 10.96.10.1 | 255.255.0.0 | No | Yes | Pass | Passed |

| | | | | | | | | |
|----|----|-------------|------------|-------------|-----|-----|------|--------|
| 35 | Up | 10.98.16.66 | 10.95.11.1 | 255.255.0.0 | No | No | Pass | Passed |
| 36 | Up | 10.95.11.13 | 10.95.11.1 | 255.255.0.0 | Yes | Yes | Pass | Passed |
| 37 | Up | 10.98.18.10 | 10.98.16.1 | 255.255.0.0 | No | No | Pass | Passed |
| 38 | Up | 10.98.18.10 | 10.96.10.1 | 255.255.0.0 | Yes | Yes | Pass | Failed |
| 39 | Up | 10.95.11.13 | 10.95.11.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 40 | Up | 10.98.16.66 | 10.95.11.1 | 255.255.0.0 | No | Yes | Pass | Failed |
| 41 | Up | 10.96.10.14 | 10.96.10.1 | 255.255.0.0 | No | No | Pass | Passed |
| 42 | Up | 10.98.16.66 | 10.95.11.1 | 255.255.0.0 | No | Yes | Pass | Passed |
| 43 | Up | 10.95.11.13 | 10.96.10.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 44 | Up | 10.98.18.10 | 10.96.10.1 | 255.255.0.0 | Yes | Yes | Pass | Passed |
| 45 | Up | 10.95.11.13 | 10.96.10.1 | 255.255.0.0 | No | No | Pass | Passed |
| 46 | Up | 10.95.11.13 | 10.98.16.1 | 255.255.0.0 | No | Yes | Pass | Passed |
| 47 | Up | 10.98.16.66 | 10.95.11.1 | 255.255.0.0 | Yes | No | Pass | Passed |
| 48 | Up | 10.96.10.14 | 10.98.16.1 | 255.255.0.0 | Yes | Yes | Pass | Failed |

Table 9. Manual Test Generation Incremental configuration

| Time Interval (days) | Incremental Configuration Change for Test Generation |
|-----------------------------|---|
| (2-4) | Addition of permitip parameter value |
| (4-6) | Addition of Gateway parameter value 10.98.16.1 |
| (6-8) | Addition of Destination IP address value 10.98.16.66 |
| (8-10) | Addition of redirect parameter value |
| (10-12) | Addition of Destination IP address value 10.98.18.10 |
| (12-14) | Addition of Gateway parameter value 10.96.10.1 |
| (14-16) | Addition of Destination IP address value 10.96.10.14 |
| (16-18) | Addition of Gateway parameter value 10.95.11.1 |
| (18-20) | Addition of Destination IP address value 10.95.11.13 |

Table 10. Manual Test Execution Incremental configuration

| Time Interval (days) | Incremental Configuration Change for Test Execution |
|-----------------------------|--|
| (5-10) | Removal of Destination IP address 10.98.18.10 from the system |
| (10-15) | Removal of Destination IP address 10.95.11.13 with use of port 2 |
| (15-20) | Addition of Destination IP address 10.96.10.14 to permitip list |
| (20-25) | Removal of Destination IP address 10.98.18.10 for check with port 2 |
| (25-30) | Addition of Destination IP address 10.98.16.66 to permitip list |
| (30-35) | Removal of Destination IP address 10.95.11.13 with use of port 1 |
| (35-40) | Addition of Destination IP address 10.98.18.10 to the system for check with port 1 |
| (40-45) | Removal of Destination IP address 10.96.10.14 with check of port 2 |

Industrial Data

* Please note that, in the following tables to preserve the confidentiality of proprietary data it is presented in a sanitized form. However, this sanitization preserved the phenomenon and relation between the values without revealing actual values.

Table 11. Failure Coverage data -Industrial

| Number of test cases | Normalized No of failures Automated test generation | Normalized No of failures Manual Testing |
|----------------------|--|---|
| 100 | 0.2405 | 0.1519 |
| 200 | 0.7468 | 0.3924 |
| 300 | 1.08 | 0.4937 |
| 400 | 1.2405 | 0.6708 |
| 500 | 1.3037 | 0.8607 |
| 600 | 1.5189 | 1.08 |
| 700 | 1.6708 | 1.1266 |
| 800 | | 1.2785 |
| 900 | | 1.4177 |
| 1000 | | 1.5696 |

For the purpose of calculations the total numbers of failures are assured to be sanitized value of 1.6708

Resources data

Table 12. Test Generation- Industrial Resources

| No. Of Months | Total Resources (cost) Automated Test Generation | Total Resources (cost) Manual Testing |
|---------------|---|--|
| 2 | 0.4574 | 0.1829 |
| 3 | 0.7858 | 0.3068 |
| 4 | 0.5167 | 0.3929 |
| 5 | 0.4844 | 0.5005 |
| 6 | 0.4951 | 0.5651 |
| 7 | 0.5059 | 0.6835 |
| 8 | | 0.7911 |
| 9 | | 0.8772 |

Maximum amount of normalized resources available each month in \$ = 1

* The actual values of resources consumed each month are calculated in dollars (\$)

* The maximum amount of resources available each month for automated as well as manual test generation is same

Resources data

Table 13. Test Execution- Industrial Resources

| No of Months | Total Resources Automation | Total Resources Manual Test Execution |
|---------------------|-----------------------------------|--|
| 2 | 0.5392 | 0.2096 |
| 4 | 0.6582 | 0.3752 |
| 6 | 0.4570 | 0.4413 |
| 8 | 0.4297 | 0.5848 |
| 10 | 0.4574 | 0.6403 |
| 12 | 0.4662 | 0.7945 |
| 14 | 1.0414 | 0.9987 |

Maximum amount of normalized resources available per month \$: 1

* The actual values of resources consumed each period are calculated in dollars (\$)

* The maximum amount of resources available each period for automated as well as manual test generation is same

Test generation

Table 14. Evolvability –Industrial (Test Generation)
(Involves Modification Complexity)

| Time Interval Months | Δ Efforts in consecutive time period Automated Test Generation | Δ Efforts in consecutive time period Manual Test Generation | No. Of Test Cases Cumulative Automated | No. Of Test Cases Cumulative Manual |
|-----------------------------|---|--|---|--|
| 0.5 | 0.32 | 0.6201 | 0.0934 | 0.1318 |
| 1 | 0.4651 | 0.7752 | 0.2197 | 0.3076 |
| 1.5 | 0.6201 | 0.9302 | 0.3516 | 0.5055 |
| 2 | 1.395 | 1.085 | 0.5329 | 0.7252 |
| 2.5 | 2.945 | 0.6201 | 0.7472 | 0.9780 |
| 3 | 1.705 | 0.7752 | 0.9065 | 1.1978 |
| 3.5 | 2.015 | 0.7752 | 1.1098 | 1.4945 |
| 4 | 1.860 | 0.9302 | 1.3351 | 1.7857 |
| 4.5 | 0.4651 | 0.7752 | 1.4835 | 2.0439 |
| 5 | 0.1550 | 0.7752 | 1.6758 | 2.3516 |

* The actual values of Δ Efforts in each consecutive time period are calculated in dollars (\$)

Test Execution

Table 15. Evolvability – Industrial (Test Execution)
(Involves Modification Complexity)

| Time Interval Months | Δ Efforts in consecutive time periods Automated Test Execution | Δ Efforts in consecutive time periods Manual Test Execution | No. Of Modifications Cumulative Automated | No. Of Modifications Cumulative Manual |
|---------------------------------|---|--|--|---|
| 1 | 0.2037 | 0.6112 | 0.1353 | 0.1729 |
| 2 | 0.4075 | 1.018 | 0.2857 | 0.3835 |
| 3 | 0.9168 | 1.1426 | 0.3834 | 0.6315 |
| 4 | 1.7318 | 1.629 | 0.4962 | 0.8721 |
| 5 | 1.8541 | 0.8965 | 0.6315 | 1.1654 |
| 6 | 2.6284 | 0.7539 | 0.7481 | 1.4135 |
| 7 | 0.3463 | 0.9576 | 0.8533 | 1.6315 |
| 8 | 0.2852 | 1.6911 | 0.9925 | 1.9248 |
| 9 | 0.2649 | 0.9983 | 1.1654 | 2.2443 |
| 10 | 0.1426 | 1.2428 | 1.345 | 2.5789 |

* The actual values of Δ Efforts in each consecutive time period are calculated in dollars (\$)

Appendix B

The screenshots displayed below are taken from AETG Web only for the research purposes of this thesis. These screenshots are discussed in section 3.10, 3.11 and 3.12. These screenshots are generated only for the research purposes of this thesis. The screenshots in Figure 26, 27 and 28 are proprietary to Telcordia Technologies and are reprinted with specific permission for this research.

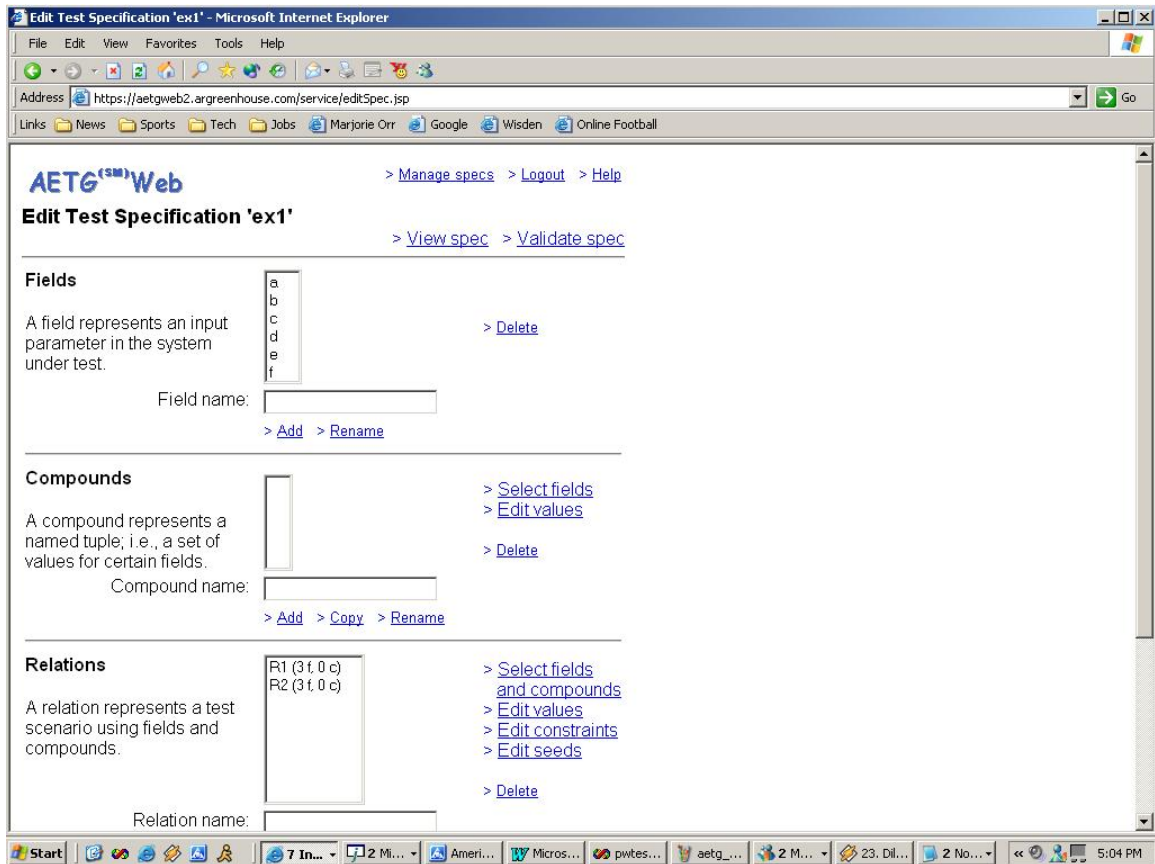


Figure 26. Edit Parameters from AETG

* Copyright © 2003, Telcordia Technologies; reprinted with specific permission

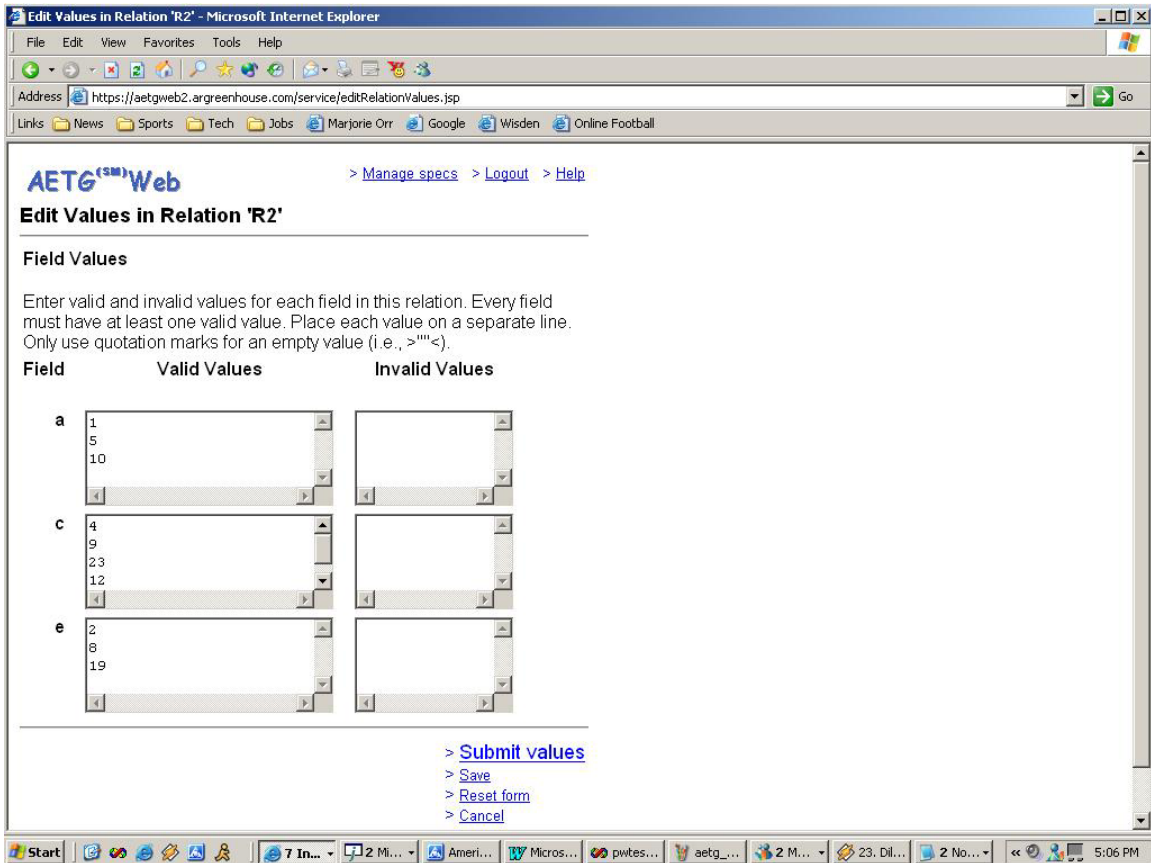


Figure 27. Edit Relations from AETG

* Copyright © 2003, Telcordia Technologies; reprinted with specific permission

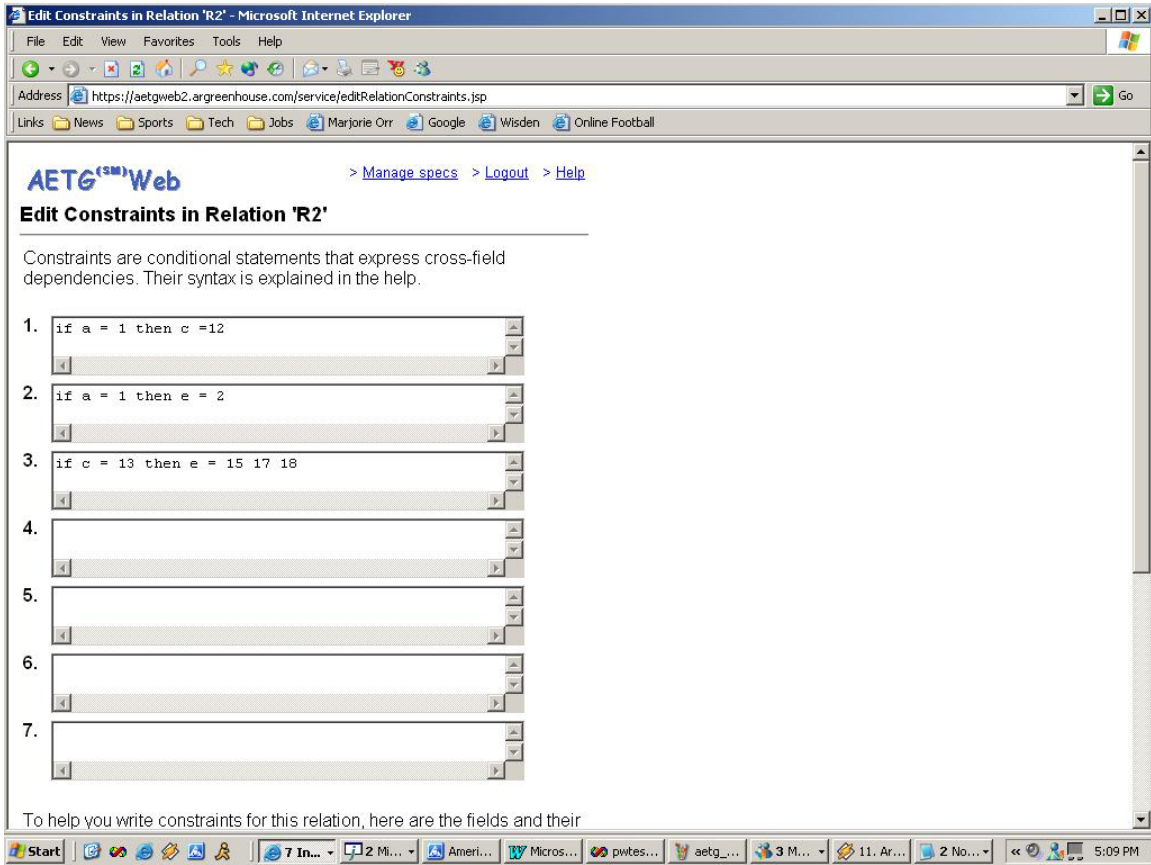


Figure 28. Edit constraints from AETG

* Copyright © 2003, Telcordia Technologies; reprinted with specific permission

The PairTest and AETG test suites listed below are the test suites generated for parameter, relation and constraints as listed in section 3.14.2 for case-study 1. These test suites were generated by inputting the respective parameters, relation and constraints into PairTest and AETG respectively. PairTest Generated 336 test cases where as AETG generated 326 test cases. The actual values of the respective parameters and data are listed in section 3.14.2 under case-study 1.

PairTest File

```

PARAMETERS
Number of Parameters :6
param:a-1|5|10|2|3|4|6|7|8|9|11|12|13|14|15|16|17|18|19|20|21|
param:b-3|13|
param:c-4|9|23|12|34|56|67|78|99|89|71|96|82|95|87|22|
param:d-11|
param:e-2|8|19|
param:f-7|17|27|
RELATIONS
Number of Relations :3
relation:0-(a,b,c,d,)
CONSTRAINTS
Number of Constraints :1
constraint:0-<10,13,96,11,>
relation:1-(e,f,d,)
CONSTRAINTS
Number of Constraints :1
constraint:0-<19,17,11,>
relation:2-(b,e,)
CONSTRAINTS
Number of Constraints :1
constraint:0-<13,19,>
[TESTSET]
NUM OF VARS:6
NUM OF TESTS:336
0:a(0:1,1:5,2:10,3:2,4:3,5:4,6:6,7:7,8:8,9:9,10:11,11:12,12:13,13:14
,14:15,15:16,16:17,17:18,18:19,19:20,20:21,)
1:b(0:3,1:13,)
2:c(0:4,1:9,2:23,3:12,4:34,5:56,6:67,7:78,8:99,9:89,10:71,11:96,12:8
2,13:95,14:87,15:22,)
3:d(0:11,)
4:e(0:2,1:8,2:19,)
5:f(0:7,1:17,2:27,)
0 0 0 0 0 0
16 0 0 0 0 0
17 0 0 0 0 0
18 0 0 0 0 0
19 0 0 0 0 0
20 0 0 0 0 0
4 0 0 0 0 0
5 0 0 0 0 0
5 0 5 0 0 0
5 0 1 0 0 0

```

2 0 1 0 0 0
3 0 1 0 0 0
4 0 1 0 0 0
6 0 1 0 0 0
6 0 6 0 0 0
6 0 0 0 0 0
7 0 0 0 0 0
7 0 7 0 0 0
7 0 1 0 0 0
8 0 1 0 0 0
8 0 8 0 0 0
8 0 0 0 0 0
9 0 0 0 0 0
10 0 0 0 0 0
11 0 0 0 0 0
12 0 0 0 0 0
13 0 0 0 0 0
14 0 0 0 0 0
14 0 14 0 0 0
14 0 1 0 0 0
9 0 1 0 0 0
9 0 9 0 0 0
9 0 2 0 0 0
0 0 2 0 0 0
1 0 2 0 0 0
4 0 2 0 0 0
5 0 2 0 0 0
6 0 2 0 0 0
7 0 2 0 0 0
8 0 2 0 0 0
10 0 2 0 0 0
10 0 10 0 0 0
10 0 1 0 0 0
11 0 1 0 0 0
11 0 11 0 0 0
11 0 2 0 0 0
12 0 2 0 0 0
12 0 12 0 0 0
12 0 1 0 0 0
13 0 1 0 0 0
13 0 13 0 0 0
13 0 2 0 0 0
14 0 2 0 0 0
15 0 2 0 0 0
15 0 0 0 0 0
15 0 1 0 0 0
15 0 15 0 0 0
15 0 3 0 0 0
0 0 3 0 0 0
1 0 3 0 0 0
4 0 3 0 0 0
5 0 3 0 0 0
6 0 3 0 0 0
7 0 3 0 0 0
8 0 3 0 0 0
9 0 3 0 0 0
10 0 3 0 0 0

11 0 3 0 0 0
12 0 3 0 0 0
13 0 3 0 0 0
14 0 3 0 0 0
16 0 3 0 0 0
16 0 2 0 0 0
17 0 2 0 0 0
18 0 2 0 0 0
19 0 2 0 0 0
20 0 2 0 0 0
20 0 3 0 0 0
17 0 3 0 0 0
18 0 3 0 0 0
19 0 3 0 0 0
19 0 4 0 0 0
0 0 4 0 0 0
1 0 4 0 0 0
2 0 4 0 0 0
3 0 4 0 0 0
6 0 4 0 0 0
7 0 4 0 0 0
8 0 4 0 0 0
9 0 4 0 0 0
10 0 4 0 0 0
11 0 4 0 0 0
12 0 4 0 0 0
13 0 4 0 0 0
14 0 4 0 0 0
15 0 4 0 0 0
16 0 4 0 0 0
17 0 4 0 0 0
18 0 4 0 0 0
20 0 4 0 0 0
20 0 5 0 0 0
0 0 5 0 0 0
1 0 5 0 0 0
2 0 5 0 0 0
3 0 5 0 0 0
6 0 5 0 0 0
7 0 5 0 0 0
8 0 5 0 0 0
9 0 5 0 0 0
10 0 5 0 0 0
11 0 5 0 0 0
12 0 5 0 0 0
13 0 5 0 0 0
14 0 5 0 0 0
15 0 5 0 0 0
16 0 5 0 0 0
17 0 5 0 0 0
18 0 5 0 0 0
19 0 5 0 0 0
19 0 6 0 0 0
0 0 6 0 0 0
1 0 6 0 0 0
2 0 6 0 0 0
3 0 6 0 0 0

4 0 6 0 0 0
5 0 6 0 0 0
8 0 6 0 0 0
9 0 6 0 0 0
10 0 6 0 0 0
11 0 6 0 0 0
12 0 6 0 0 0
13 0 6 0 0 0
14 0 6 0 0 0
15 0 6 0 0 0
16 0 6 0 0 0
17 0 6 0 0 0
18 0 6 0 0 0
20 0 6 0 0 0
20 0 7 0 0 0
0 0 7 0 0 0
1 0 7 0 0 0
2 0 7 0 0 0
3 0 7 0 0 0
4 0 7 0 0 0
5 0 7 0 0 0
8 0 7 0 0 0
9 0 7 0 0 0
10 0 7 0 0 0
11 0 7 0 0 0
12 0 7 0 0 0
13 0 7 0 0 0
14 0 7 0 0 0
15 0 7 0 0 0
16 0 7 0 0 0
17 0 7 0 0 0
18 0 7 0 0 0
19 0 7 0 0 0
19 0 8 0 0 0
0 0 8 0 0 0
1 0 8 0 0 0
2 0 8 0 0 0
3 0 8 0 0 0
4 0 8 0 0 0
5 0 8 0 0 0
6 0 8 0 0 0
7 0 8 0 0 0
10 0 8 0 0 0
11 0 8 0 0 0
12 0 8 0 0 0
13 0 8 0 0 0
14 0 8 0 0 0
15 0 8 0 0 0
16 0 8 0 0 0
17 0 8 0 0 0
18 0 8 0 0 0
20 0 8 0 0 0
20 0 9 0 0 0
0 0 9 0 0 0
1 0 9 0 0 0
2 0 9 0 0 0
3 0 9 0 0 0

4 0 9 0 0 0
5 0 9 0 0 0
6 0 9 0 0 0
7 0 9 0 0 0
10 0 9 0 0 0
11 0 9 0 0 0
12 0 9 0 0 0
13 0 9 0 0 0
14 0 9 0 0 0
15 0 9 0 0 0
16 0 9 0 0 0
17 0 9 0 0 0
18 0 9 0 0 0
19 0 9 0 0 0
19 0 10 0 0 0
0 0 10 0 0 0
1 0 10 0 0 0
2 0 10 0 0 0
3 0 10 0 0 0
4 0 10 0 0 0
5 0 10 0 0 0
6 0 10 0 0 0
7 0 10 0 0 0
8 0 10 0 0 0
9 0 10 0 0 0
12 0 10 0 0 0
13 0 10 0 0 0
14 0 10 0 0 0
15 0 10 0 0 0
16 0 10 0 0 0
17 0 10 0 0 0
18 0 10 0 0 0
20 0 10 0 0 0
20 0 11 0 0 0
0 0 11 0 0 0
1 0 11 0 0 0
2 0 11 0 0 0
3 0 11 0 0 0
4 0 11 0 0 0
5 0 11 0 0 0
6 0 11 0 0 0
7 0 11 0 0 0
8 0 11 0 0 0
9 0 11 0 0 0
12 0 11 0 0 0
13 0 11 0 0 0
14 0 11 0 0 0
15 0 11 0 0 0
16 0 11 0 0 0
17 0 11 0 0 0
18 0 11 0 0 0
19 0 11 0 0 0
19 0 12 0 0 0
0 0 12 0 0 0
1 0 12 0 0 0
2 0 12 0 0 0
3 0 12 0 0 0

4 0 12 0 0 0
5 0 12 0 0 0
6 0 12 0 0 0
7 0 12 0 0 0
8 0 12 0 0 0
9 0 12 0 0 0
10 0 12 0 0 0
11 0 12 0 0 0
14 0 12 0 0 0
15 0 12 0 0 0
16 0 12 0 0 0
17 0 12 0 0 0
18 0 12 0 0 0
20 0 12 0 0 0
20 0 13 0 0 0
0 0 13 0 0 0
1 0 13 0 0 0
2 0 13 0 0 0
3 0 13 0 0 0
4 0 13 0 0 0
5 0 13 0 0 0
6 0 13 0 0 0
7 0 13 0 0 0
8 0 13 0 0 0
9 0 13 0 0 0
10 0 13 0 0 0
11 0 13 0 0 0
14 0 13 0 0 0
15 0 13 0 0 0
16 0 13 0 0 0
17 0 13 0 0 0
18 0 13 0 0 0
19 0 13 0 0 0
19 0 14 0 0 0
0 0 14 0 0 0
1 0 14 0 0 0
2 0 14 0 0 0
3 0 14 0 0 0
4 0 14 0 0 0
5 0 14 0 0 0
6 0 14 0 0 0
7 0 14 0 0 0
8 0 14 0 0 0
9 0 14 0 0 0
10 0 14 0 0 0
11 0 14 0 0 0
12 0 14 0 0 0
13 0 14 0 0 0
16 0 14 0 0 0
17 0 14 0 0 0
18 0 14 0 0 0
20 0 14 0 0 0
20 0 15 0 0 0
0 0 15 0 0 0
1 0 15 0 0 0
2 0 15 0 0 0
3 0 15 0 0 0

4 0 15 0 0 0
5 0 15 0 0 0
6 0 15 0 0 0
7 0 15 0 0 0
8 0 15 0 0 0
9 0 15 0 0 0
10 0 15 0 0 0
11 0 15 0 0 0
12 0 15 0 0 0
13 0 15 0 0 0
16 0 15 0 0 0
17 0 15 0 0 0
18 0 15 0 0 0
19 0 15 0 0 0
14 1 15 0 0 0
7 1 6 0 0 0
12 1 13 0 0 0
19 1 1 0 0 0
16 1 1 0 0 0
20 1 1 0 0 0
17 1 1 0 0 0
18 1 1 0 0 0
0 1 1 0 0 0
5 1 4 0 0 0
15 1 14 0 0 0
11 1 10 0 0 0
10 1 11 0 0 0
6 1 7 0 0 0
9 1 8 0 0 0
13 1 12 0 0 0
3 1 2 0 0 0
8 1 9 0 0 0
4 1 5 0 0 0
1 1 0 0 1 0
1 0 1 0 1 1
3 0 3 0 0 1
3 0 0 x 2 1
2 0 0 0 2 0
2 0 2 0 2 2
2 1 3 0 1 2
4 0 4 0 0 2

AETG Test Set

AETG test set was generated using AETG Web only for the research in this thesis. This test set is is reprinted with specific permission only for research purposes of this thesis. AETG Web. Copyright © 2003, Telcordia Technologies; reprinted with specific permission

The AETG Test Set is given as below.

Valid Test Cases

View as: > [XML](#) > [DIF](#)

| # | a | b | c | d | e | f |
|----|----|----|----|----|----|----|
| 1 | 9 | 13 | 56 | 11 | 2 | 17 |
| 2 | 9 | 3 | 67 | 11 | 8 | 7 |
| 3 | 9 | 13 | 78 | 11 | 19 | 17 |
| 4 | 9 | 13 | 89 | 11 | 8 | 17 |
| 5 | 9 | 3 | 71 | 11 | 2 | 7 |
| 6 | 9 | 13 | 96 | 11 | 19 | 7 |
| 7 | 9 | 3 | 82 | 11 | 2 | 17 |
| 8 | 9 | 13 | 87 | 11 | 8 | 7 |
| 9 | 9 | 3 | 22 | 11 | 19 | 17 |
| 10 | 11 | 3 | 4 | 11 | 8 | 17 |
| 11 | 11 | 13 | 9 | 11 | 2 | 7 |
| 12 | 11 | 3 | 23 | 11 | 19 | 7 |
| 13 | 11 | 3 | 12 | 11 | 2 | 17 |
| 14 | 11 | 13 | 34 | 11 | 8 | 7 |
| 15 | 11 | 13 | 56 | 11 | 19 | 17 |
| 16 | 11 | 13 | 78 | 11 | 8 | 17 |
| 17 | 11 | 13 | 99 | 11 | 2 | 7 |
| 18 | 11 | 3 | 89 | 11 | 19 | 7 |
| 19 | 11 | 13 | 96 | 11 | 2 | 17 |
| 20 | 11 | 3 | 82 | 11 | 8 | 7 |
| 21 | 11 | 13 | 95 | 11 | 19 | 17 |
| 22 | 11 | 13 | 87 | 11 | 8 | 17 |
| 23 | 11 | 3 | 22 | 11 | 2 | 7 |
| 24 | 12 | 13 | 4 | 11 | 19 | 7 |
| 25 | 12 | 13 | 9 | 11 | 2 | 17 |
| 26 | 12 | 3 | 23 | 11 | 8 | 7 |
| 27 | 12 | 13 | 12 | 11 | 19 | 17 |
| 28 | 12 | 13 | 56 | 11 | 8 | 17 |
| 29 | 12 | 3 | 67 | 11 | 2 | 7 |
| 30 | 12 | 13 | 78 | 11 | 19 | 7 |
| 31 | 12 | 13 | 89 | 11 | 2 | 17 |
| 32 | 12 | 13 | 71 | 11 | 8 | 7 |
| 33 | 12 | 3 | 96 | 11 | 19 | 17 |
| 34 | 12 | 3 | 82 | 11 | 8 | 17 |
| 35 | 12 | 3 | 95 | 11 | 2 | 7 |
| 36 | 12 | 13 | 87 | 11 | 19 | 7 |
| 37 | 12 | 3 | 22 | 11 | 2 | 17 |
| 38 | 13 | 13 | 4 | 11 | 8 | 7 |
| 39 | 13 | 3 | 9 | 11 | 19 | 17 |
| 40 | 13 | 13 | 23 | 11 | 8 | 17 |
| 41 | 13 | 13 | 12 | 11 | 2 | 7 |
| 42 | 13 | 13 | 34 | 11 | 19 | 7 |
| 43 | 13 | 3 | 56 | 11 | 2 | 17 |
| 44 | 13 | 3 | 67 | 11 | 8 | 7 |
| 45 | 13 | 13 | 78 | 11 | 19 | 17 |
| 46 | 13 | 3 | 89 | 11 | 8 | 17 |
| 47 | 13 | 13 | 71 | 11 | 2 | 7 |

| | | | | | | |
|----|----|----|----|----|----|----|
| 48 | 13 | 3 | 96 | 11 | 19 | 7 |
| 49 | 13 | 13 | 82 | 11 | 2 | 17 |
| 50 | 13 | 13 | 95 | 11 | 8 | 7 |
| 51 | 13 | 3 | 87 | 11 | 19 | 17 |
| 52 | 14 | 13 | 4 | 11 | 8 | 17 |
| 53 | 14 | 3 | 9 | 11 | 2 | 7 |
| 54 | 14 | 13 | 23 | 11 | 19 | 7 |
| 55 | 14 | 13 | 12 | 11 | 2 | 17 |
| 56 | 14 | 13 | 34 | 11 | 8 | 7 |
| 57 | 14 | 13 | 56 | 11 | 19 | 17 |
| 58 | 14 | 3 | 67 | 11 | 8 | 17 |
| 59 | 14 | 13 | 78 | 11 | 2 | 7 |
| 60 | 14 | 13 | 89 | 11 | 19 | 7 |
| 61 | 14 | 13 | 71 | 11 | 2 | 17 |
| 62 | 14 | 13 | 82 | 11 | 8 | 7 |
| 63 | 14 | 3 | 95 | 11 | 19 | 17 |
| 64 | 14 | 13 | 87 | 11 | 8 | 17 |
| 65 | 14 | 3 | 22 | 11 | 2 | 7 |
| 66 | 15 | 13 | 4 | 11 | 19 | 7 |
| 67 | 15 | 13 | 9 | 11 | 2 | 17 |
| 68 | 15 | 13 | 23 | 11 | 8 | 7 |
| 69 | 15 | 13 | 12 | 11 | 19 | 17 |
| 70 | 15 | 13 | 34 | 11 | 8 | 17 |
| 71 | 15 | 13 | 56 | 11 | 2 | 7 |
| 72 | 15 | 3 | 67 | 11 | 19 | 7 |
| 73 | 15 | 13 | 99 | 11 | 2 | 17 |
| 74 | 15 | 3 | 89 | 11 | 8 | 7 |
| 75 | 15 | 13 | 71 | 11 | 19 | 17 |
| 76 | 15 | 13 | 96 | 11 | 8 | 17 |
| 77 | 15 | 13 | 95 | 11 | 2 | 7 |
| 78 | 15 | 3 | 87 | 11 | 19 | 7 |
| 79 | 15 | 3 | 22 | 11 | 2 | 17 |
| 80 | 16 | 3 | 4 | 11 | 8 | 7 |
| 81 | 16 | 3 | 9 | 11 | 19 | 17 |
| 82 | 16 | 13 | 23 | 11 | 8 | 17 |
| 83 | 16 | 13 | 12 | 11 | 2 | 7 |
| 84 | 16 | 13 | 56 | 11 | 19 | 7 |
| 85 | 16 | 3 | 67 | 11 | 2 | 17 |
| 86 | 16 | 13 | 78 | 11 | 8 | 7 |
| 87 | 16 | 3 | 99 | 11 | 19 | 17 |
| 88 | 16 | 3 | 89 | 11 | 8 | 17 |
| 89 | 16 | 3 | 71 | 11 | 2 | 7 |
| 90 | 16 | 3 | 96 | 11 | 19 | 7 |
| 91 | 16 | 13 | 82 | 11 | 2 | 17 |
| 92 | 16 | 3 | 95 | 11 | 8 | 7 |
| 93 | 16 | 3 | 22 | 11 | 19 | 17 |
| 94 | 17 | 13 | 4 | 11 | 8 | 17 |
| 95 | 17 | 13 | 9 | 11 | 2 | 7 |
| 96 | 17 | 3 | 23 | 11 | 19 | 7 |

| | | | | | | |
|-----|----|----|----|----|----|----|
| 97 | 17 | 3 | 12 | 11 | 2 | 17 |
| 98 | 17 | 13 | 34 | 11 | 8 | 7 |
| 99 | 17 | 3 | 56 | 11 | 19 | 17 |
| 100 | 17 | 13 | 67 | 11 | 8 | 17 |
| 101 | 17 | 13 | 78 | 11 | 2 | 7 |
| 102 | 17 | 13 | 99 | 11 | 19 | 7 |
| 103 | 17 | 13 | 89 | 11 | 2 | 17 |
| 104 | 17 | 13 | 71 | 11 | 8 | 7 |
| 105 | 17 | 13 | 82 | 11 | 19 | 17 |
| 106 | 17 | 13 | 95 | 11 | 8 | 17 |
| 107 | 17 | 13 | 22 | 11 | 2 | 7 |
| 108 | 18 | 13 | 9 | 11 | 19 | 7 |
| 109 | 18 | 3 | 23 | 11 | 2 | 17 |
| 110 | 18 | 3 | 12 | 11 | 8 | 7 |
| 111 | 18 | 3 | 34 | 11 | 19 | 17 |
| 112 | 18 | 13 | 56 | 11 | 8 | 17 |
| 113 | 18 | 13 | 67 | 11 | 2 | 7 |
| 114 | 18 | 13 | 78 | 11 | 19 | 7 |
| 115 | 18 | 3 | 99 | 11 | 2 | 17 |
| 116 | 18 | 3 | 89 | 11 | 8 | 7 |
| 117 | 18 | 3 | 71 | 11 | 19 | 17 |
| 118 | 18 | 3 | 96 | 11 | 8 | 17 |
| 119 | 18 | 13 | 82 | 11 | 2 | 7 |
| 120 | 18 | 13 | 87 | 11 | 19 | 7 |
| 121 | 18 | 3 | 22 | 11 | 2 | 17 |
| 122 | 19 | 13 | 4 | 11 | 8 | 7 |
| 123 | 19 | 3 | 23 | 11 | 19 | 17 |
| 124 | 19 | 3 | 12 | 11 | 8 | 17 |
| 125 | 19 | 3 | 34 | 11 | 2 | 7 |
| 126 | 19 | 13 | 56 | 11 | 19 | 7 |
| 127 | 19 | 3 | 67 | 11 | 2 | 17 |
| 128 | 19 | 13 | 78 | 11 | 8 | 7 |
| 129 | 19 | 13 | 99 | 11 | 19 | 17 |
| 130 | 19 | 13 | 89 | 11 | 8 | 17 |
| 131 | 19 | 13 | 71 | 11 | 2 | 7 |
| 132 | 19 | 3 | 82 | 11 | 19 | 7 |
| 133 | 19 | 3 | 95 | 11 | 2 | 17 |
| 134 | 19 | 3 | 87 | 11 | 8 | 7 |
| 135 | 19 | 13 | 22 | 11 | 19 | 17 |
| 136 | 20 | 3 | 4 | 11 | 8 | 17 |
| 137 | 20 | 3 | 9 | 11 | 2 | 7 |
| 138 | 20 | 3 | 23 | 11 | 19 | 7 |
| 139 | 20 | 3 | 34 | 11 | 2 | 17 |
| 140 | 20 | 13 | 56 | 11 | 8 | 7 |
| 141 | 20 | 3 | 78 | 11 | 19 | 17 |
| 142 | 20 | 3 | 99 | 11 | 8 | 17 |
| 143 | 20 | 13 | 89 | 11 | 2 | 7 |
| 144 | 20 | 3 | 71 | 11 | 19 | 7 |
| 145 | 20 | 13 | 96 | 11 | 2 | 17 |

| | | | | | | |
|-----|----|----|----|----|----|----|
| 146 | 20 | 3 | 82 | 11 | 8 | 7 |
| 147 | 20 | 13 | 95 | 11 | 19 | 17 |
| 148 | 20 | 3 | 87 | 11 | 8 | 17 |
| 149 | 20 | 13 | 22 | 11 | 2 | 7 |
| 150 | 21 | 3 | 4 | 11 | 19 | 7 |
| 151 | 21 | 3 | 9 | 11 | 2 | 17 |
| 152 | 21 | 3 | 12 | 11 | 8 | 7 |
| 153 | 21 | 3 | 34 | 11 | 19 | 17 |
| 154 | 21 | 13 | 56 | 11 | 8 | 17 |
| 155 | 21 | 3 | 67 | 11 | 2 | 7 |
| 156 | 21 | 13 | 78 | 11 | 19 | 7 |
| 157 | 21 | 3 | 99 | 11 | 2 | 17 |
| 158 | 21 | 13 | 89 | 11 | 8 | 7 |
| 159 | 21 | 13 | 71 | 11 | 19 | 17 |
| 160 | 21 | 13 | 96 | 11 | 8 | 17 |
| 161 | 21 | 3 | 95 | 11 | 2 | 7 |
| 162 | 21 | 13 | 87 | 11 | 19 | 7 |
| 163 | 21 | 3 | 22 | 11 | 2 | 17 |
| 164 | 9 | 3 | 34 | 11 | 8 | 7 |
| 165 | 9 | 13 | 12 | 11 | 19 | 17 |
| 166 | 9 | 3 | 9 | 11 | 8 | 17 |
| 167 | 9 | 13 | 4 | 11 | 2 | 7 |
| 168 | 8 | 13 | 22 | 11 | 19 | 7 |
| 169 | 8 | 3 | 87 | 11 | 2 | 17 |
| 170 | 8 | 13 | 82 | 11 | 8 | 7 |
| 171 | 8 | 13 | 96 | 11 | 19 | 17 |
| 172 | 8 | 3 | 71 | 11 | 8 | 17 |
| 173 | 8 | 3 | 89 | 11 | 2 | 7 |
| 174 | 8 | 13 | 78 | 11 | 19 | 7 |
| 175 | 8 | 13 | 56 | 11 | 2 | 17 |
| 176 | 8 | 3 | 34 | 11 | 8 | 7 |
| 177 | 8 | 3 | 12 | 11 | 19 | 17 |
| 178 | 8 | 3 | 23 | 11 | 8 | 17 |
| 179 | 8 | 13 | 9 | 11 | 2 | 7 |
| 180 | 8 | 3 | 4 | 11 | 19 | 7 |
| 181 | 7 | 13 | 87 | 11 | 2 | 17 |
| 182 | 7 | 13 | 95 | 11 | 8 | 7 |
| 183 | 7 | 3 | 82 | 11 | 19 | 17 |
| 184 | 7 | 13 | 96 | 11 | 8 | 17 |
| 185 | 7 | 13 | 71 | 11 | 2 | 7 |
| 186 | 7 | 13 | 78 | 11 | 19 | 7 |
| 187 | 7 | 3 | 67 | 11 | 2 | 17 |
| 188 | 7 | 3 | 56 | 11 | 8 | 7 |
| 189 | 7 | 3 | 34 | 11 | 19 | 17 |
| 190 | 7 | 13 | 12 | 11 | 8 | 17 |
| 191 | 7 | 3 | 23 | 11 | 2 | 7 |
| 192 | 7 | 13 | 9 | 11 | 19 | 7 |
| 193 | 7 | 3 | 4 | 11 | 2 | 17 |
| 194 | 6 | 13 | 22 | 11 | 8 | 7 |

| | | | | | | |
|-----|---|----|----|----|----|----|
| 195 | 6 | 13 | 87 | 11 | 19 | 17 |
| 196 | 6 | 13 | 95 | 11 | 8 | 17 |
| 197 | 6 | 13 | 82 | 11 | 2 | 7 |
| 198 | 6 | 13 | 96 | 11 | 19 | 7 |
| 199 | 6 | 3 | 71 | 11 | 2 | 17 |
| 200 | 6 | 3 | 78 | 11 | 8 | 7 |
| 201 | 6 | 13 | 67 | 11 | 19 | 17 |
| 202 | 6 | 3 | 56 | 11 | 8 | 17 |
| 203 | 6 | 3 | 34 | 11 | 2 | 7 |
| 204 | 6 | 3 | 23 | 11 | 19 | 7 |
| 205 | 6 | 3 | 9 | 11 | 2 | 17 |
| 206 | 6 | 13 | 4 | 11 | 8 | 7 |
| 207 | 4 | 13 | 22 | 11 | 19 | 17 |
| 208 | 4 | 3 | 87 | 11 | 8 | 17 |
| 209 | 4 | 3 | 95 | 11 | 2 | 7 |
| 210 | 4 | 3 | 82 | 11 | 19 | 7 |
| 211 | 4 | 13 | 96 | 11 | 2 | 17 |
| 212 | 4 | 13 | 89 | 11 | 8 | 7 |
| 213 | 4 | 13 | 78 | 11 | 19 | 17 |
| 214 | 4 | 3 | 67 | 11 | 8 | 17 |
| 215 | 4 | 13 | 34 | 11 | 2 | 7 |
| 216 | 4 | 13 | 12 | 11 | 19 | 7 |
| 217 | 4 | 13 | 23 | 11 | 2 | 17 |
| 218 | 4 | 3 | 9 | 11 | 8 | 7 |
| 219 | 4 | 13 | 4 | 11 | 19 | 17 |
| 220 | 3 | 3 | 22 | 11 | 8 | 17 |
| 221 | 3 | 13 | 87 | 11 | 2 | 7 |
| 222 | 3 | 3 | 95 | 11 | 19 | 7 |
| 223 | 3 | 3 | 82 | 11 | 2 | 17 |
| 224 | 3 | 13 | 96 | 11 | 8 | 7 |
| 225 | 3 | 13 | 71 | 11 | 19 | 17 |
| 226 | 3 | 13 | 67 | 11 | 8 | 17 |
| 227 | 3 | 13 | 56 | 11 | 2 | 7 |
| 228 | 3 | 13 | 34 | 11 | 19 | 7 |
| 229 | 3 | 13 | 12 | 11 | 2 | 17 |
| 230 | 3 | 13 | 23 | 11 | 8 | 7 |
| 231 | 3 | 3 | 9 | 11 | 19 | 17 |
| 232 | 3 | 3 | 4 | 11 | 8 | 17 |
| 233 | 2 | 13 | 22 | 11 | 2 | 7 |
| 234 | 2 | 3 | 87 | 11 | 19 | 7 |
| 235 | 2 | 3 | 95 | 11 | 2 | 17 |
| 236 | 2 | 3 | 82 | 11 | 8 | 7 |
| 237 | 2 | 13 | 96 | 11 | 19 | 17 |
| 238 | 2 | 13 | 71 | 11 | 8 | 17 |
| 239 | 2 | 13 | 89 | 11 | 2 | 7 |
| 240 | 2 | 13 | 67 | 11 | 19 | 7 |
| 241 | 2 | 13 | 56 | 11 | 2 | 17 |
| 242 | 2 | 3 | 12 | 11 | 8 | 7 |
| 243 | 2 | 13 | 23 | 11 | 19 | 17 |

| | | | | | | |
|-----|----|----|----|----|----|----|
| 244 | 2 | 13 | 9 | 11 | 8 | 17 |
| 245 | 2 | 13 | 4 | 11 | 2 | 7 |
| 246 | 10 | 13 | 22 | 11 | 19 | 7 |
| 247 | 10 | 3 | 87 | 11 | 2 | 17 |
| 248 | 10 | 13 | 95 | 11 | 8 | 7 |
| 249 | 10 | 3 | 82 | 11 | 19 | 17 |
| 250 | 10 | 13 | 96 | 11 | 8 | 17 |
| 251 | 10 | 3 | 71 | 11 | 2 | 7 |
| 252 | 10 | 3 | 89 | 11 | 19 | 7 |
| 253 | 10 | 13 | 78 | 11 | 2 | 17 |
| 254 | 10 | 13 | 67 | 11 | 8 | 7 |
| 255 | 10 | 13 | 34 | 11 | 19 | 17 |
| 256 | 10 | 3 | 12 | 11 | 8 | 17 |
| 257 | 10 | 3 | 9 | 11 | 2 | 7 |
| 258 | 10 | 13 | 4 | 11 | 19 | 7 |
| 259 | 5 | 13 | 22 | 11 | 2 | 17 |
| 260 | 5 | 13 | 87 | 11 | 8 | 7 |
| 261 | 5 | 3 | 95 | 11 | 19 | 17 |
| 262 | 5 | 13 | 82 | 11 | 8 | 17 |
| 263 | 5 | 3 | 96 | 11 | 2 | 7 |
| 264 | 5 | 3 | 71 | 11 | 19 | 7 |
| 265 | 5 | 13 | 89 | 11 | 2 | 17 |
| 266 | 5 | 3 | 78 | 11 | 8 | 7 |
| 267 | 5 | 3 | 67 | 11 | 19 | 17 |
| 268 | 5 | 3 | 56 | 11 | 8 | 17 |
| 269 | 5 | 13 | 34 | 11 | 2 | 7 |
| 270 | 5 | 3 | 12 | 11 | 19 | 7 |
| 271 | 5 | 3 | 23 | 11 | 2 | 17 |
| 272 | 1 | 3 | 22 | 11 | 8 | 7 |
| 273 | 1 | 13 | 87 | 11 | 19 | 17 |
| 274 | 1 | 13 | 95 | 11 | 8 | 17 |
| 275 | 1 | 13 | 82 | 11 | 2 | 7 |
| 276 | 1 | 13 | 96 | 11 | 19 | 7 |
| 277 | 1 | 13 | 71 | 11 | 2 | 17 |
| 278 | 1 | 13 | 89 | 11 | 8 | 7 |
| 279 | 1 | 3 | 78 | 11 | 19 | 17 |
| 280 | 1 | 13 | 67 | 11 | 8 | 17 |
| 281 | 1 | 13 | 56 | 11 | 2 | 7 |
| 282 | 1 | 3 | 34 | 11 | 19 | 7 |
| 283 | 1 | 13 | 12 | 11 | 2 | 17 |
| 284 | 1 | 13 | 4 | 11 | 8 | 7 |
| 285 | 21 | 3 | 23 | 11 | 19 | 17 |
| 286 | 19 | 13 | 9 | 11 | 8 | 17 |
| 287 | 18 | 13 | 95 | 11 | 2 | 7 |
| 288 | 17 | 3 | 96 | 11 | 19 | 7 |
| 289 | 16 | 13 | 34 | 11 | 2 | 17 |
| 290 | 15 | 13 | 78 | 11 | 8 | 7 |
| 291 | 14 | 3 | 99 | 11 | 19 | 17 |
| 292 | 11 | 3 | 67 | 11 | 8 | 17 |

| | | | | | | |
|-----|----|----|----|----|----|----|
| 293 | 9 | 3 | 23 | 11 | 2 | 7 |
| 294 | 8 | 3 | 95 | 11 | 19 | 7 |
| 295 | 7 | 13 | 89 | 11 | 2 | 17 |
| 296 | 13 | 13 | 22 | 11 | 8 | 7 |
| 297 | 12 | 13 | 99 | 11 | 19 | 17 |
| 298 | 6 | 13 | 89 | 11 | 8 | 17 |
| 299 | 4 | 3 | 71 | 11 | 2 | 7 |
| 300 | 3 | 13 | 78 | 11 | 19 | 7 |
| 301 | 2 | 13 | 34 | 11 | 2 | 17 |
| 302 | 10 | 3 | 56 | 11 | 8 | 7 |
| 303 | 5 | 3 | 9 | 11 | 19 | 17 |
| 304 | 1 | 3 | 23 | 11 | 8 | 17 |
| 305 | 20 | 3 | 67 | 11 | 2 | 7 |
| 306 | 21 | 13 | 82 | 11 | 19 | 7 |
| 307 | 20 | 13 | 12 | 11 | 2 | 17 |
| 308 | 19 | 3 | 96 | 11 | 8 | 7 |
| 309 | 18 | 3 | 4 | 11 | 19 | 17 |
| 310 | 17 | 13 | 87 | 11 | 8 | 17 |
| 311 | 16 | 3 | 87 | 11 | 2 | 7 |
| 312 | 15 | 3 | 82 | 11 | 19 | 7 |
| 313 | 14 | 13 | 96 | 11 | 2 | 17 |
| 314 | 13 | 3 | 99 | 11 | 8 | 7 |
| 315 | 11 | 13 | 71 | 11 | 19 | 17 |
| 316 | 9 | 13 | 95 | 11 | 8 | 17 |
| 317 | 8 | 13 | 67 | 11 | 2 | 7 |
| 318 | 7 | 3 | 22 | 11 | 19 | 7 |
| 319 | 6 | 3 | 12 | 11 | 2 | 17 |
| 320 | 4 | 13 | 56 | 11 | 8 | 7 |
| 321 | 3 | 3 | 89 | 11 | 19 | 17 |
| 322 | 2 | 3 | 78 | 11 | 8 | 17 |
| 323 | 10 | 13 | 23 | 11 | 2 | 7 |
| 324 | 5 | 13 | 4 | 11 | 19 | 7 |
| 325 | 1 | 13 | 9 | 11 | 2 | 17 |
| 326 | 12 | 3 | 34 | 11 | 8 | 7 |

* All values and combinations are valid.

Invalid Constraint Test Cases

View as: > [XML](#) > [DIF](#)

| # | a | b | c | d | e | f |
|----|----|----|----|----|----|----|
| 1 | 1 | 13 | 99 | 11 | 2 | 17 |
| 2 | 5 | 13 | 99 | 11 | 8 | 7 |
| 3 | 10 | 13 | 99 | 11 | 19 | 17 |
| 4 | 2 | 13 | 99 | 11 | 8 | 17 |
| 5 | 3 | 3 | 99 | 11 | 2 | 7 |
| 6 | 4 | 3 | 99 | 11 | 19 | 7 |
| 7 | 6 | 13 | 99 | 11 | 2 | 17 |
| 8 | 7 | 13 | 99 | 11 | 8 | 7 |
| 9 | 8 | 13 | 99 | 11 | 19 | 17 |
| 10 | 9 | 13 | 99 | 11 | 8 | 17 |
| 11 | 11 | 13 | 9 | 11 | 2 | 27 |
| 12 | 11 | 3 | 23 | 11 | 8 | 27 |
| 13 | 11 | 3 | 12 | 11 | 19 | 27 |

* All values are valid, but tuples violate constraints.

Appendix C

PairTest Manual

==== PairTest Testing Tool =====

PairTest Ver 1.1 / 6/16/2002

Copyright (c) North Carolina State University
from 1998 - 2002

The PairTest tool was developed by Dr.K. C. Tai and his students Ho-Yen Chang and Yu Lei at North Carolina State University, with the support of a grant from IBM through CACC (Center for Advanced Computing and Communication) at NCSU.

The PairTest tool was modified to add additional features by Dr.M.A. Vouk and his student Yatin Chalke at North Carolina State University.

* Guide to PairTest *

Version 1 Released on

February 8, 1998

K. C. Tai

Computer Science Department
North Carolina State University
Raleigh, NC 27695-7534 USA
kct@csc.ncsu.edu, (919) 515-7146

Version 1.1 Released on

June 28, 2002

Dr. Mladen Vouk

Computer Science Department
North Carolina State University
Raleigh, NC 27695-7534 USA
vouk@csc.ncsu.edu, (919) 515-7886

1. Introduction

PairTest is a software tool that generates a test set satisfying the pairwise testing strategy for a system. Pairwise testing is a specification-based testing strategy and requires, in principle, that every combination of valid values of any two input parameters of a system be covered by at least one test case. Empirical results show that pairwise testing is practical and effective for various types of software systems [Coh94, Coh96,

Coh97, Tai98].

Major features of PairTest include the following:

- PairTest supports the generation of pairwise test sets for systems with or without existing test sets and for systems modified due to changes of input parameters and/or values.
- PairTest provides information for planning the effort of testing and the order of applying test cases.
- PairTest provide a graphical user interface (GUI) to make the tool easy to use.
- PairTest also provides an option for reading Parameters, Relations or Constraints directly from file thus avoiding manul parameter manipulation with GUI.
- PairTest also provides an option to use GUI or to run the tool from Command Line by specifying the files during runtime to read necessary parameters.
- PairTest is written in Java and thus can run on different platforms.

The PairTest tool was developed by Dr. K. C. Tai and his students Ho-Yen Chang and Yu Lei at North Carolina State University, with the support of a grant from IBM through CACC (Center for Advanced Computing and Communication) at NCSU.

The PairTest tool was modified to add additional features by Dr.M.A. Vouk and his student Yatin Chalke at North Carolina State University.

Section 2 provides details of the pairwise testing strategy.

Section 3 discusses applications of PairTest for different types of test generation. Section 4 explains the functionality of PairTest.

2. The Pairwise Testing Strategy

Below is a procedure for applying pairwise testing to a system with a number of input parameters: (This procedure is slightly different from the one described in [Coh97].)

- (a) For each input parameter, specify a number of valid input values. If a parameter has a large number of valid values, choose representative and boundary values. The first value of each parameter must be a representative value of the parameter.
- (b) Specify a number of relations for input parameters, where a relation is a set of two or more related input parameters. An input parameter may be in two or more relations. If an input parameter does not appear in any relation, it is called a non-interacting parameter. For each relation, constraints can be provided to specify prohibited combinations of values of some parameters in the relation. Each constraint is defined as a set of values for distinct parameters.
- (c) Generate a test set for the system to satisfy the following requirements:
 - (c.1) For each relation, every allowed combination of values of any two parameters in the relation is covered by at least one test,
 - (c.2) For each non-interacting parameter, every value of the parameter is covered by at least one test,
 - (c.3) Each test does not satisfy any constraint for any relation, and
 - (c.4) The first test contains the first value of each parameter.

To illustrate the pairwise testing strategy, consider a system having parameters A, B, and C with their valid values shown

below:

A B C

A1 B1 C1

A2 B2 C2

A3 B3

Now we apply the pairwise testing strategy to each of the following cases.

Case (1): The system has one relation $\{A,C\}$. Six tests are needed for covering combinations of values of A and C. These six tests for $\{A,C\}$ can be extended to cover three values of B. Below is a minimum pairwise test set for the system:

A B C

A1 B1 C1

A1 B2 C2

A2 B3 C1

A2 B3 C2

A3 B3 C1

A3 B3 C2

Case (2): The system has two relations $\{A,C\}$ and $\{B,C\}$. Below is a minimum pairwise test for the system: (The following test set can also be used for case (1).)

A B C

A1 B1 C1

A1 B1 C2

A2 B2 C1

A2 B2 C2

A3 B3 C1
A3 B3 C2

Case (3): The system has one relation {A,B,C}. Nine tests are needed for covering combinations of values of A and B. These nine tests for {A,B} can be extended to cover six combinations of values of A and C as well as six combinations of values of B and C. Below is a minimum pairwise test set for the system:

A B C

A1 B1 C1
A1 B2 C2
A1 B3 C1
A2 B1 C2
A2 B2 C1
A2 B3 C2
A3 B1 C1
A3 B2 C2
A3 B3 C1

Case (4): The system has one relation {A,B,C} with constraint (A3,B3). By deleting (A3,B3,C1) from the test set for case (3), the resulting test set, shown below, is a minimum pairwise test set for the system. Note that the deletion of (A3,B3,C1) does not affect the coverage of (A3,C1) and (B3,C1), since they are covered by (A3,B1,C1) and (A1,B3,C1), respectively.

A B C

A1 B1 C1
A1 B2 C2
A1 B3 C1

A2 B1 C2
A2 B2 C1
A2 B3 C2
A3 B1 C1
A3 B2 C2

3. Applications of PairTest

PairTest supports the following types of specification-based test generation:

- (a) For a system without an existing test set, PairTest generates a pairwise test set, according to the pairwise testing procedure described in section 2. (The test generation algorithm used in PairTest is different from the one in [Coh97] and will be described in a separate report.)
- (b) For a system with an existing test set, PairTest allows two options. One is to generate a new pairwise test set, without considering the existing test set. The other is to keep the existing test set and generate additional tests, if necessary, such that the combined test set is a pairwise test set. Note that the combined test set produced by the second option may be larger than the new test set produced by the first option.
- (c) For a system that has an existing test set and is being modified due to changes of parameters, values, relations and constraints, PairTest allows two options. One is to generate a new pairwise test set, without considering the existing test set. The other is to modify the existing test set, if necessary, and then generate additional tests, if necessary, such that the combined test set is a pairwise test set.

In addition, PairTest allows the generated pairwise test set to

be sorted in increasing distance from the first test, where the distance between two tests is defined as the number of different corresponding values between the two tests. (Note that the first test in the generated test set contains the first value of each parameter.) Applying tests in a sorted test set according to the given order makes debugging easier. After a successful execution of the first test, we apply tests that have the smallest distance >from the first test. If one of these tests results in an erroneous execution, the differences between such a test and the first test can be used to determine possible causes of this erroneous execution. Later, whenever a test has an erroneous execution, we can compare this test with the tests that have been executed successfully and use the information to help debugging.

PairTest can be used for different levels of specification-based testing, including module testing, integration testing, and system testing. Different levels of testing for a system have different sets of input parameters. The number of tests generated for pairwise testing of a program unit depends upon the number of input parameters, the number of values chosen for each input parameter, the number of relations, and the number of parameters in each relation.

PairTest is also useful for specification-based regression testing. To illustrate this, consider the system used earlier, which has parameters A, B, and C with their valid values shown below:

| A | B | C |
|-------|----|----|
| ----- | | |
| A1 | B1 | C1 |
| A2 | B2 | C2 |

A3 B3

Assume that the system has been tested by using the pairwise test set generated by PairTest according to relation $\{A,B,C\}$. Now the system is modified by adding two parameters D and E, as shown below:

A B C D E

A1 B1 C1 D1 E1

A2 B2 C2 D2 E2

A3 B3 D3

Below are some choices for generating tests for the revised system:

- (1) Use PairTest to generate a pairwise test set according to relation $\{A,B,C,D,E\}$. The generated test set contains ten tests. This choice is to check interactions between all parameters and thus ignores the earlier effort for testing interactions between old parameters.
- (2) Use PairTest to generate a pairwise test set according to relation $\{D,E\}$. The generated test set contains six tests. This choice is the opposite of choice (1) and focuses on checking interactions between new parameters only.
- (3) Use PairTest to generate a pairwise test set according to relation $\{D,E\}$ and other relations involving some new and old parameters. This is an approach to checking limited interactions between new and old parameters.
- (4) Delete some values of parameters A, B and C and then use PairTest to generate a pairwise test set according to relation $\{A,B,C,D,E\}$. This is a different approach to checking limited interactions between new and old parameters.

To help the user control the number of generated tests, after the

completion of test generation for a program unit, PairTest shows the number of tests generated for each relation and indicates the relations having the maximum number of tests. Based on this information, the user can decide how to increase or decrease the number of tests, if necessary.

4. Use of PairTest

4.1 Execution of PairTest

(a) To compile PairTest, type "javac TestSet.java" first and then "javac PW_TestMenuFrame.java".

(b) To run PairTest, type "java PW_TestMenuFrame".

The User will be asked to enter a choice regarding if he wants to use the GUI based version or the Command Line version.

If choice "Yes" is entered then a window for PairTest will appear on the screen.

If choice "No" is entered then the Pairtest will be run from the Command line by asking User to enter filenames from which the necessary parameters will be read.

Ex:

Enter ur choice :

Yes for GUI

No to read from file

Enter CHOICE (YES/ NO) :

If yes is entered the PairTest window will appear on screen.

If No is entered User will be prompted from the Command line to enter the appropriate Filenames for appropriate parameters.

The Option (Yes/ No) entered is Case Insensitive.

4.2 Contents of a PairTest File

A file created by PairTest, called a PairTest file, contains the following three parts::

- parameters
- relations and constraints
- test set (optional)

The first two parts together are referred to as the system specification in a PairTest file. If a test set is present, each value in a test is represented by a non-negative integer, with "0" denoting the first value of a parameter, "1" the second value of a parameter, and so on. The "test set" part in a PairTest file begins with "mapping information", which indicates the number of parameters, the number of tests, the parameter name for each column number and the value name for each value index of a parameter.

Below are the contents of the PairTest file for case (4) shown in section 2.

PARAMETERS

Number of Parameters :3

param:A-A1|A2|A3|

param:B-B1|B2|B3|

param:C-C1|C2|

RELATIONS

Number of Relations :1

relation:0-(A,B,C,)

CONSTRAINTS

Number of Constraints :1

constraint:0-<A3,B3,*,>

[TESTSET]

NUM OF PARAMETERS:3

NUM OF TESTS:8

0:A(0:A1,1:A2,2:A3,)

1:B(0:B1,1:B2,2:B3,)

2:C(0:C1,1:C2,)

0 0 0

2 0 0

1 0 1

1 2 1

1 1 0

2 1 1

0 1 1

0 2 0

The contents of a PairTest file are not easy to read. PairTest provides a GUI to simplify the creation and editing of a PairTest file. Also, PairTest allows the user to "view" a PairTest file or any of the three parts of a PairTest file in a readable format. Below is the view of the test set part of the above PairTest file.

Test Set (Unsorted)

Number of Parameters:3

Number of Tests:8

Number of tests for each relation:

Relation 0 => 8,

Relations with the largest number of tests:0,

A B C

- - -
- (0) A1 B1 C1
 - (1) A3 B1 C1
 - (2) A2 B1 C2
 - (3) A2 B3 C2
 - (4) A2 B2 C1
 - (5) A3 B2 C2
 - (6) A1 B2 C2
 - (7) A1 B3 C1

Note that in the above view of the test set, the information about whether the test set is sorted and about numbers of tests for relations is provided when test generation is complete, but it is not saved in the PairTest file. Also, the information about numbers of tests for relations is computed according to newly generated tests, without considering tests the existed before test generation.

4.3 File formats for Reading parameters from File.

PairTest can read parameters from file in its GUI/No GUI based Versions thus saving Manual Labour and Time needed to enter large number of parameters.

In the GUI based run of PairTest there will be Text Boxes provided to enter the name of the file for the appropriate window.

For Ex: In the Window to add new parameters we can also enter the name of the file from which the parameters can be read in addition to manually entering the parameters.

In the NON GUI based run of PairTest User will be prompted from Command Line to enter the Filenames to enter the appropriate parameters.

Format of the File to enter PARAMETERS and their VALUES from the file:

PairTest can read parameters from file when to avoid repeated entry of large number of parameters manually.

The name of the File should be provided in the Text Box provided to enter the name of the file in window for Gui based Run or when prompted by user in NO GUI based Run.

The file should contain the name of paramter to be entered as the beginning letter of each line of the file.

The name of the parameter should be followed by its values on the same line, each separated by a single space from each other as well as parameter name.

Each line for the parameter should be terminated by a Carriage return ('\r') or a Line Feed ('\n') character.

Ex:

To enter paramters and their values from a file.

a 1 5 10

b 3 13

c 4 9 23 12

d 11

e 2 8 19

f 7 17 27

| | | |

| -----

| |

| values for the parameter Separated by Single Space and terminated

| by Line Feed Characters

|

Parameter name

Format of the File to enter RELATIONS from the file:

PairTest can read Relations from file when to avoid repeated entry of large number of Interconnected Relations involving same parameters manually.

The name of the File should be provided in the Text Box provided to enter the name of the file in window for GUI based Run or when prompted by user in NO GUI based Run.

The file should contain a single relation on each line of the file. The parameters involved in the relation should be written on the line for corresponding relation separated by a single space from each other.

Each line for the Relation should be terminated by a Carriage return ('\r') or a Line Feed ('\n') character.

Ex:

To enter Relations from a file.

a b c d

e f d

b e

a c e d

||||

|

Single relation Involving paramters a, c, e & d.

Note:

Please note that ther should not be any space after the last parameter in the relation.

Please note that only parameters added earlier can be used to construct a Relation.

Format of the File to enter CONSTRAINTS from the file:

PairTest can read Constraints from file when to avoid repeated entry of large number of Constraints involving same parameters manually.

The name of the File should be provided in the Text Box provided to enter the name of the file in window for GUI based Run or when

prompted by user in NO GUI based Run.

The file should contain a single Constraint on each line of the file. The each constraint should start with the identifier depicting corresponding relation. The Relation Identifiers range from 0 to n-1 for n relations.

After the Identifier the Constraint values for the relation must be placed on, each separated from each other and identifier by a single space, in the appropriate order for each parameter of the relation.

Each line for the Constraint should be terminated by a Carriage return ('\r') or a Line Feed ('\n') character.

Ex:

```
0 1 * 4 *
1 19 27 11
2 3 *
| |
| ---
| |
| Constraint values for parameters ( b =3 , e = *)
|
Identifier for 2 nd Relation which is (b,e)
```

Note:

Please note that there should not be any space after the last parameter value in the constraint.

Please note that only Relations added earlier can be used to construct a Constraint.

4.4 Menus in the PairTest Window

The menu bar in the PairTest window contains six symbols: File, Edit, View, TestGen, and Help. Below we explain the menu for each of these symbols.

File:

- New: to create a new PairTest file. PairTest provides a dialog for entering the name of the new file. This operation closes the file used earlier. PairTest does not support the use of multiple PairTest files at the same time.
- Open: to open an existing PairTest file. The operation closes the file used earlier. The PairTest window shows the complete view of the file, which contains three parts: parameters, relations & constraints, and test set. The user can choose to view only one of these parts by using the "View" menu, which is described later.
- Save: to save the current contents of the PairTest file by overriding the original file.
- Save as: to save the current contents of the PairTest file into a new file.
- Read: to read (not "view") the contents of a selected file. This operation does not affect the file being used. PairTest creates a new window to show the contents of the selected file. The user can execute this operation multiple times in order to read several files at the same time.
- Print: to print the current contents of the PairTest window.
- Close: to close the current PairTest file.

- Exit: to exit PairTest and close all related files and windows.

Edit: To edit parameters, parameter values, relations and constraints in the current PairTest file.

If the file has an existing test set, the user must click "Run" in the "TestGen" menu before closing the file. Otherwise, the system specification and the test set in the file may be inconsistent.

- Replace only: If checked, the user is allowed to replace the names of parameters and/or their values in the current PairTest file. If not checked, the user is allowed to add or delete parameters, parameter values, relations and constraints in the current PairTest file. Note that if this option is checked and the file has an existing test set, the user must click "Run" in the "TestGen" menu before closing the file or changing the value of this option for performing more editing operations. Otherwise, the system specification and the test set in the file may be inconsistent.

- Parameters: to add, delete, and replace parameters and their values in the current PairTest file. PairTest provides a window for performing these operations.

- Relations & Constraints: to add and delete relations and their constraints in the current PairTest file. PairTest provides a window for performing these operations.

Notes:

- (1) The name of a parameter, value or relation can be any sequence of digits and characters (including blanks), excluding the following characters: ":", "-", "|", ",", "(", ")", "<", and ">". Different parameters can have the same value name.
- (2) Relations and constraints can only be defined for parameters

and values that already exist in the file.

- (3) The replacement of the name of a parameter or value by another name is applied to all occurrences of this parameter or value in relations and constraints. The deletion of a parameter results in the deletion of all relations containing this parameter. The deletion of a parameter value results in the deletion of all constraints containing this value.
- (4) Constraints are not allowed to prohibit the test that contains the first value of each parameter.
- (5) PairTest performs editing operations on-the-fly. Thus, after an editing operation has been entered, PairTest immediately performs the operation.
- (6) PairTest does not provide editing operations for creating or modifying a test set in a PairTest file. For example, if the user wants to create a test set in a PairTest file according to a test set that existed before using PairTest, he or she has to do this manually. Manual creation or modification of a test set in a PairTest file, however, is tedious and error-prone.

View:

- Parameters: to view parameters and their values in the current PairTest file.
- Relations & Constraints: to view relations and their constraints in the current PairTest file.
- Test set: to view the test set in the current PairTest file. If the test set was just generated by clicking "Run" in the "TestGen" menu, additional information is provided (see instructions for "Run"), but such information is not saved in the PairTest file.
- Complete file: to view the complete contents of the current PairTest file.

TestGen: To select some of the following options, if necessary, before clicking "Run".

- Max#Tests: to specify the maximum number of tests for a system.

The default value is 500. PairTest does not have other limits except that the machine memory must be large enough to hold all data.

- Sort: If checked, the generated test set is sorted in increasing distance from the first test. If not checked, the generated test set is not sorted. The default value is "checked".

- Extend: This option is considered only if the current PairTest file has an existing test set. If checked, PairTest modifies the existing test set, if necessary, and then generates additional tests, if necessary, such that the combined test set is a pairwise test set. The default value is "not checked".

Below are four possible combinations of values of "Replace only" and "Extend":

- "Replace only" checked and "Extend" not checked: PairTest modifies only the "mapping information" at the beginning of the "test set" part according to the revised PairTest file.

- both "Replace only" and "Extend" checked: PairTest modifies the "mapping information" as described above and then generates additional tests, if necessary, such that the combined test set is a pairwise test set. (This case is useless if the existing test set is a pairwise test set generated earlier by PairTest.)

- "Replace only" not checked and "Extend" checked: see notes below.

- both "Replace only" and "Extend" not checked: PairTest generates a new pairwise test set, without considering the

existing test set.

- Run: to run the test generation algorithm. After completion of test generation, the PairTest window shows the generated test set and the following additional information:

- whether the test set is sorted
- the number of newly generated tests for each relation
- relations having the maximum number of tests
- "*" attached to each test that was also in the previous test set in the PairTest file (Such tests exist only if "Extend" is checked. They may have been modified, according to the notes below, and, if "Sort" is not checked, appear before newly generated tests.)

The above additional information, however, is not saved in the PairTest file.

Notes:

For a PairTest file with an existing test set, if the user has edited the file and then clicks "Run" with "Extend" checked and "Replace only" not checked, PairTest performs the following:

- (1) Compare the old and new lists of parameters. If a parameter appears in the old list, but not in the new list, it is considered to have been deleted. If a parameter appears in the new list, but not in the old list, it is considered to have been added.
- (2) For each parameter in both the old and new lists of parameters, compare the old and new lists of values of this parameter. If a value appears in the old list, but not in the new list, it is considered to have been deleted. If a value appears in the new list, but not in the old list, it is considered to have been added.
- (3) If some parameters have been deleted, modify each test by deleting values for the deleted parameters.

- (4) If some parameter values have been deleted, delete all tests that contain one or more deleted parameter values.
- (5) Delete tests that violate constraints in the revised PairTest file.
- (6) Extend each test by adding one value for each added parameter.
- (7) Add more tests, if necessary, such that the combined test set is a pairwise test set.

Help:

to show this document in a new window. (The name of the file containing this document is "help.txt".)

Additional Help for "No GUI" Version of the Tool:

Inputs in NO GUI Version of Tool :

===== Pairwise Testing Tool =====

Please Enter an Option shown below :

- 1 . To Edit Max#Test Value
- 2 . To Enter Parameters/relations/Constraints from File
- 3 . To Run The code/Generate Test Cases
- 4 . To Read & Load data from a File

- 5 . To Save a File
- 6 . Set the 'Replace Only' flag & Replace Parameters or Their Values
- 7 . Set the 'Extend' flag
- 8 . Edit the File
- 9 . Display Help File
- 10 . Exit

Enter CHOICE :

=====

For 1 . To Edit Max#Test Value

Old system limit is: 500

Please Enter new Limit for Max#Tests

Enter New Limit for Maximum # of Tests :

To change Max#Test Value.

=====

2 . To Enter Parameters/relations/Constraints from File

To Enter Parameters/ Relations / Constraints from Files

Please Enter an Option shown below :

- 1 . To Enter Parameters from File
- 2 . To Enter Relations from File
- 3 . To Enter Constraints from File
- 4 . Exit

Enter CHOICE :

=====

- 3 . To Run The code/Generate Test Cases

To Run the code and Generate the Test cases.

=====

- 4 . To Read & Load data from a File

To Read from a pairwise File and Load corresponding data .

Enter the Name of the File to be Read :

=====

- 5 . To Save a File

Enter the Name for the File to be Saved :

6 . Set the 'Replace Only' flag & Replace Parameters or Their Values

Setting the 'Replace Only' flag

Do you want to Replace Parameter or Value?

Enter 'P' to Replace Parameter

Enter 'V' to Replace Parameter Value

Enter 'Q' to Quit

Enter CHOICE (P/V/Q):

7 . Set the 'Extend' flag

To set the Extend Flag.

Be sure to run the code to generate new test cases after Setting of Extend flag.

8 . Edit the File

==== Pairwise Testing Tool =====

Please Enter an Option shown below :

- 1 . To Add New Parameter and its Values
- 2 . To Delete a parameter
- 3 . To Delete a parameter Value
- 4 . To Add a relation
- 5 . To delete a Relation
- 6 . To Add a Constraint
- 7 . To delete a Constraint
- 8 . To View contents of file
- 9 . Return to Main Menu

Enter CHOICE :

- 1 . To Add New Parameter and its Values

Enter the New Parameter :

Enter the new parameter name.

- 2 . To Delete a parameter

Enter the parameter Name to delete.

Enter the Paramter to be Deleted :

3 . To Delete a parameter Value

Enter the Parameter for which Value is to be Deleted :

4 . To Add a relation

Please Enter the Relation on a Single line with the Parameters in the Relation Separated by Space.

There MUST NOT be any space after the Last parameter in the Relation.

Enter the Relation :

Sample i/p:

Enter the Relation :a b c

This will enter relation (a, b, c).

5 . To delete a Relation

Enter the Relation ID to Delete The Relation :

Sample i/p :

Enter the Relation ID to Delete The Relation :2

This will delete Relation 2.

Relation ID can be seen by "View Contents of File " Option.

6 . To Add a Constraint

Please Enter the Constraint on a Single line starting with the Relation ID for which Constraint is to be added.

Relation ID MUST BE followed by Parameter Values for the Relation Separated by a single Space.

There MUST NOT be any space after the Last parameter in the Constraint.

Enter the Constraint :

Sample i/p:

Enter the Constraint :2 3 * 4

This will add Constraint for Relation "2" and the constraint added will be (3, *, 4).

7 . To delete a Constraint

Please Enter the Constraint on a Single line with Parameter Values for the Relation Separated by a single Space.

There MUST NOT be any space after the Last parameter in the Constraint.

Enter the Relation ID for which Constraint is to be deleted :

Enter the Constraint to be Deleted:

Sample i/p:

Enter the Relation ID for which Constraint is to be deleted :2

Enter the Constraint to be Deleted:3 * 4

This will delete the Constraint (3, *, 4) for Relation "2".

8 . To View contents of file

To display the contents of current Pairtest File.

9 . Return to Main Menu

=====

9 . Display Help File

To display Help File.

=====

10. Exit

To Exit the tool

Acknowledgments

Isaac Allen, Prakash Devalapalli, and James Williams, Jr. at IBM have provided helpful suggestions on the design of the PairTest tool.

Dr.K. C. Tai and his students Ho-Yen Chang and Yu Lei at North Carolina State University for making the Tool available for further modifications.

Dr.Vouk for his very helpful guidance and suggestions to add new features in the Tool.